**SQL NOTES**

Keep in Mind That...

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

In this tutorial we will write all SQL keywords in upper-case.

Semicolon after SQL Statements?

Some database systems require a semicolon at the end of each SQL statement.

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

In this tutorial, we will use semicolon at the end of each SQL statement.

Some of The Most Important SQL Commands

* **SELECT** - extracts data from a database
* **UPDATE** - updates data in a database
* **DELETE** - deletes data from a database
* **INSERT INTO** - inserts new data into a database
* **CREATE DATABASE** - creates a new database
* **ALTER DATABASE** - modifies a database
* **CREATE TABLE** - creates a new table
* **ALTER TABLE** - modifies a table
* **DROP TABLE** - deletes a table
* **CREATE INDEX** - creates an index (search key)
* **DROP INDEX** - deletes an index

## The SQL SELECT Statement

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

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

### **SQL SELECT Syntax**

SELECT column\_name,column\_name  
FROM table\_name;

and

SELECT \* FROM table\_name;

## The SQL SELECT DISTINCT Statement

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

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

### **SQL SELECT DISTINCT Syntax**

SELECT DISTINCT column\_name,column\_name  
FROM table\_name;

## The SQL WHERE Clause

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

### **SQL WHERE Syntax**

SELECT column\_name,column\_name  
FROM table\_name  
WHERE column\_name operator value;

## The SQL AND & OR Operators

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

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

## The SQL ORDER BY Keyword

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

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

### **SQL ORDER BY Syntax**

SELECT column\_name,column\_name  
FROM table\_name  
ORDER BY column\_name,column\_name ASC|DESC;

🡪 SELECT \* FROM People ORDER BY FirstName DESC, YearOfBirth ASC

## The SQL INSERT INTO Statement

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

### **SQL INSERT INTO Syntax**

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

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

INSERT INTO table\_name  
VALUES (value1,value2,value3,...);

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

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

**🡪** The CustomerID column is automatically updated with a unique number for each record in the table.

## The SQL UPDATE Statement

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

### **SQL UPDATE Syntax**

UPDATE table\_name  
SET column1=value1,column2=value2,...  
WHERE some\_column=some\_value;

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

## The SQL DELETE Statement

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

### **SQL DELETE Syntax**

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

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

## The SQL SELECT TOP Clause

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

The SELECT TOP clause can be very useful on large tables with thousands of records. Returning a large number of records can impact on performance.

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

### **SQL Server / MS Access Syntax**

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

### **MySQL Syntax**

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

## The SQL LIKE Operator

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

### **SQL LIKE Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name LIKE pattern;

## The SQL LIKE Operator

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

### **SQL LIKE Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name LIKE pattern;

**#**Selects all customers with a City starting with the letter "s":

**🡪** SELECT \* FROM Customers WHERE City LIKE 's%';

**#**Selects all customers with a City ending with the letter "s":

**🡪** SELECT \* FROM Customers WHERE City LIKE '%s';

**#**Selects all customers with a Country containing the pattern "land":

**🡪** SELECT \* FROM Customers WHERE Country LIKE '%land%';

**#**Selects all customers with a Country NOT containing the pattern "land":

**🡪** SELECT \* FROM Customers WHERE Country NOT LIKE '%land%';

## SQL Wildcard Characters

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

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

With SQL, the wildcards are:

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

**#** Selects all customers with a City starting with any character, followed by "erlin":

**🡪** SELECT \* FROM Customers WHERE Country LIKE '\_erlin';

**#** Selects all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on":

**🡪** SELECT \* FROM Customers WHERE City LIKE 'L\_n\_on';

**#** Selects all customers with a City starting with "b", "s", or "p":

**🡪** SELECT \* FROM Customers WHERE City LIKE '[bsp]%';

**#** Selects all customers with a City starting with "b", "s", or "p":

**🡪** SELECT \* FROM Customers WHERE City LIKE '[a-c]%';

**#** Selects all customers with a City not starting with "b", "s", or "p":

**🡪** SELECT \* FROM Customers WHERE City NOT LIKE '[a-c]%';

🡪 SELECT \* FROM Customers WHERE City LIKE '[!a-c]%';

## 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,...);

## The SQL BETWEEN Operator

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

### **SQL BETWEEN Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_nameBETWEEN value1 AND value2;

## SQL Aliases

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

Basically aliases are created to make column names more readable.

### **SQL Alias Syntax for Columns**

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

### **SQL Alias Syntax for Tables**

SELECT column\_name(s)  
FROM table\_nameAS alias\_name;

## SQL JOIN

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

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

Different SQL JOINs

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

## SQL INNER JOIN Keyword

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

### **SQL INNER JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
INNER JOIN table2  
ON table1.column\_name=table2.column\_name;

SELECT column\_name(s)  
FROM table1  
JOIN table2  
ON table1.column\_name=table2.column\_name;

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

**Note:** The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are rows in the "Customers" table that do not have matches in "Orders", these customers will NOT be listed.

## SQL LEFT JOIN Keyword

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

### **SQL LEFT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2  
ON table1.column\_name=table2.column\_name;

SELECT column\_name(s)  
FROM table1  
LEFT OUTER JOIN table2  
ON table1.column\_name=table2.column\_name;

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

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

## SQL RIGHT JOIN Keyword

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

### **SQL RIGHT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2  
ON table1.column\_name=table2.column\_name;

SELECT column\_name(s)  
FROM table1  
RIGHT OUTER JOIN table2  
ON table1.column\_name=table2.column\_name;

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

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

## SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword returns all rows from the left table (table1) and from the right table (table2).

The FULL OUTER JOIN keyword combines the result of both LEFT and RIGHT joins.

### **SQL FULL OUTER JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
FULL OUTER JOIN table2  
ON table1.column\_name=table2.column\_name;

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

## The SQL UNION Operator

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

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

### **SQL UNION Syntax**

SELECT column\_name(s) FROM table1  
UNION  
SELECT column\_name(s) FROM table2;

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

### **SQL UNION ALL Syntax**

SELECT column\_name(s) FROM table1  
UNION ALL  
SELECT column\_name(s) FROM table2;

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

**Note:** UNION cannot be used to list ALL cities from the two tables. If several table\_1 and table\_2 share the same city, each city will only be listed once. UNION selects only distinct values. Use UNION ALL to also select duplicate values!

## The SQL SELECT INTO Statement

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

### **SQL SELECT INTO Syntax**

We can copy all columns into the new table:

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

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

SELECT column\_name(s)  
INTO newtable [IN externaldb]  
FROM table1;

The new table will be created with the column-names and types as defined in the SELECT statement. You can apply new names using the AS clause.

## SQL SELECT INTO Examples

Create a backup copy of Customers:

SELECT \*  
INTO CustomersBackup2013  
FROM Customers;

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

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

Copy only a few columns into the new table:

SELECT CustomerName, ContactName  
INTO CustomersBackup2013  
FROM Customers;

Copy only the German customers into the new table:

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

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

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

**Tip:** The SELECT INTO statement can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

SELECT \*  
INTO newtable  
FROM table1  
WHERE 1=0;

## The SQL INSERT INTO SELECT Statement

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

### **SQL INSERT INTO SELECT Syntax**

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

INSERT INTO table2  
SELECT \* FROM table1;

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

INSERT INTO table2  
(column\_name(s))  
SELECT column\_name(s)  
FROM table1;

## The SQL CREATE DATABASE Statement

The CREATE DATABASE statement is used to create a database.

### **SQL CREATE DATABASE Syntax**

CREATE DATABASE dbname;

## The SQL CREATE TABLE Statement

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

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

### **SQL CREATE TABLE Syntax**

CREATE TABLE table\_name  
(  
column\_name1 data\_type(size),  
column\_name2 data\_type(size),  
column\_name3 data\_type(size),  
....  
);

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

## SQL Constraints

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

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

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

### **SQL CREATE TABLE + CONSTRAINT Syntax**

CREATE TABLE table\_name  
(  
column\_name1 data\_type(size) constraint\_name,  
column\_name2 data\_type(size) constraint\_name,  
column\_name3 data\_type(size) constraint\_name,  
....  
);

 In SQL, we have the following constraints:

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

## SQL NOT NULL Constraint

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.

## 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:

**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)  
)

**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)  
)

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.

Most tables 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)  
)

**Note:** In the example above there is only ONE PRIMARY KEY (pk\_PersonID). However, the value of the pk\_PersonID is made up of two columns (P\_Id and 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 invalid data from 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 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 TABLE Persons  
ALTER City SET DEFAULT 'SANDNES'

**SQL Server / MS Access:**

ALTER TABLE Persons  
ALTER COLUMN City SET DEFAULT 'SANDNES'

**Oracle:**

ALTER TABLE Persons  
MODIFY City DEFAULT 'SANDNES'

## To DROP a DEFAULT Constraint

To drop a DEFAULT constraint, use the following SQL:

**MySQL:**

ALTER TABLE Persons  
ALTER City DROP DEFAULT

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

ALTER TABLE Persons  
ALTER COLUMN City DROP DEFAULT

## The DROP TABLE Statement

The DROP TABLE statement is used to delete a table.

DROP TABLE table\_name

## The DROP DATABASE Statement

The DROP DATABASE statement is used to delete a database.

DROP DATABASE database\_name

## The TRUNCATE TABLE Statement

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

Then, use the TRUNCATE TABLE statement:

TRUNCATE TABLE table\_name

## The ALTER TABLE Statement

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

### **SQL ALTER TABLE Syntax**

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

ALTER TABLE table\_name  
ADD column\_name datatype

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

ALTER TABLE table\_name  
DROP COLUMN column\_name

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

**SQL Server / MS Access:**

ALTER TABLE table\_name  
ALTER COLUMN column\_name datatype

**My SQL / Oracle:**

ALTER TABLE table\_name  
MODIFY COLUMN column\_name datatype

**Oracle 10G and later:**

ALTER TABLE table\_name  
MODIFY 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](http://www.w3schools.com/sql/sql_datatypes.asp).

The "Persons" 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 "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 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 |

## AUTO INCREMENT a Field

Very often we would like the value of the primary key field to be created automatically every time a new record is inserted.

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

## Syntax for MySQL

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

CREATE TABLE Persons  
(  
ID int NOT NULL AUTO\_INCREMENT,  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255),  
PRIMARY KEY (ID)  
)

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

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

To let the AUTO\_INCREMENT sequence start with another value, use the following SQL statement:

ALTER TABLE Persons AUTO\_INCREMENT=100

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

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

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

## Syntax for SQL Server

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

CREATE TABLE Persons  
(  
ID int IDENTITY(1,1) PRIMARY KEY,  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255)  
)

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

In the example above, the starting value for IDENTITY is 1, and it will increment by 1 for each new record.

**Tip:** To specify that the "ID" column should start at value 10 and increment by 5, change it to IDENTITY(10,5).

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

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

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

## Syntax for Access

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

CREATE TABLE Persons  
(  
ID Integer 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.

**Tip:** To specify that the "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 "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 (ID,FirstName,LastName)  
VALUES (seq\_person.nextval,'Lars','Monsen')

The SQL statement above would insert a new record into the "Persons" table. The "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 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

**Tip:** The aggregate functions and the scalar functions will be explained in details in the next chapters.

## 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 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.

## The MAX() Function

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

### **SQL MAX() Syntax**

SELECT MAX(column\_name) FROM table\_name;

## The MIN() Function

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

### **SQL MIN() Syntax**

SELECT MIN(column\_name) FROM table\_name;

## The SUM() Function

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

### **SQL SUM() Syntax**

SELECT SUM(column\_name) FROM table\_name;

## The GROUP BY Statement

Aggregate functions often need an added 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;

## The HAVING Clause

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

### **SQL HAVING Syntax**

SELECT column\_name, aggregate\_function(column\_name)  
FROM table\_name  
WHERE column\_name operator value  
GROUP BY column\_name  
HAVING aggregate\_function(column\_name) operator value;