THE GUIDE TO WRITING SQL QUERIES

CONTENTS

INTRODUCTION TO SQL

At the heart of most database systems sits the Structured Query Language, or SQL. This is the language which allows the user to construct and manipulate the database, creating and dropping tables, working with relationships and indexes, interrogating the tables and extracting data, as well as providing much other functionality.

It is a mistake, however, to think of SQL as a single language. There are several ANSI standards for the SQL language which define the essential statements such as SELECT and CREATE and the syntax such statements must follow; these have evolved over the years, and have been extended by various manufacturers to provide their own 'dialects' or implementations. Each adds to or tweaks the functionality of the basic SQL syntax to allow for an "optimised" version for their own database products, often adding programming functionality such as declaration of variables, stored procedure parameters and flow-control constructs.

So, for SQL Server, Microsoft have created Transact SQL (or T-SQL) while Oracle use PL-SQL for their database systems. Access uses yet another dialect (JET SQL) - even though it's a Microsoft product, it doesn't use pure T-SQL. If, however, you want to use ADO within VBA to automate Access, then you'll be using JET SQL within the application front-end, and T-SQL within the VBA routines...

Don't be put off by this complexity. In very many day-to-day tasks, the language is almost identical in the various dialects, and getting to grips with many of the differences is as easy as falling off a pavement. Or sidewalk....

In these pages, the examples are written and tested in Microsoft's SQL Server, and so use T-SQL. Where there are differences which would prevent these examples from working within Microsoft Access, I have noted such differences. All the examples use Microsoft's Northwind database, which can be downloaded direct from Microsoft's website.

BASIC SELECT STATEMENTS

The select statement is one of the workhorses of SQL. It is used to interrogate tables and return data, and also to return the result of functions, variables and other system objects. However, for this section, we are concerned only with its use as a way of extracting data from tables within a database.

The basic syntax for the Select statement is as follows:

```
SELECT column names FROM table name
```

Let's say we wanted to show the following columns from the Customers table: CompanyName, ContactName, City and Country. The select statement to extract this information would be as follows:

```
SELECT companyname, contactname, city, country
FROM customers
```

This will return the values in the listed columns from all records in the table. Note that the layout of this select statement is unimportant, as is the use of upper case for keywords. However, you may find that laying out the statements as above will make your text easier to read.

There are a couple of other points to note. Firstly, if the names of your columns or tables contain spaces - not good practise, but we don't always live in an ideal world - then you'll need to enclose them within square brackets. Thus, in order to request the OrderID, ProductID, Quantity and UnitPrice columns from the Order Details table, you would use the following statement:

```
SELECT orderid, productid, quantity, unitprice
FROM [order details]
```

Secondly, what if you want to return all the columns, rather than a specified list? You don't need to enter all the names - you can simply use an asterisk as shorthand:

```
SELECT *
FROM customers
```

That's it for this section! In the next topic, we'll look at restricting your results so that you don't run the risk of returning several million rows of data!

MODIFYING THE SELECT STATEMENT

ORDER BY, TOP AND DISTINCT

The output from a SELECT statement can be modified in a number of ways through the use of the ORDER BY clause and the addition of the TOP and DISTINCT keywords.

THE ORDER BY CLAUSE

By default, a SELECT statement will retrieve records from a table according to the order in which they exist in that table, which will be governed by the order in which they were entered as well as the use of indexes (including the Primary Key).

If needed, the records can be sorted by one or more fields by the addition of an ORDER BY clause which specifies which columns should be used for sorting, and the order in which they should be sorted.

The syntax of an order by clause is as follows:

```
SELECT Columns
FROM Table
ORDER BY ColumnName [ASC | DESC]
```

There is no limit to the number of columns which can be included in an ORDER BY clause, each optionally modified with ASC or DESC, separated with commas. Furthermore, columns within the ORDER BY clause do not have to be included within the SELECT list, allowing data to be sorted by columns which are not shown.

THE TOP KEYWORD

If the ORDER BY clause is being used, the TOP keyword may also be used. This restricts the output to the number of rows specified. The syntax when using TOP is as follows:

```
SELECT TOP n [PERCENT] [WITH TIES] Columns
FROM Table
ORDER BY ColumnName [ASC | DESC]
```

Note the following important points:

1. Either an exact value may be supplied for the number of rows to be returned, or this number may be followed by the word PERCENT if a proportion of the total rows should be returned.
2. By default, the exact number of rows specified will be returned. This means that if there are, for example, 10 students in an exam table with identical scores, using TOP 5 will only show the first 5 rows in the table. Should you want to return the top five

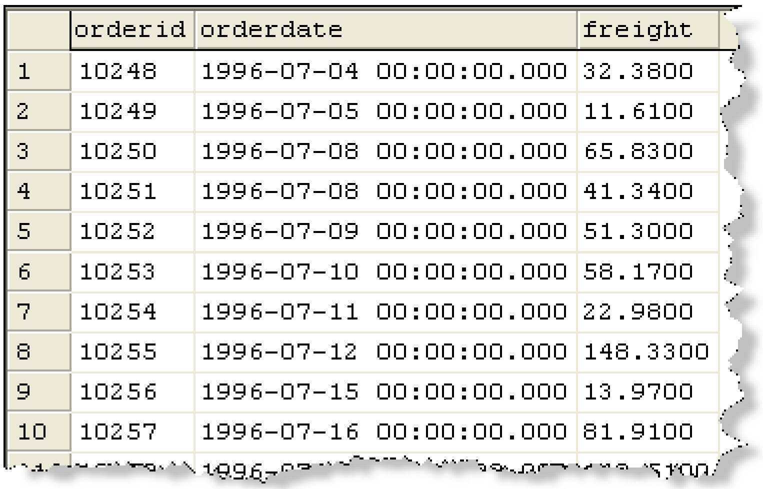
but also any other rows where the data matches (in the columns in the ORDER BY clause) then WITH TIES should also be specified prior to the column list.

Let's see some examples of ORDER BY and TOP.

The following statement:

```
SELECT orderid, orderdate, freight
FROM orders
ORDER BY orderdate
```

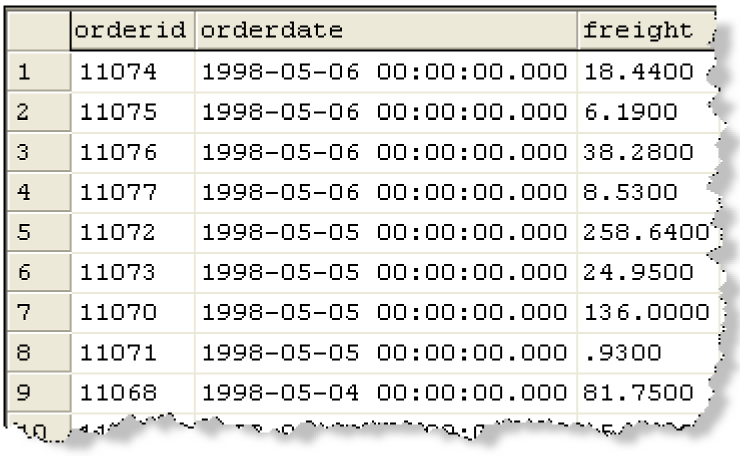
returns the following results:



	orderid	orderdate	freight
1	10248	1996-07-04 00:00:00.000	32.3800
2	10249	1996-07-05 00:00:00.000	11.6100
3	10250	1996-07-08 00:00:00.000	65.8300
4	10251	1996-07-08 00:00:00.000	41.3400
5	10252	1996-07-09 00:00:00.000	51.3000
6	10253	1996-07-10 00:00:00.000	58.1700
7	10254	1996-07-11 00:00:00.000	22.9800
8	10255	1996-07-12 00:00:00.000	148.3300
9	10256	1996-07-15 00:00:00.000	13.9700
10	10257	1996-07-16 00:00:00.000	81.9100
[illegible]	[illegible]	[illegible]	[illegible]

Adding the DESC modifier to the end of the ORDER BY clause returns the most recent date first:

```
SELECT orderid, orderdate, freight
FROM orders
ORDER BY orderdate DESC
```

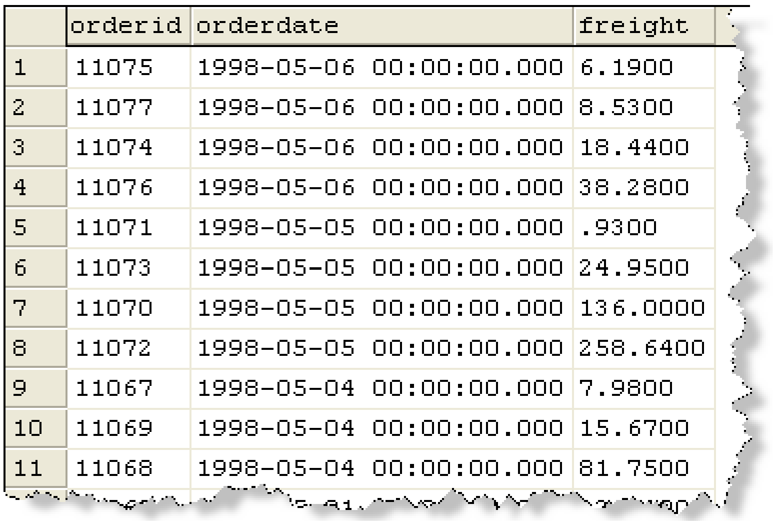


	orderid	orderdate	freight
1	11074	1998-05-06 00:00:00.000	18.4400
2	11075	1998-05-06 00:00:00.000	6.1900
3	11076	1998-05-06 00:00:00.000	38.2800
4	11077	1998-05-06 00:00:00.000	8.5300
5	11072	1998-05-05 00:00:00.000	258.6400
6	11073	1998-05-05 00:00:00.000	24.9500
7	11070	1998-05-05 00:00:00.000	136.0000
8	11071	1998-05-05 00:00:00.000	.9300
9	11068	1998-05-04 00:00:00.000	81.7500
[illegible]	[illegible]	[illegible]	[illegible]

In order to sort the output by more than one column, we simply list these columns in the ORDER BY clause:

```
SELECT orderid, orderdate, freight
FROM orders
ORDER BY orderdate DESC, freight ASC
```

This delivers an output where the rows returned are sorted firstly by date, in descending order, and then within each date by freight in ascending order:



	orderid	orderdate	freight
1	11075	1998-05-06 00:00:00.000	6.1900
2	11077	1998-05-06 00:00:00.000	8.5300
3	11074	1998-05-06 00:00:00.000	18.4400
4	11076	1998-05-06 00:00:00.000	38.2800
5	11071	1998-05-05 00:00:00.000	.9300
6	11073	1998-05-05 00:00:00.000	24.9500
7	11070	1998-05-05 00:00:00.000	136.0000
8	11072	1998-05-05 00:00:00.000	258.6400
9	11067	1998-05-04 00:00:00.000	7.9800
10	11069	1998-05-04 00:00:00.000	15.6700
11	11068	1998-05-04 00:00:00.000	81.7500

Note that we do not need to use ASC as it is the default, however it is available should you wish to use it for clarity.

Lets have a look now at the use of TOP, both with and without WITH TIES.

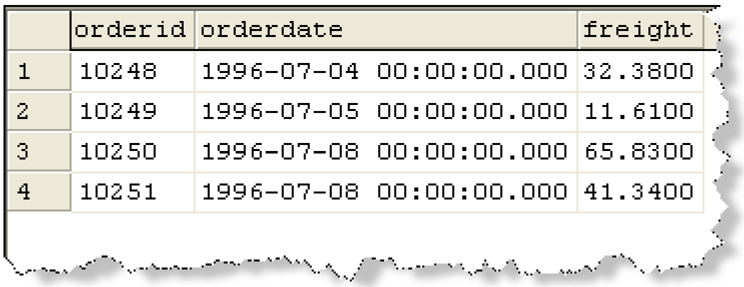
If we issue the following statement:

```
SELECT TOP 3 orderid, orderdate, freight
FROM orders
ORDER BY orderdate
```

then we see exactlty 3 records - the first three in the order as displayed in the first screenshot above. However, if we add WITH TIES:

```
SELECT TOP 3 WITH TIES orderid, orderdate, freight
FROM orders
ORDER BY orderdate
```

we now get four records return, as the fourth row matches the third when ordered by orderdate:



	orderid	orderdate	freight
1	10248	1996-07-04 00:00:00.000	32.3800
2	10249	1996-07-05 00:00:00.000	11.6100
3	10250	1996-07-08 00:00:00.000	65.8300
4	10251	1996-07-08 00:00:00.000	41.3400

Finally for this section, we can also use the DISTINCT keyword to modify the output from a SELECT statement.

If the output from the columns involved in the SELECT statement would result in duplicate rows being returned, DISTINCT removes the duplicates from the rowset. Note that the rows do not have to be duplicates in their entirety within the table - merely in the combined output of the columns within the select list.

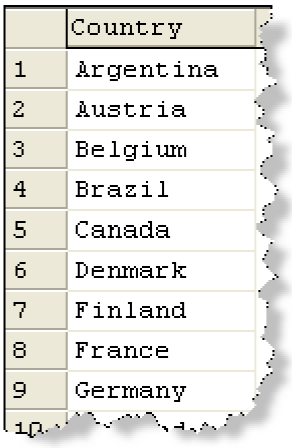
Therefore, if the following statement is issued:

```
SELECT Country
FROM Customers
```

the result would be 91 rows - the value in the Country field for each customer. However, if we modify this statement as follows:

```
SELECT DISTINCT Country
FROM Customers
```

the output would be only 21 rows - a list of the distinct countries found within this field in the customers table:



	Country
1	Argentina
2	Austria
3	Belgium
4	Brazil
5	Canada
6	Denmark
7	Finland
8	France
9	Germany

INTRODUCTION TO THE WHERE CLAUSE

So far, the various SELECT statements we've looked at have returned data from some or all of the columns within a given table. However, they have returned this information for every row in that table.

This may be acceptable if the table in question only contains a few records - if, perhaps, it is a lookup table containing country codes. But what if the table contains several hundred thousand or even several million records? It is simply not efficient - for the server, the database, the network or the end user (or their application) to process that amount of data.

The WHERE clause is used to restrict the data so that only what is actually required gets returned. This clause follows the FROM clause, and its basic construction is as follows:

```
WHERE  COLUMNNAME =  VALUE
```

Of course, "equals" is only one of many possible operators - you could just as easily use greater than (>), less than (<), not equal to (<>) or various others that we shall cover shortly. You can also combine several conditions by using AND, OR and NOT. Just remember that each condition must follow the same format - in other words, the column name must be repeated for each required value. Also bear in mind that ANDs get processed before ORs, unless you use brackets to indicate otherwise

Let's take a look at a few examples.

In order to do this...	**Issue this SQL statement...**

Select all customers in Mexico	`SELECT *`
	`FROM Customers`
	`WHERE Country = 'Mexico'`

Select all customers in the UK	`SELECT *`
	`FROM Customers`
	`WHERE Country = 'UK'`

Select all customers in the UK or Mexico	`SELECT *`
	`FROM Customers`
	`WHERE Country = 'UK' OR`
	`Country = 'Mexico'`

Select all orders with an Employee ID of 3 or 4	`SELECT *`
	`FROM Orders`
	`WHERE EmployeeID = 3 OR`
	`EmployeeID = 4`

Select all orders with an EmployeeID of 3 or 4 which also have a ship country of France	`SELECT *`
	`FROM Orders`
	`WHERE (EmployeeID = 3 OR EmployeeID = 4) AND ShipCountry = 'France'`

Select all orders placed in January 2007	`SELECT *`
	`FROM Orders`
	`WHERE OrderDate BETWEEN '01-Jan-1997' AND '31-Jan-1997'`

Select all orders placed in January 2007 (using ISO date format)	`SELECT *`
	`FROM Orders`
	`WHERE Orderdate >= '19970101' AND <= '19970131'`

Select all orders with a freight value over 100	`SELECT *`
	`FROM Orders`
	`WHERE Freight > 100`

Examples of the WHERE clause

Note that all strings must be put in quotes.

USING IN TO LOCATE MULTIPLE MATCHES

If you have several alternatives which need to be matched, there is an alternative to using multiple "OR" operators.

You may find it simpler and more elegant to use the IN operator, providing a list of the desired matches. This list should be comma separated, and should be within brackets. The syntax is as follows:

```
WHERE Country IN ('France', 'Germany', 'UK', 'Spain',
'Portugal')
```

This can also be used with numeric values, as follows:

```
WHERE ProductID IN (1,4,11,24,44,45,46,60,68)
```

USING WILDCARDS

So far, we have seen the use of the WHERE clause to look for specific matches, or specific ranges of matches. But what happens when you need to look for something less clearly defined. For example, what if you want to look for a customer, but you aren't sure how they spell their name? Or you want to look for products supplied by "ABC Industries", "ABC Supplies" and "ABC Products Ltd"?

It would certainly be possible to build WHERE clauses which use multiple ORs to select these matches, and to look for all possible spellings of your customer's name. But the use of wildcards (sometimes known as "Pattern Matching") makes this much simpler.

In case you haven't seen or used these before, a wildcard is a symbol which is used in place of one or more missing characters. If you want all customers whose company name begins with ABC, then you might use the syntax below:

```
WHERE CompanyName LIKE 'ABC%'
```

In this example, the % symbol is used to represent any group of zero or more characters. Thus, "ABC Industries", "ABC Supplies" and "ABC Products Ltd" would all be returned - as would a company simply called "ABC".

A different wildcard symbol is used when you want to search for a specific number of missing characters. For example, if you were searching for "The Smith Organisation", you might choose to replace the "S" in organi**s**ation with a wildcard, in order to ensure that you also returned "The Smith Organi**z**ation".

To achieve this, you would use the underscore symbol, indicating that you are searching for a single character, rather than zero or more characters, as in this example:

```
WHERE CompanyName LIKE 'The Smith Organi_ation'
```

NOTE: The wildcards mentioned in this article are used in TSQL (SQL Server) environments. If you are writing queries within MS Access, different wildcards apply. Within the Microsoft Access query grid, you should use an asterisk (*) to represent zero or more characters, and a question mark (?) to represent a single character. This picture is slightly complicated by the fact that if you are using ADO to programme MS Access, you should revert to using the TSQL wildcards as mentioned in this article!

USING [ ] WITH WILDCARDS

The use of square brackets around an expression allows for a wildcard to be restricted in the characters that are permitted. This may be useful when the alternative is a lot of OR operators.

As an example, say you were looking for any product name beginning with the letter A. It would be simple to write a WHERE clause like this:

```
WHERE ProductName like 'A%'
```

But what if the product name should be restricted to those beginning A, D, P, T and W? Using an OR for each of these would make the WHERE clause very cumbersome, so in these circumstances, square brackets would be more efficient:

```
WHERE ProductName LIKE '[ADPTW]%'
```

Note that in this syntax, unlike the IN clause, the characters within the square brackets are not comma separated.

This technique can also be extended to include a range of characters, such as in the situation where the product name should begin with any character from D to P:

```
WHERE ProductName LIKE '[D-P]%'
```

The logic of such a where clause can be reversed in two ways - firstly with the use of the word NOT preceding the LIKE operator, and secondly through the use of the ^ symbol within the square brackets to disallow the characters or range of characters provided:

```
WHERE ProductName NOT LIKE '[D-P]%'
WHERE ProductName LIKE '[^D-P]%'
```

The above where clauses are synonymous.

USING [ ] TO ESCAPE WILDCARD CHARACTERS

So far, we have seen how to use wildcard characters in order to search for a pattern. We see that % is used to specify any group of characters (or no character at all) and that underscore is used to specify any single character.

However, this poses a problem when we need to search for a company such as "The 5 * Burger Company" (in MS Access) or a discount field containing "75%" (in T-SQL). However, square brackets once again come to our rescue! In this instance, the character within the square brackets is "escaped" or stripped of its special meaning. So, we can now look for a discount of 75% with the following syntax:

```
WHERE Discount = '75[%]'
```

or, in our Access example,

```
WHERE CompanyName LIKE 'The 5 [*]*'
```

Note that in this example, the first asterisk is used literally, being within square brackets, whereas the second is being used as a wildcard, allowing us to search for any company whose name begins "The 5 *"

INTRODUCTION TO SQL FUNCTIONS AND ALIASES

Just as in any programming language, SQL offers a variety of functions which can be used to manipulate and process data.

These can be used within the SELECT list and within WHERE clause, as well as in various other places such as GROUP BY and HAVING clauses.

These function can be grouped into various categories, such as string, date, aggregate and mathematical functions, and over the next few pages we will look at each of these in turn.

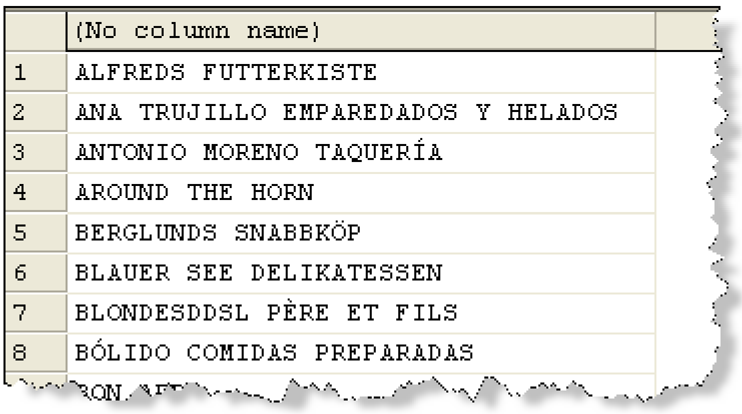
It should be noted that when one or more columns (or other pieces of data) are passed to a function, the resulting output from that function does not inherit the column names. Instead, SQL Server which will output the data without a column name. Microsoft Access will typically generate a name for the column such as "Expr1" (Expression 1).

This behaviour can be modified by using the AS keyword to create an alias for the column. Let's take a look at an example both with and without the use of aliases.

Suppose we wanted to return a list of company names from the customers table, but we wanted them in upper case. SQL provides a function to do exactly this - the UPPER function (more information on this and other functions for working with strings can be found in the next section). In order to achieve our objective we would write a query like this:

```
SELECT UPPER(CompanyName)
FROM Customers
```

This will return an output like this within SQL Server:

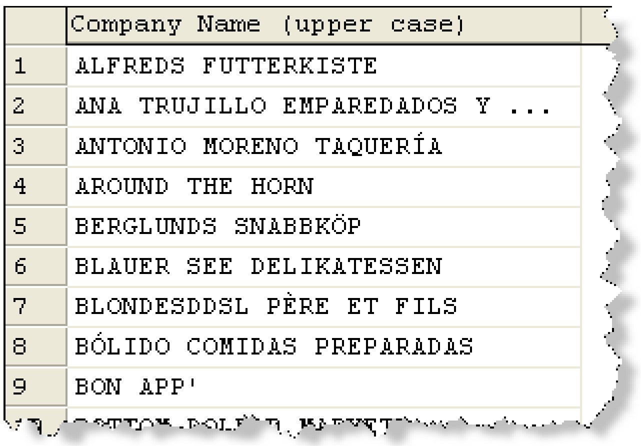


```
   (No column name)
1  ALFREDS FUTTERKISTE
2  ANA TRUJILLO EMPAREDADOS Y HELADOS
3  ANTONIO MORENO TAQUERÍA
4  AROUND THE HORN
5  BERGLUNDS SNABBKÖP
6  BLAUER SEE DELIKATESSEN
7  BLONDESDDSL PÈRE ET FILS
8  BÓLIDO COMIDAS PREPARADAS
```

However, if we want to create the same output, naming the column "Company Name (upper case)", we can modify our query as follows:

```
SELECT UPPER(CompanyName) AS 'Company Name (upper case)'
FROM Customers
```

which will return this output:



```
   Company Name (upper case)
1  ALFREDS FUTTERKISTE
2  ANA TRUJILLO EMPAREDADOS Y ...
3  ANTONIO MORENO TAQUERÍA
4  AROUND THE HORN
5  BERGLUNDS SNABBKÖP
6  BLAUER SEE DELIKATESSEN
7  BLONDESDDSL PÈRE ET FILS
8  BÓLIDO COMIDAS PREPARADAS
9  BON APP'
```

SQL STRING FUNCTIONS

On the pages that follow are some of the more common string functions, along with examples of the usage of each. For more information on these and other string functions, see the relevant page within MSDN.

CONCATENATION

In order to concatenate (join together) two or more strings, use the concatenation operator. In the ANSI SQL standard, || is used to concatenate fields, but this practice is not followed in either T-SQL or Microsoft Access. Both T-SQL and Access allow the use of the plus symbol (+) as a concatenation operator, and Access also allows the use of ampersand (&).

This statement...	Generates this result...
`SELECT LastName + FirstName`	NancyDavolio AndrewFuller ...ETC
`SELECT LastName + ' ' + FirstName`	Nancy Davolio Andrew Fuller
`SELECT LastName + ' (' + City + ' - ' + Country + ')'`	Davolio (Seattle - USA) Fuller (Tacoma - USA)
`SELECT TitleOfCourtesy & ' ' & LastName`	Ms. Davolio Dr. Fuller (ONLY IN MS ACCESS)

`SELECT FirstName`
`FROM Employees`
`WHERE TitleOfCourtesy + ' ' + LastName = 'Dr. Fuller'`
	Andrew

The concatenation operator. All examples use the Employees table

CHANGING THE CASE OF A STRING

There are two function in T-SQL for converting a string to upper or lower case. The UPPER and LOWER functions take a single string as their input and return the same string in upper or lower case respectively. Note that the use of UPPER and LOWER is restricted to T-SQL - to achieve the same result in MS Access, you need to use the UCase and LCase functions. Apart from the function names, these functions operate identically to their T-SQL equivalents.

This statement...	**Generates this result...**
`SELECT FirstName, UPPER(LastName)`	Nancy DAVOLIO
`SELECT LOWER(FirstName), LOWER(LastName)`	nancy davolio
`SELECT FirstName, UCase(LastName)`	Nancy DAVOLIO (IN MS ACCESS)
`SELECT LCase(FirstName), LCase(LastName)`	nancy davolio (IN MS ACCESS)

Converting the case of a string with UPPER, LOWER, UCase and LCase.
All examples use the Employees table

It is worth noting that these functions can be used in such a way as to instruct a database to ignore case (SQL Server can be configured at the Server, Database, Table and Column level to be case sensitive or case insensitive). If, for example, the Employees table is configured to be case sensitive, but you want to retrieve records for anyone with the last name "Fuller" - whether stored as "Fuller", "FULLER", "fuller" or even "FullER", then the following statement would do the job:

```
SELECT *
FROM Employees
WHERE UPPER(LastName) = 'FULLER'
```

SQL STRING FUNCTIONS

RETURNING A PORTION OF A STRING

The LEFT and RIGHT functions

The LEFT and RIGHT functions are similar in very many ways. Both take as their input parameters a string and a positive integer. The LEFT function then returns the number of characters specified from the beginning of the string, the RIGHT function returns the number of characters specified from the end of the string.

Examples of these functions follow:

This statement...	Generates this result...
`SELECT FirstName, LastName, LEFT(FirstName, 1) + LEFT(LastName,1)`	Nancy Davolio ND Andrew Fuller AF etc.
`SELECT FirstName, LastName, RIGHT(FirstName, 1) + RIGHT(LastName,1)`	Nancy Davolio yo Andrew Fuller wr

LEFT and RIGHT function examples. All examples use the Employees table.

THE SUBSTRING FUNCTION

The substring function is used when the portion of a string to be returned is not at either end of the string. For example, a person called Catherine Joanne Wright might have the driving license number WRIGH805216**CJ**9FV, in which the CJ (highlighted) represents her initials. In order to extract these initials from the driving license number, the SUBSTRING function would be used.

The SUBSTRING function takes three parameters. The first is the string (or column) to be processed. The second is a positive integer representing the first character to be returned from the string. The third is a positive integer representing the number of characters to be returned. Thus, to return the initials from the driving license as above, the following statement would be executed:

```
SELECT SUBSTRING('WRIGH805216CJ9FV',12,2)
```

Other examples of the output from this function follow. Note that where more characters are specified to return than actually exist in the string, no error is generated and all remaining characters are returned (as in the second example below).

This statement...	Generates this result...
`SELECT FirstName, LastName, SUBSTRING(FirstName, 2,3)`	Nancy Davolio anc Andrew Fuller dnr ETC.
`SELECT FirstName, LastName, SUBSTRING(LastName,5,3)`	Nancy Davolio lio Andrew Fuller er

SUBSTRING function examples. All examples use the Employees table.

SQL STRING FUNCTIONS

LOCATING PART OF A STRING

THE CHARINDEX FUNCTION

CHARINDEX is the function to use when you need to locate (or determine the existence of) one string within another. It can be used with either two parameters or three, depending upon your requirements.

If two parameters are supplied, the first is the string being sought. The second is the string (or column) within which to search. The return value will be the positive integer representing the starting point of the search string within the string being searched. If the string is not located, the return value is zero.

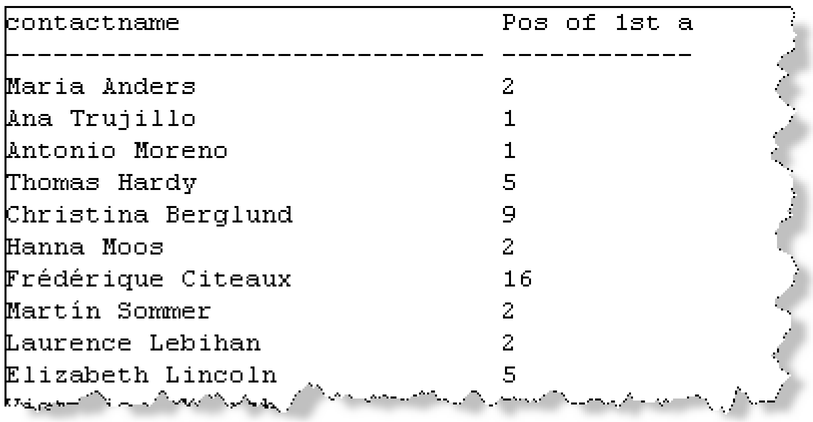
If a third parameter is supplied, this is a positive integer representing the character within the second string at which to begin the search for the first string. If this parameter is supplied, but is either a negative integer or a zero, the search begins at the start of the second string. Note that if this parameter is supplied, the return value still represents the location of the first string within the second STARTING FROM THE BEGINNING OF THE SECOND STRING. For example, if the third parameter is supplied as the number 4 and the function returns the value 6, this means that the first occurrence of the first string within the second is at the 6th character - NOT at the 10th character (ie the sixth after character 4).

Let's see some examples.

The SELECT statement:

```
SELECT contactname,
CHARINDEX('a',contactname) as 'Pos of 1st a'
FROM Customers
```

will generate the following output:

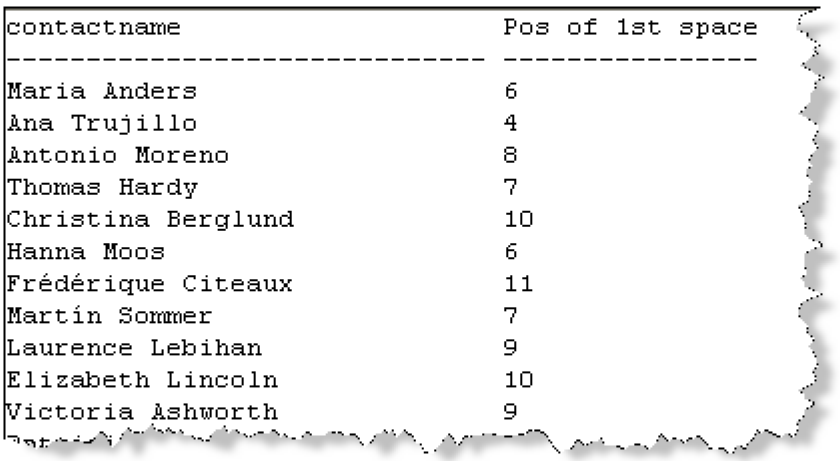


```
contactname                    Pos of 1st a
------------------------------ ------------
Maria Anders                   2
Ana Trujillo                   1
Antonio Moreno                 1
Thomas Hardy                   5
Christina Berglund             9
Hanna Moos                     2
Frédérique Citeaux             16
Martín Sommer                  2
Laurence Lebihan               2
Elizabeth Lincoln              5
```

The SELECT statement:

```
SELECT contactname,
CHARINDEX(' ',ContactName) as 'Pos of 1st space'
FROM Customers
```

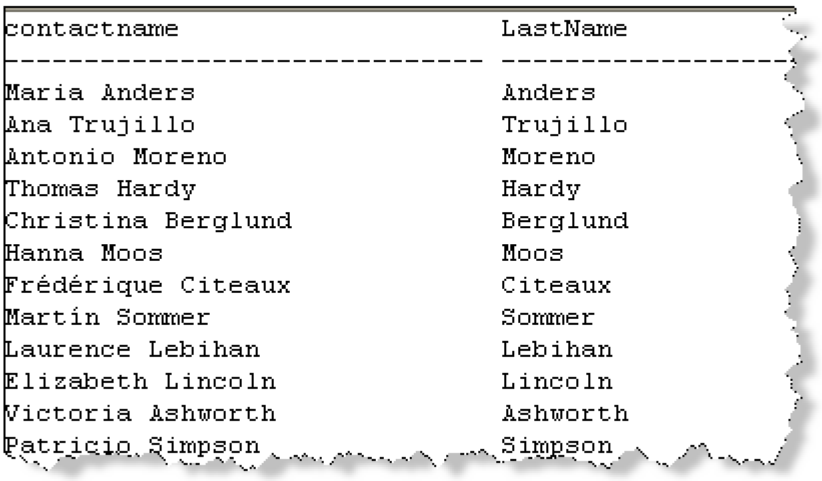
will generate the following output:



```
contactname                    Pos of 1st space
------------------------------ ----------------
Maria Anders                   6
Ana Trujillo                   4
Antonio Moreno                 8
Thomas Hardy                   7
Christina Berglund             10
Hanna Moos                     6
Frédérique Citeaux             11
Martín Sommer                  7
Laurence Lebihan               9
Elizabeth Lincoln              10
Victoria Ashworth              9
```

We can now combine CHARINDEX with SUBSTRING as discussed in the previous page in order to locate the Last Name of our customer contacts:

```
SELECT contactname,
SUBSTRING(ContactName,
CHARINDEX(' ',ContactName)+1,LEN(ContactName)) as 'LastName'
FROM Customers
```



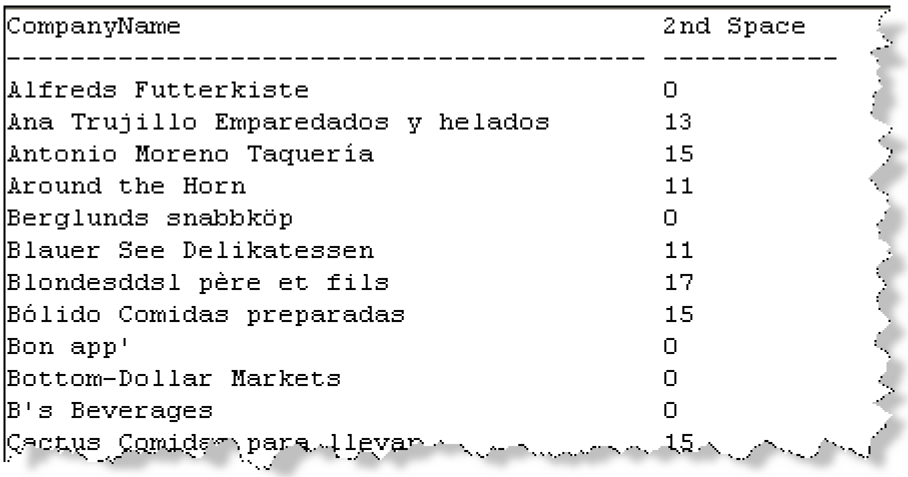
```
contactname                    LastName
------------------------------ ------------------
Maria Anders                   Anders
Ana Trujillo                   Trujillo
Antonio Moreno                 Moreno
Thomas Hardy                   Hardy
Christina Berglund             Berglund
Hanna Moos                     Moos
Frédérique Citeaux             Citeaux
Martín Sommer                  Sommer
Laurence Lebihan               Lebihan
Elizabeth Lincoln              Lincoln
Victoria Ashworth              Ashworth
Patricio Simpson               Simpson
```

Note that in this example, the second parameter of the SUBSTRING function is the result of the CHARINDEX function plus 1. This 1 is added so that the output begins with the character after the space rather than the space itself. Also note that the LEN function (not covered in these pages, but fairly self-explanatory - it simply returns the length of the given string) is used as the third parameter of SUBSTRING. This ensures that the entire last name is returned. It would be possible to use the LEN function and subtract a second call to the CHARINDEX function to determine the length of the last name. However, this adds unnecessarily to the complexity of our stantement, and no error is generated by asking SUBSTRING to return more characters than actually exist within a string.

A final example of CHARINDEX shows how it can be nested within another CHARINDEX function in order to locate the second instance of a string. Once again, you need to add 1 to the nested CHARINDEX in order to begin the search for the second instance at the character after the first instance:

```
SELECT CompanyName,
Charindex(' ', CompanyName, CHARINDEX(' ',CompanyName)+1) as
'2nd Space'
FROM Customers
```

This generates the result below:



```
CompanyName                                2nd Space
----------------------------------------- -----------
Alfreds Futterkiste                        0
Ana Trujillo Emparedados y helados         13
Antonio Moreno Taquería                    15
Around the Horn                            11
Berglunds snabbköp                         0
Blauer See Delikatessen                    11
Blondesddsl père et fils                   17
Bólido Comidas preparadas                  15
Bon app'                                   0
Bottom-Dollar Markets                      0
B's Beverages                              0
Cactus Comidas para llevar                 15
```

Note that a zero value is returned where there are less than two spaces within a company name.

SQL DATE FUNCTIONS

GETDATE, DAY, MONTH AND YEAR

There are a number of functions within SQL for processing dates. These include functions for determining the difference between two dates, for returning information about a date and for adding a value to a date to return a new date.

In this section, we will take a look at some of the basic date functions - GETDATE, DAY, WEEKDAY, MONTH and YEAR. Over the next pages we will look at DATEPART, DATENAME, DATEDIFF and DATEADD.

GETDATE

GETDATE is an unusual function in the sense that it does not process a given date, or indeed need any other input parameter. It simply returns the current system date and time. Its syntax is simple:

```
SELECT GETDATE()
```

Note that the use of the brackets to indicate the use of a function is required - an error will be returned if the brackets are omitted. The value returned is of the DATETIME data type and will include the date and the time.

GETDATE is frequently used when comparing a given date to the current date, for example in determining how long ago an order date was, or how far in the future a shipment date is. For more information on accomplishing these tasks, see the section on DATEDIFF.

DAY, MONTH AND YEAR

These functions are similar in their functionality. Each takes a single input parameter and returns an integer.

DAY returns a value between 1 and 31 indicating the day of the month for the given date.

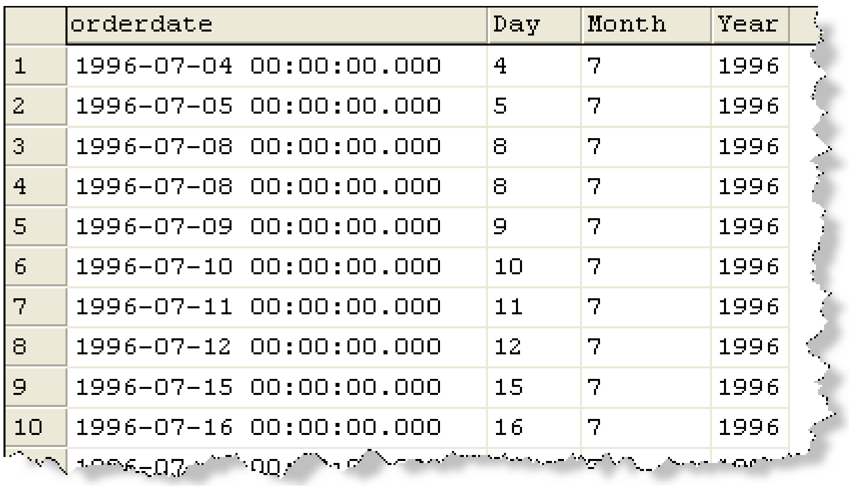
MONTH returns a value between 1 and 12 indicating the month of a given date.

YEAR returns the year portion of the given date.

The following code allows us to see the output of each of these functions:

```
SELECT orderdate,
DAY(orderdate) as 'Day',
MONTH(orderdate) as 'Month',
YEAR(orderdate) as 'Year'
FROM orders
```

This returns the following:



	orderdate	Day	Month	Year
1	1996-07-04 00:00:00.000	4	7	1996
2	1996-07-05 00:00:00.000	5	7	1996
3	1996-07-08 00:00:00.000	8	7	1996
4	1996-07-08 00:00:00.000	8	7	1996
5	1996-07-09 00:00:00.000	9	7	1996
6	1996-07-10 00:00:00.000	10	7	1996
7	1996-07-11 00:00:00.000	11	7	1996
8	1996-07-12 00:00:00.000	12	7	1996
9	1996-07-15 00:00:00.000	15	7	1996
10	1996-07-16 00:00:00.000	16	7	1996

In the next section, we will see the use of DATEPART and DATENAME. DATEPART is able to return values which are synonymous with DAY, MONTH and YEAR, but as we shall see, it also has additional capabilities.

SQL DATE FUNCTIONS

DATEPART AND DATENAME

Note: These functions do not exist in Access (JET SQL) .

The **datename** and **datepart** functions are used within T-SQL for extracting information about a date, in much the same way as the date functions discussed in the previous section. They are, however, slightly more complex, in that unlike DATE or MONTH or YEAR they require more than one parameter.

Both functions are similar in their construction - they take parameters indicating the type of information required and the date to be analysed. The difference between them is that DATENAME returns a string indicating a portion of a date - such as 'Monday' or 'March' whereas DATEPART returns an integer, such as 2 for Monday or 3 for March. In many instances (for example when returning the year) both functions will return a number. However, in all cases DATEPART returns an integer, whereas DATENAME returns a string - specifically the T-SQL nvarchar data type.

Perhaps the hardest part is remembering the values for the first parameter - that which indicates which portion of the date to return. The table below (taken from SQL Server help) gives this information, and is followed by some examples of using both DATEPART and DATENAME.

Datepart	Abbreviations
Year	yy, yyyy
Quarter	qq, q
Month	mm, m
Dayofyear	dy, y
Day	dd, d
Week	wk, ww

Weekday	dw
Hour	hh
Minute	mi, n
Second	ss, s
Millisecond	ms

Values for the first argument of DATENAME and DATEPART.

So, given that GETDATE() returns the current date and time, which at the time of writing is Thursday, 22 May 2011 at 07:59:23.347, let's take a look at the output from these functions. In the first column in the table below is the value for the first argument of either function, which may be replaced by any of the abbreviations in the table above. The second column displays the return value from the DATEPART function, which would be written as follows (for the first example):

```
SELECT DATEPART(year, GETDATE())
```

The third column displays the return value from the DATENAME function, which would be written as follows, again for the first example:

```
SELECT DATENAME(year, GETDATE())
```

First argument	DATEPART return	DATENAME return
year	2011	2011
quarter	2	2
month	5	May
dayofyear	143	143

day	22	22
week	21	21
weekday	5	Thursday
hour	7	7
minute	59	59
second	23	23
millisecond	347	347

Return values from DATEPART and DATENAME

USES OF DATEPART AND DATENAME

Given that in many cases DATEPART and DATENAME seem to return identical values, when is it necessary to use one function rather than the other?

Firstly, there are the obvious cases of using **month** and **weekday** as the first parameter - if the name of the day or month is required, then obviously you would use DATENAME. However, this can pose problems when the output needs to be sorted by month - if an ORDER BY clause is followed by the DATENAME function, the output will be sorted in alphabetical order, making April the first month, followed by August, then December and so on. In this instance, it is often necessary to include the DATENAME function in the select list, but use the DATEPART function in the ORDER BY clause to ensure proper sorting.

Beyond that, the key lies in the data type being returned. As mentioned above, the return value from DATEPART is always **int** whereas from DATENAME it is always **nvarchar**. There will be occasions when, for example, creating stored procedures (which is beyond the scope of this article, but may be a topic for a future article) when you want to return a numeric value rather than a varchar, or vice versa. In such cases the use of the appropriate function here will negate the need for use of the CONVERT function to convert data from one type to another.

MONTH(DATE) or DATEPART(M, DATE) ?

In the previous section, we discussed the MONTH, DAY and YEAR functions. So, which should you use - DATEPART with an appropriate first parameter, or MONTH, DAY and YEAR? In short, it doesn't matter - the two functions are synonymous when returning integers representing the current day of the month, month number or year.

Do bear in mind, however, that DATENAME and DATEPART do not exist in Microsoft Access. Therefore if you want to write a script which will work in both Access and SQL Server, you will need to use DAY, MONTH and YEAR in preference to DATEPART.

DATEDIFF AND DATEADD

DATEDIFF and DATEADD both work on the basis of determining the difference between two dates in a given unit of time. With DATEDIFF, you provide the two dates and the return value will be the difference in the unit you specify; with DATEADD you provide the first date and the difference, and the return value will be the second date.

DATEDIFF

The DATEDIFF function determines the difference (the 'interval') between two date values. The function takes three arguments, as follows:

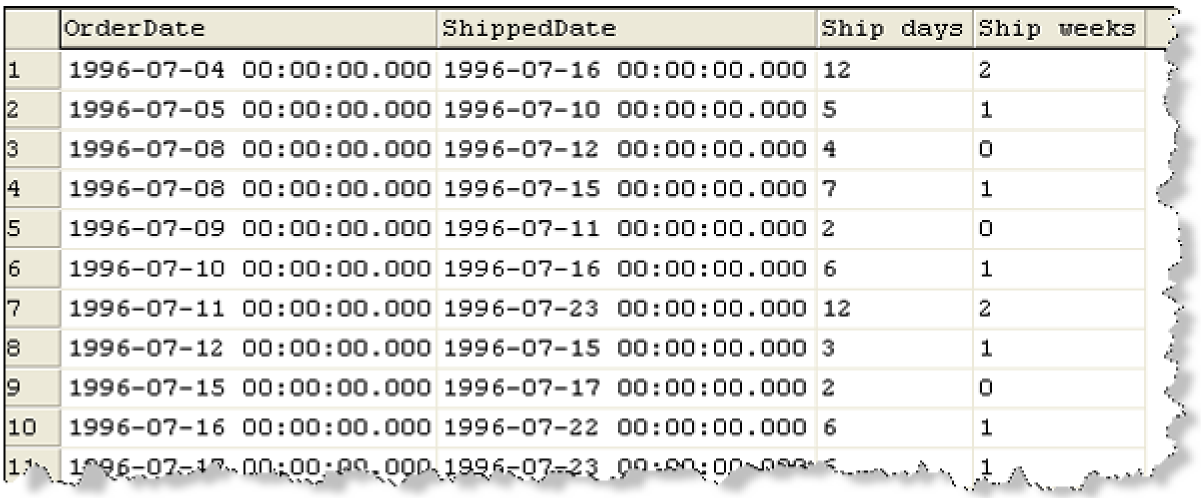
```
DATEDIFF(interval, date1, date2)
```

The first parameter is any valid datepart value, as listed in the section on DATENAME and DATPART. This determines the unit in which the difference should be calculated. The second and third parameters are the date values to be compared. If date1 is before date2, the result will be a positive value, if date1 is after date2 the result will be a negative value.

Let's take a look at an example of the output for various parameter values. In the following script, the first two columns extract the OrderDate and the ShippedDate values from the Orders table. The third column expresses the difference between these two dates in days, the fourth expresses it in weeks.

```
SELECT OrderDate, ShippedDate,
DATEDIFF(d,OrderDate, ShippedDate) as 'Ship days',
DATEDIFF(wk, OrderDate, ShippedDate) as 'Ship weeks'
FROM Orders
```

Here's the output from this query:

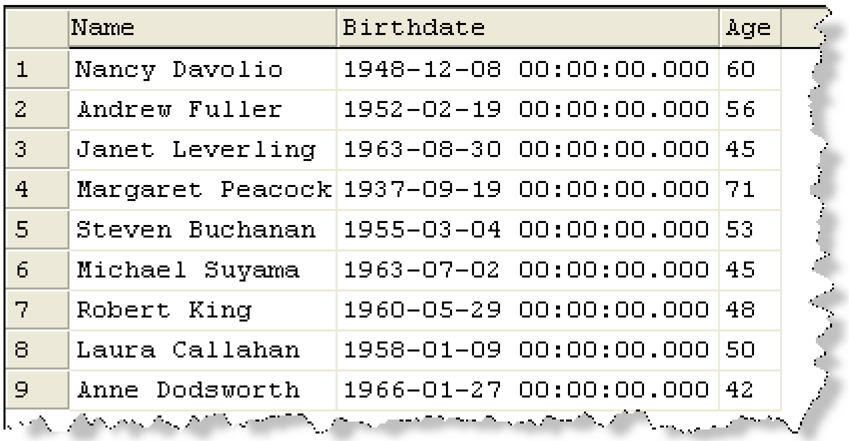


	OrderDate	ShippedDate	Ship days	Ship weeks
1	1996-07-04 00:00:00.000	1996-07-16 00:00:00.000	12	2
2	1996-07-05 00:00:00.000	1996-07-10 00:00:00.000	5	1
3	1996-07-08 00:00:00.000	1996-07-12 00:00:00.000	4	0
4	1996-07-08 00:00:00.000	1996-07-15 00:00:00.000	7	1
5	1996-07-09 00:00:00.000	1996-07-11 00:00:00.000	2	0
6	1996-07-10 00:00:00.000	1996-07-16 00:00:00.000	6	1
7	1996-07-11 00:00:00.000	1996-07-23 00:00:00.000	12	2
8	1996-07-12 00:00:00.000	1996-07-15 00:00:00.000	3	1
9	1996-07-15 00:00:00.000	1996-07-17 00:00:00.000	2	0
10	1996-07-16 00:00:00.000	1996-07-22 00:00:00.000	6	1
11	1996-07-17 00:00:00.000	1996-07-23 00:00:00.000	6	1

In the next example, we are calculating the age of each of the employees listed in the Employees table. We are using DATEDIFF to calculate the difference in years between the BirthDate field and the current date as returned by the GETDATE function. Heres the SQL statement:

```
SELECT FirstName +' ' +LastName as 'Name',
Birthdate,
DATEDIFF(yy, Birthdate, GETDATE()) as 'Age'
FROM Employees
```

And here's the output:



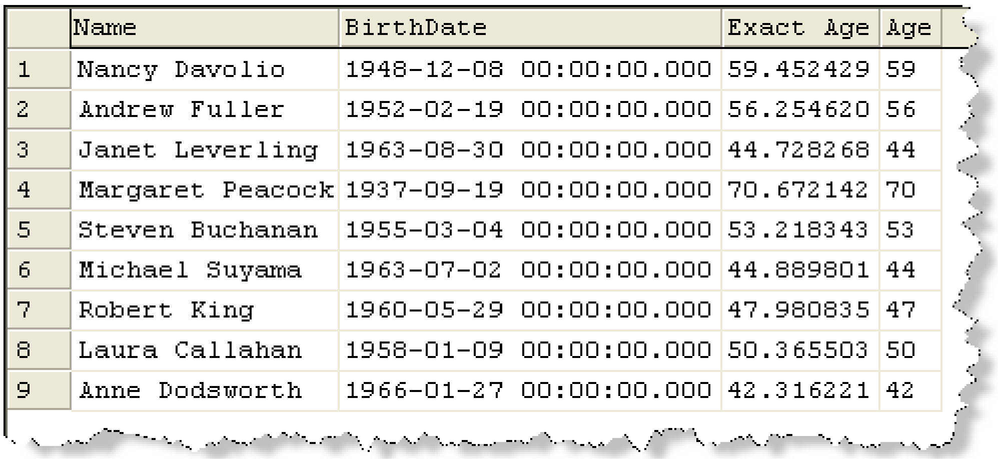
	Name	Birthdate	Age
1	Nancy Davolio	1948-12-08 00:00:00.000	60
2	Andrew Fuller	1952-02-19 00:00:00.000	56
3	Janet Leverling	1963-08-30 00:00:00.000	45
4	Margaret Peacock	1937-09-19 00:00:00.000	71
5	Steven Buchanan	1955-03-04 00:00:00.000	53
6	Michael Suyama	1963-07-02 00:00:00.000	45
7	Robert King	1960-05-29 00:00:00.000	48
8	Laura Callahan	1958-01-09 00:00:00.000	50
9	Anne Dodsworth	1966-01-27 00:00:00.000	42

If we look carefully at this output, there appear to be some anomalies. The date on which this query was run was 21 May 2008... and yet Nancy Davolio is calculated as being 60, despite the fact that her birthday is not until December. Why is this? It's certainly not rounding, as she's actually closer to her 59th birthday than her 60th.

In fact DATEDIFF calculates *how many boundaries have been crossed* of the specified type. So, given that we're looking for years in this example, the first boundary in the first row is crossed on December 31 1947 - only 23 days after Nancy's date of birth. And we've crossed another one each year, including one on 31 December 2007 - the 60th boundary to be crossed in the 59 years of her birth.

You may therefore find it to be a more accurate measure to use **dd** as the interval parameter, and then divide the result by 365.25 in order to determine the number of years. This will result in a decimal being returned, so the FLOOR function may be used to round this value down to the nearest whole number, as follows:

```
SELECT FIRSTNAME + ' ' + LASTNAME as 'Name',
BIRTHDATE,
DATEDIFF(dd,BIRTHDATE,GETDATE())/365.25 as 'Exact Age',
FLOOR(DATE(dd,DATEDIFF,DATEDIFF())/365.25) as 'Age'
FROM Employees
```



	Name	BirthDate	Exact Age	Age
1	Nancy Davolio	1948-12-08 00:00:00.000	59.452429	59
2	Andrew Fuller	1952-02-19 00:00:00.000	56.254620	56
3	Janet Leverling	1963-08-30 00:00:00.000	44.728268	44
4	Margaret Peacock	1937-09-19 00:00:00.000	70.672142	70
5	Steven Buchanan	1955-03-04 00:00:00.000	53.218343	53
6	Michael Suyama	1963-07-02 00:00:00.000	44.889801	44
7	Robert King	1960-05-29 00:00:00.000	47.980835	47
8	Laura Callahan	1958-01-09 00:00:00.000	50.365503	50
9	Anne Dodsworth	1966-01-27 00:00:00.000	42.316221	42

DATEADD

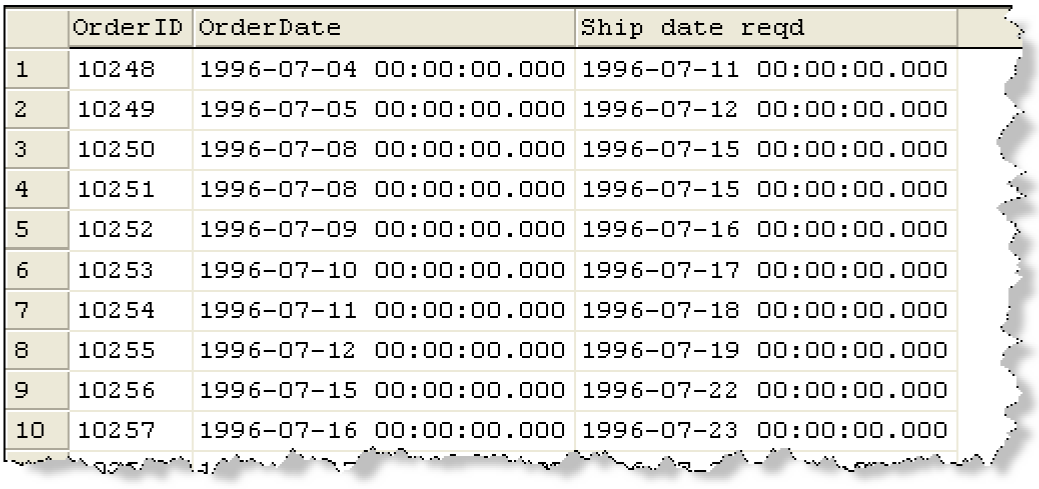
DATEADD is used when an interval is to be added to a given date in order to return a second date. The values allowed for the interval argument are identical to those for the DATEDIFF function. The full syntax is as follows:

```
DATEADD(Interval, Number, Date)
```

As an example, consider a requirement that all orders are shipped within seven days. The following statement shows the date seven days after the order date for each order in the orders table:

```
SELECT OrderID, OrderDate,
DATEADD(dd,7,Orderdate) as 'Ship date reqd'
FROM Orders
```

This generates the following output:



	OrderID	OrderDate	Ship date reqd
1	10248	1996-07-04 00:00:00.000	1996-07-11 00:00:00.000
2	10249	1996-07-05 00:00:00.000	1996-07-12 00:00:00.000
3	10250	1996-07-08 00:00:00.000	1996-07-15 00:00:00.000
4	10251	1996-07-08 00:00:00.000	1996-07-15 00:00:00.000
5	10252	1996-07-09 00:00:00.000	1996-07-16 00:00:00.000
6	10253	1996-07-10 00:00:00.000	1996-07-17 00:00:00.000
7	10254	1996-07-11 00:00:00.000	1996-07-18 00:00:00.000
8	10255	1996-07-12 00:00:00.000	1996-07-19 00:00:00.000
9	10256	1996-07-15 00:00:00.000	1996-07-22 00:00:00.000
10	10257	1996-07-16 00:00:00.000	1996-07-23 00:00:00.000

SQL AGGREGATE FUNCTIONS

SQL supports a number of aggregate functions, all of which work in broadly similar ways. In this section, we shall consider some of the more common aggregate functions - SUM, MAX, MIN, AVG and COUNT.

All except COUNT take as their expression a single value, which is typically the name of a column and must be numeric. The return value of the data type varies depending on the data type of the expression being evaluated - for example if SUM is passed a column of INT type, the return value will be an INT; if MAX is passed a column of MONEY type, the return value will also be MONEY.

COUNT works slightly differently, in that its input expression need not be numeric. COUNT returns a count of the items in the expression passed - for example the number of values in a LastName column - and therefore these items need not be numeric. Furthermore, all these functions other than COUNT ignore null values, whereas COUNT may not, depending on how it is used. Let's take a look in a little more detail.

SUM

Returns the total of the non-null values in its expression. For example, SUM(UnitPrice) returns the total of all non-null values in the UnitPrice column.

MAX

Returns the largest of the non-null values in its expression. For example, MAX(UnitPrice) returns the largest non-null value in the UnitPrice column.

MIN

Returns the smallest of the non-null values in its expression. For example, MIN(UnitPrice) returns the smallest non-null value in the UnitPrice column.

AVG

Returns the mean average of the non-null values in its expression. For example, AVG(UnitPrice) returns the mean average of all non-null values in the UnitPrice column.

COUNT(expression)

Returns the number of non-null values in its expression. For example, COUNT(UnitPrice) returns the number of non-null values in the UnitPrice column.

COUNT(*)

Returns the number of rows in a given table. Rather than counting values in a particular field as in COUNT(expression) this will return the number of rows in a table, and therefore is not affected by the presence of nulls in any particular field.

In all these cases, the expression can be modified with the use of ALL or DISTINCT. The use of ALL is not required as it is the default. DISTINCT is used to specify that the aggregate function should ignore duplicate values in the expression being evaluated.

The use of these functions is restricted to

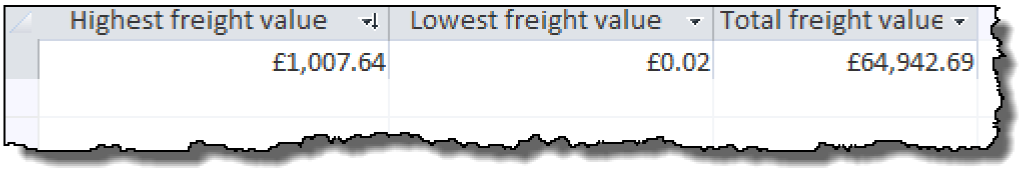
- SELECT lists (see example below)
- HAVING clauses
- COMPUTE clauses

Note, therefore, that aggregate functions may NOT be used in a where clause. For more information on restricting values by comparison with an aggregate function, see the section. on HAVING and GROUP BY.

The following query and output demonstrates the use of these aggregate functions:

```
SELECT MAX(freight) as 'Highest freight value',
MIN(freight) as 'Lowest freight value',
SUM(freight) as 'Total freight value'
FROM Orders
```

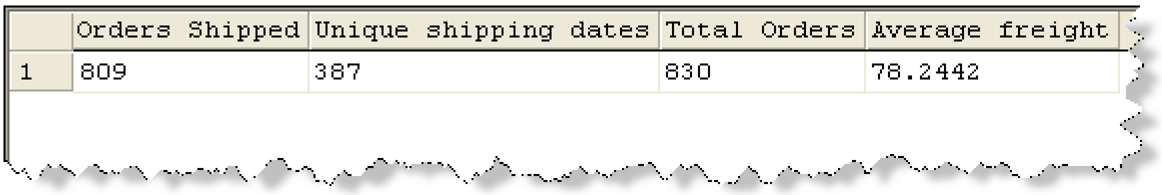
generates the following output:



Highest freight value	Lowest freight value	Total freight value
£1,007.64	£0.02	£64,942.69

```
SELECT COUNT(ShippedDate) as 'Orders Shipped',
COUNT(Distinct ShippedDate) as 'Unique shipping dates',
COUNT(*) as 'Total Orders',
AVG(freight) as 'Average freight'
FROM orders
```

generates the folllowing output:



	Orders Shipped	Unique shipping dates	Total Orders	Average freight
1	809	387	830	78.2442

THE GROUP BY CLAUSE

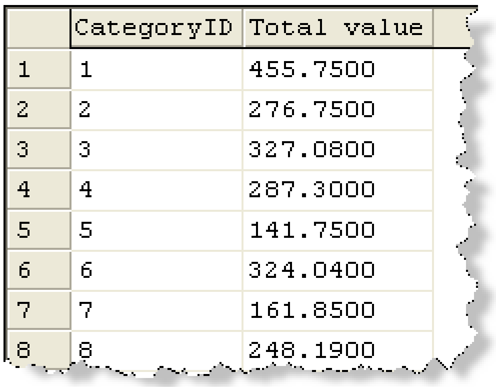
The GROUP BY clause is used when an aggregate function is used within a select statement to summarise data across rows. Data which required either in a SELECT list, or within an ORDER BY clause, but is not being summarised through the use of aggregate functions must be included in the GROUP BY clause.

The GROUP BY clause should follow the FROM and WHERE clauses, and precede any HAVING and ORDER BY clauses.

Suppose, for example, a query is required which will show the total unit price for all products in each category in the Products table. In order to achieve this, the following query could be written:

```
SELECT CategoryID, SUM(UnitPrice) as 'Total value'
FROM Products
GROUP BY CategoryID
```

This would result in the following output:



	CategoryID	Total value
1	1	455.7500
2	2	276.7500
3	3	327.0800
4	4	287.3000
5	5	141.7500
6	6	324.0400
7	7	161.8500
8	8	248.1900

It should be noted that this groups all products together by category, before producing a total value for the sum of their unit prices. If other columns were included in the select list, they would also have to be included in the GROUP BY clause (unless they were also calculated as part of an aggregate function), which might result in less of a grouping.

For example, suppose the statement was modified as follows:

```
SELECT CategoryID, SUPPLIERID, SUM(UnitPrice) as 'Total val'
FROM Products
GROUP BY CategoryID, SUPPLIERID
```

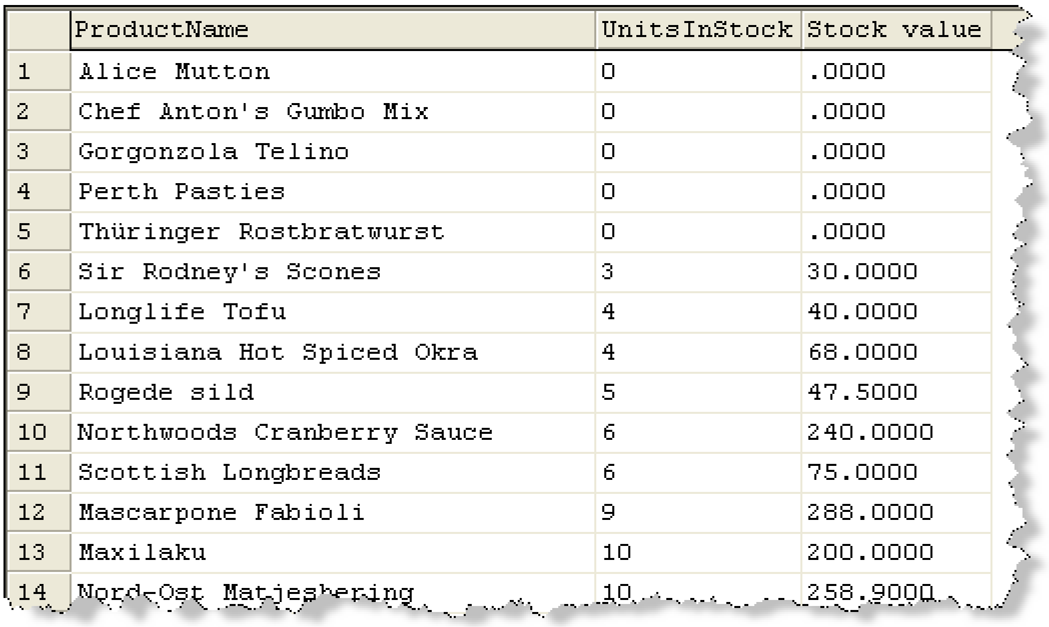
This would result in a total unit price per category per supplier, which might not be what was required (it would, in this instance, result in 49 rows of data, rather than the original 8).

GROUP BY AND CALCULATED FIELDS

There are two ways in which GROUP BY can be used with reference to calculated fields. Firstly, the calculated fields themselves can be used in an aggregate function, with GROUP BY being used to determine the scope of the records to be grouped, as follows:

```
SELECT ProductName, UnitsInStock,
SUM(UnitsInStock * UnitPrice) as 'Stock value'
FROM Products
GROUP BY ProductName, UnitsInStock
```

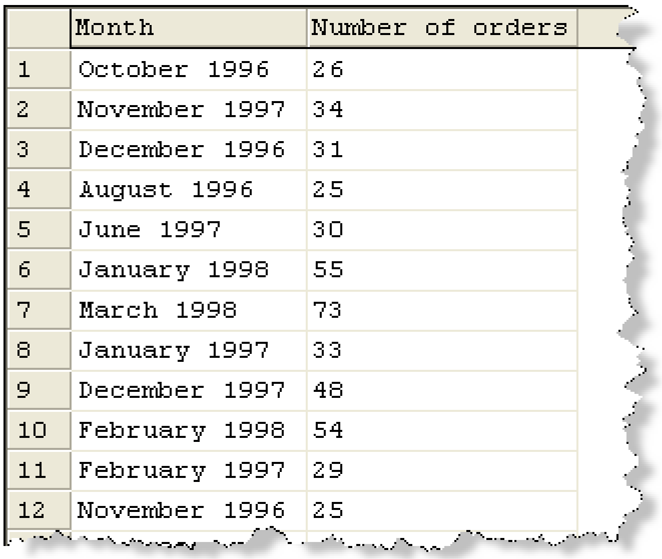
This results in the following output:



	ProductName	UnitsInStock	Stock value
1	Alice Mutton	0	.0000
2	Chef Anton's Gumbo Mix	0	.0000
3	Gorgonzola Telino	0	.0000
4	Perth Pasties	0	.0000
5	Thüringer Rostbratwurst	0	.0000
6	Sir Rodney's Scones	3	30.0000
7	Longlife Tofu	4	40.0000
8	Louisiana Hot Spiced Okra	4	68.0000
9	Rogede sild	5	47.5000
10	Northwoods Cranberry Sauce	6	240.0000
11	Scottish Longbreads	6	75.0000
12	Mascarpone Fabioli	9	288.0000
13	Maxilaku	10	200.0000
14	Nord-Ost Matjeshering	10	258.9000

Secondly, calculated fields themselves may be included in GROUP BY clauses, as in the following example, which shows the number of orders placed per month:

```
SELECT DATENAME(month,Orderdate) + ' '
 + DATENAME(year,orderdate) AS 'Month',
COUNT(*) as 'Number of orders'
FROM Orders
GROUP BY DATENAME(month,Orderdate) + ' '
 + DATENAME(year,orderdate)
```

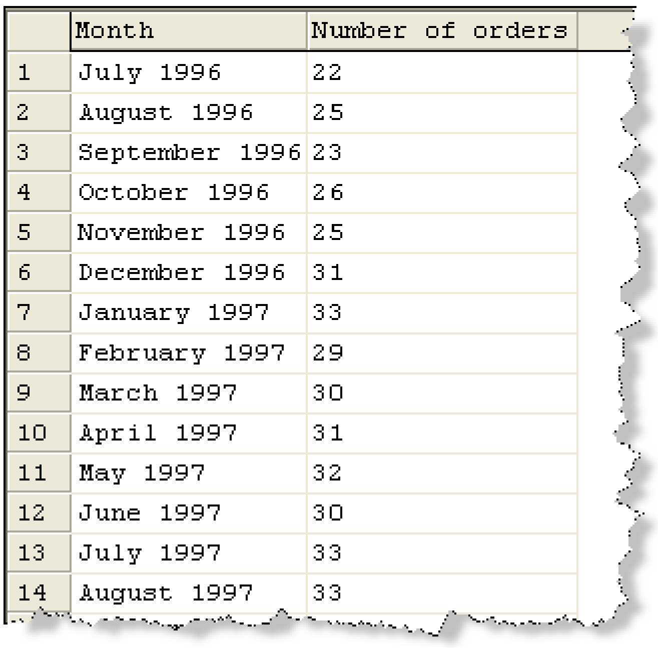


	Month	Number of orders
1	October 1996	26
2	November 1997	34
3	December 1996	31
4	August 1996	25
5	June 1997	30
6	January 1998	55
7	March 1998	73
8	January 1997	33
9	December 1997	48
10	February 1998	54
11	February 1997	29
12	November 1996	25

You will notice that this example returns accurate data, but the information is probably not ordered as desired. Therefore, we need to include an ORDER BY clause in order to order the output more appropriately. Remember - if a column is included in a SELECT list or an ORDER BY clause but is not part of an aggregate function, it must be included in the GROUP BY clause.

```
SELECT DATENAME(month,Orderdate) + ' '
+ DATENAME(year,orderdate),
COUNT(*) as 'Number of orders'
FROM Orders
GROUP BY DATENAME(month,Orderdate) + ' '
+ DATENAME(year,orderdate),
year(orderdate), month(orderdate)
ORDER BY year(orderdate), month(orderdate)
```

The output from this query is more usefully sorted:



	Month	Number of orders
1	July 1996	22
2	August 1996	25
3	September 1996	23
4	October 1996	26
5	November 1996	25
6	December 1996	31
7	January 1997	33
8	February 1997	29
9	March 1997	30
10	April 1997	31
11	May 1997	32
12	June 1997	30
13	July 1997	33
14	August 1997	33

THE HAVING CLAUSE

Just as the WHERE clause is used to restrict the rows to be returned based on values in one or more individual columns, HAVING is used to restrict the rows returned, based on the value of an aggregate function calculated over a group.

Let's just take a step back and remind ourselves of the WHERE clause. The following query will only return CustomerIDs, OrderIDs and Freight values for all orders where the freight value is over 100:

```
SELECT CustomerID, OrderID, Freight
FROM Orders
WHERE Freight > 100
```

Suppose we want to view total freight values for each customer, rather than information for each order. We could modify the query as follows:

```
SELECT CustomerID, SUM(Freight)
FROM Orders
WHERE Freight > 100
GROUP BY CustomerID
```

Although this now shows the total freight value per customer, the individual records being included in the calculation are still those where the freight value is over 100. So, if a customer placed three orders, with freight values of 50, 100 and 150, the total freight value would show as 150 - the sum of freight value of all orders where the freight value is over 100. Because the first two orders do not have a freight value over 100, they are discarded from the query before the aggregate function is calculated.

Of course, this type of "row level" filtering is often exactly what is required. Perhaps you want to filter the results to show total order values where the customer's city is Berlin, or the EmployeeID for the order is number 4.

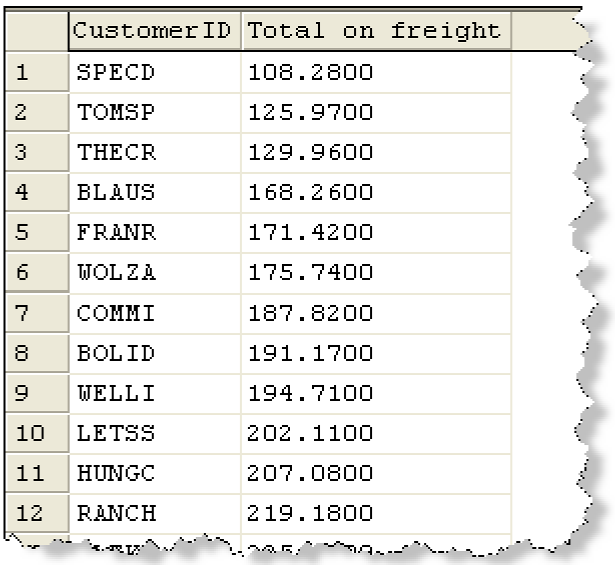
But what if you actually wanted to filter the results so that they only show those customers whose orders have had a TOTAL freight value over 100? The results for this would be rather different - all orders should now be included in the calculation, rather than just those with an individual freight value over 100. But the only rows returned will be those with a SUM calculation resulting in a value over 100.

This is where the HAVING clause comes in. HAVING filters results based on the results of an aggregate function. The HAVING clause immediately follows the GROUP BY clause, and precedes any ORDER BY clause. It includes the aggregate function or calculation being used as the filter and the expression to be matched.

Here's the query to return all customers whose orders have a total freight value over 100, ordered by the second column so that the accuracy of the results is more clearly visible:

```
SELECT CustomerID, SUM(Freight) AS 'Total on freight'
FROM Orders
GROUP BY CustomerID
HAVING SUM(Freight) > 100
ORDER BY 2
```

Here's the output of this query:



	CustomerID	Total on freight
1	SPECD	108.2800
2	TOMSP	125.9700
3	THECR	129.9600
4	BLAUS	168.2600
5	FRANR	171.4200
6	WOLZA	175.7400
7	COMMI	187.8200
8	BOLID	191.1700
9	WELLI	194.7100
10	LETSS	202.1100
11	HUNGC	207.0800
12	RANCH	219.1800

THE COMPUTE CLAUSE

Note: This clause is not available in Microsoft Access (JET SQL).

The COMPUTE clause is like a combination of a simple SELECT statement which selects columns and rows of data as specified, and a GROUP BY clause which uses aggregates to produce a summary, all rolled up into one!

Let's say that you wanted to know the total value of the orders in our Order Details table. You could write the following:

```
SELECT SUM(Quantity * UnitPrice)
FROM [Order Details]
```

However, this will show the total only - not the rows of data that make up the total.

The COMPUTE clause allows you to specify a column list within your select statement, and then a calculation to be shown at the end of the returned rows. The syntax is as follows:

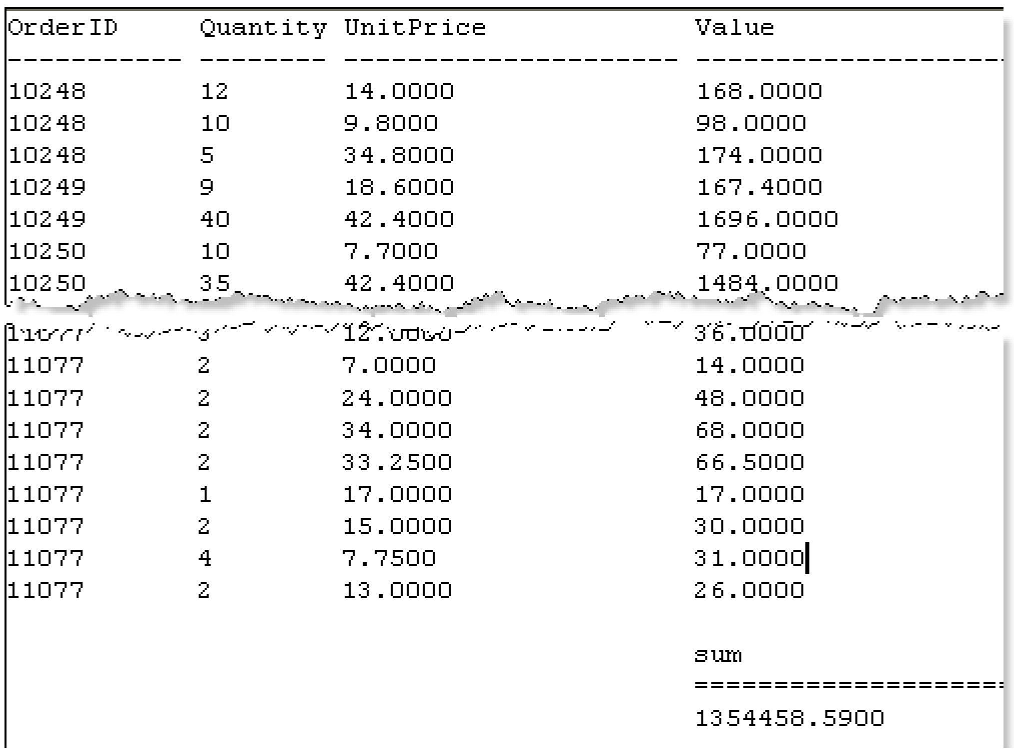
```
COMPUTE
{ { AVG | COUNT | MAX | MIN | STDEV | STDEVP
| VAR | VARP | SUM }
( expression ) } [ ,...n ]
[ BY expression [ ,...n ] ]
```

You will see that the various aggregate functions are shown here, followed by an "expression" to be calculated. (The use of "BY" is covered below)

So, to see this in action, let's amend the above query to show the individual order information, and also a grand total value for our orders:

```
SELECT OrderID, Quantity, UnitPrice, Quantity * UnitPrice as
'Value'
FROM [Order Details]
COMPUTE SUM(Quantity * UnitPrice)
```

The output from this query is shown below - note that the output for this query was switched from "Grid" to "Text" to make the results easier to see:



```
OrderID     Quantity UnitPrice             Value
----------- -------- --------------------- --------------------
10248       12       14.0000               168.0000
10248       10       9.8000                98.0000
10248       5        34.8000               174.0000
10249       9        18.6000               167.4000
10249       40       42.4000               1696.0000
10250       10       7.7000                77.0000
10250       35       42.4000               1484.0000
11077       3        12.0000               36.0000
11077       2        7.0000                14.0000
11077       2        24.0000               48.0000
11077       2        34.0000               68.0000
11077       2        33.2500               66.5000
11077       1        17.0000               17.0000
11077       2        15.0000               30.0000
11077       4        7.7500                31.0000
11077       2        13.0000               26.0000

                                           sum
                                           ====================
                                           1354458.5900
```

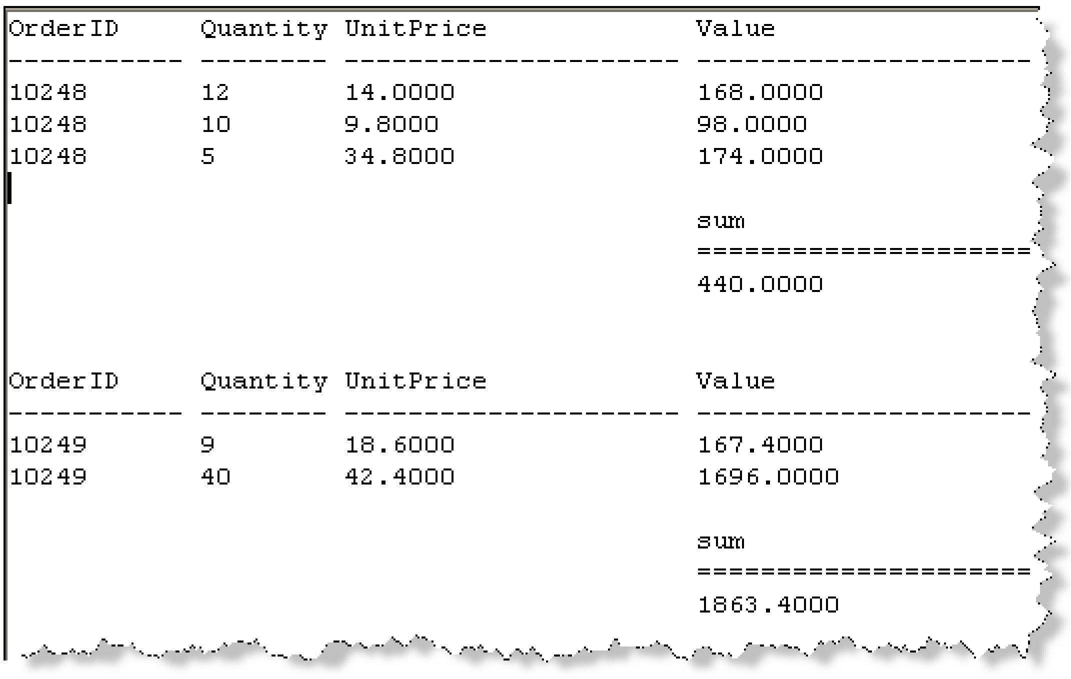
This can be very useful. But suppose we want to see the total PER ORDER rather than over all? This is where the COMPUTE BY clause comes into play.

The first thing to note is that if you want to get summary information for information in a particular column, the ORDER BY clause must be used to order the output by the information to be summarised with the COMPUTE BY clause. So, in our case, we must include ORDER BY OrderID in our query. The same column is then included after the BY keyword to specify that the aggregate function should be calculated over groupings in this column.

The full query to return the order information, summarised by each order is as follows:

```
SELECT OrderID, Quantity, UnitPrice, Quantity * UnitPrice as
'Value'
FROM [Order Details]
ORDER BY OrderID
COMPUTE SUM(Quantity * UnitPrice) BY OrderID
```

The output from this query is as follows:



```
OrderID     Quantity UnitPrice             Value
----------- -------- --------------------- ---------------------
10248       12       14.0000               168.0000
10248       10       9.8000                98.0000
10248       5        34.8000               174.0000

                                           sum
                                           =====================
                                           440.0000


OrderID     Quantity UnitPrice             Value
----------- -------- --------------------- ---------------------
10249       9        18.6000               167.4000
10249       40       42.4000               1696.0000

                                           sum
                                           =====================
                                           1863.4000
```

JOINING TABLES IN A QUERY

There will frequently be times when the information you need to retrieve is not all located in one table. Suppose, for example, you want a list of company names, and the dates of the orders each company has placed. In the Northwind database, the CompanyName field is in the Customers table, and the OrderDate field is in the Orders table. If you want to include the names and quantities of those products ordered, this involves two more tables - Order Details and Products.

So far, the queries we've dealt with all extract their data from a single table, but in this situation, this won't be enough. Over the next couple of pages, we'll take a look at drawing information from two or more tables by using the JOIN clause.

Let's start by taking a look at the first example mentioned above - a requirement to show every company name and the order dates for their orders.

The basic select statement is fairly straightforward in this example - you simply list the field names required:

```
SELECT CompanyName, OrderDate
```

However, the data comes from the Customers table and from the Orders table, which are related. The next step is to reflect that relationship in the FROM clause:

```
FROM Customers JOIN Orders
```

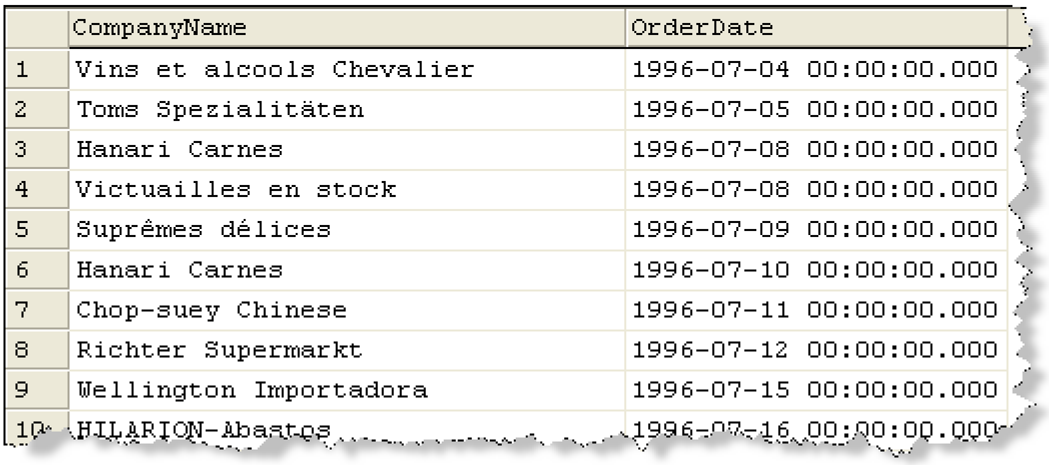
However, this doesn't specify how the tables are related - merely that they are. We must go on to use ON to specify which fields are involved in the relationships:

```
ON Customers.CustomerID = Orders.CustomerID
```

So, our final query looks like this:

```
SELECT CompanyName, OrderDate
FROM Customers JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
```

Hey presto - we get the output we wanted!

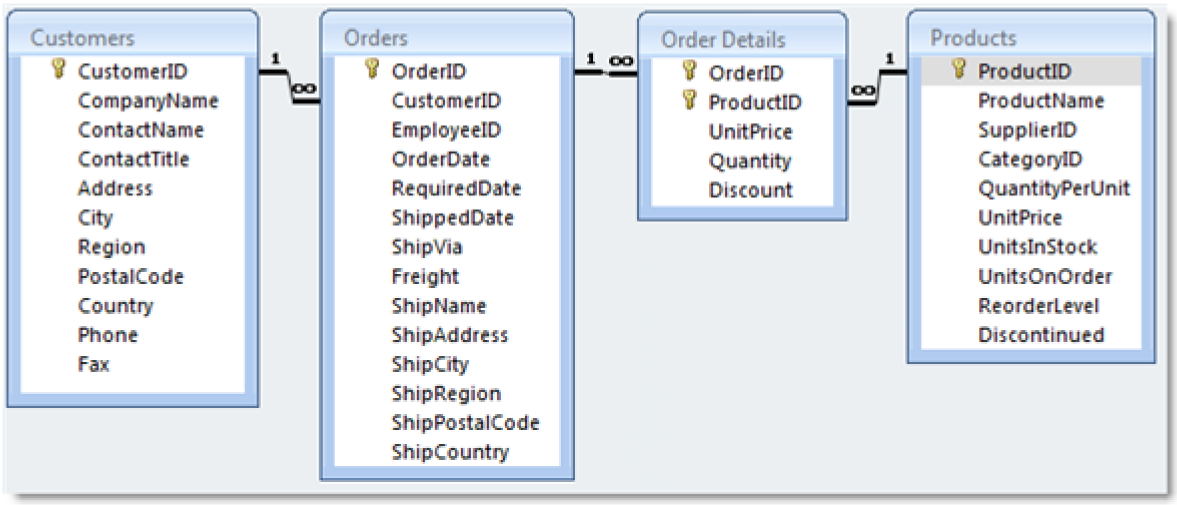


	CompanyName	OrderDate
1	Vins et alcools Chevalier	1996-07-04 00:00:00.000
2	Toms Spezialitäten	1996-07-05 00:00:00.000
3	Hanari Carnes	1996-07-08 00:00:00.000
4	Victuailles en stock	1996-07-08 00:00:00.000
5	Suprêmes délices	1996-07-09 00:00:00.000
6	Hanari Carnes	1996-07-10 00:00:00.000
7	Chop-suey Chinese	1996-07-11 00:00:00.000
8	Richter Supermarkt	1996-07-12 00:00:00.000
9	Wellington Importadora	1996-07-15 00:00:00.000
10	HILARION-Abastos	1996-07-16 00:00:00.000

It should be noted that what we have here is actually an INNER JOIN - within T-SQL the use of the word INNER is optional (although it is not within Jet SQL). See the page on outer joins for a discussion of the differences between inner and outer joins.

Now that we've seen how to join two tables, we can return to our more complex example mentioned above - we want to return the CompanyName, OrderDate, ProductName and Quantity.

The tables involved are Customers, Orders, Order Details and Products, with the relationships as shown in this diagram:



Note: The above diagram was taken from the Access version of Northwind. The tables and relationships are identical to those in the SQL Server version of Northwind, but within Access the relationship diagram indicates which fields are involved, whereas with the SQL Server database diagram, only the tables can be seen - it is not immediately obvious which fields are related.

The principles of what we need to do are exactly the same here as in our previous example - we just need to extend the SELECT statement to include the extra tables and relationships.

As before, we begin with the list of columns we want to retrieve:

```
SELECT CompanyName, OrderDate, ProductName, Quantity
```

Next we add the FROM clause, specifying each table and join in turn. First, let's do the Customers and Orders tables, just as before:

```
FROM Customers JOIN Orders
ON Customers.CustomerID=Orders.CustomerID
```

Next we repeat the JOIN ... ON clause to specify the next relationship, which is between the Orders table and the Order details table. As you can see from the above diagram, these are joined through the OrderID column, so our statement continues like this (not that Order Details is enclosed in square brackets to avoid an error being caused by the space in the table name):

```
JOIN [Order Details]
ON Orders.OrderID = [Order Details].OrderID
```

Finally, we add in the relationship, based on the ProductID field, between the Order Details table and the Products table:

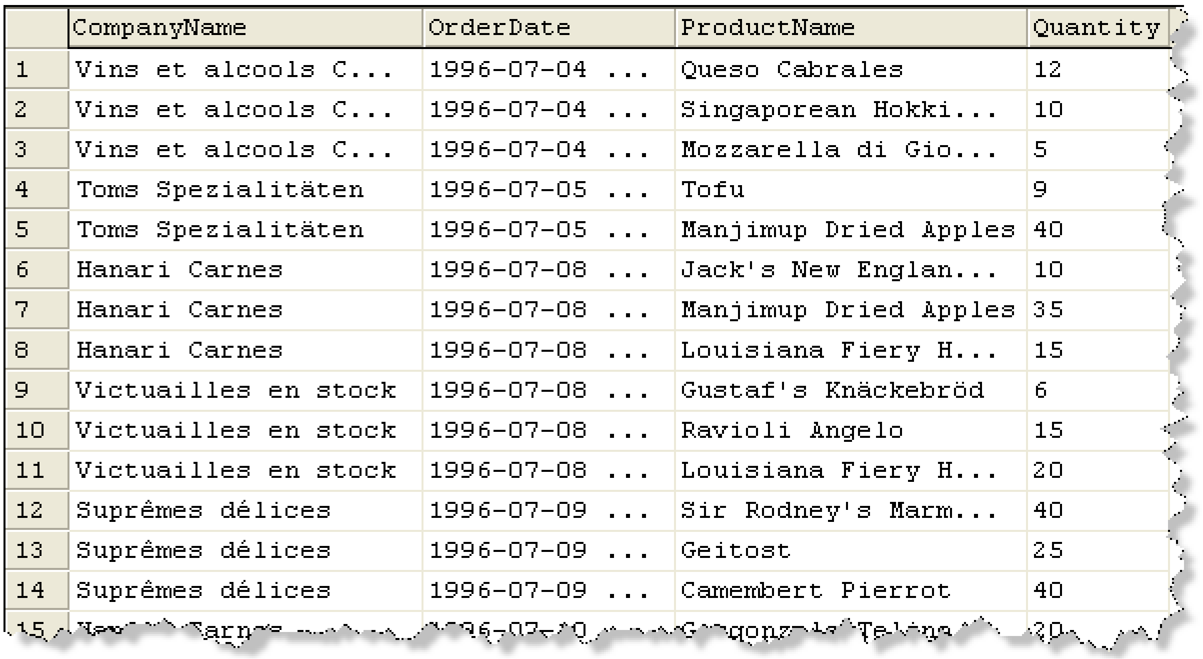
```
JOIN Products
ON [Order Details].ProductID = Products.ProductID
```

So, our final SELECT statement looks like this:

```
SELECT CompanyName, OrderDate, ProductName, Quantity
FROM Customers JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
    JOIN [Order Details]
    ON Orders.OrderID = [Order Details].OrderID
        JOIN Products
        ON [Order Details].ProductID = Products.ProductID
```

Note that in the above example the code has been indented in order to make the relationships easier to see. This is not a requirement, but you may find it helps the readability of your SELECT statements to do so.

The results of this query are as follows:



	CompanyName	OrderDate	ProductName	Quantity
1	Vins et alcools C...	1996-07-04 ...	Queso Cabrales	12
2	Vins et alcools C...	1996-07-04 ...	Singaporean Hokki...	10
3	Vins et alcools C...	1996-07-04 ...	Mozzarella di Gio...	5
4	Toms Spezialitäten	1996-07-05 ...	Tofu	9
5	Toms Spezialitäten	1996-07-05 ...	Manjimup Dried Apples	40
6	Hanari Carnes	1996-07-08 ...	Jack's New Englan...	10
7	Hanari Carnes	1996-07-08 ...	Manjimup Dried Apples	35
8	Hanari Carnes	1996-07-08 ...	Louisiana Fiery H...	15
9	Victuailles en stock	1996-07-08 ...	Gustaf's Knäckebröd	6
10	Victuailles en stock	1996-07-08 ...	Ravioli Angelo	15
11	Victuailles en stock	1996-07-08 ...	Louisiana Fiery H...	20
12	Suprêmes délices	1996-07-09 ...	Sir Rodney's Marm...	40
13	Suprêmes délices	1996-07-09 ...	Geitost	25
14	Suprêmes délices	1996-07-09 ...	Camembert Pierrot	40
15	[illegible]	[illegible]	[illegible]	[illegible]

This SELECT statement obviously works, and achieves what we set out to achieve. However, when dealing with multiple tables, particularly those with long or problematic names (as with Order Details above) it is often useful to use table aliases in order to simplify referral to those tables. An alias for a table is created through the use of AS within the FROM clause, and all references to the aliased table should then be to the alias rather than the full table name. (For information on column aliases, see the section on Functions and aliases.)

The above query can be rewritten to use an alias for Order Details as follows (the creation and use of the alias is highlighted in red):

```
SELECT CompanyName, OrderDate, ProductName, Quantity
FROM Customers JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
    JOIN [Order Details] AS OD
    ON Orders.OrderID = OD.OrderID
        JOIN Products
        ON OD.ProductID = Products.ProductID
```

The final issue to be considered is that of ambiguous column names in the SELECT list. So far, all our required column names have been unique - CompanyName can only come from the Customers table because that's the only table with such a column name, and similarly for the other fields.

But what happens if we want to select, for example, the OrderID column? This could be the OrderID column from either the Orders table or from the Order Details table. It doesn't matter that in practice the Order IDs will be the same in either table; without more information the database server is not able to determine which column should be retrieved.

In this scenario, it is necessary to "disambiguate" the column reference by prefixing it with the table name (or the alias) of the table to be used.

So, if we want to return the CustomerID (from the Customers table), the OrderID (from the Order Details table), the Order date (which can only come from the Orders table, the ProductID (from the Products table) and the UnitPrice (from the Order Details table) we must write our query as follows:

```
SELECT Customers.CustomerID, OD.OrderID, OrderDate,
Products.ProductID, OD.UnitPrice
FROM Customers JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
    JOIN [Order Details] AS OD
    ON Orders.OrderID = OD.OrderID
        JOIN Products
        ON OD.ProductID = Products.ProductID
```

So, we can now retrieve data from multiple tables. We can create aliases for awkward table names, and we can specify which column should be used when there is a choice.

In the next section, we'll compare and contrast these INNER JOINS that we've used so far with OUTER JOINS and see when you might use an outer join rather than an inner!

OUTER JOINS

WHAT IS AN OUTER JOIN?

When creating a join as we have until now, we have simply said "Select the following fields from these tables which are joined." We have not specified any particular type of join - we've merely said "FROM TableA JOIN TableB". Thus, we have been using the default join type, which is an INNER join.

It should be noted that if you are writing Jet SQL within Access, the use of the word INNER is required unlike in T-SQL, where the use of INNER is not required.

What does this mean? Simply that for a row to be returned, there must be a record to return from BOTH tables involved in the join.

Take as an example our requirement to see a list of Customers, and for each one, a list of their order dates. We wrote such a query like this in the previous section of this guide:

```
SELECT CompanyName, OrderDate
FROM Customers JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
```

But in fact, if we are to treat the requirement for "a list of Customers, and for each one, a list of their order dates" precisely, this isn't quite what we have here. Why not? Well, we only have a list of **those customers who have placed orders**, and for each of those, the dates of their orders. In other words, there is no mention here of, say, customers who phoned us, and from whom we took details, but who never got around to placing an order. Again, the reason for this is that the INNER JOIN **only** shows rows where there are related records in both tables.

If we extend this query to include information from the Order Details table, then we're further restricting our returned rows to those where we have a customer for whom we have a record in the Orders table, **and where that order also has related information in the order details table.** So, if we have a customer who only ever placed one order, but for that order we have no details (must be down to poor data entry staff!) in the Order Details table **we will see no information for that customer, nor for their order information in the orders table.**

In practice, of course, this is most often what was required. The likelihood is that in writing the above SELECT statement, we wanted customer and order information for our orders. But what if we really did want "a list of customers [all of them] and for each one, a list of their order dates [where they've placed orders]."

For this we would need to use an OUTER join.

However, the picture is perhaps muddied somewhat by the fact that there are two types of outer join - the LEFT OUTER join and the RIGHT OUTER join. The distinction between the two comes down to an understanding of which is the parent table in the relationship and which is the child. Let's examine this idea in a little more detail before returning to the concept of LEFT OUTER joins versus RIGHT OUTER joins.

The relationship between Customers and Orders is a one-to-many relationship - that is, for a single customer, there may be many related orders, but for a single order there may only be one related customer. Similarly, the relationship between Orders and Order Details is one-to-many: a single order may have many details (the 'lines' of an order, such as 34 pencils at 28 pence each) but a single order detail can only belong to one order.

In each of these cases, and indeed in any one-to-many relationship, **the one side is always the parent; the many side is always the child.** To put it another way, the parent table is the one containing the Primary Key, the child table is the one containing the Foreign Key.

What about where the tables are involved in a one-to-one relationship? Take, for example, two tables relating to staff. One has the basic contact / HR information for every member of staff, the other has information pertaining to staff involvement in the company pension scheme. Of 2000 staff, only 400 are in the company pension scheme, so rather than putting all the information in a single table with the attendant empty columns for the 1600 staff to whom this information does not apply, we've created a separate table to hold the pension information (provider code, maturity date, monthly payment amount and suchlike).

In this scenario, a single member of staff in the StaffHR table will have a single matching record in the StaffPension table, and a single record in the StaffPension table will have just one matching record in the StaffHR table. This, then, would be a one-to-one relationship. So, which is the parent? In a one-to-one relationship, the parent is the 'dominant' table - in this case, the one with ALL the staff - the StaffHR table.

If there are matching records for all staff (perhaps one table is all the 'public' information, such as name, position, phone and the other is 'private' information, such as next of kin name, home address etc) then the decision is less obvious.

However, whether the 'dominant' table within a relationship is obvious or not - or indeed whether the relationship is one-to-one or one-to-many - the details of which table is the parent and which is the child can be determined by viewing the relationship properties, as in this diagram (taken from SQL Server 2000). Again, the parent table is the one containing the Primary Key, the child contains the Foreign Key:

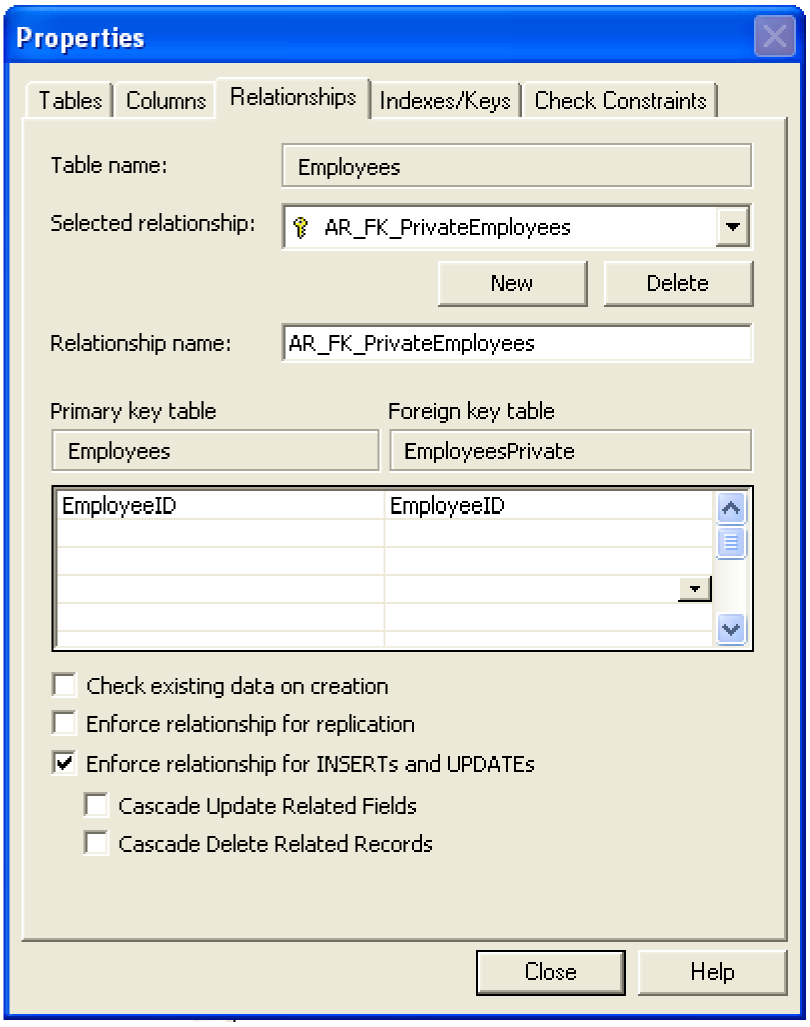


Table name: Employees

Selected relationship: AR_FK_PrivateEmployees

Relationship name: AR_FK_PrivateEmployees

Primary key table	Foreign key table
Employees	EmployeesPrivate
EmployeeID	EmployeeID

[ ] Check existing data on creation
[ ] Enforce relationship for replication
[✓] Enforce relationship for INSERTs and UPDATEs
[ ] Cascade Update Related Fields
[ ] Cascade Delete Related Records

So, let's get back to our initial question... How does all this affect our decision regarding the use of a LEFT or a RIGHT join?

If you want to see ALL the records from the parent (or primary key) table, along with any related records from the child (or foreign key) table, you will need a LEFT OUTER join.

If you want to see all the records from the child (foreign key) table, along with any that match in the parent (primary key) table, then use a RIGHT OUTER join. Of course, the relationship constraints within your database will, in most cases, prevent records from existing in a child table where there is no related record in a parent, but that doesn't make RIGHT OUTER joins redundant. It might be, for example, that you need to show in a query information from two tables which don't have a relationship, but which need to be temporarily joined within the SELECT statement. Or it could be that you are in the process of importing data from an old, badly-designed database which had badly created (or non-existent!) relationships.

So let's return once more to our original example. Remember, we want to see ALL our customers - including any that have never placed an order, but for those orders that have been placed, we want to see the order dates.

To achieve this, we amend the join to be a LEFT OUTER join, or, given that the word OUTER is not required within T-SQL, simply a LEFT JOIN.

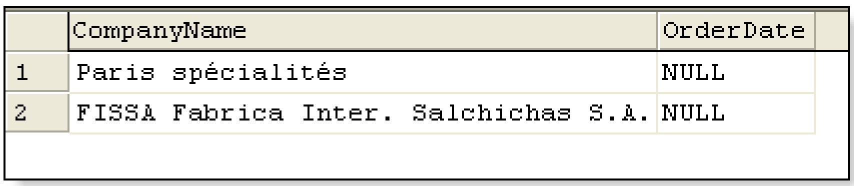
If we re-write our query as follows:

```
SELECT CompanyName, OrderDate
FROM Customers LEFT JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
```

we will now see ALL rows from the customers table, and any related rows from the Orders table. Whereas our original query returned 830 records, this new one returns 832 records - implying that two new records have been added, presumably for those customers who are in the customers table but never placed an order.

Adding a WHERE clause to the query to filter out any records where there IS an order date confirms this:

```
SELECT CompanyName, OrderDate
FROM Customers LEFT JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
WHERE OrderDate IS NULL
```



```
  CompanyName                         OrderDate
1 Paris spécialités                   NULL
2 FISSA Fabrica Inter. Salchichas S.A. NULL
```

Finally, we get our desired results - we see which customers have never placed an order! And presumably we begin to entice them with 'first order' special offers...