SQL COMMAND

SQL SELECT Statement

Is used to select data from a database. The data returned is stored in a result table, called the result-set.

SELECT *column1*, *column2, ...*
FROM *table_name*;

Select all the fields available in the table

SELECT * FROM *table_name*;

SQL SELECT DISTINCT Statement

Is used to return only distinct (different) values.

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

SQL WHERE Clause

Is used to filter records and extract only those records that fulfill a specified condition.

SELECT *column1*, *column2, ...*
FROM *table_name*
WHERE *condition*;

Is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!

Operators in WHERE Clause

The following operators can be used in the WHERE clause:

=	Equal
<>	Not equal. (may be written as !=)
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

SQL AND, OR and NOT Operators

WHERE clause can be combined with AND, OR, and NOT operators.

AND and OR operators are used to filter records based on more than one condition:

- AND operator displays a record if all the conditions separated by AND is TRUE.

- OR operator displays a record if any of the conditions separated by OR is TRUE.

NOT operator displays a record if the condition(s) is NOT TRUE.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

SQL ORDER BY Keyword

Is used to sort the result-set in ascending or descending order.

Sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

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

SQL INSERT INTO Statement

#1

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

#2

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

NULL Value

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

SQL UPDATE Statement

Is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

SQL DELETE Statement

Is used to delete existing records in a table.

```
DELETE FROM table_name
WHERE condition;
```

SQL SELECT TOP Clause

Is used to specify the number of records to return.

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

SQL MIN() and MAX() Functions

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

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

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

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

SQL COUNT(), AVG() and SUM() Functions

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

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

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

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

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

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

Wildcard Characters

Is used to substitute any other character(s) in a string.

SQL LIKE Operator

Is used in a WHERE clause to search for a specified pattern in a column.

Two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

Some examples showing different LIKE operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"

WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

SQL IN Operator

Allows you to specify multiple values in a WHERE clause.

Is a shorthand for multiple OR conditions.

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

or:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

SQL BETWEEN Operator

Selects values within a given range. The values can be numbers, text, or dates.

Is inclusive: begin and end values are included.

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

EXAMPLE

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

```
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

```
SELECT * FROM Products
WHERE (Price BETWEEN 10 AND 20)
AND NOT CategoryID IN (1,2,3);
```

```
SELECT * FROM Products
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'
ORDER BY ProductName;
```

```
SELECT * FROM Products
WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'
ORDER BY ProductName;
```

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN #07/04/1996# AND #07/09/1996#;
```

SQL Aliases

Are used to give a table, or a column in a table, a temporary name.

Are often used to make column names more readable.

An alias only exists for the duration of the query.

Alias Column

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

Alias Table

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

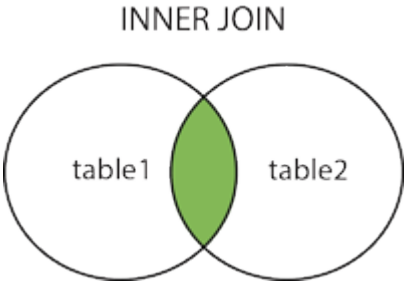
SQL JOIN

Is used to combine rows from two or more tables, based on a related column between them.

SQL INNER JOIN Keyword

Selects records that have matching values in both tables.

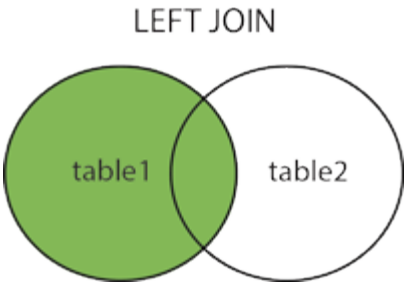
```
SELECT column_name(s)
FROM table1
INNER JOIN table2 ON table1.column_name = table2.column_name;
```



SQL LEFT JOIN Keyword

Returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

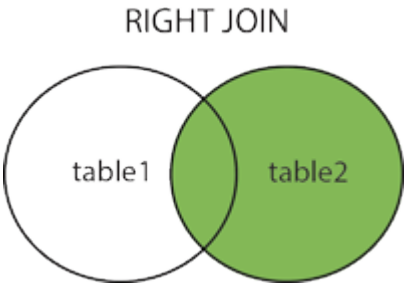
```
SELECT column_name(s)
FROM table1
LEFT JOIN table2 ON table1.column_name = table2.column_name;
```



SQL RIGHT JOIN Keyword

Returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

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

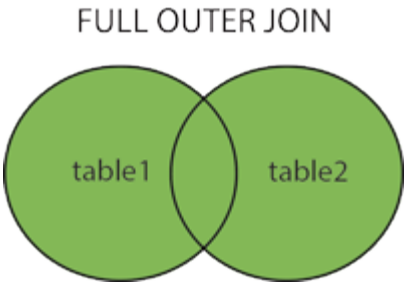


SQL FULL OUTER JOIN Keyword

Return all records when there is a match in either left (table1) or right (table2) table records.

FULL OUTER JOIN can potentially return very large result-sets!

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



SQL Self JOIN

Is a regular join, but the table is joined with itself.

SELECT *column_name(s)*
FROM *table1 T1, table1 T2*
WHERE *condition*;

SQL UNION Operator

Is used to combine the result-set of two or more SELECT statements.

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

SELECT *column_name(s)* FROM *table1*
UNION
SELECT *column_name(s)* FROM *table2*;

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

SELECT *column_name(s)* FROM *table1*
UNION ALL
SELECT *column_name(s)* FROM *table2*;

SQL GROUP BY Statement

Is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG…) to group the result-set by one or more columns.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

SQL HAVING Clause

Was added to SQL because the WHERE keyword could not be used with aggregate functions.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

SQL SELECT INTO Statement

Copies data from one table into a new table.

Copy all columns into a new table:

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

Copy only some columns into a new table:

```
SELECT column1, column2, column3, ...
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

SQL INSERT INTO SELECT Statement

Copies data from one table and inserts it into another table.

- INSERT INTO SELECT requires that data types in source and target tables match
- The existing records in the target table are unaffected

Copy all columns from one table to another table:

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

Copy only some columns from one table into another table:

```
INSERT INTO table2 (column1, column2, column3, ...)
SELECT column1, column2, column3, ...
FROM table1
WHERE condition;
```

SQL CREATE DATABASE Statement

Is used to create a new SQL database.

```
CREATE DATABASE databasename;
```

SQL DROP DATABASE Statement

Is used to drop an existing SQL database.

```
DROP DATABASE databasename;
```

SQL CREATE TABLE Statement

Is used to create a new table in a database.

```
CREATE TABLE table_name (
   column1 datatype,
   column2 datatype,
   column3 datatype,
  ....
);
```

SQL DROP TABLE Statement

Is used to drop an existing table in a database.

```
DROP TABLE table_name;
```

SQL ALTER TABLE Statement

Is used to add, delete, or modify columns in an existing table.

Is also used to add and drop various constraints on an existing table.

ALTER TABLE - ADD Column

To add a column in a table

ALTER TABLE *table_name*
ADD *column_name datatype*;

ALTER TABLE - DROP COLUMN

To delete a column in a table

ALTER TABLE *table_name*
DROP COLUMN *column_name*;

ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table

ALTER TABLE *table_name*
MODIFY COLUMN *column_name datatype*;

OR

ALTER TABLE *table_name*
ALTER COLUMN *column_name datatype*;

SQL Constraints

SQL Create Constraints

Can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement. CREATE TABLE *table_name* (
column1 datatype constraint,
column2 datatype constraint,
column3 datatype constraint,
....

);

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Uniquely identifies a row/record in another table
- CHECK - Ensures that all values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column when no value is specified
- INDEX - Use to create and retrieve data from the database very quickly

SQL NOT NULL Constraint

By default, a column can hold NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values.

If the table has already been created, you can add a NOT NULL constraint to a column with the ALTER TABLE statement.

SQL UNIQUE Constraint

The UNIQUE constraint ensures that all values in a column are different.

SQL UNIQUE Constraint on CREATE TABLE

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

```
CREATE TABLE Persons (
   ID int NOT NULL UNIQUE,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int
);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
```

```
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

SQL UNIQUE Constraint on ALTER TABLE

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

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName)
```

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE Persons
DROP INDEX UC_Person;
```

OR

```
ALTER TABLE Persons
DROP CONSTRAINT UC_Person;
```

SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only one primary key, which may consist of single or multiple fields.

SQL PRIMARY KEY on CREATE TABLE

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

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

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

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

SQL PRIMARY KEY on ALTER TABLE

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

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

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

```
ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
```

DROP a PRIMARY KEY Constraint

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

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

OR

```
ALTER TABLE Persons
DROP CONSTRAINT PK_Person;
```

SQL FOREIGN KEY Constraint

A FOREIGN KEY is a key used to link two tables together.

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

SQL FOREIGN KEY on CREATE TABLE

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

```
CREATE TABLE Orders (
  OrderID int NOT NULL,
  OrderNumber int NOT NULL,
  PersonID int,
  PRIMARY KEY (OrderID),
  FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

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 (
  OrderID int NOT NULL,
  OrderNumber int NOT NULL,
  PersonID int,
  PRIMARY KEY (OrderID),
  CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
  REFERENCES Persons(PersonID)
);
```

SQL FOREIGN KEY on ALTER TABLE

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

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

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

```
ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

DROP a FOREIGN KEY Constraint

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

```
ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```

OR

```
ALTER TABLE Orders
DROP CONSTRAINT FK_PersonOrder;
```

SQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

SQL CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that you can not have any person below 18 years:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   CHECK (Age>=18)
);
```

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

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
```

```
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

SQL CHECK on ALTER TABLE

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

```
ALTER TABLE Persons
ADD CHECK (Age>=18);
```

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_PersonAge CHECK (Age>=18 AND City='Sandnes');
```

DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

```
ALTER TABLE Persons
DROP CONSTRAINT CHK_PersonAge;
```

OR

```
ALTER TABLE Persons
DROP CHECK CHK_PersonAge;
```

SQL DEFAULT Constraint

The DEFAULT constraint is used to provide a default value for a column.

The default value will be added to all new records IF no other value is specified.

SQL DEFAULT on CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

```
CREATE TABLE Persons (
    ID int NOT NULL,
```

```
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

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

```
CREATE TABLE Orders (
    ID int NOT NULL,
    OrderNumber int NOT NULL,
    OrderDate date DEFAULT GETDATE()
);
```

SQL DEFAULT on ALTER TABLE

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

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';
```

DROP a DEFAULT Constraint

To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

SQL CREATE INDEX Statement

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

Indexes are used to retrieve data from the database very fast. The users cannot see the indexes, they are just used to speed up searches/queries.

Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

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

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

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

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

DROP INDEX Statement

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

```
ALTER TABLE table_name
DROP INDEX index_name;
```

SQL AUTO INCREMENT Field

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

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),
  Age int,
  PRIMARY KEY (ID)
);
```

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".

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.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

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 Updating a View

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

SQL Dropping a View

You can delete a view with the DROP VIEW command.

```
DROP VIEW view_name;
```

SQL Function

SQL Aggregate Functions

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

Function	Description
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
ROUND()	Rounds a numeric field to the number of decimals specified

SUM()	Returns the sum

SQL String Functions

Function	Description
CHARINDEX	Searches an expression in a string expression and returns its starting position if found
CONCAT()	
LEFT()	
LEN() / LENGTH()	Returns the length of the value in a text field
LOWER() / LCASE()	Converts character data to lower case
LTRIM()	
SUBSTRING() / MID()	Extract characters from a text field

PATINDEX()	
REPLACE()	
RIGHT()	
RTRIM()	
UPPER() / UCASE()	Converts character data to upper case

SQL Date Functions

‹ Previous Next ›

MySQL Date Functions

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

Function	Description
NOW()	Returns the current date and time

CURDATE()	Returns the current date
CURTIME()	Returns the current time
DATE()	Extracts the date part of a date or date/time expression
EXTRACT()	Returns a single part of a date/time
DATE_ADD()	Adds a specified time interval to a date
DATE_SUB()	Subtracts a specified time interval from a date
DATEDIFF()	Returns the number of days between two dates
DATE_FORMAT()	Displays date/time data in different formats

SQL Server Date Functions

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

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

SQL Date and Time Data Types and Functions

Function	Description
FORMAT()	Formats how a field is to be displayed
NOW()	Returns the current system date and time

SQL Dates

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

As long as your data contains only the date portion, your queries will work as expected. However, if a time portion is involved, it gets more complicated.

SQL Date Data Types

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

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

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

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