SQL Tutorial

SQL is a standard language for accessing databases. This SQL tutorial will teach you how to use SQL to access and manipulate data in: MySQL, SQL Server, Access, Oracle, Sybase, DB2, and other database systems.

1) Introduction to SQL

SQL is a standard language for accessing and manipulating databases.

What is SQL?

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

What Can SQL do?

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

Although SQL is an ANSI (American National Standards Institute) standard, there are different versions of the SQL language. However, to be compliant with the ANSI standard, they all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE) in a similar manner.

Note: Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard!

Using SQL in Your Web Site

To build a web site that shows data from a database, you will need:

- An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
- To use a server-side scripting language, like PHP or ASP
- To use SQL to get the data you want
- To use HTML / CSS

RDBMS

RDBMS stands for Relational Database Management System. RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access. The data in RDBMS is stored in database objects called **tables**. A table is a collection of related data entries and it consists of columns and rows.

2) SQL Syntax

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

In this tutorial we will use the well-known Northwind sample database (included in MS Access and MS SQL Server). 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

(Note: all examples in this tutorial uses the above Customer Table)

SQL Statements

Most of the actions you need to perform on a database are done with SQL statements.

The following SQL statement selects all the records in the "Customers" table:

```
SELECT * FROM Customers;
```

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

3) 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;

Or

*SELECT * FROM table_name;*

Select Column example

SELECT CustomerName,City FROM Customers;

*Select * example*

*SELECT * FROM Customers;*

Navigation in a Result-set

Most database software systems allow navigation in the result-set with programming functions, like: Move-To-First-Record, Get-Record-Content, Move-To-Next-Record, etc. Programming functions like these are not a part of this tutorial.

4) SQL SELECT DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values. 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;

Select Distinct example

SELECT DISTINCT City FROM Customers;

5) 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;
```

Sql Where clause example

```
SELECT * FROM Customers
WHERE Country='Mexico';
```

Text Fields vs. Numeric Fields

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

Text field in Where clause example

```
SELECT * FROM Customers
WHERE CustomerID=1;
```

Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

Operator	Description
=	Equal
<>	Not equal. **Note:** In some versions of SQL this operator 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

6) SQL AND & OR Operators

The AND & OR operators are used to filter records based on more than one condition.

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

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

SQL AND Operator Example

The following SQL statement selects all customers from the country "Germany" AND the city "Berlin", in the "Customers" table:

```
SELECT * FROM Customers
WHERE Country='Germany' AND City='Berlin';
```

SQL OR Operator Example

The following SQL statement selects all customers from the city "Berlin" OR "München", in the "Customers" table:

```
SELECT * FROM Customers
WHERE City='Berlin'
OR City='München';
```

Combining AND & OR

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

The following SQL statement selects all customers from the country "Germany" AND the city must be equal to "Berlin" OR "München", in the "Customers" table:

```
SELECT * FROM Customers
WHERE Country='Germany'
AND (City='Berlin' OR City='München');
```

7) 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;
```

ORDER BY Example (Default ASC)

```
SELECT * FROM Customers
ORDER BY Country;
```

ORDER BY DESC Example

```
SELECT * FROM Customers
ORDER BY Country,CustomerName;
```

8) 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.

a) 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,...);

b) 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,...);

INSERT INTO Example (without specifying column names)

INSERT INTO Customers

VALUES ('Cardinal','Tom B. Erichsen','Skagen 21','Stavanger','4006','Norway');

INSERT INTO Example (including column names)

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)

VALUES ('Cardinal','Tom B. Erichsen','Skagen 21','Stavanger','4006','Norway');

Note: we did not insert any number into the CustomerID field? The CustomerID column is an AutoNumber field and is automatically updated with a unique number for each record in the table. AutoNumber is a type of data used in Microsoft Access tables to generate an automatically incremented numeric counter. The default AutoNumber type has a start value of 1 and an increment of 1.

Insert Data Only in Specified Columns

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

The following SQL statement will insert a new row, but only insert data in the "CustomerName", "City", and "Country" columns (and the CustomerID field will of course also be updated automatically).

INSERT INTO Customers (CustomerName, City, Country)

VALUES ('Cardinal', 'Stavanger', 'Norway');

9) 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;

SQL UPDATE Example

Assume we wish to update the customer "Alfreds Futterkiste" with a new contact person and city.

```
UPDATE Customers
SET ContactName='Alfred Schmidt', City='Hamburg'
WHERE CustomerName='Alfreds Futterkiste';
```

10) 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;
```

SQL DELETE Example

Assume we wish to delete the customer "Alfreds Futterkiste" from the "Customers" table.

```
DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste' AND ContactName='Maria Anders';
```

Delete All Data

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

```
DELETE FROM table_name;
or
DELETE * FROM table_name;
```

11) 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;
```

SQL SELECT TOP Equivalent in MySQL and Oracle

MySQL Syntax

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

MySQL Example

```
SELECT *
FROM Persons
LIMIT 5;
```

Oracle Syntax

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

Oracle Example

```
SELECT *
FROM Persons
WHERE ROWNUM <=5;
```

SQL SELECT TOP Example

The following SQL stament selects the two first records from the "Customers" table:

```
SELECT TOP 2 * FROM Customers;
```

SQL SELECT TOP PERCENT Example

The following SQL stament selects the first 50% of the records from the "Customers" table:

```
SELECT TOP 50 PERCENT * FROM Customers;
```

12) SQL LIKE Operator

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

SQL LIKE Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern;
```

SQL LIKE Operator Examples

The following SQL statement selects all customers with a City starting with the letter "s":

```
SELECT * FROM Customers
WHERE City LIKE 's%';
```

Tip: The "%" sign is used to define wildcards (missing letters) both before and after the pattern. You will learn more about wildcards in the next topic.

The following SQL statement selects all customers with a City ending with the letter "s":

```
SELECT * FROM Customers
WHERE City LIKE '%s';
```

The following SQL statement selects all customers with a Country containing the pattern "land":

*SELECT * FROM Customers*

WHERE Country LIKE '%land%';

Using the NOT keyword allows you to select records that does NOT match the pattern.

The following SQL statement selects all customers with a Country NOT containing the pattern "land":

*SELECT * FROM Customers*

WHERE Country NOT LIKE '%land%';

13) SQL Wildcards *(New)*

A wildcard character can be used to substitute for any other character(s) in a string.

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

Using the SQL % Wildcard Examples

The following SQL statement selects all customers with a City starting with "ber":

*SELECT * FROM Customers*

WHERE City LIKE 'ber%';

The following SQL statement selects all customers with a City containing the pattern "es":

*SELECT * FROM Customers*

WHERE City LIKE '%es%';

The following SQL statement 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';

Using the SQL [charlist] Wildcard Examples

The following SQL statement selects all customers with a City starting with "b", "s", or "p":

*SELECT * FROM Customers*

WHERE City LIKE '[bsp]%';

The following SQL statement selects all customers with a City starting with "a", "b", or "c":

*SELECT * FROM Customers*

WHERE City LIKE '[a-c]%';

The following SQL statement selects all customers with a City NOT starting with "b", "s", or "p":

*SELECT * FROM Customers*

WHERE City LIKE '[!bsp]%';

14) SQL IN Operator

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

SQL IN Syntax

SELECT column_name(s)

FROM table_name

WHERE column_name IN (value1,value2,...);

IN Operator Example

The following SQL statement selects all customers with a City of "Paris" or "London":

*SELECT * FROM Customers*

WHERE City IN ('Paris','London');

15) 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_name BETWEEN value1 AND value2;

Demo Table (Product) from NorthWind database

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	1	2	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	1	2	36 boxes	21.35

BETWEEN Operator Example

The following SQL statement selects all products with a price BETWEEN 10 and 20:

*SELECT * FROM Products*

WHERE Price BETWEEN 10 AND 20;

NOT BETWEEN Operator Example

To display the products outside the range of the previous example, use NOT BETWEEN:

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

BETWEEN Operator with IN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20, but products with a CategoryID of 1,2, or 3 should not be displayed:

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

BETWEEN Operator with Text Value Example

The following SQL statement selects all products with a ProductName beginning with any of the letter BETWEEN 'C' and 'M':

```
SELECT * FROM Products
WHERE ProductName BETWEEN 'C' AND 'M';
```

NOT BETWEEN Operator with Text Value Example

The following SQL statement selects all products with a ProductName beginning with any of the letter NOT BETWEEN 'C' and 'M':

```
SELECT * FROM Products
WHERE ProductName NOT BETWEEN 'C' AND 'M';
```

BETWEEN Operator with Date Value Example

The following SQL statement selects all orders with an OrderDate BETWEEN '04-July-1996' and '09-July-1996':

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

Notice that the BETWEEN operator can produce different result in different databases! In some databases, BETWEEN selects fields that are between and excluding the test values. In other databases, BETWEEN selects fields that are between and including the test values. And in other databases, BETWEEN selects fields between the test values, including the first test value and excluding the last test value.

16) SQL Aliases

SQL aliases are used to give a database table or a column in a table a temporarily 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_name AS alias_name;
```

Alias Example for Table Columns

The following SQL statement specifies two aliases, one for the CustomerName column and one for the ContactName column. **Tip:** It require double quotation marks or square brackets if the column name contains spaces:

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person]
FROM Customers;
```

Alias Example for Table Columns

The following SQL statement specifies two aliases, one for the CustomerName column and one for the ContactName column. **Tip:** It require double quotation marks or square brackets if the column name contains spaces:

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person]
FROM Customers;
```


In the following SQL statement we combine four columns (Address, City, PostalCode, and Country) and create an alias named "Address":

```
SELECT CustomerName, Address+', '+City+', '+PostalCode+', '+Country AS Address
FROM Customers;
```

Alias Example for Tables

The following SQL statement selects all the orders from the customer "Alfreds Futterkiste". We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we have used aliases to make the SQL shorter):

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName='Alfreds Futterkiste';
```

Aliases can be useful when:

- There are more than one table involved in a query
- Functions are used in the query
- Column names are big or not very readable

- Two or more columns are combined together

17) SQL Joins

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

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

18) 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:

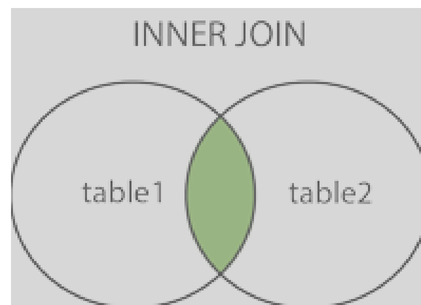
SELECT column_name(s)

FROM table1

JOIN table2

ON table1.column_name=table2.column_name;

PS! INNER JOIN is the same as JOIN.



SQL INNER JOIN Example

The following SQL statement will return all customers with orders:

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

INNER JOIN Orders

ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

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.

19) 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:

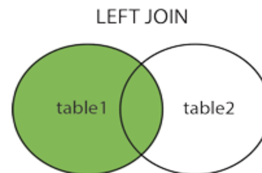
SELECT column_name(s)

FROM table1

LEFT OUTER JOIN table2

ON table1.column_name=table2.column_name;

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



SQL LEFT JOIN Example

The following SQL statement will return all customers, and any orders they might have:

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders

ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

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).

20) 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:

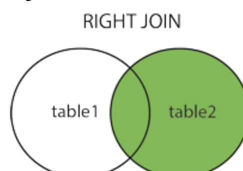
SELECT column_name(s)

FROM table1

RIGHT OUTER JOIN table2

ON table1.column_name=table2.column_name;

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



SQL RIGHT JOIN Example

The following SQL statement will return all orders, and any customers that might have placed them:

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

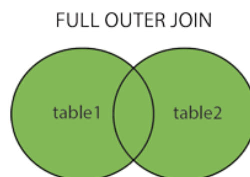
Note: The RIGHT JOIN keyword returns all the rows from the right table (Orders), even if there are no matches in the left table (Customers).

21) 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;
```



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.

22) 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.

SQL UNION Example

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

```
SELECT City FROM Customers
UNION
```

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

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

SQL UNION ALL Example

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

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

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:

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

23) SQL SELECT INTO Statement

With SQL, you can copy information from one table into another. The SELECT INTO statement copies 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;
```

24) SQL INSERT INTO SELECT Statement

With SQL, you can copy information from one table into another. 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;
```

SQL INSERT INTO SELECT Examples

Copy only a few columns from "Suppliers" into "Customers":

```
INSERT INTO Customers (CustomerName, Country)
SELECT SupplierName, Country FROM Suppliers;
```

Copy only the German suppliers into "Customers":

```
INSERT INTO Customers (CustomerName, Country)
SELECT SupplierName, Country FROM Suppliers
WHERE Country='Germany';
```

25) 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.

26) 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 CREATE TABLE Example

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

```
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.

27) 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

28) SQL NOT NULL Constraint

By default, a table column can hold NULL values.

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)
)
```

29) 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
```

30) 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
```

31) SQL FOREIGN KEY Constraint

A FOREIGN KEY in one table points to a PRIMARY KEY in another 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
```

32) SQL CHECK Constraint

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

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

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

SQL CHECK Constraint on CREATE TABLE

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

MySQL:

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CHECK (P_Id>0)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
P_Id int NOT NULL CHECK (P_Id>0),
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

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

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CONSTRAINT chk_Person CHECK (P_Id>0 AND City='Sandnes')
)
```

SQL CHECK Constraint on ALTER TABLE

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

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CHECK (P_Id>0)
```

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

MySQL / SQL Server / Oracle / MS Access:

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

To DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

SQL Server / Oracle / MS Access:

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

MySQL:

```
ALTER TABLE Persons
DROP CHECK chk_Person
```

33) 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

34) SQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables. Indexes allow the database application to find data fast; without reading the whole table

Indexes

An index can be created in a table to find data more quickly and efficiently. The users cannot see the indexes, they are just used to speed up searches/queries.

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

SQL CREATE INDEX Syntax

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

CREATE INDEX index_name

ON table_name (column_name)

SQL CREATE UNIQUE INDEX Syntax

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

CREATE UNIQUE INDEX index_name

ON table_name (column_name)

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

CREATE INDEX Example

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

CREATE INDEX PIndex

ON Persons (LastName)

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

CREATE INDEX PIndex

ON Persons (LastName, FirstName)

35) SQL DROP INDEX, DROP TABLE, and DROP DATABASE

Indexes, tables, and databases can easily be deleted/removed with the DROP statement.

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

36) SQL ALTER TABLE Statement

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

SQL ALTER TABLE Syntax

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

ALTER TABLE table_name

ADD column_name datatype

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

ALTER TABLE table_name

DROP COLUMN column_name

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

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

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

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
```

37) SQL AUTO INCREMENT Field

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

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

Syntax for MySQL

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

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

MySQL uses the AUTO_INCREMENT keyword to perform an auto-increment feature. By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.

To let the AUTO_INCREMENT sequence start with another value, use the following SQL statement:

```
ALTER TABLE Persons AUTO_INCREMENT=100
```

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

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

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

Syntax for SQL Server

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

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

The MS SQL Server uses the IDENTITY keyword to perform an auto-increment feature. By default, the starting value for IDENTITY is 1, and it will increment by 1 for each new record.

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

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

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

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

Syntax for Access

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

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

The MS Access uses the AUTOINCREMENT keyword to perform an auto-increment feature. By default, the starting value for AUTOINCREMENT is 1, and it will increment by 1 for each new record.

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

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

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

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

Syntax for Oracle

In Oracle the code is a little bit more tricky.

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

Use the following CREATE SEQUENCE syntax:

```
CREATE SEQUENCE seq_person
MINVALUE 1
START WITH 1
INCREMENT BY 1
CACHE 10
```

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

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

```
INSERT INTO Persons (P_Id,FirstName,LastName)
VALUES (seq_person.nextval,'Lars','Monsen')
```

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

38) SQL Views

A view is a virtual table. This topic shows how to create, update, and delete a view.

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

39) SQL Date Functions

The most difficult part when working with dates is to be sure that the format of the date you are trying to insert, matches the format of the date column in the database. As long as your data contains only the date portion, your queries will work as expected. However, if a time portion is involved, it gets complicated. Before talking about the complications of querying for dates, we will look at the most important built-in functions for working with dates.

MySQL Date Functions

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

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

SQL Server Date Functions

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

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

SQL Date Data Types

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

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

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

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

Note: The date types are chosen for a column when you create a new table in your database!

40) SQL NULL Values

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

If a column in a table is optional, we can insert a new record or update an existing record without adding a value to this column. This means that the field will be saved with a NULL value.

NULL values are treated differently from other values. NULL is used as a placeholder for unknown or inapplicable values. Note: It is not possible to compare NULL and 0; they are not equivalent

SQL Working with NULL Values

How can we test for NULL values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

SQL IS NULL

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

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

SQL IS NOT NULL

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

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

41) SQL NULL Functions

SQL ISNULL(), NVL(), IFNULL() and COALESCE() Functions

We have the following SELECT statement:

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

In the example above, if any of the "UnitsOnOrder" values are NULL, the result is NULL. Microsoft's ISNULL() function is used to specify how we want to treat NULL values. The NVL(), IFNULL(), and COALESCE() functions can also be used to achieve the same result.

In this case we want NULL values to be zero. Below, if "UnitsOnOrder" is NULL it will not harm the calculation, because ISNULL() returns a zero if the value is NULL:

SQL Server / MS Access

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

Oracle

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

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

MySQL

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

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

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

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

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