The SQL CREATE DATABASE Statement

The CREATE DATABASE statement is used to create a database.

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;

Database tables can be added with the CREATE TABLE statement.

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.

Tip: For an overview of the data types available in MS Access, MySQL, and SQL Server, go to our complete Data Types Reference.

SQL CREATE TABLE Example

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

We use the following CREATE TABLE statement:

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

The PersonID column is of type int and will hold an integer.

The LastName, FirstName, Address, and City columns are of type varchar and will hold characters, and the maximum length for these fields is 255 characters.

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

PersonID	LastName	FirstName	Address	City

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

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 rows 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

The next chapters will describe each constraint in detail.

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)
)
```

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
```

The DROP INDEX 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:

SQL Server / MS Access:

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype
```

My SQL / Oracle:

```
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.

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

Microsoft Access Data Types

Data type	Description	Storage
Text	Use for text or combinations of text and numbers. 255 characters maximum	
Memo	Memo is used for larger amounts of text. Stores up to 65,536 characters. **Note:** You cannot sort a memo field. However, they are searchable	
Byte	Allows whole numbers from 0 to 255	1 byte

Integer	Allows whole numbers between -32,768 and 32,767	2 bytes
Long	Allows whole numbers between -2,147,483,648 and 2,147,483,647	4 bytes
Single	Single precision floating-point. Will handle most decimals	4 bytes
Double	Double precision floating-point. Will handle most decimals	8 bytes
Currency	Use for currency. Holds up to 15 digits of whole dollars, plus 4 decimal places. **Tip:** You can choose which country's currency to use	8 bytes
AutoNumber	AutoNumber fields automatically give each record its own number, usually starting at 1	4 bytes
Date/Time	Use for dates and times	8 bytes
Yes/No	A logical field can be displayed as Yes/No, True/False, or On/Off. In code, use the constants True and False (equivalent to -1 and 0). **Note:** Null values are not allowed in Yes/No fields	1 bit
Ole Object	Can store pictures, audio, video, or other BLOBs (Binary Large OBjects)	up to 1GB
Hyperlink	Contain links to other files, including web pages	
Lookup Wizard	Let you type a list of options, which can then be chosen from a drop-down list	4 bytes

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

String types:

Data type	Description	Storage
char(n)	Fixed width character string. Maximum 8,000 characters	Defined width
varchar(n)	Variable width character string. Maximum 8,000 characters	2 bytes + number of chars
varchar(max)	Variable width character string. Maximum 1,073,741,824 characters	2 bytes + number of chars
text	Variable width character string. Maximum 2GB of text data	4 bytes + number of chars
nchar	Fixed width Unicode string. Maximum 4,000 characters	Defined width x 2
nvarchar	Variable width Unicode string. Maximum 4,000 characters	
nvarchar(max)	Variable width Unicode string. Maximum 536,870,912 characters	
ntext	Variable width Unicode string. Maximum 2GB of text data	
bit	Allows 0, 1, or NULL	
binary(n)	Fixed width binary string. Maximum 8,000 bytes	
varbinary	Variable width binary string. Maximum 8,000 bytes	

varbinary(max)	Variable width binary string. Maximum 2GB
image	Variable width binary string. 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

« Previous
Next Chapter »

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.

SQL AVG() Function

« Previous

Next Chapter »

The AVG() Function

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

SQL AVG() Syntax

```
SELECT AVG(column_name) FROM table_name
```

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10

4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	21.35
5	Chef Anton's Gumbo Mix	2	2	36 boxes	25

SQL AVG() Example

The following SQL statement gets the average value of the "Price" column from the "Products" table:

Example

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

SQL COUNT() Function

« Previous

Next Chapter »

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.

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10265	7	2	1996-07-25	1
10266	87	3	1996-07-26	3
10267	25	4	1996-07-29	1

SQL COUNT(column_name) Example

The following SQL statement counts the number of orders from "CustomerID"=7 from the "Orders" table:

Example

```
SELECT COUNT(CustomerID) AS OrdersFromCustomerID7 FROM Orders
WHERE CustomerID=7;
```

SQL MAX() Function

« Previous

Next Chapter »

The MAX() Function

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

SQL MAX() Syntax

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

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	21.35
5	Chef Anton's Gumbo Mix	2	2	36 boxes	25

SQL MAX() Example

The following SQL statement gets the largest value of the "Price" column from the "Products" table:

Example

SELECT MAX(Price) AS HighestPrice FROM Products;

SQL MIN() Function

« Previous

Next Chapter »

The MIN() Function

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

SQL MIN() Syntax

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

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	21.35
5	Chef Anton's Gumbo Mix	2	2	36 boxes	25

SQL MIN() Example

The following SQL statement gets the smallest value of the "Price" column from the "Products" table:

Example

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

SQL SUM() Function

« Previous

Next Chapter »

The SUM() Function

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

SQL SUM() Syntax

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

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "OrderDetails" table:

OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10

3	10248	72	5
4	10249	14	9
5	10249	51	40

SQL SUM() Example

The following SQL statement finds the sum of all the "Quantity" fields for the "OrderDetails" table:

Example

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

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;
```

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

And a selection from the "Shippers" table:

ShipperID	ShipperName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo	Notes
1	Davolio	Nancy	1968-12-08	EmpID1.pic	Education includes a BA....
2	Fuller	Andrew	1952-02-19	EmpID2.pic	Andrew received his BTS....
3	Leverling	Janet	1963-08-30	EmpID3.pic	Janet has a BS degree....

SQL GROUP BY Example

Now we want to find the number of orders sent by each shipper.

The following SQL statement counts as orders grouped by shippers:

Example

```
SELECT Shippers.ShipperName,COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders
LEFT JOIN Shippers
```

```
ON Orders.ShipperID=Shippers.ShipperID
GROUP BY ShipperName;
```

Try it yourself »

GROUP BY More Than One Column

We can also use the GROUP BY statement on more than one column, like this:

Example

```
SELECT Shippers.ShipperName, Employees.LastName,
COUNT(Orders.OrderID) AS NumberOfOrders
FROM ((Orders
INNER JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID)
INNER JOIN Employees
ON Orders.EmployeeID=Employees.EmployeeID)
GROUP BY ShipperName,LastName;
```

SQL HAVING Clause

« Previous

Next Chapter »

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;
```

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo	Notes
1	Davolio	Nancy	1968-12-08	EmpID1.pic	Education includes a BA....
2	Fuller	Andrew	1952-02-19	EmpID2.pic	Andrew received his BTS....
3	Leverling	Janet	1963-08-30	EmpID3.pic	Janet has a BS degree....

SQL HAVING Example

Now we want to find if any of the customers have a total order of less than 2000.

We use the following SQL statement:

The following SQL statement finds if any of the employees has registered more than 10 orders:

Example

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders FROM (Orders
INNER JOIN Employees
ON Orders.EmployeeID=Employees.EmployeeID)
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 10;
```

Try it yourself »

Now we want to find the if the employees "Davolio" or "Fuller" have more than 25 orders

We add an ordinary WHERE clause to the SQL statement:

Example

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders
INNER JOIN Employees
ON Orders.EmployeeID=Employees.EmployeeID
WHERE LastName='Davolio' OR LastName='Fuller'
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 25;
```

The 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;
```

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

---

SQL UCASE() Example

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table, and converts the "CustomerName" column to uppercase:

Example

```
SELECT UCASE(CustomerName) AS Customer, City
FROM Customers;
```

SQL Joins

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.

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, have a look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Notice that the "CustomerID" column in the "Orders" table refers to the customer in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, if we run the following SQL statement (that contains an INNER JOIN):

Example

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers
ON Orders.CustomerID=Customers.CustomerID;
```

it will produce something like this:

OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996
10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996

Different SQL JOINs

Before we continue with examples, we will list the types the different SQL JOINs you can use:

- **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*;

or:

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

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*;

or:

SELECT *column_name(s)*
FROM *table1*
LEFT OUTER JOIN *table2*
ON *table1.column_name=table2.column_name*;

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*;

or:

SELECT *column_name(s)*
FROM *table1*
RIGHT OUTER JOIN *table2*
ON *table1.column_name=table2.column_name*;

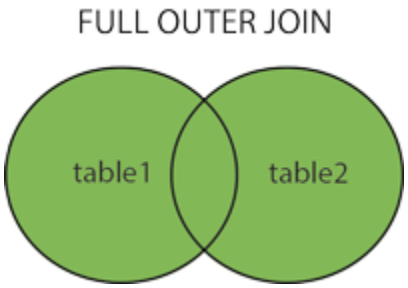
SQL FULL OUTER JOIN Keyword

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*;



Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1

10310 77 8 1996-09-20 2

SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders
ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

A selection from the result set may look like this:

CustomerName	OrderID
Alfreds Futterkiste	
Ana Trujillo Emparedados y helados	10308
Antonio Moreno Taquería	10365
	10382
	10351

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.

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Suppliers" table:

SupplierID	SupplierName	ContactName	Address	City	PostalCode	Country

1	Exotic Liquid	Charlotte Cooper	49 Gilbert St.	Londona	EC1 4SD	UK
2	New Orleans Cajun Delights	Shelley Burke	P.O. Box 78934	New Orleans	70117	USA
3	Grandma Kelly's Homestead	Regina Murphy	707 Oxford Rd.	Ann Arbor	48104	USA

SQL UNION Example

The following SQL statement selects all the **different** cities (only distinct values) from the "Customers" and the "Suppliers" tables:

Example

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

SQL UNION ALL Example

The following SQL statement uses UNION ALL to select **all** (duplicate values also) cities from the "Customers" and "Suppliers" tables:

Example

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

Try it yourself »

SQL UNION ALL With WHERE

The following SQL statement uses UNION ALL to select **all** (duplicate values also) **German** cities from the "Customers" and "Suppliers" tables:

Example

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

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;
```

Tip: 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;
```