**Keep in Mind That...**

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

**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.

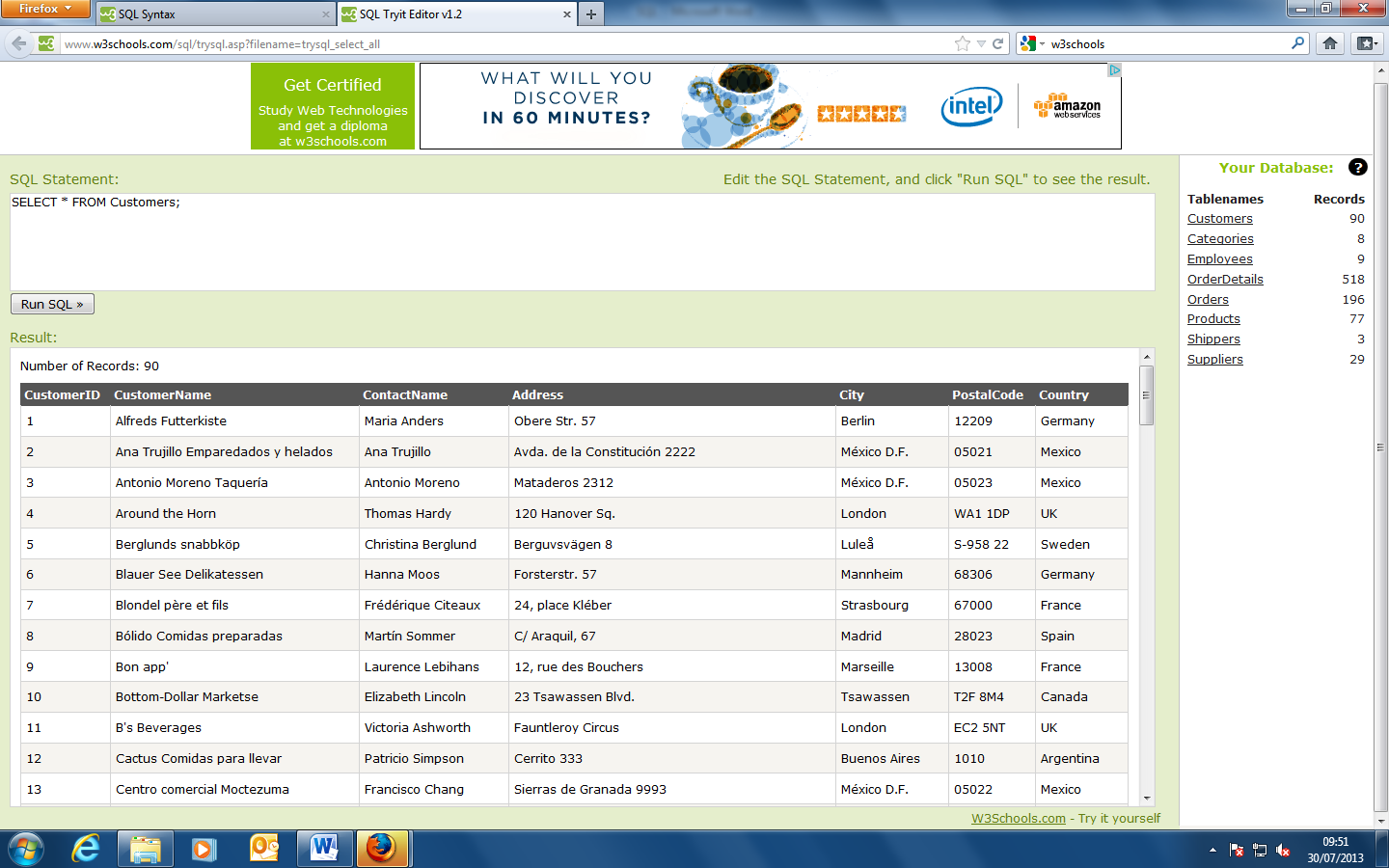
In this tutorial, we will use semicolon at the end of each SQL statement.

**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

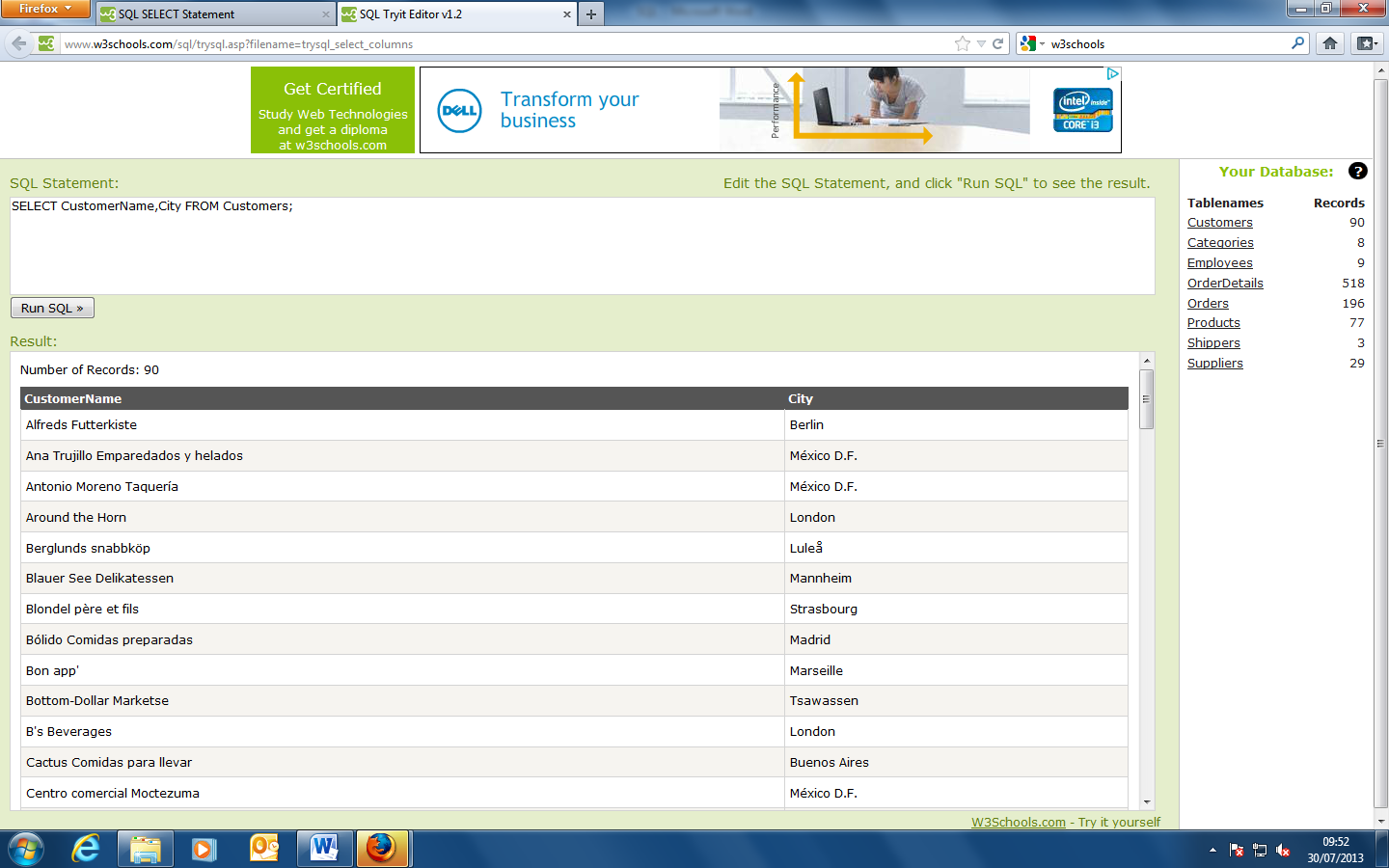
SELECT \* FROM Customers;

This selects the whole customers page from the database.



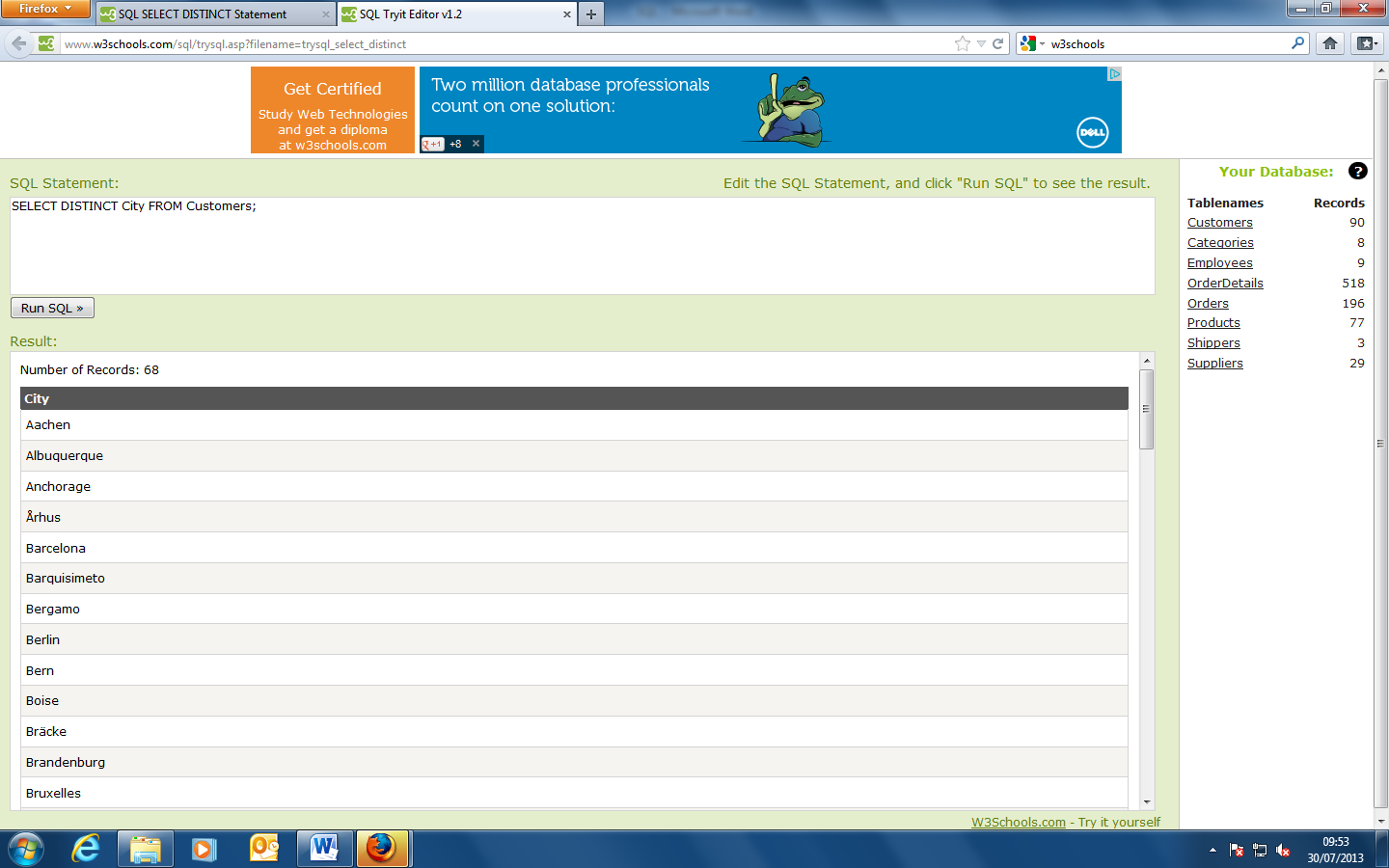
SELECT CustomerName,City, customerID FROM Customers;

This selects the columns Customername, City and Customer ID, from the customers page of the database. To add more fields to the query use a comma and then add another field in the relevant database.



SELECT DISTINCT City FROM Customers;

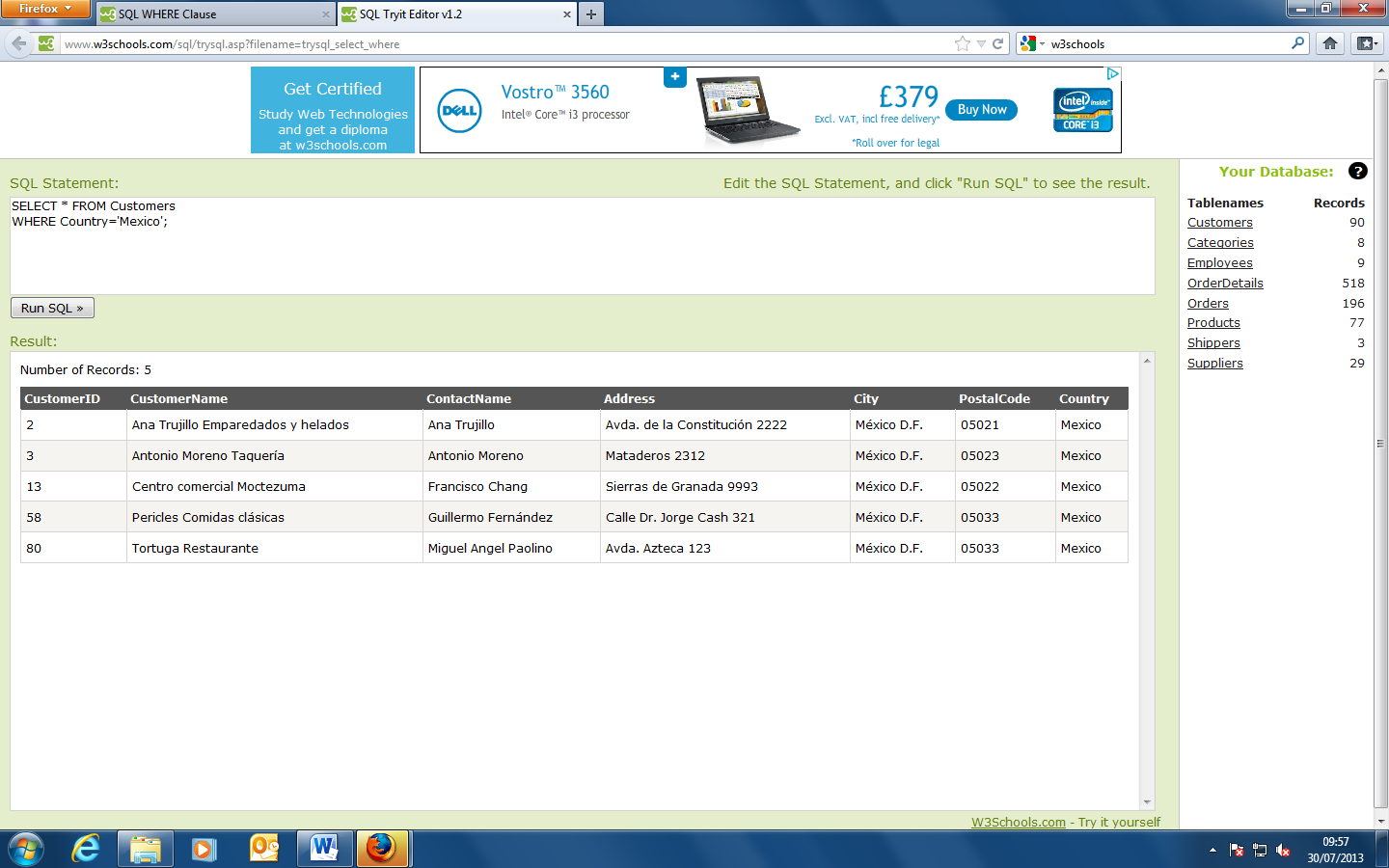
This command shows only different cities. It cuts out any repeats.



SELECT \* FROM Customers

WHERE Country='Mexico';

This selects all the information contained in the rows containing the information ‘Mexico’. Note that SQL requires single quotes around text values (most database systems will also allow double quotes).

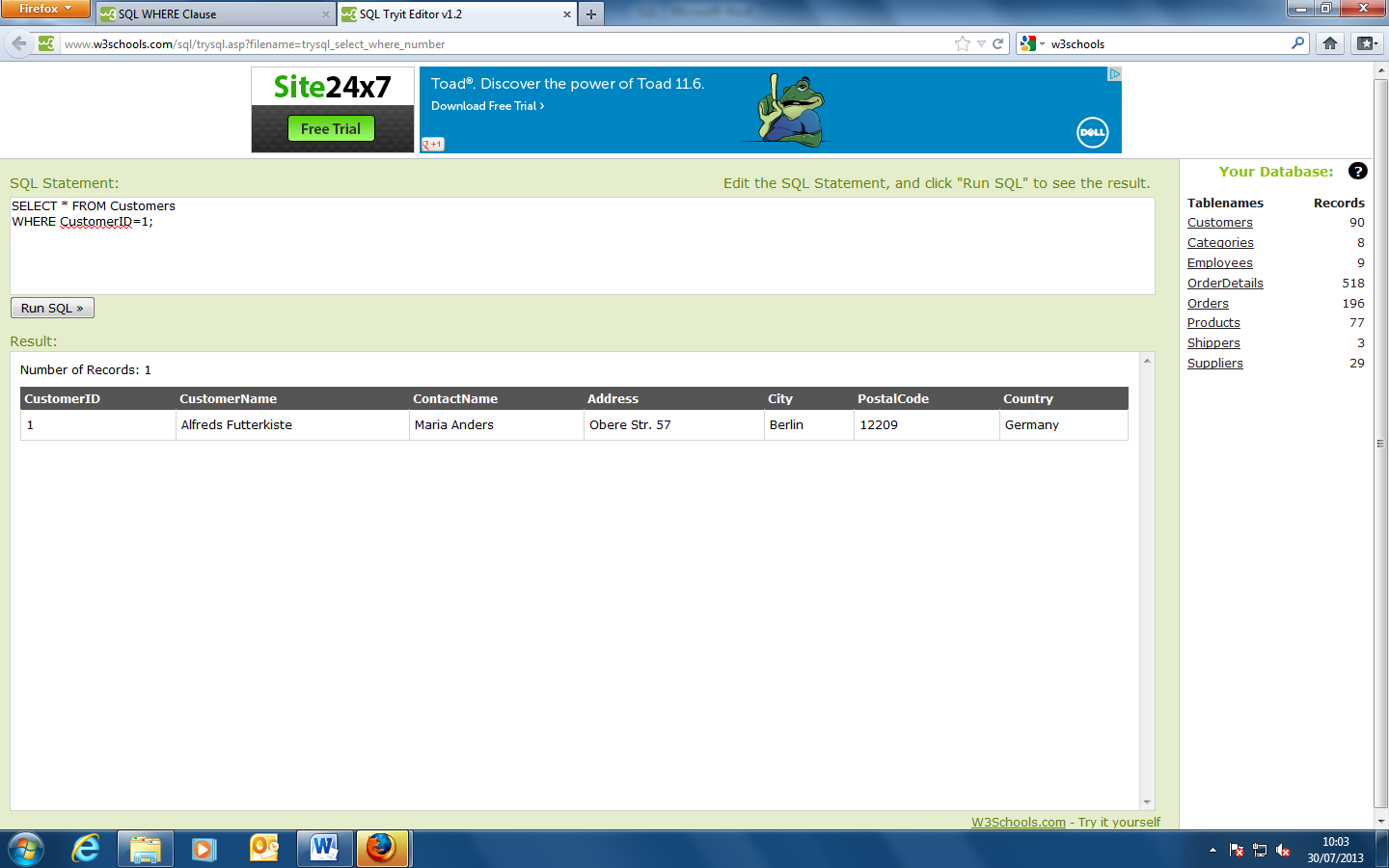


However, numeric fields should not be enclosed in quotes:

SELECT \* FROM Customers

WHERE CustomerID=1;

This selects the information relating to CustomerID = 1 in the customers data base page.



**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

**SQL AND & OR Operators**

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

The SQL AND & OR Operators

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

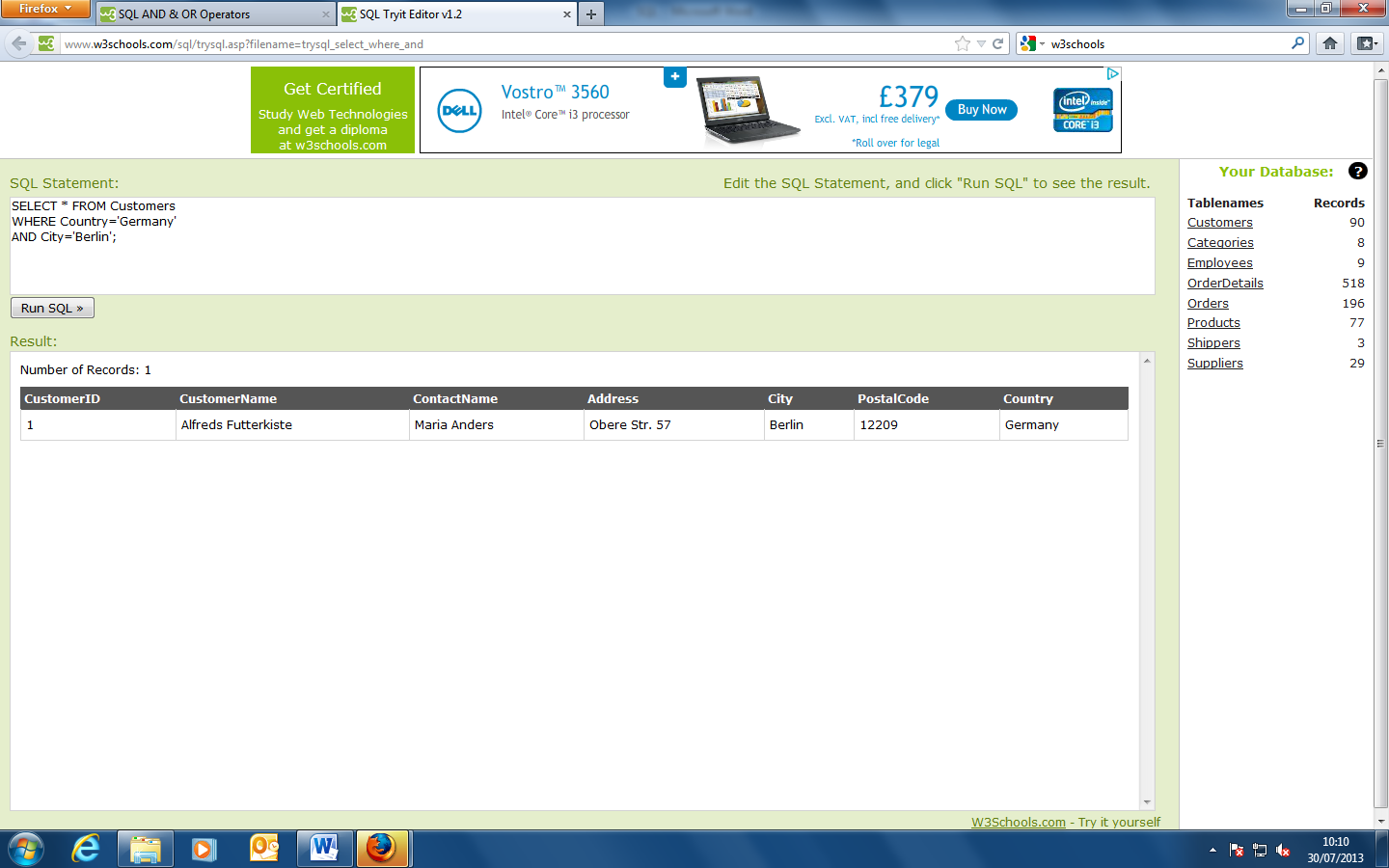
The OR operator displays a record if either the first condition OR the second condition is true.

SELECT \* FROM Customers

WHERE Country='Germany'

AND City='Berlin';

This selects the fields that are specifically related to Germany and Berlin. The records must contain the criteria Germany and Berlin to be displayed.

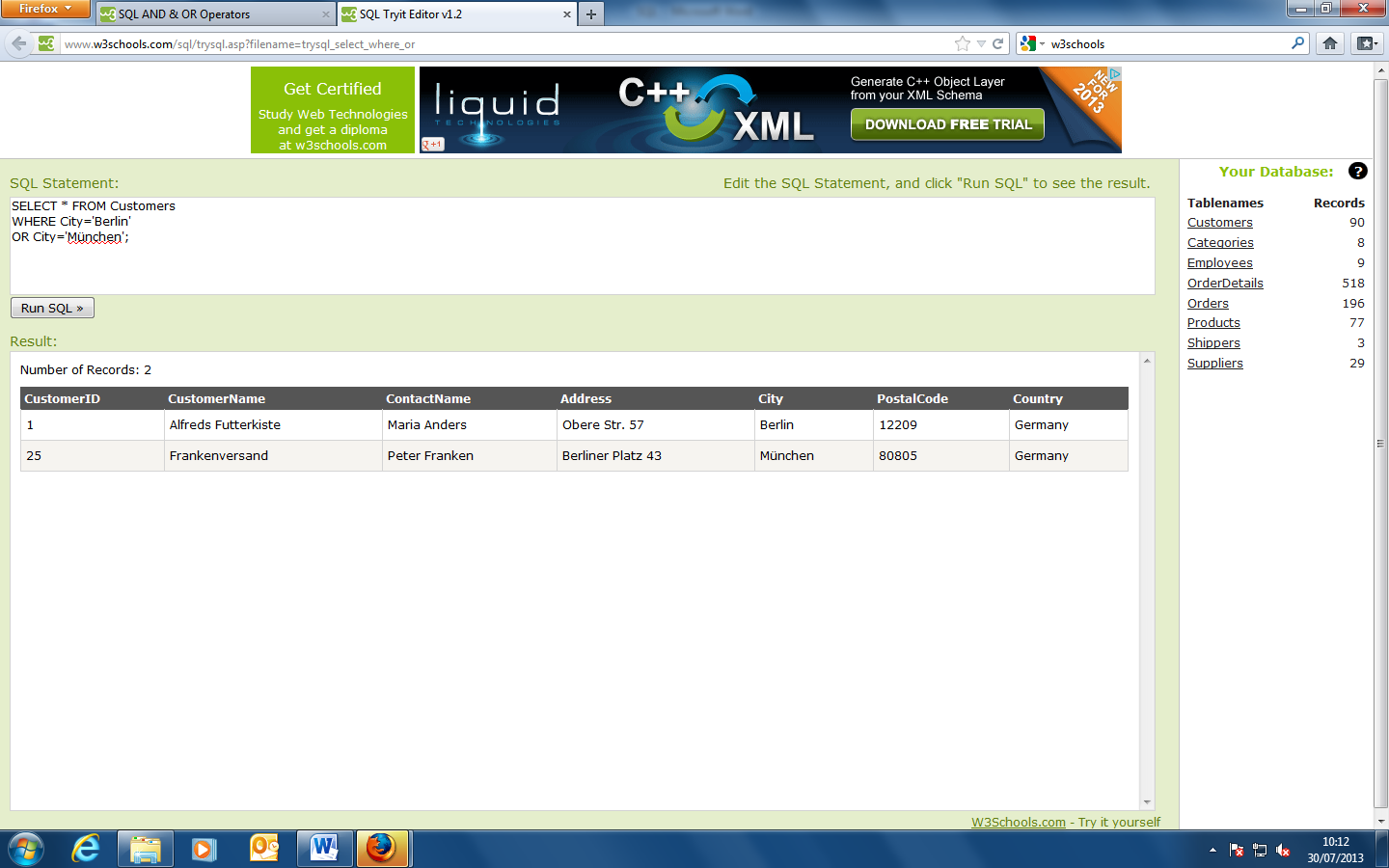


SELECT \* FROM Customers

WHERE City='Berlin'

OR City='München';

This selects any records that fulfil the criteria of containing either berlin or Munchen in the Customers database.



**Combining AND & OR**

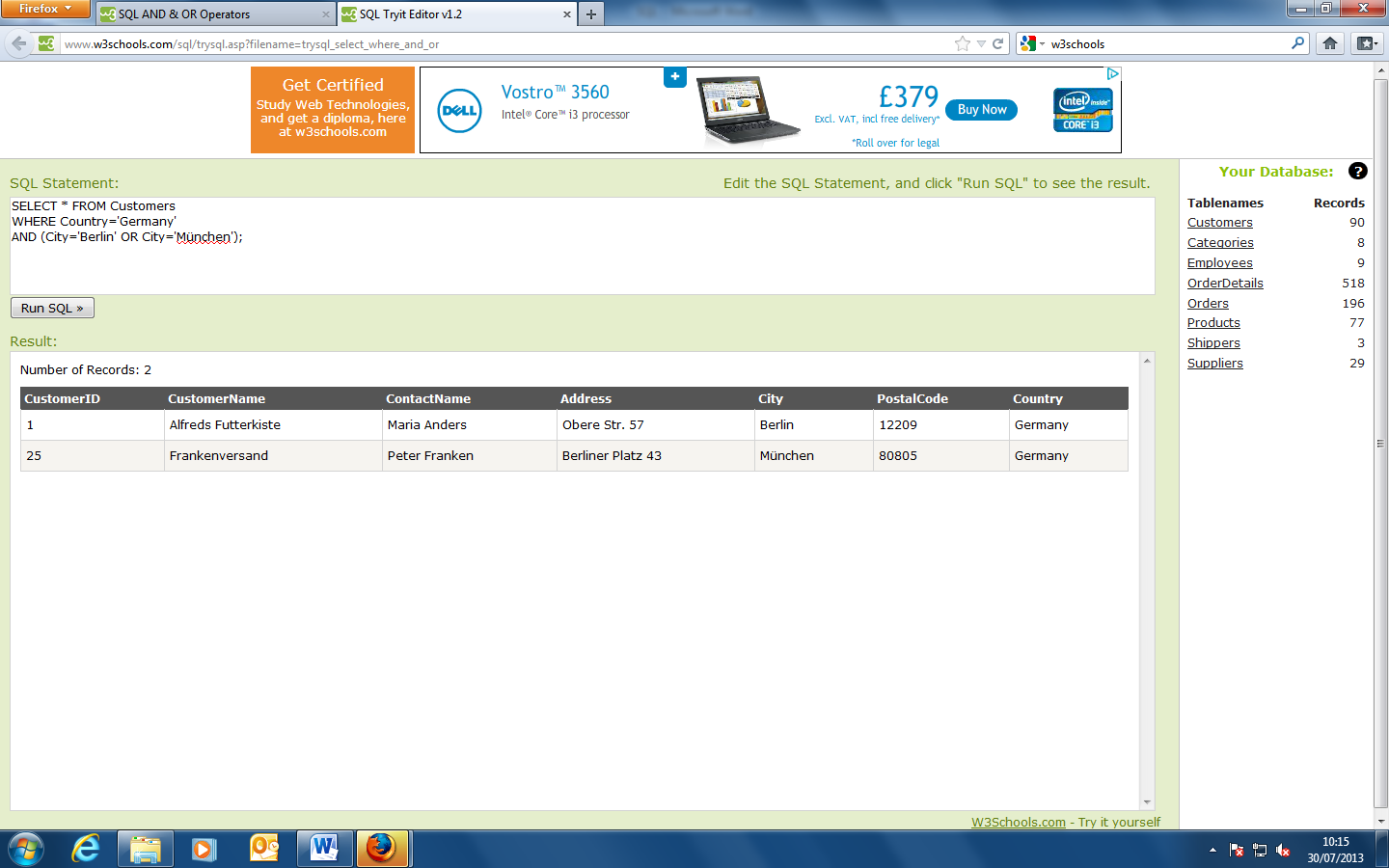
You can also combine AND and OR (use parenthesis to form complex expressions).

SELECT \* FROM Customers

WHERE Country='Germany'

AND (City='Berlin' OR City='München');

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



**The SQL ORDER BY Keyword**

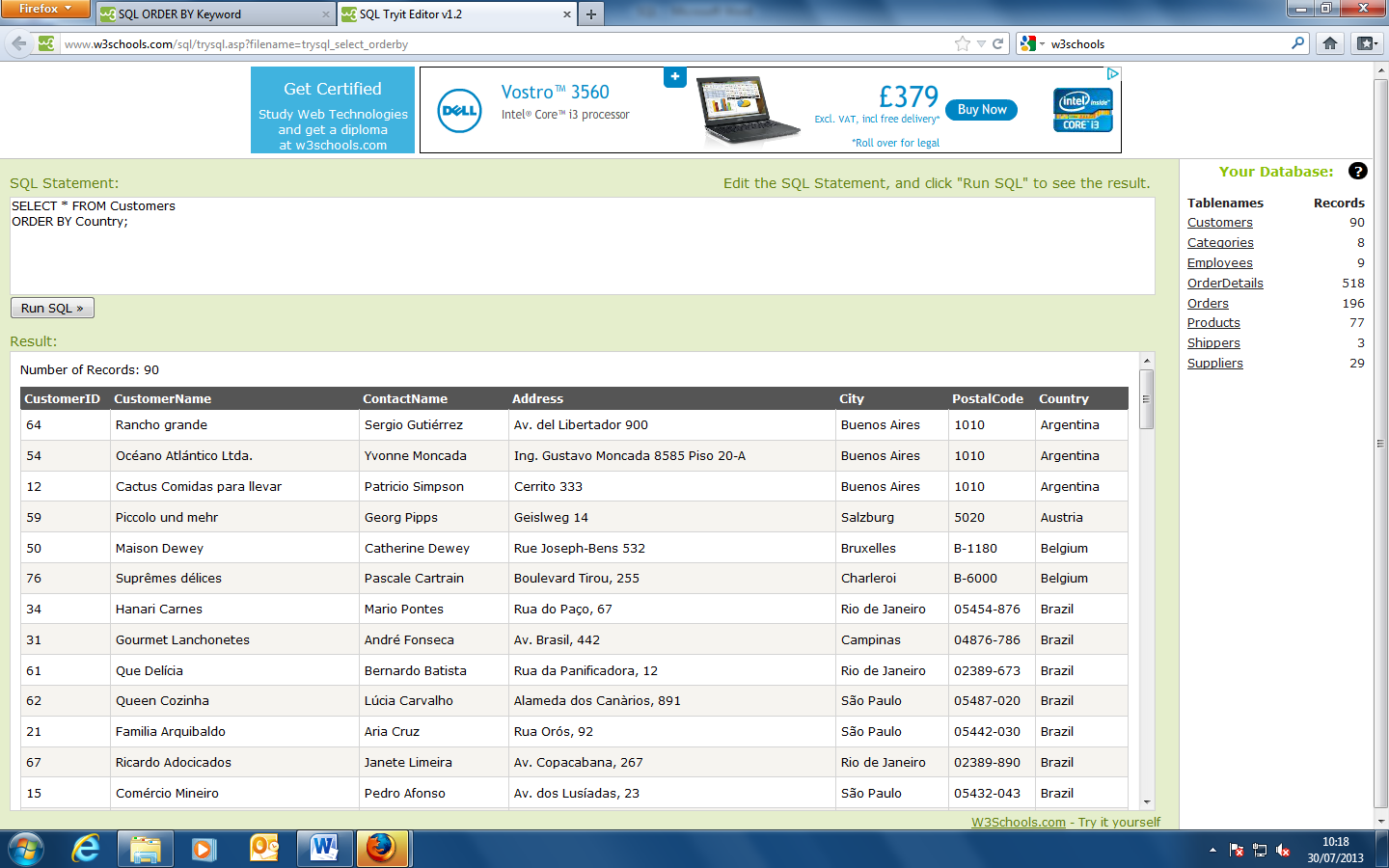
The ORDER BY keyword is used to sort the result-set by one or more columns.

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

SELECT \* FROM Customers

ORDER BY Country;

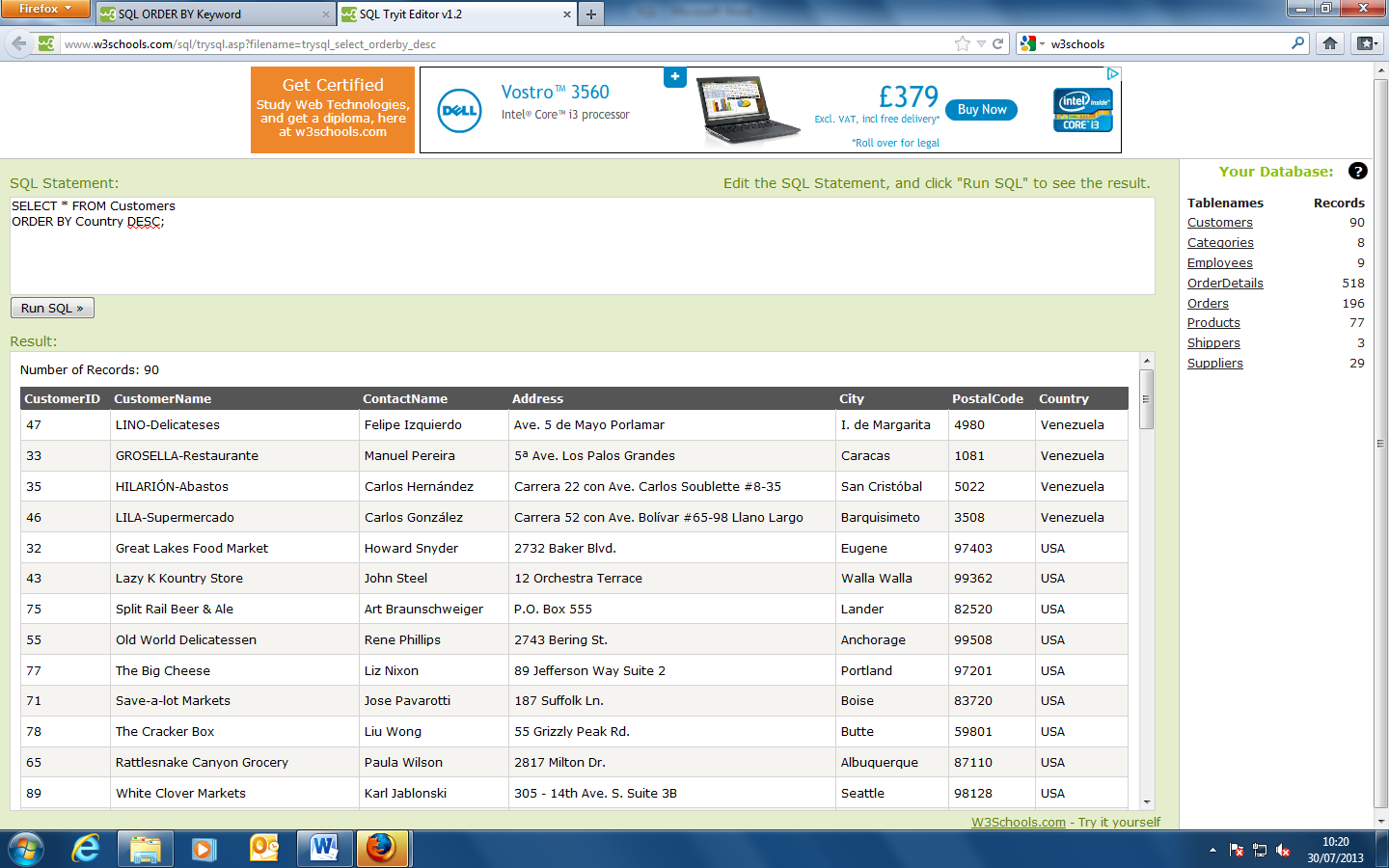
This command orders the customer database in alphabetical order by country.



SELECT \* FROM Customers

ORDER BY Country DESC;

This command orders the customer database by country in descending alphabetical order



SELECT \* FROM Customers

ORDER BY Country,CustomerName;

This statement orders the customer database in alphabetical order by country and then by customer name. The table shows the countries in alphabetical order and then the customers within each country in alphabetical order.



**The SQL INSERT INTO Statement**

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

SQL INSERT INTO Syntax

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

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

**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** |
| 87 | Wartian Herkku | Pirkko Koskitalo | Torikatu 38 | Oulu | 90110 | Finland |
| 88 | Wellington Importadora | Paula Parente | Rua do Mercado, 12 | Resende | 08737-363 | Brazil |
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |

**INSERT INTO Example**

Assume we wish to insert a new row in the "Customers" table.

We can use the following SQL statement (without specifying column names):

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

or this SQL statement (including column names):

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)  
VALUES ('Cardinal','Tom B. Erichsen','Skagen 21','Stavanger','4006','Norway');

The selection from the "Customers" table will now look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 87 | Wartian Herkku | Pirkko Koskitalo | Torikatu 38 | Oulu | 90110 | Finland |
| 88 | Wellington Importadora | Paula Parente | Rua do Mercado, 12 | Resende | 08737-363 | Brazil |
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |
| 92 | Cardinal | Tom B. Erichsen | Skagen 21 | Stavanger | 4006 | Norway |

|  |  |
| --- | --- |
| **lamp** | **Did you notice that 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');

The selection from the "Customers" table will now look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 87 | Wartian Herkku | Pirkko Koskitalo | Torikatu 38 | Oulu | 90110 | Finland |
| 88 | Wellington Importadora | Paula Parente | Rua do Mercado, 12 | Resende | 08737-363 | Brazil |
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |
| 92 | Cardinal |  |  | Stavanger |  | Norway |

**The SQL UPDATE Statement**

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

**SQL UPDATE Syntax**

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

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

**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 UPDATE Example**

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

We use the following SQL statement:

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

The selection from the "Customers" table will now look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Alfred Schmidt | Obere Str. 57 | Hamburg | 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 |

**Update Warning!**

Be careful when updating records. If we had omitted the WHERE clause in the example above, like this:

UPDATE Customers  
SET ContactName='Alfred Schmidt', City='Hamburg';

The "Customers" table would have looked like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Alfred Schmidt | Obere Str. 57 | Hamburg | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Alfred Schmidt | Avda. de la Constitución 2222 | Hamburg | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Alfred Schmidt | Mataderos 2312 | Hamburg | 05023 | Mexico |
| 4 | Around the Horn | Alfred Schmidt | 120 Hanover Sq. | Hamburg | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Alfred Schmidt | Berguvsvägen 8 | Hamburg | S-958 22 | Sweden |

**The SQL DELETE Statement**

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

**SQL DELETE Syntax**

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

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

**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 DELETE Example**

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

We use the following SQL statement:

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

The "Customers" table will now look like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 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 |

**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*;

**Note:** Be very careful when deleting records. You cannot undo this statement!

**The SQL SELECT TOP Clause**

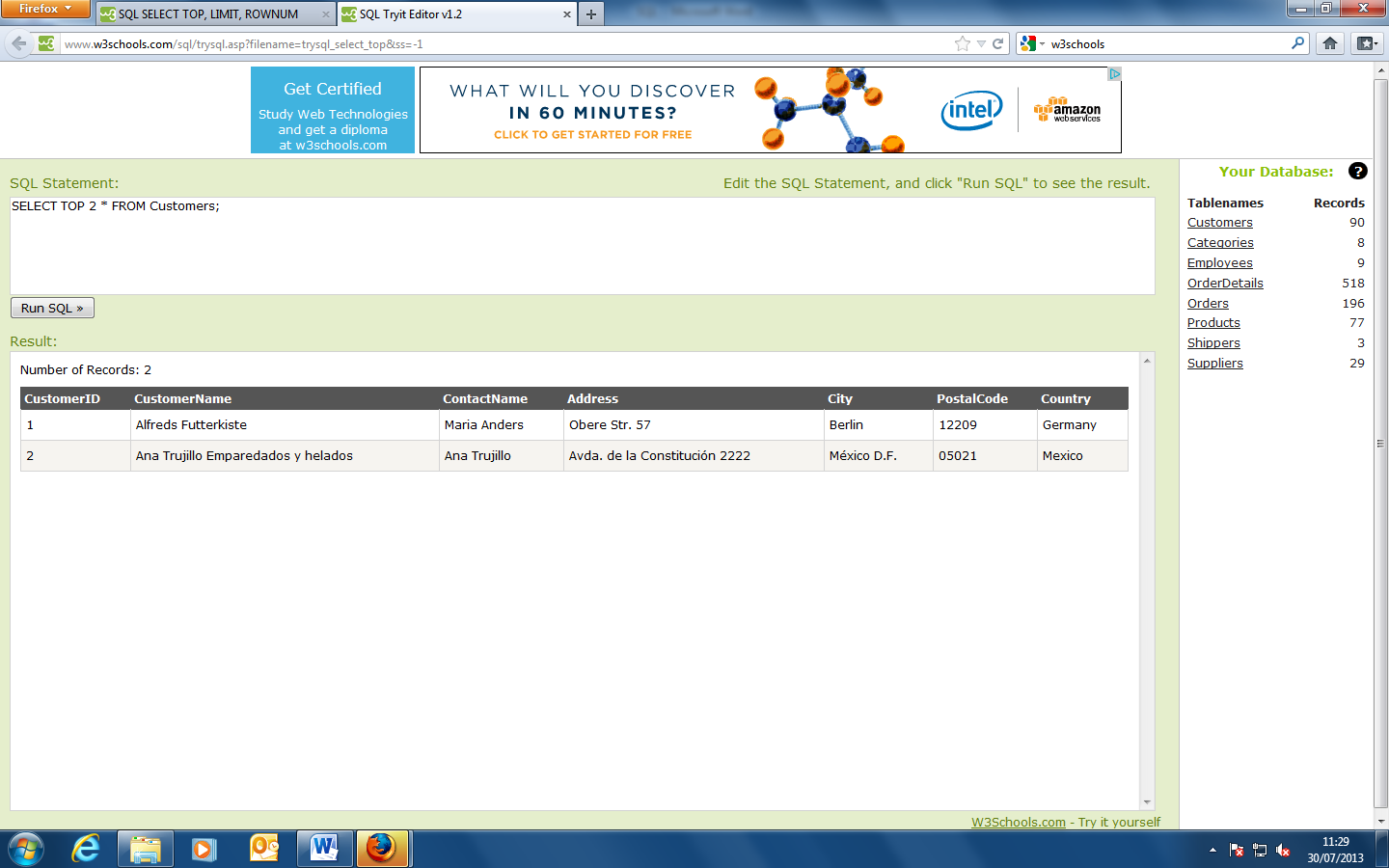
The SELECT TOP clause is used to specify the number of records to return.

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

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

SELECT TOP 2 \* FROM Customers;

This command selects the top two records from the database Customers



SELECT TOP 50 PERCENT \* FROM Customers;

Selects the top 50% from the database customers.

**The SQL LIKE Operator**

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

SQL LIKE Syntax

SELECT column\_name(s)

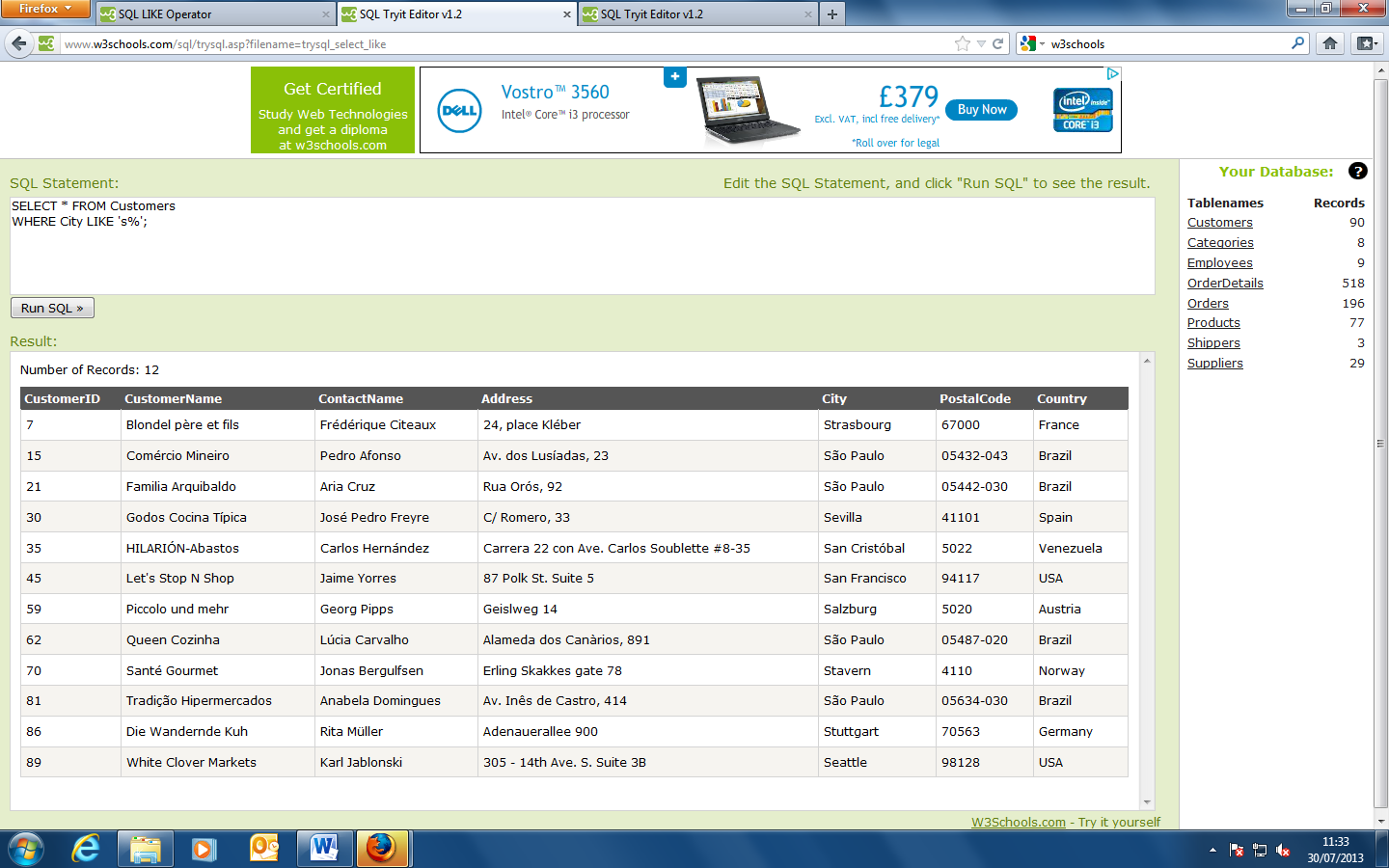
FROM table\_name

WHERE column\_name LIKE pattern;

SELECT \* FROM Customers

WHERE City LIKE 's%';

This command selects all cities starting with S from the customers database.

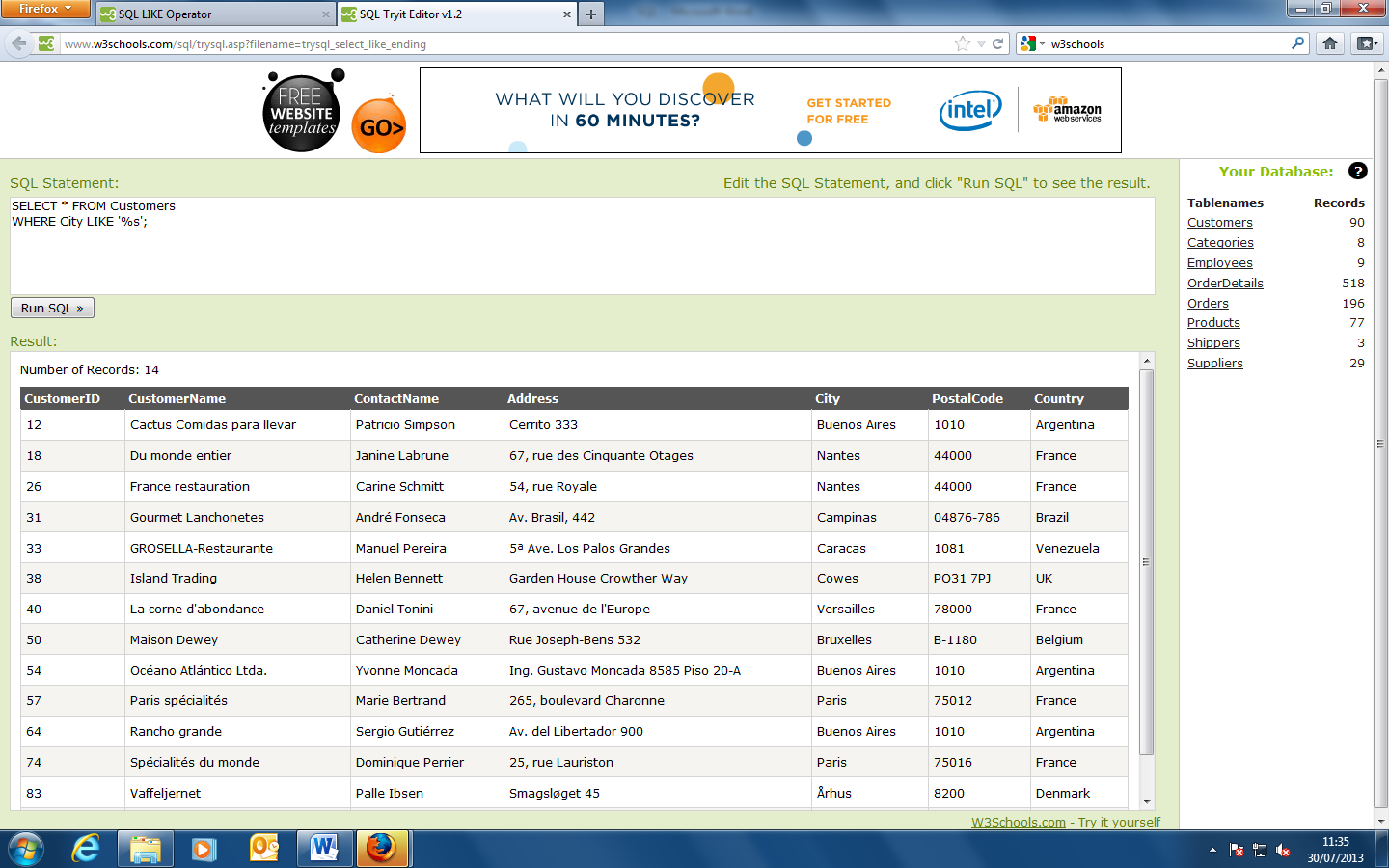


**Tip:** The "%" sign is used to define wildcards (missing letters) both before and after the pattern. You will learn more about wildcards in the next chapter.

SELECT \* FROM Customers

WHERE City LIKE '%s';

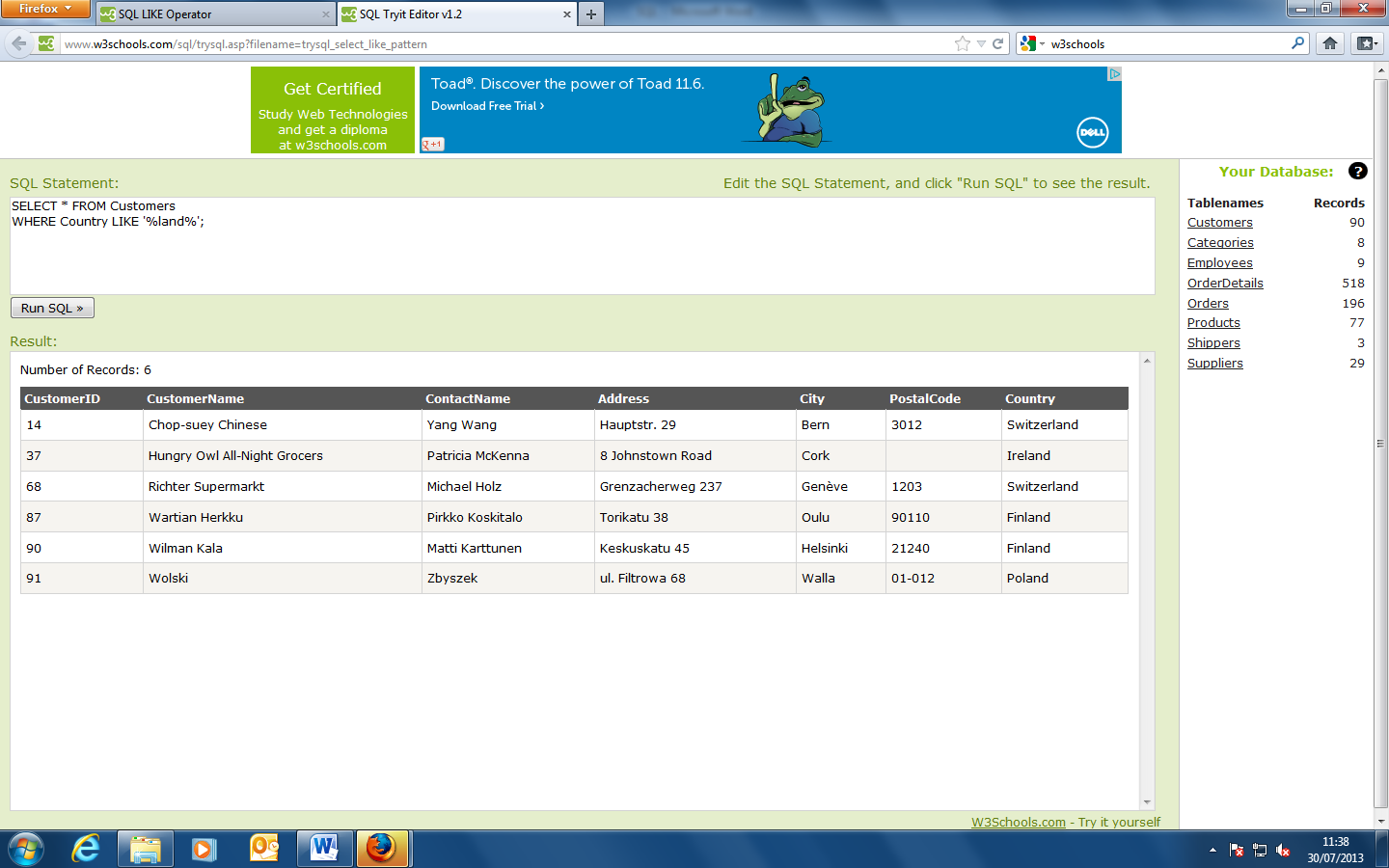
This command selects all cities ending with S from the customers database.



SELECT \* FROM Customers

WHERE Country LIKE '%land%';

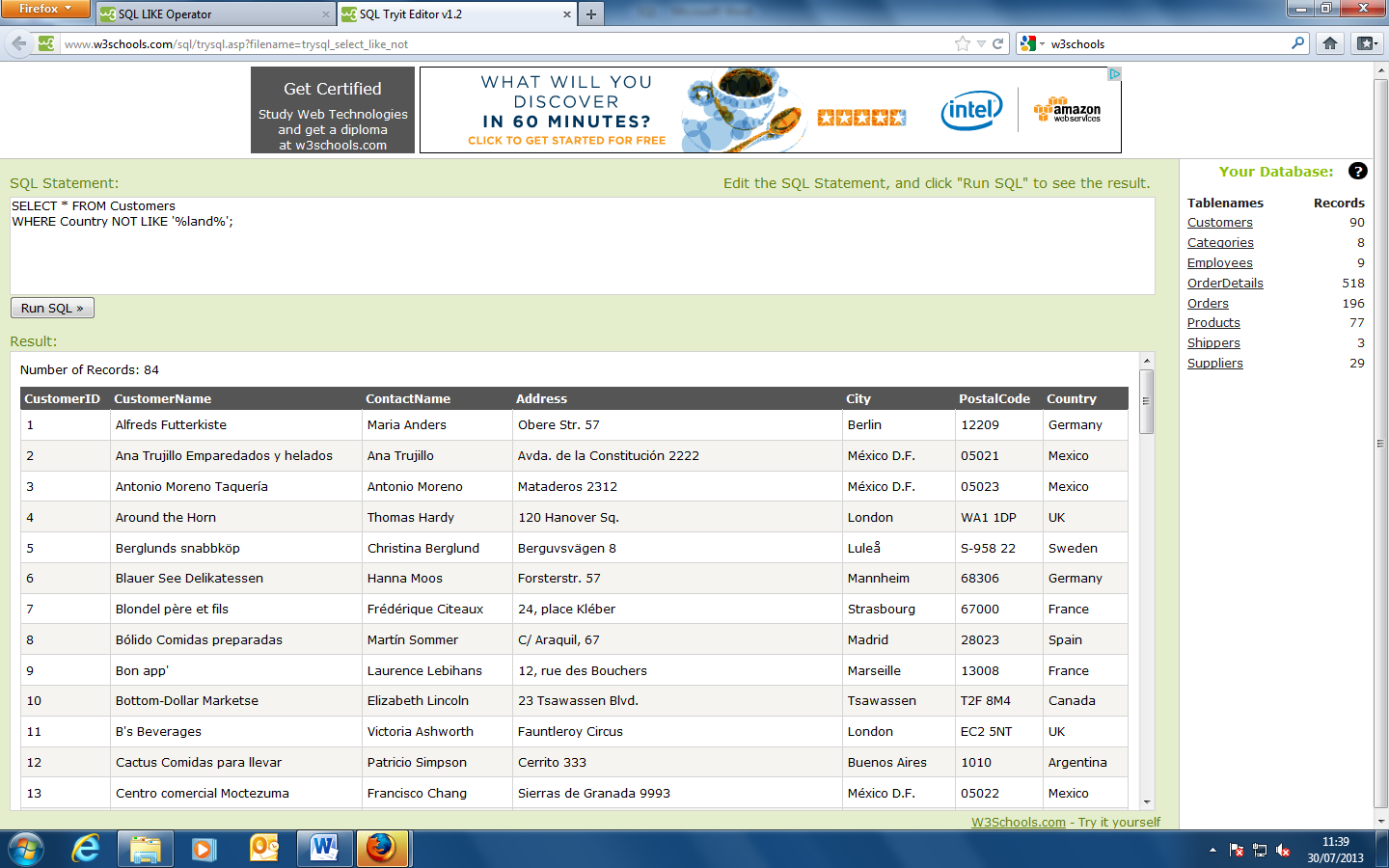
The use of % allows the search to be confined to parts of words in a given attribute. This command searches for any country names containing ‘land’.



SELECT \* FROM Customers

WHERE Country NOT LIKE '%land%';

Conversely this command finds all countries not containing ‘land’ in their names.



**SQL Wildcard Characters**

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

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

With SQL, the wildcards are:

**Wildcard Description**

% A substitute for zero or more characters

\_ A substitute for a single character

[charlist] Sets and ranges of characters to match

[^charlist]

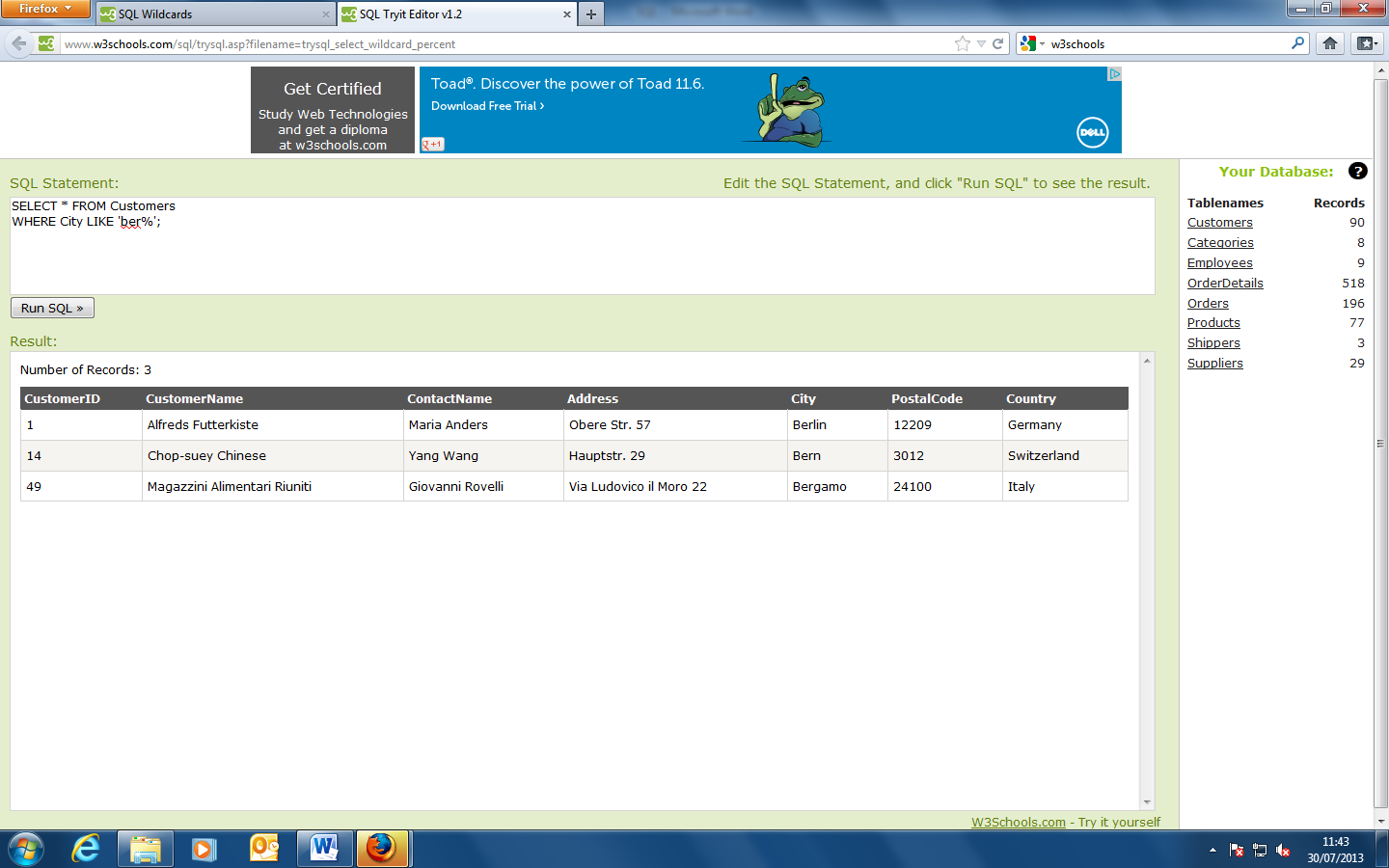
or

[!charlist] Matches only a character NOT specified within the brackets

SELECT \* FROM Customers

WHERE City LIKE 'ber%';

Selects only cities beginning with ‘ber’



SELECT \* FROM Customers

WHERE City LIKE '%es%';

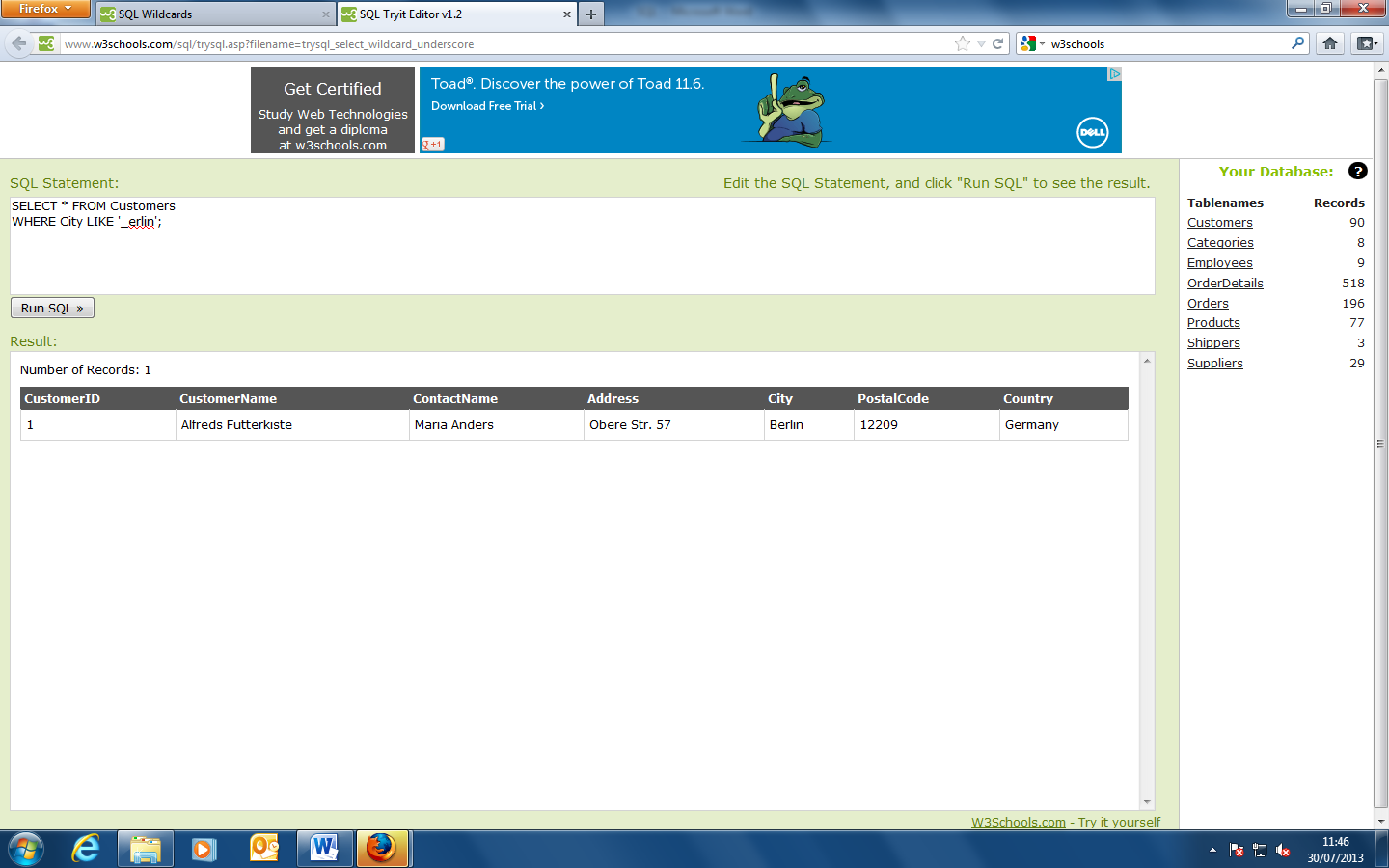
Selects only cities containing the pattern ’es’



SELECT \* FROM Customers

WHERE City LIKE '\_erlin';

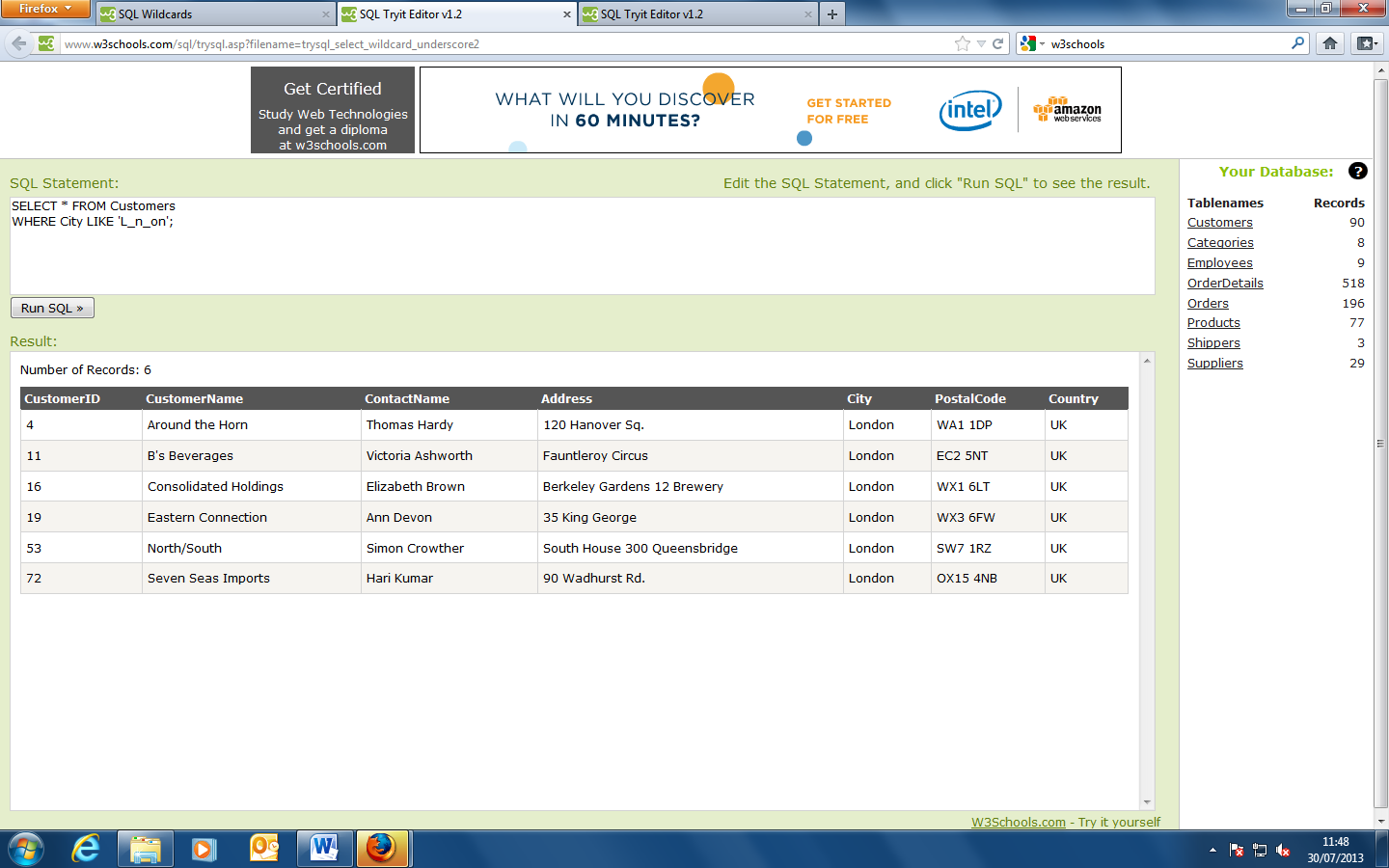
This SQL statement selects all customers with a City starting with any character, followed by "erlin":



SELECT \* FROM Customers

WHERE City LIKE 'L\_n\_on';

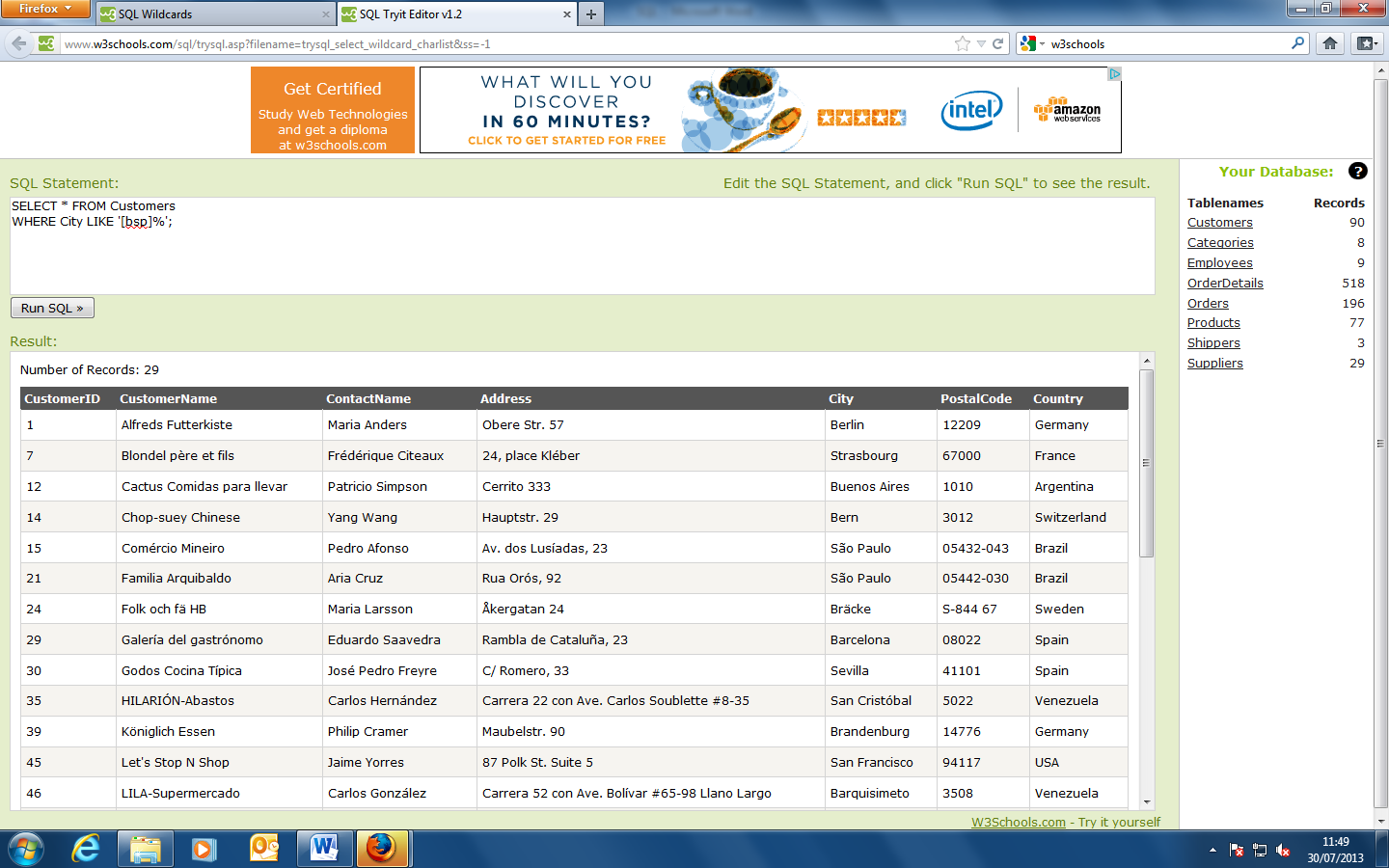
This SQL statement selects all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on".



SELECT \* FROM Customers

WHERE City LIKE '[bsp]%';

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



SELECT \* FROM Customers

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

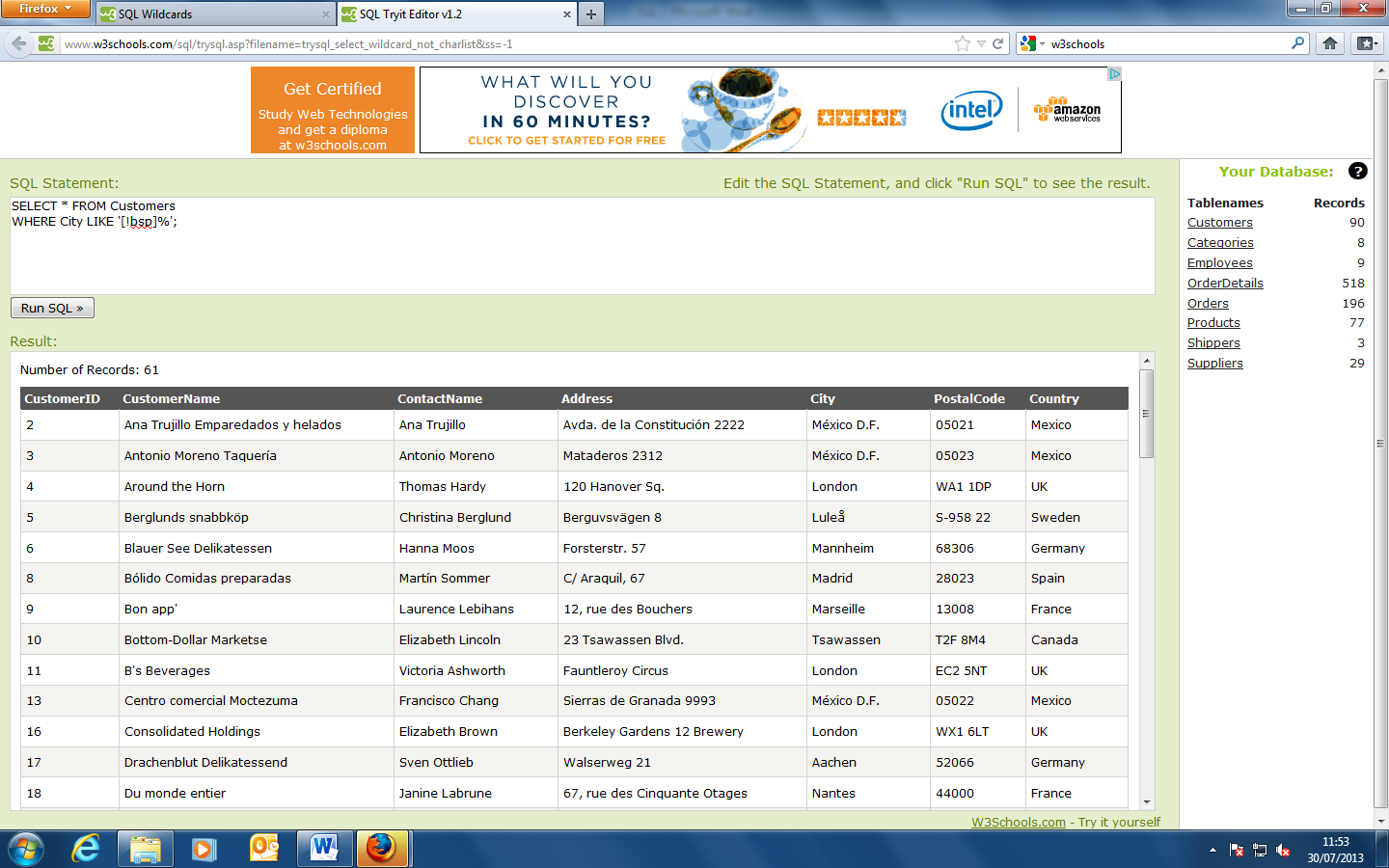
This SQL statement selects all customers with a City starting with "a", "b", or "c":



SELECT \* FROM Customers

WHERE City LIKE '[!bsp]%';

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



**The IN Operator**

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

SQL IN Syntax

SELECT column\_name(s)

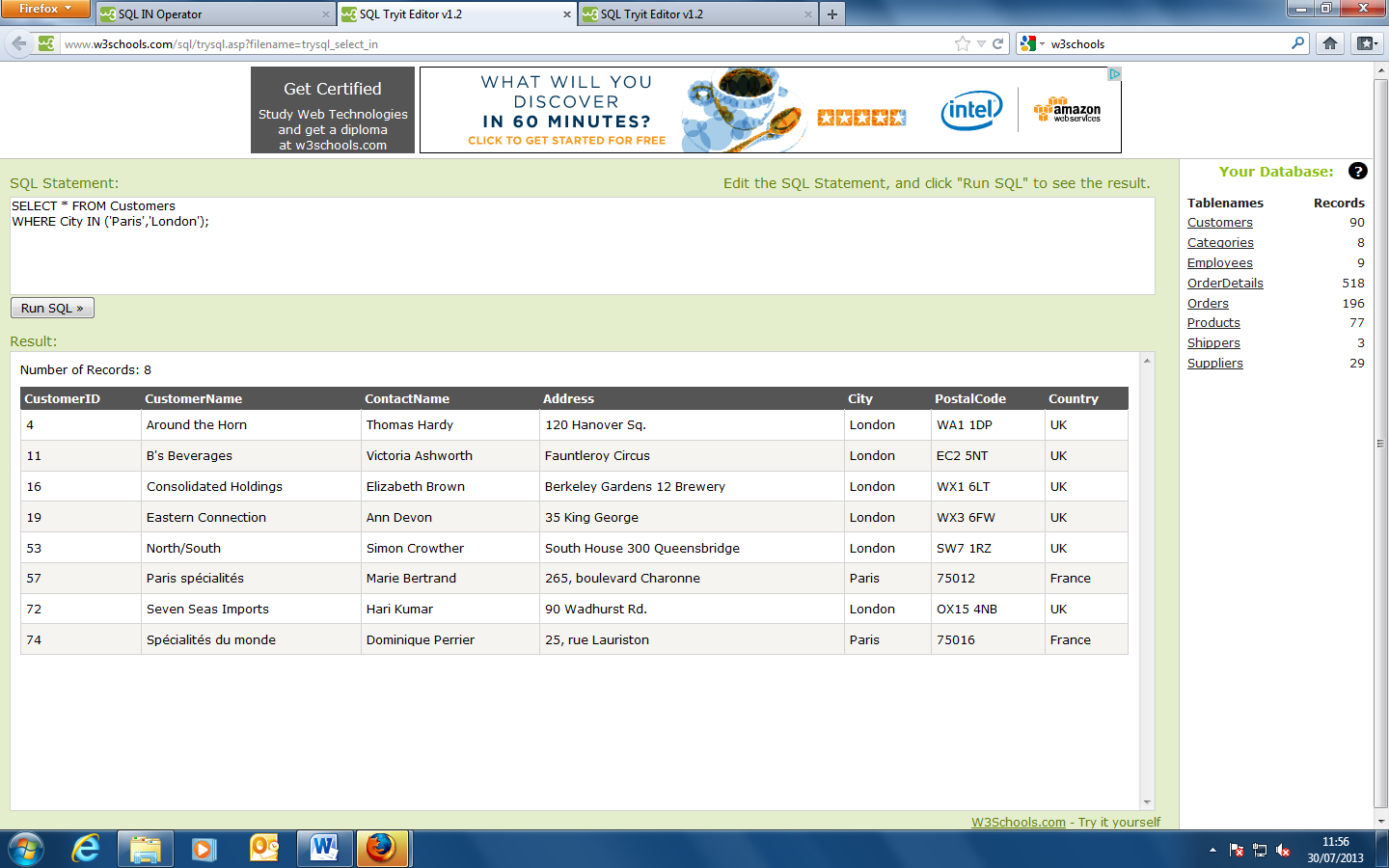
FROM table\_name

WHERE column\_name IN (value1,value2,...);

SELECT \* FROM Customers

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

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



**The SQL BETWEEN Operator**

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

SQL BETWEEN Syntax

SELECT column\_name(s)

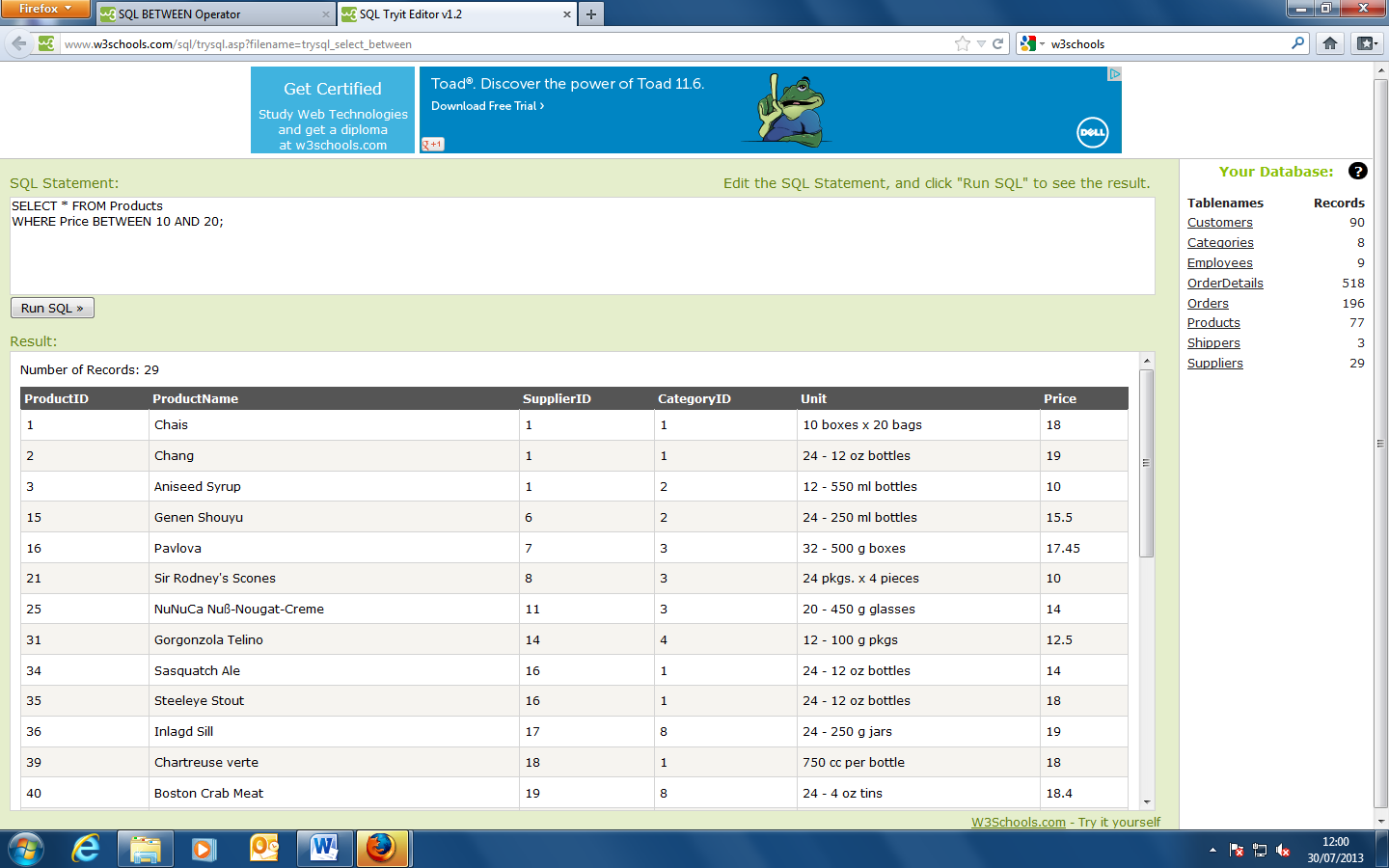
FROM table\_name

WHERE column\_name BETWEEN value1 AND value2;

SELECT \* FROM Products

WHERE Price BETWEEN 10 AND 20;

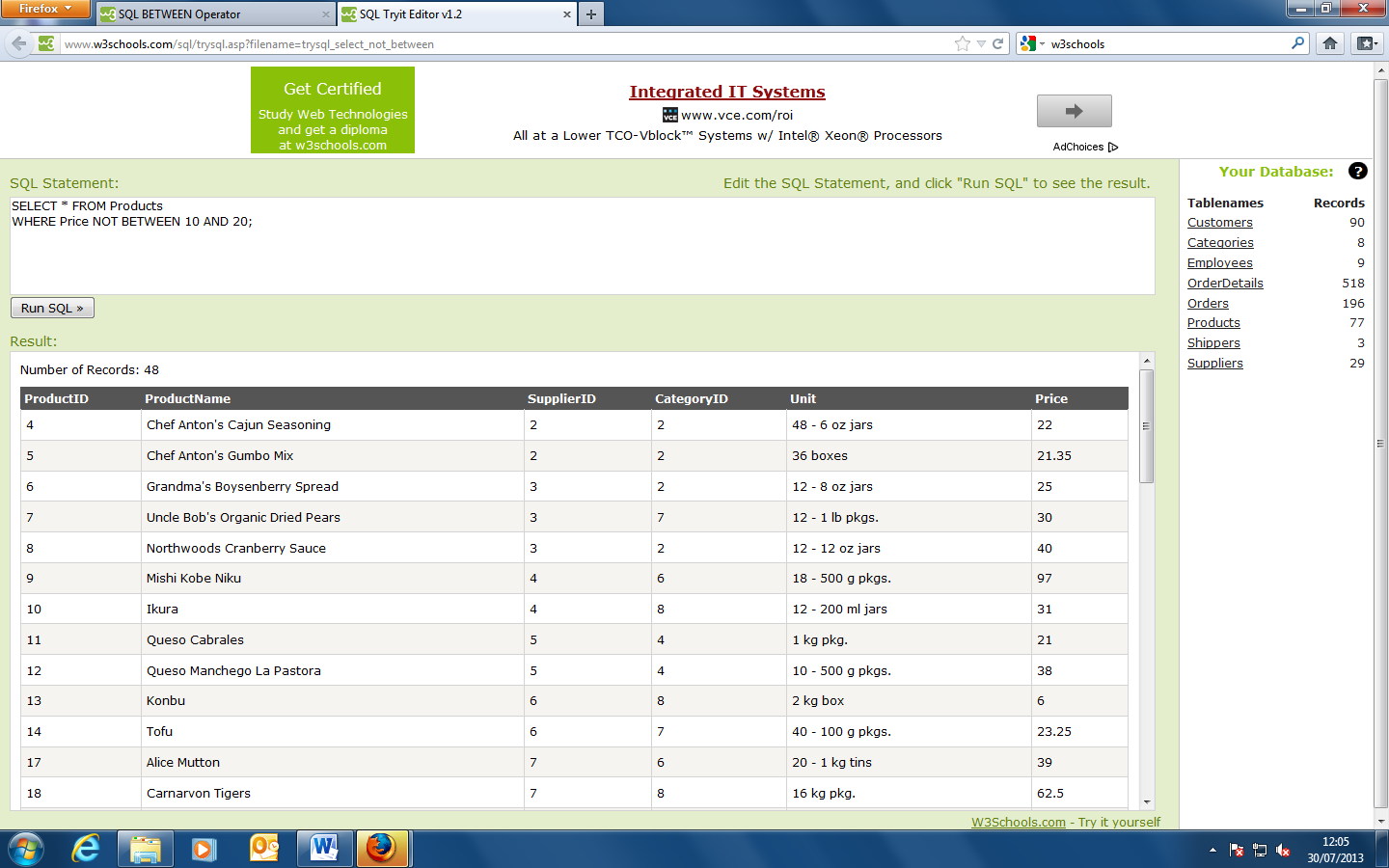
This SQL statement selects all products with a price BETWEEN 10 and 20:



SELECT \* FROM Products

WHERE Price NOT BETWEEN 10 AND 20;

This SQL statment displays the products outside the range of the previous example, use NOT BETWEEN:

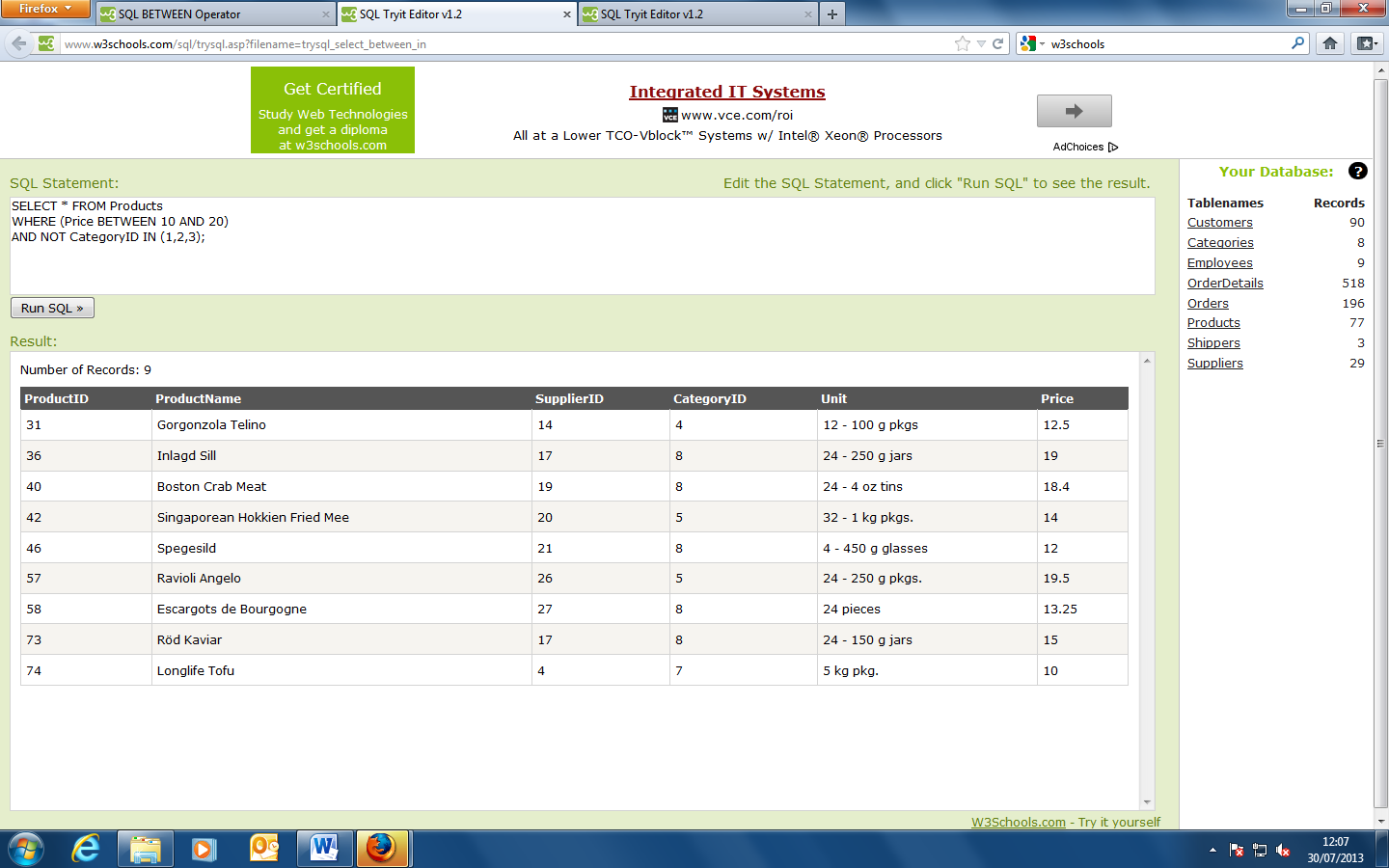


SELECT \* FROM Products

WHERE (Price BETWEEN 10 AND 20)

AND NOT CategoryID IN (1,2,3);

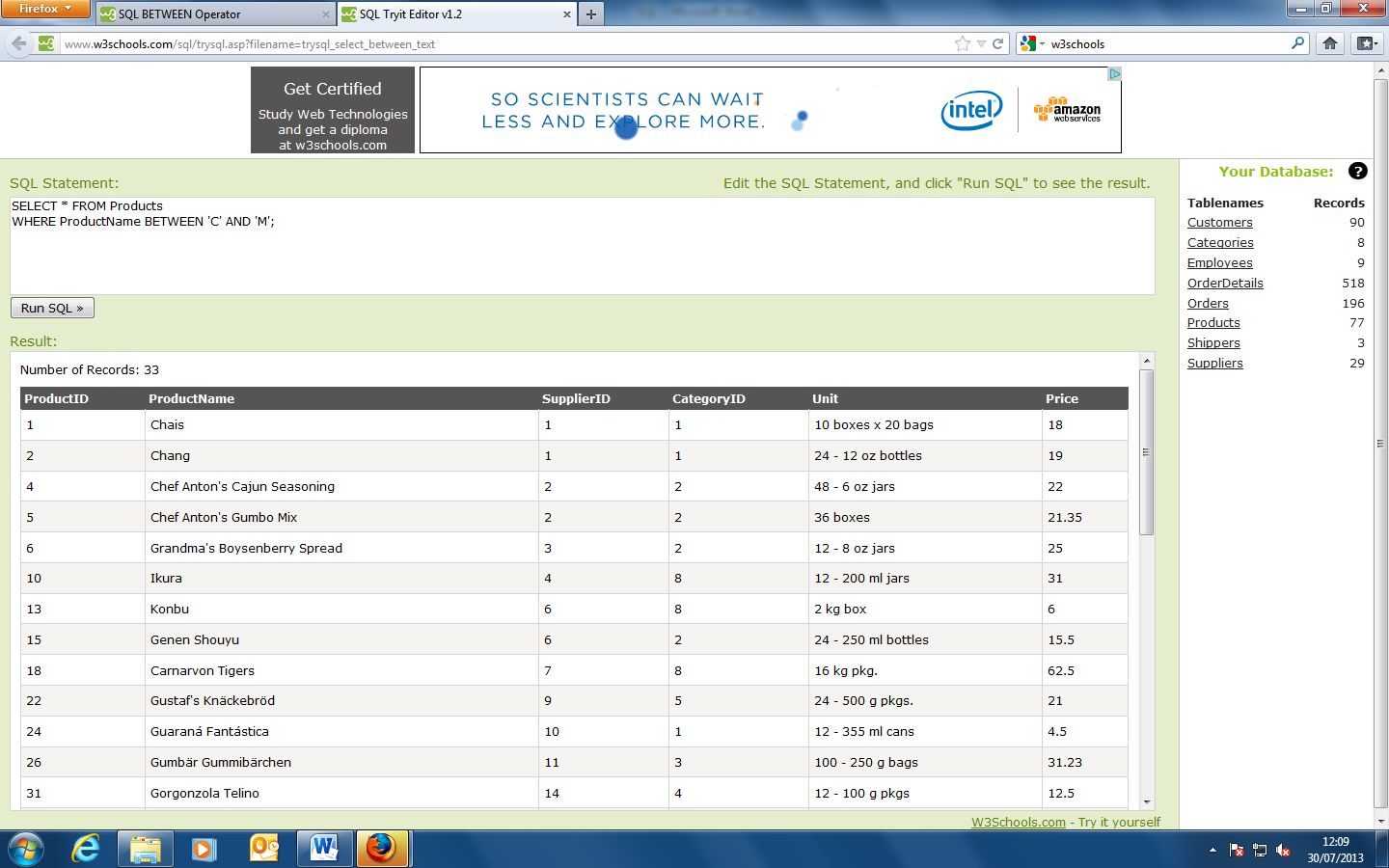
ThisSQL statement selects all products with a price BETWEEN 10 and 20, but products with a CategoryID of 1,2, or 3 should not be displayed:



SELECT \* FROM Products

WHERE ProductName BETWEEN 'C' AND 'M';

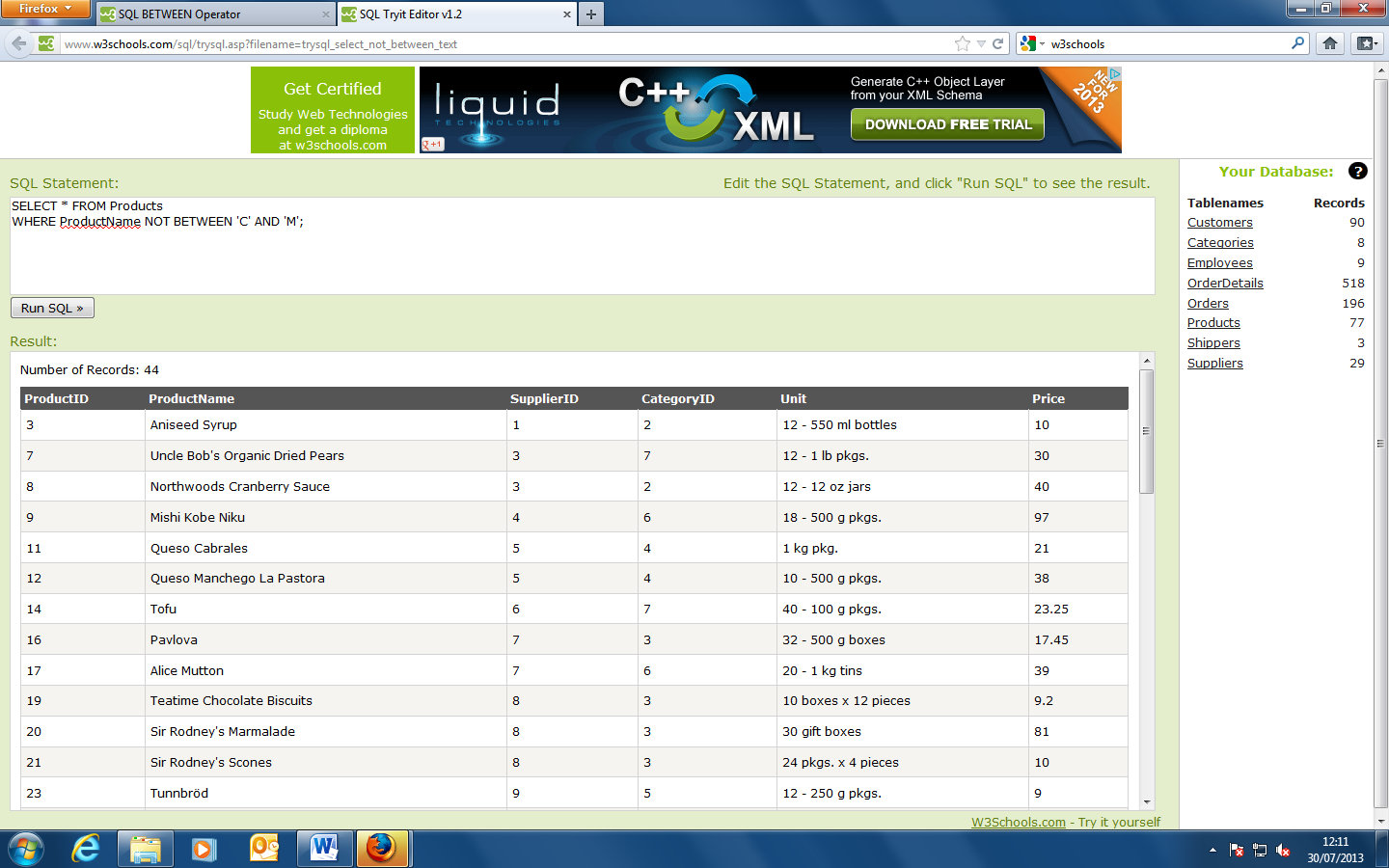
This SQL statement selects all products with a ProductName beginning with any of the letter BETWEEN 'C' and 'M'



SELECT \* FROM Products

WHERE ProductName NOT BETWEEN 'C' AND 'M';

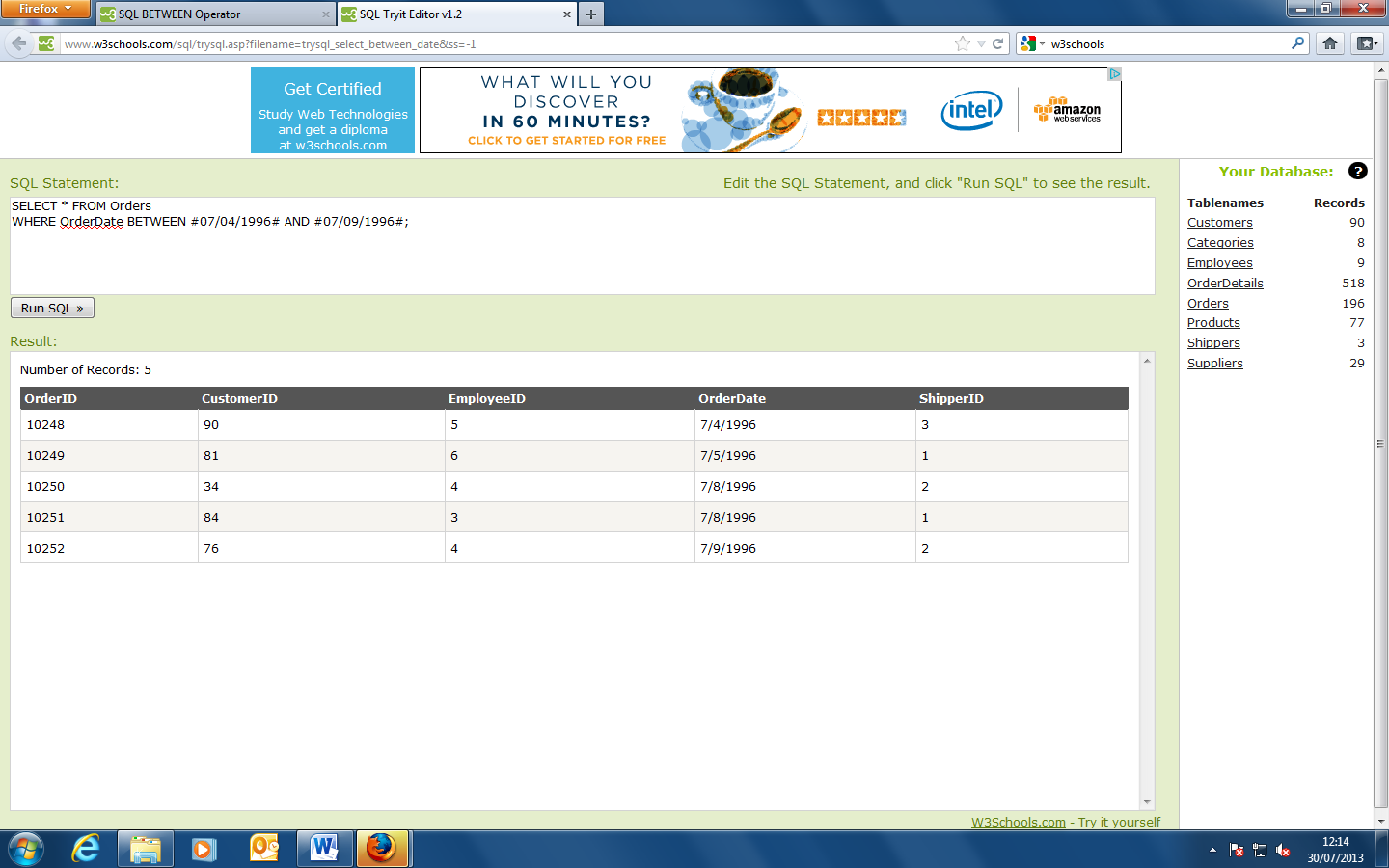
This SQL statement selects all products with a ProductName beginning with any of the letter NOT BETWEEN 'C' and 'M':



SELECT \* FROM Orders

WHERE OrderDate BETWEEN #07/04/1996# AND #07/09/1996#;

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



**SQL Aliases**

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

Basically aliases are created to make column names more readable.

SQL Alias Syntax for Columns

SELECT column\_name AS alias\_name

FROM table\_name;

SQL Alias Syntax for Tables

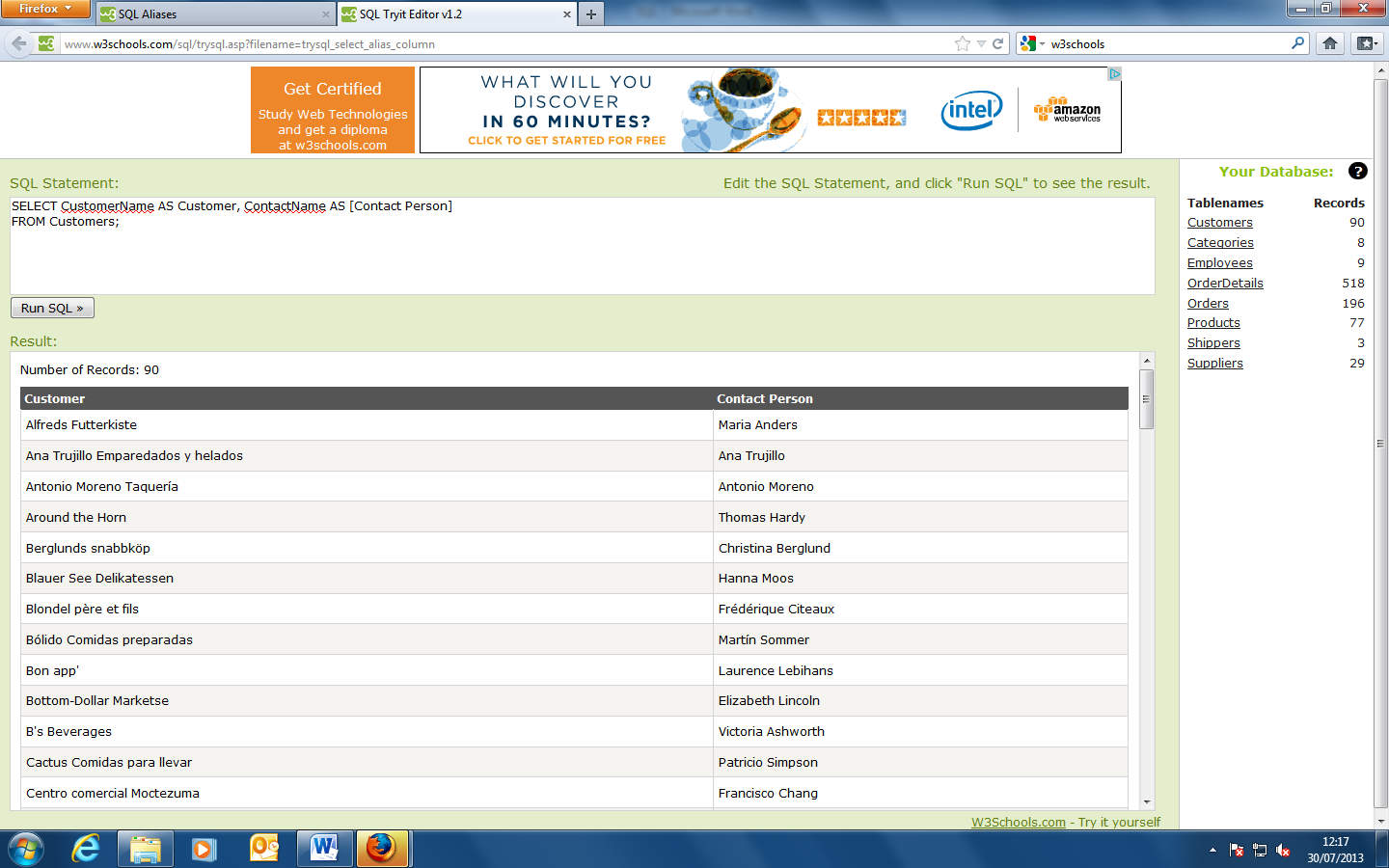
SELECT column\_name(s)

FROM table\_name AS alias\_name;

SELECT CustomerName AS Customer, ContactName AS [Contact Person]

FROM Customers;

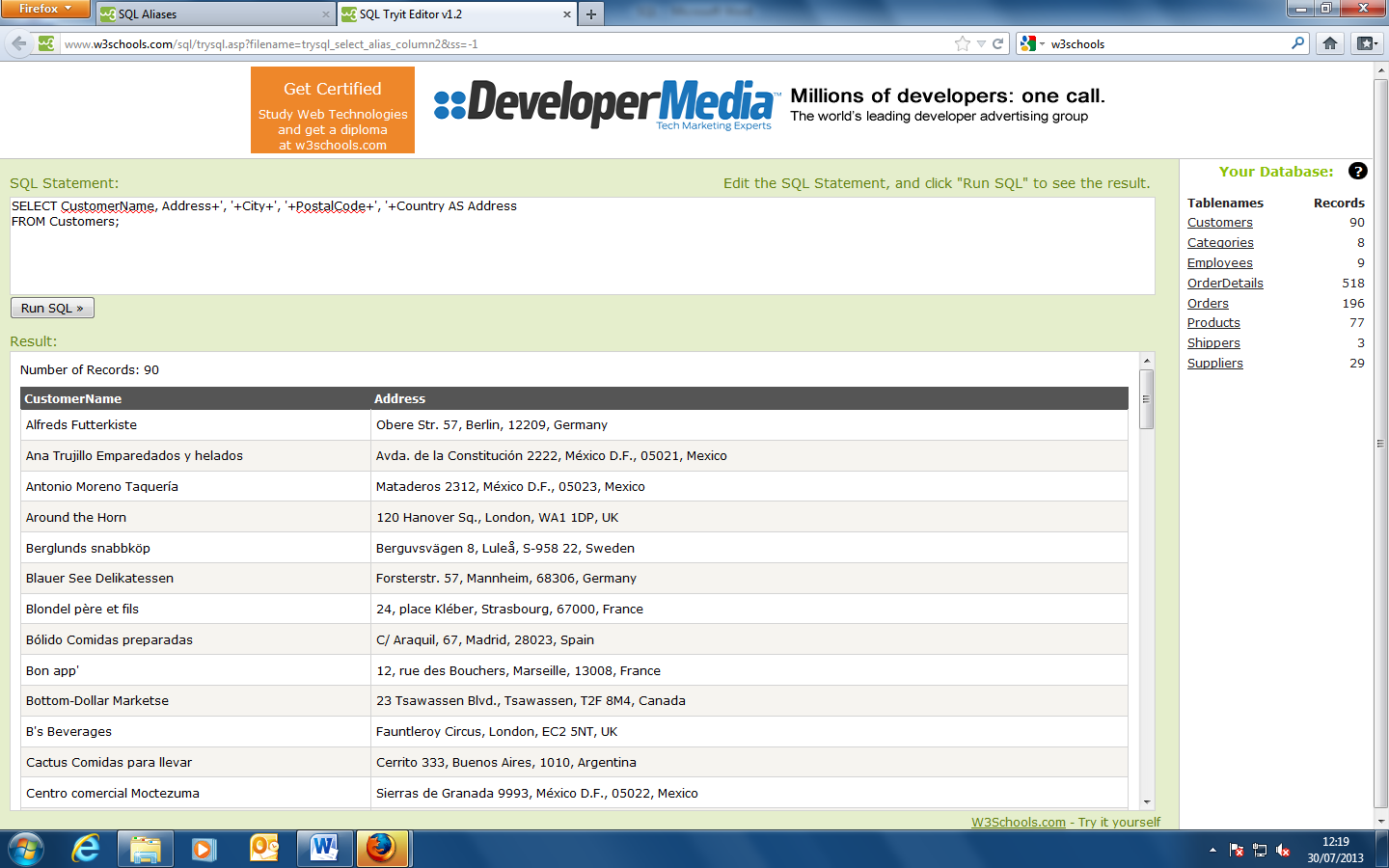
This SQL statement specifies two aliases, one for the CustomerName column and one for the ContactName column. **Tip:** It require double quotation marks or square brackets if the column name contains spaces.



SELECT CustomerName, Address+', '+City+', '+PostalCode+', '+Country AS Address

FROM Customers;

This SQL statement we combine four columns (Address, City, PostalCode, and Country) and create an alias named "Address".



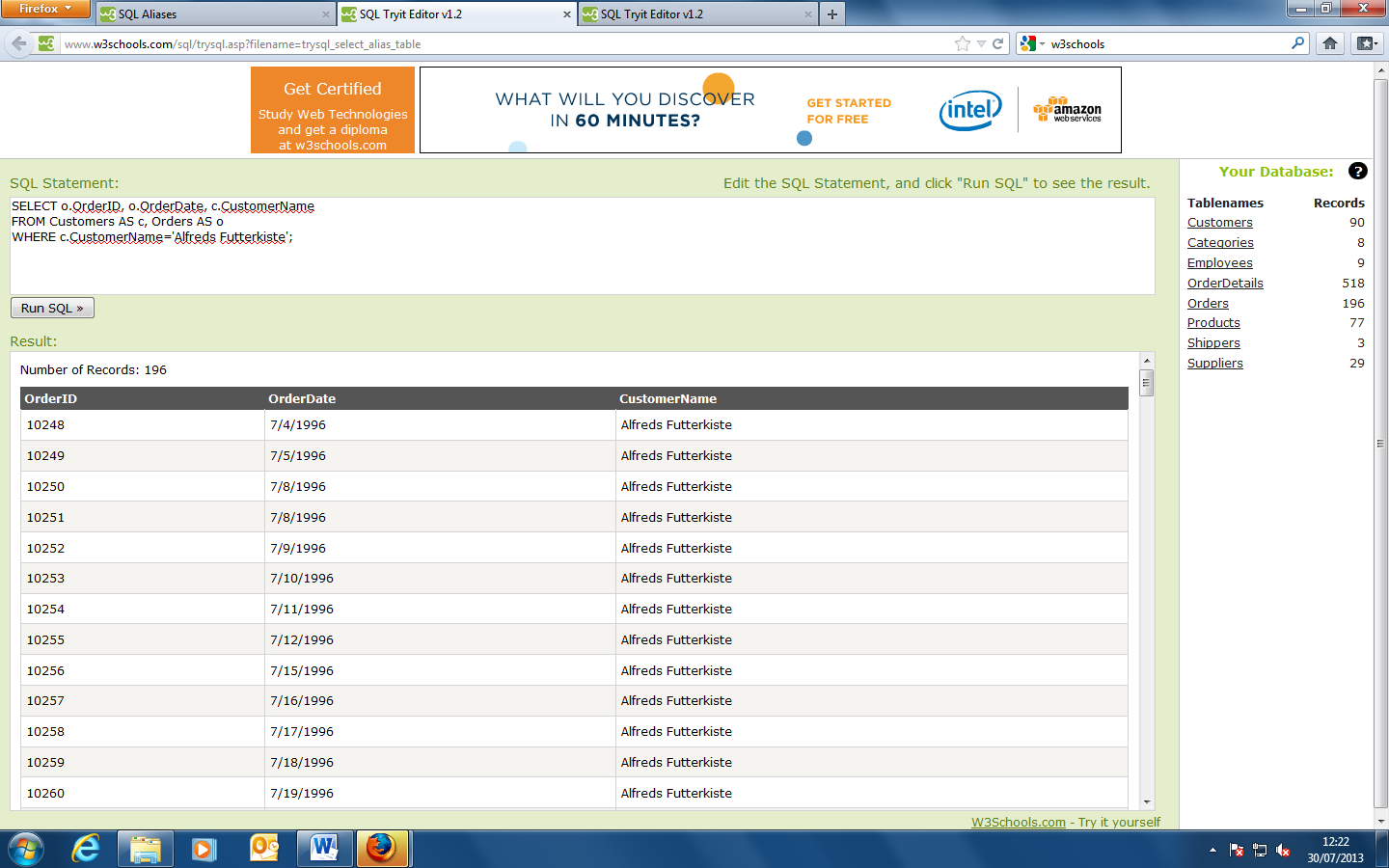
**Alias Example for Tables**

SELECT o.OrderID, o.OrderDate, c.CustomerName

FROM Customers AS c, Orders AS o

WHERE c.CustomerName='Alfreds Futterkiste';

This 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).

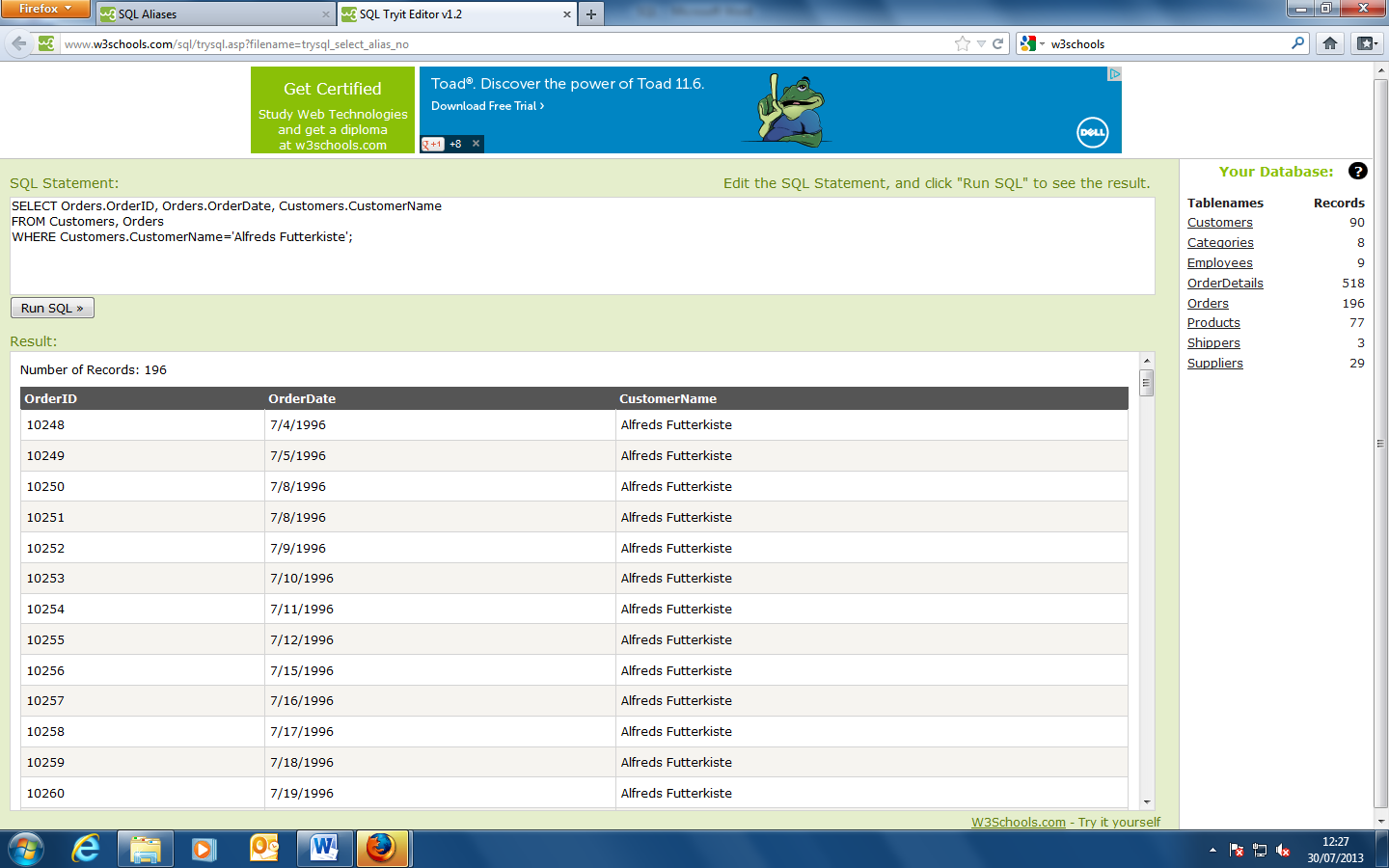


SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName

FROM Customers, Orders

WHERE Customers.CustomerName='Alfreds Futterkiste';

This is the same SQL statement without aliases:



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

**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.

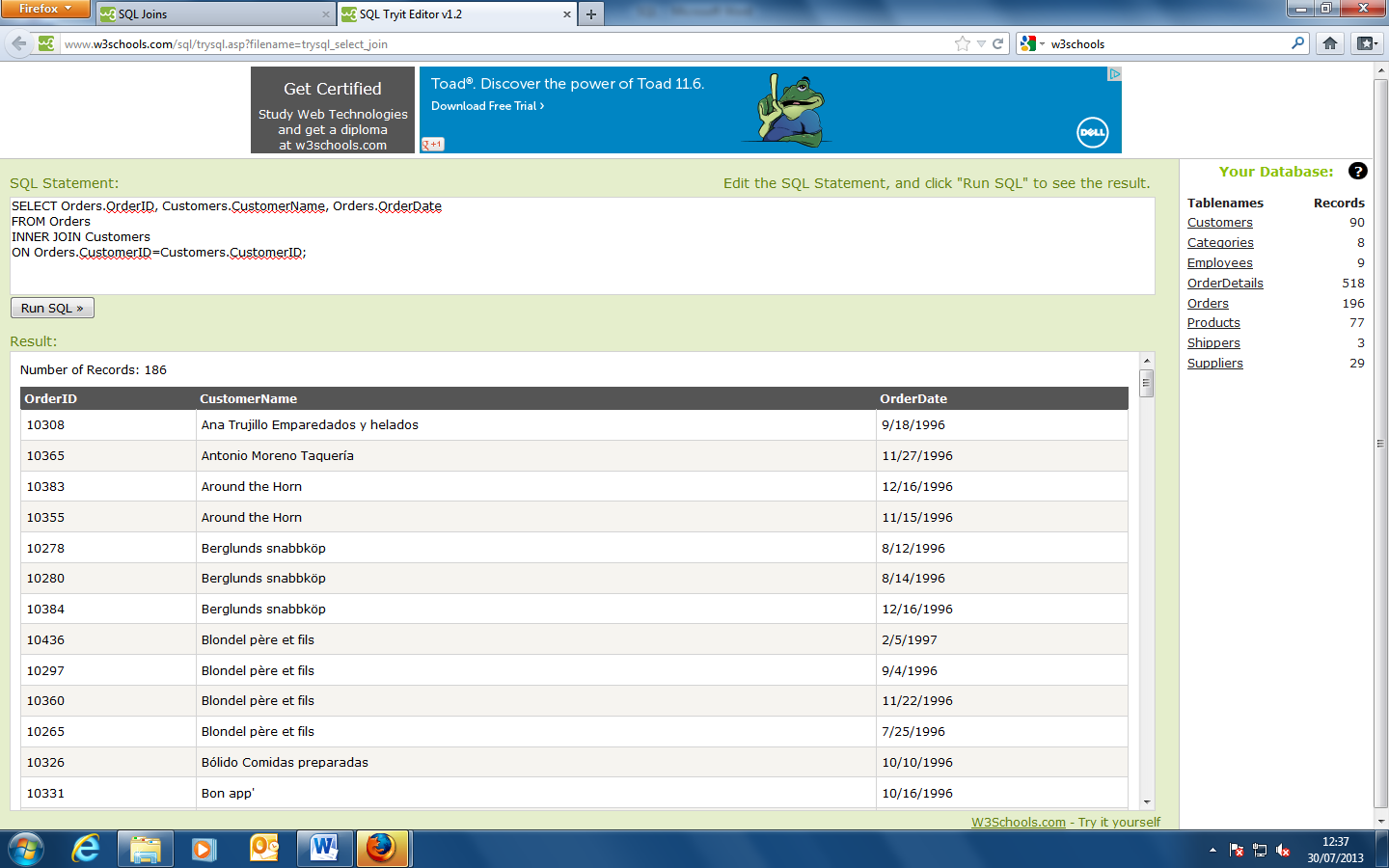
Then, if we run the following SQL statement (that contains an INNER JOIN):

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate

FROM Orders

INNER JOIN Customers

ON Orders.CustomerID=Customers.CustomerID;



**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.

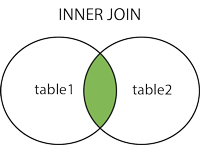
**SQL INNER JOIN Syntax**

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*;

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



SELECT Customers.CustomerName, Orders.OrderID

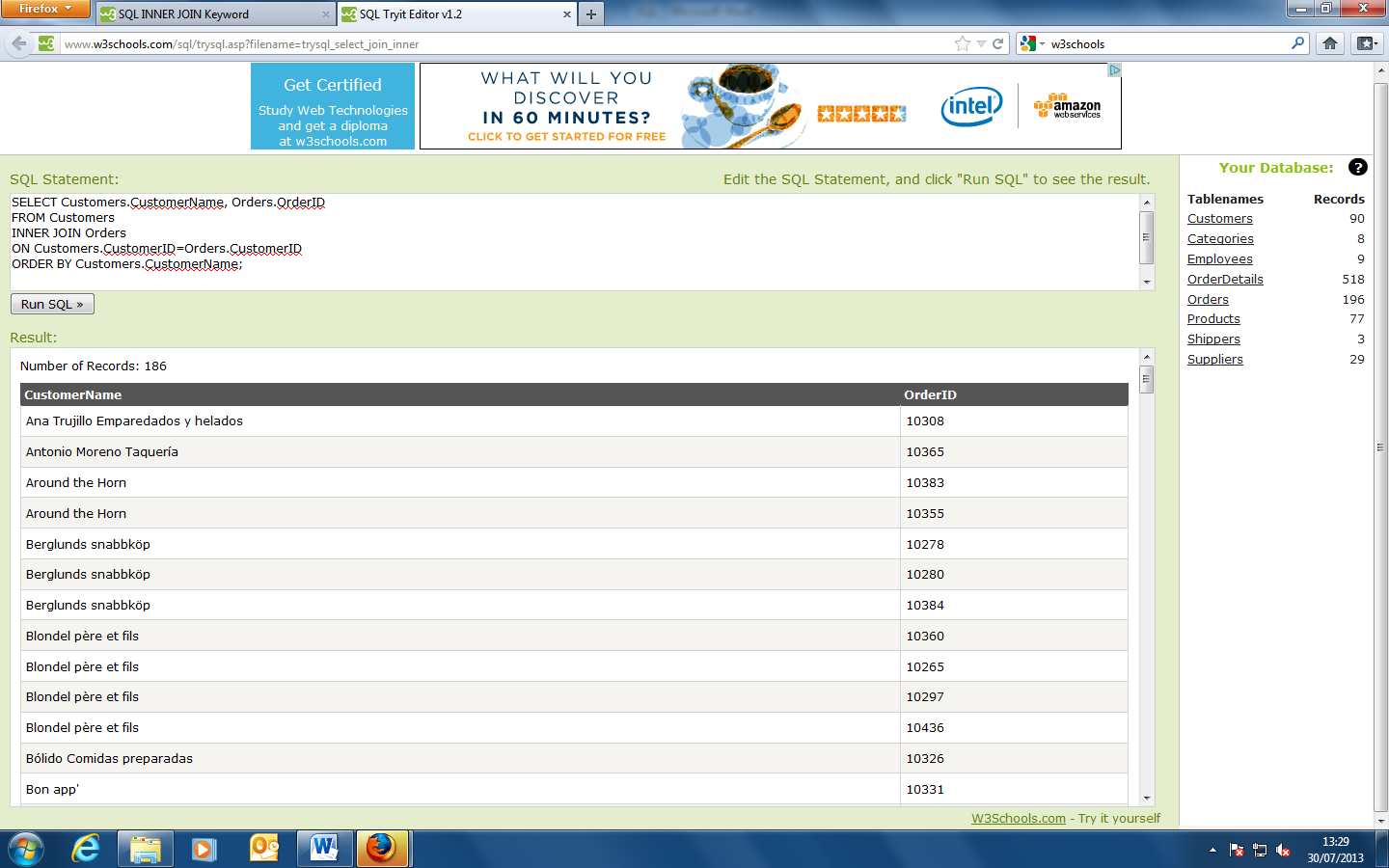
FROM Customers

INNER JOIN Orders

ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

This SQL statement will return all customers with orders:



**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.

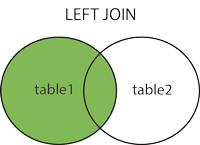
**SQL LEFT JOIN Syntax**

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.



SELECT Customers.CustomerName, Orders.OrderID

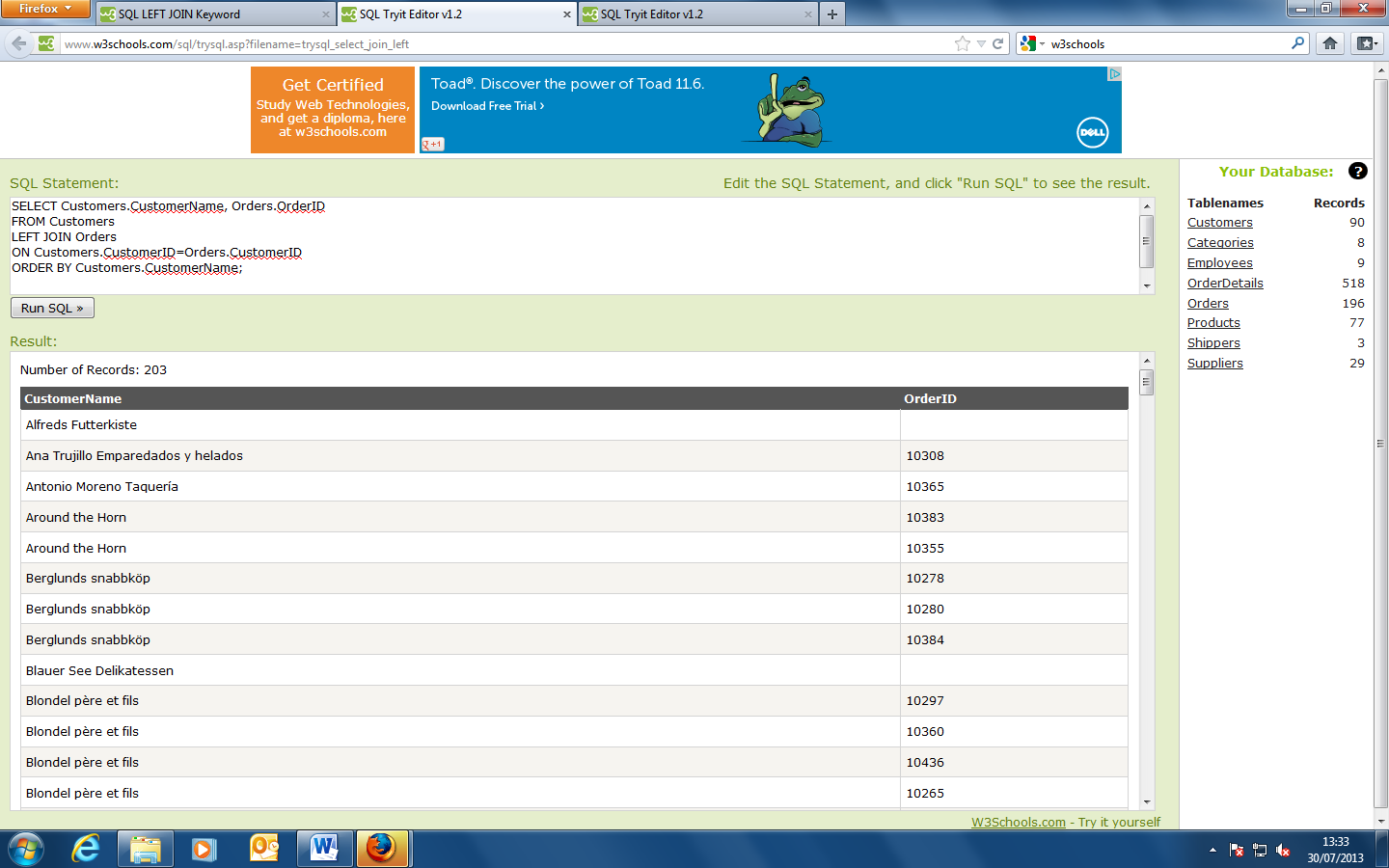
FROM Customers

LEFT JOIN Orders

ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

This SQL statement will return all customers, and any orders they might have



**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.

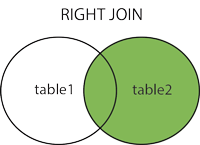
**SQL RIGHT JOIN Syntax**

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.



SELECT Customers.CustomerName, Orders.OrderID

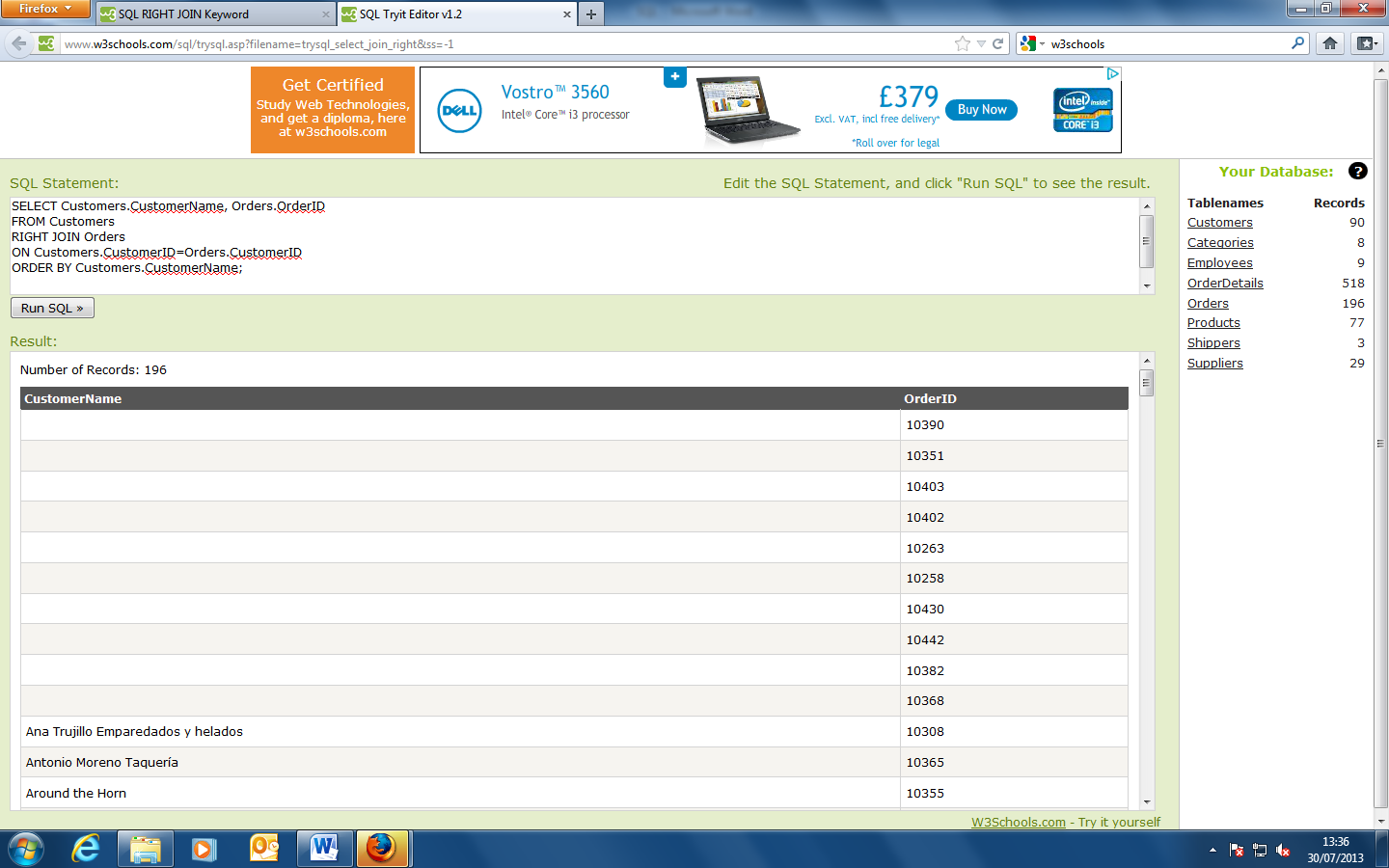
FROM Customers

RIGHT JOIN Orders

ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

This SQL statement will return all orders, and any customers that might have placed them



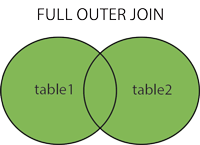
**SQL FULL OUTER JOIN Keyword**

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.

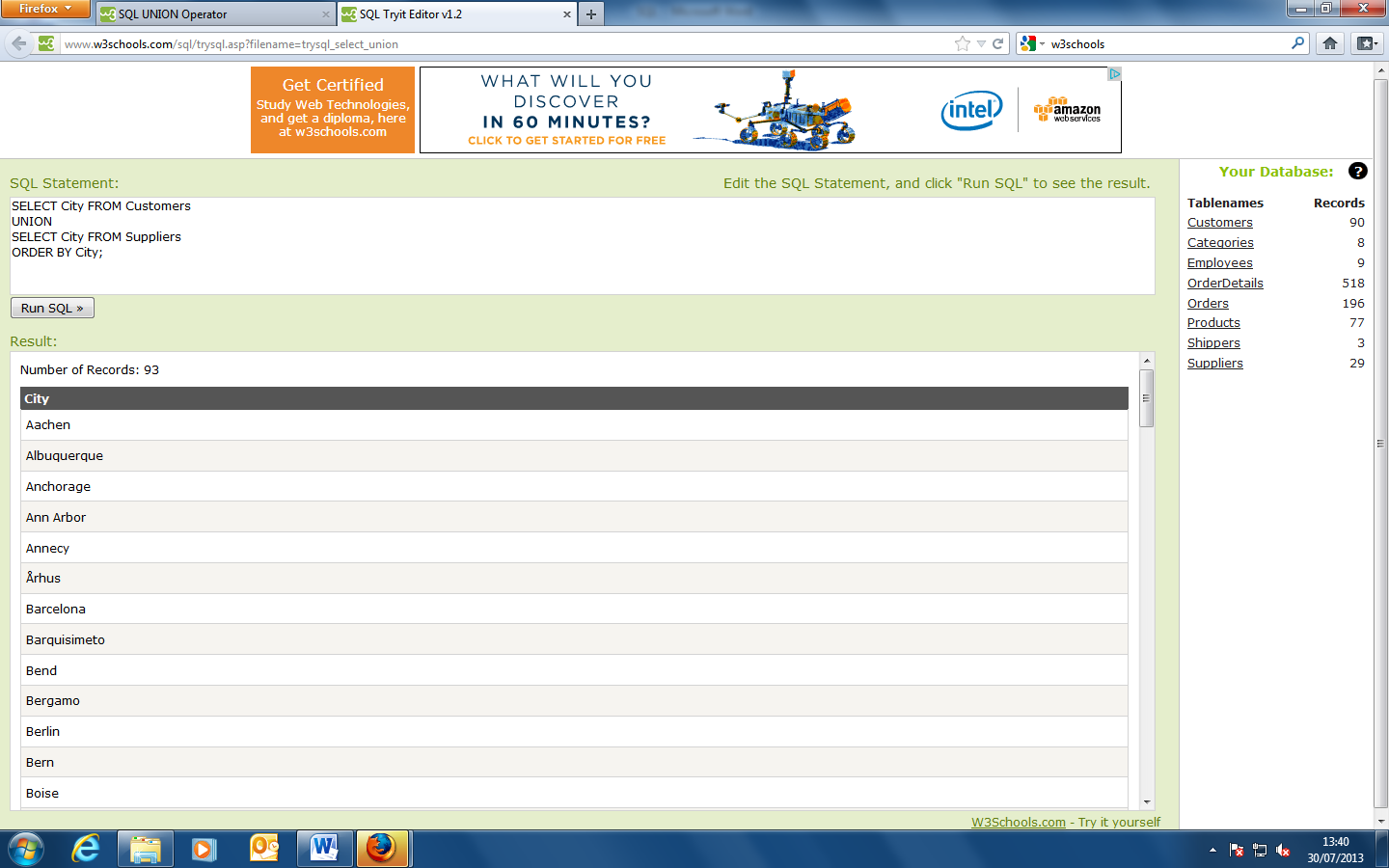
SELECT City FROM Customers

UNION

SELECT City FROM Suppliers

ORDER BY City;

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



**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!

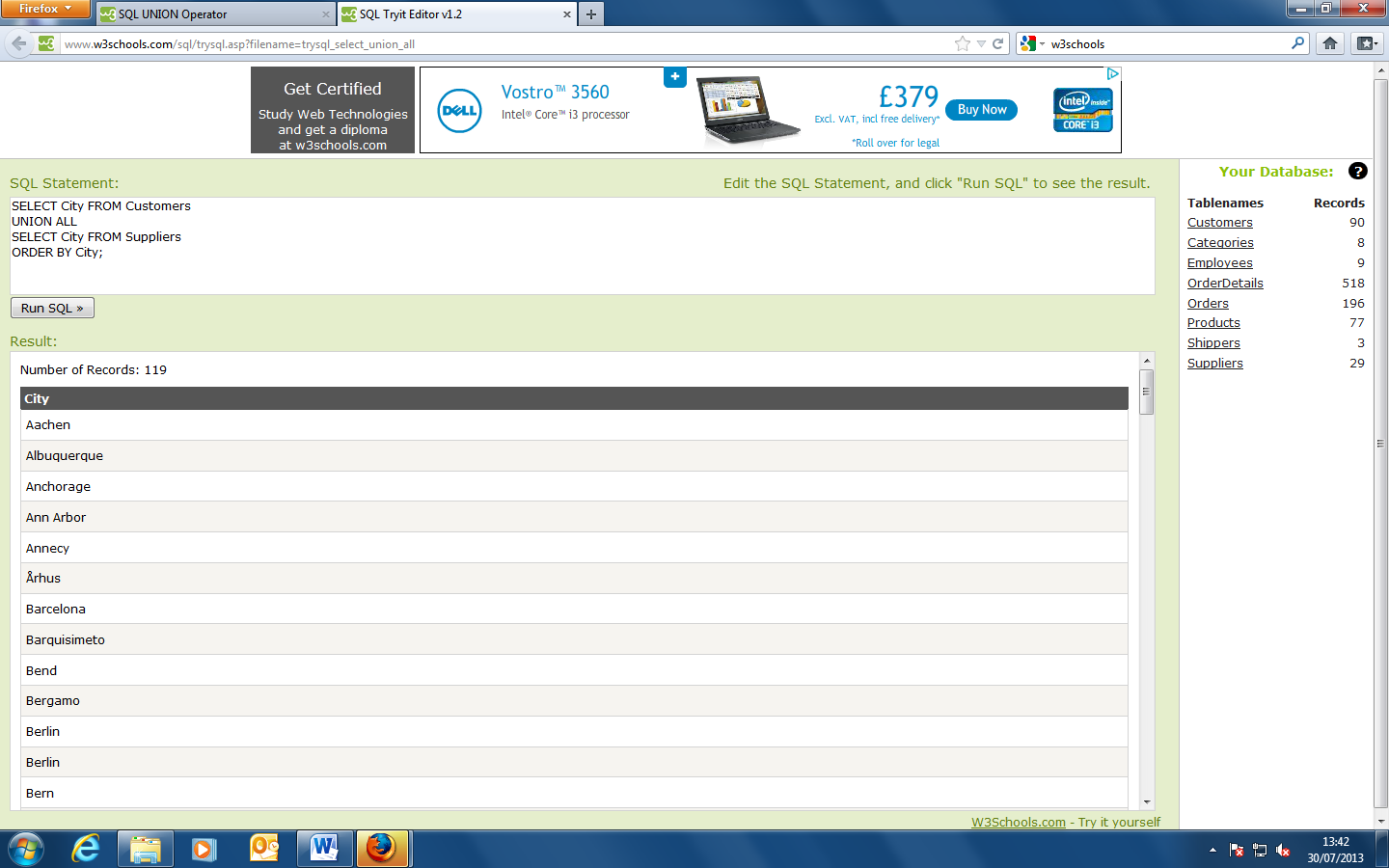
SELECT City FROM Customers

UNION ALL

SELECT City FROM Suppliers

ORDER BY City;

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



SELECT City, Country FROM Customers

WHERE Country='Germany'

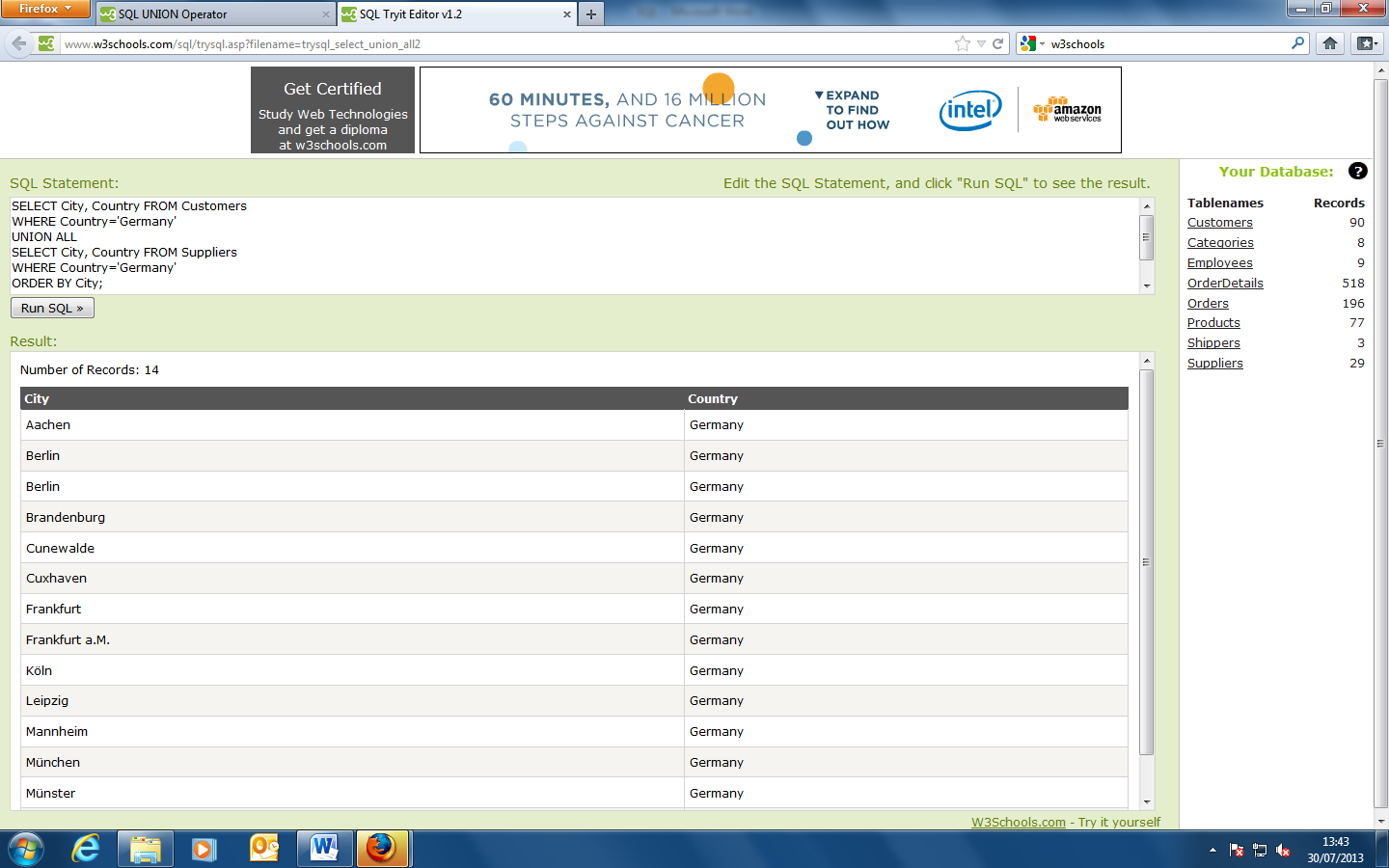
UNION ALL

SELECT City, Country FROM Suppliers

WHERE Country='Germany'

ORDER BY City;

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



**SQL SELECT INTO Examples**

Create a backup copy of Customers:

SELECT \*  
INTO CustomersBackup2013  
FROM Customers;

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

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

Copy only a few columns into the new table:

SELECT CustomerName, ContactName  
INTO CustomersBackup2013  
FROM Customers;

Copy only the German customers into the new table:

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

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

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

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

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

**The SQL INSERT INTO SELECT Statement**

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

**SQL INSERT INTO SELECT Syntax**

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

INSERT INTO *table2*  
SELECT \* FROM *table1;*

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

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

**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';

**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](http://www.w3schools.com/sql/sql_datatypes.asp).

**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

**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

**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

**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)

**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](http://www.w3schools.com/sql/sql_datatypes.asp).

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 | |  |  |  |  |