This documentation is archived and is not being maintained.

Writing SQL Queries: Let's Start with the Basics

Jackie Goldstein

Renaissance Computer Systems

November 2005

Summary: Learn to be more productive with SQL Server 2005 Express Edition with this quick introduction to the T-SQL language and the basics of getting information from the database using the SELECT statement.

Introduction

With the availability of ever more powerful programming tools and environments such as Visual Basic and Visual Studio.NET, as well as the availability of powerful database engines such as the free SQL Server 2005 Express Edition, more and more people find themselves having to learn the basics of SQL queries and statements. Sometimes they are professional developers who are experienced in other types of programming, and sometimes they are individuals whose expertise lies in other areas, but they suddenly find themselves programming database applications for fun and/or profit. If you fall into one of these categories, or are just curious about database programming, then this article is for you.

SQL Server 2005 Express offers you the opportunity to dive deeply into advanced databases and database applications, while still being free of charge. It is the same core database engine as all of the other versions in the SQL Server 2005, but it allows for easier setup and distribution all at no cost. It supports all of the advanced database features including, views, stored procedures, triggers, functions, native XML support, full T-SQL support, and high performance.

The purpose of this article is to lay out the basic structure and use of SQL **SELECT** queries and statements. These statements are part of Transact-SQL (T-SQL) language specification and are central to the use of Microsoft SQL Server. T-SQL is an extension to the ANSI SQL standard and adds improvements and capabilities, making T-SQL an efficient, robust, and secure language for data access and manipulation.

Although many tools are available for designing your queries visually, such as the Visual Database Tools that are available with Microsoft Visual Studio, it is still worthwhile and important to understand the SQL language. There is a real benefit to understanding what the visual tools are doing and why. There are also times when manually writing the necessary SQL statement is the only, or simply the fastest, way to achieve what you want. It is also an ideal way to learn how to use the full power of a relational database such as SQL Express.

Relational Databases: A 30 Second Review

Although there exist many different types of database, we will focus on the most common type—the relational database. A relational database consists of one or more **tables**, where each **table** consists of 0 or more **records**, or **rows**, of data. The data for each row is organized into discrete units of information, known as **fields** or **columns**. When we want to show the fields of a table, let's say the Customers table, we will often show it like this:

Many of the tables in a database will have **relationships**, or links, between them, either in a one-to-one or a one-to-many relationship. The connection between the tables is made by a **Primary Key** – **Foreign Key** pair, where a Foreign Key field(s) in a given table is the Primary Key of another table. As a typical example, there is a one-to-many relationship between Customers and Orders. Both tables have a CustID field, which is the Primary Key of the Customers table and is a Foreign Key of the Orders Table. The related fields do not need to have the identical name, but it is a good practice to keep them the same.

Fetching Data: SQL SELECT Queries

It is a rare database application that doesn't spend much of its time fetching and displaying data. Once we have data in the database, we want to "slice and dice" it every which way. That is, we want to look at the data and analyze it in an endless number of different ways, constantly varying the filtering, sorting, and calculations that we apply to the raw data. The SQL **SELECT** statement is what we use to choose, or select, the data that we want returned from the database to our application. It is the language we use to formulate our question, or query, that we want answered by the database. We can start out with very simple queries, but the **SELECT** statement has many different options and extensions, which provide the great flexibility that we may ultimately need. Our goal is to help you understand the structure and most common elements of a **SELECT** statement, so that later you will be able to understand the many options and nuances and apply them to your specific needs. We'll start with the bare minimum and slowly add options for greater functionality.

> **Note:** For our illustrations, we will use the Employees table from the Northwind sample database that has come with MS Access, MS SQL Server and is available for download at the Microsoft Download Center.

A SQL **SELECT** statement can be broken down into numerous elements, each beginning with a keyword. Although it is not necessary, common convention is to write these keywords in all capital letters. In this article, we will focus on the most fundamental and common elements of a **SELECT** statement, namely

- **SELECT**
- **FROM**
- **WHERE**
- **ORDER BY**

The SELECT ... FROM Clause

The most basic SELECT statement has only 2 parts: (1) what columns you want to return and (2) what table(s) those columns come from.

If we want to retrieve all of the information about all of the customers in the Employees table, we could use the asterisk (*) as a shortcut for all of the columns, and our query looks like

```
SELECT * FROM Employees
```

If we want only specific columns (as is usually the case), we can/should explicitly specify them in a comma-separated list, as in

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
```

which results in the specified fields of data for *all* of the rows in the table:

EmployeeID	FirstName	LastName	HireDate	City
1	Nancy	Davolio	1/5/1992 12:00:00 AM	Seattle
2	Andrew	Fuller	14/8/1992 12:00:00 AM	Tacoma
3	Janet	Leverling	1/4/1992 12:00:00 AM	Kirkland
4	Margaret	Peacock	3/5/1993 12:00:00 AM	Redmond
5	Steven	Buchanan	17/10/1993 12:00:00 AM	London
6	Michael	Suyama	17/10/1993 12:00:00 AM	London
7	Robert	King	2/1/1994 12:00:00 AM	London
8	Laura	Callahan	5/3/1994 12:00:00 AM	Seattle
9	Anne	Dodsworth	15/11/1994 12:00:00 AM	London

Explicitly specifying the desired fields also allows us to control the order in which the fields are returned, so that if we wanted the last name to appear before the first name, we could write

```
SELECT EmployeeID, LastName, FirstName, HireDate, City FROM Employees
```

The WHERE Clause

The next thing we want to do is to start limiting, or filtering, the data we fetch from the database. By adding a **WHERE** clause to the **SELECT** statement, we add one (or more) conditions that must be met by the selected data. This will limit the number of rows that answer the query and are fetched. In many cases, this is where most of the "action" of a query takes place.

We can continue with our previous query, and limit it to only those employees living in London:

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
WHERE City = 'London'
```

resulting in

EmployeeID	FirstName	LastName	HireDate	City
5	Steven	Buchanan	17/10/1993 12:00:00 AM	London
6	Michael	Suyama	17/10/1993 12:00:00 AM	London
7	Robert	King	2/1/1994 12:00:00 AM	London
9	Anne	Dodsworth	15/11/1994 12:00:00 AM	London

If you wanted to get the opposite, the employees who do *not* live in London, you would write

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
WHERE City <> 'London'
```

It is not necessary to test for equality; you can also use the standard equality/inequality operators that you would expect. For example, to get a list of employees who where hired on or after a given date, you would write

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
WHERE HireDate >= '1-july-1993'
```

and get the resulting rows

EmployeeID	FirstName	LastName	HireDate	City
5	Steven	Buchanan	17/10/1993 12:00:00 AM	London
6	Michael	Suyama	17/10/1993 12:00:00 AM	London
7	Robert	King	2/1/1994 12:00:00 AM	London
8	Laura	Callahan	5/3/1994 12:00:00 AM	Seattle
9	Anne	Dodsworth	15/11/1994 12:00:00 AM	London

Of course, we can write more complex conditions. The obvious way to do this is by having multiple conditions in the **WHERE** clause. If we want to know which employees were hired between two given dates, we could write

```
SELECT      EmployeeID, FirstName, LastName, HireDate, City
FROM        Employees
WHERE       (HireDate >= '1-june-1992') AND (HireDate <= '15-december-1993')
```

resulting in

EmployeeID	FirstName	LastName	HireDate	City
2	Andrew	Fuller	14/8/1992 12:00:00 AM	Tacoma
4	Margaret	Peacock	3/5/1993 12:00:00 AM	Redmond
5	Steven	Buchanan	17/10/1993 12:00:00 AM	London
6	Michael	Suyama	17/10/1993 12:00:00 AM	London

Note that SQL also has a special BETWEEN operator that checks to see if a value is between two values (including equality on both ends). This allows us to rewrite the previous query as

```
SELECT     EmployeeID, FirstName, LastName, HireDate, City
FROM       Employees
WHERE      HireDate BETWEEN '1-june-1992' AND '15-december-1993'
```

We could also use the **NOT** operator, to fetch those rows that are *not* between the specified dates:

```
SELECT     EmployeeID, FirstName, LastName, HireDate, City
FROM       Employees
WHERE      HireDate NOT BETWEEN '1-june-1992' AND '15-december-1993'
```

Let us finish this section on the **WHERE** clause by looking at two additional, slightly more sophisticated, comparison operators.

What if we want to check if a column value is equal to more than one value? If it is only 2 values, then it is easy enough to test for each of those values, combining them with the **OR** operator and writing something like

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
WHERE City = 'London' OR City = 'Seattle'
```

However, if there are three, four, or more values that we want to compare against, the above approach quickly becomes messy. In such cases, we can use the **IN** operator to test against a set of values. If we wanted to see if the City was either Seattle, Tacoma, or Redmond, we would write

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
WHERE City IN ('Seattle', 'Tacoma', 'Redmond')
```

producing the results shown below.

EmployeeID	FirstName	LastName	HireDate	City
1	Nancy	Davolio	1/5/1992 12:00:00 AM	Seattle
2	Andrew	Fuller	14/8/1992 12:00:00 AM	Tacoma
4	Margaret	Peacock	3/5/1993 12:00:00 AM	Redmond
8	Laura	Callahan	5/3/1994 12:00:00 AM	Seattle

As with the **BETWEEN** operator, here too we can reverse the results obtained and query for those rows where City is not in the specified list:

SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees

WHERE City NOT IN ('Seattle', 'Tacoma', 'Redmond')

Finally, the **LIKE** operator allows us to perform basic pattern-matching using wildcard characters. For Microsoft SQL Server, the wildcard characters are defined as follows:

Wildcard	Description
_ (underscore)	matches any single character
%	matches a string of one or more characters
[ ]	matches any single character within the specified range (e.g. [a-f]) or set (e.g. [abcdef]).
[^]	matches any single character not within the specified range (e.g. [^a-f]) or set (e.g. [^abcdef]).

A few examples should help clarify these rules.

- **WHERE** FirstName **LIKE** '_im' finds all three-letter first names that end with 'im' (e.g. Jim, Tim).
- **WHERE** LastName **LIKE** '%stein' finds all employees whose last name ends with 'stein'

- **WHERE** LastName **LIKE** '%stein%' finds all employees whose last name includes 'stein' anywhere in the name.
- **WHERE** FirstName **LIKE** '[JT]im' finds three-letter first names that end with 'im' and begin with either 'J' or 'T' (that is, *only* Jim and Tim)
- **WHERE** LastName **LIKE** 'm[^c]%' finds all last names beginning with 'm' where the following (second) letter is not 'c'.

Here too, we can opt to use the NOT operator: to find all of the employees whose first name does not start with 'M' or 'A', we would write

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
WHERE (FirstName NOT LIKE 'M%') AND (FirstName NOT LIKE 'A%')
```

resulting in

EmployeeID	FirstName	LastName	HireDate	City
1	Nancy	Davolio	1/5/1992 12:00:00 AM	Seattle
3	Janet	Leverling	1/4/1992 12:00:00 AM	Kirkland
5	Steven	Buchanan	17/10/1993 12:00:00 AM	London
7	Robert	King	2/1/1994 12:00:00 AM	London
8	Laura	Callahan	5/3/1994 12:00:00 AM	Seattle

The ORDER BY Clause

Until now, we have been discussing filtering the data: that is, defining the conditions that determine which rows will be included in the final set of rows to be fetched and returned from the database. Once we have determined which columns and rows will be included in the results of our **SELECT** query, we may want to control the order in which the rows appear—sorting the data.

To sort the data rows, we include the **ORDER BY** clause. The **ORDER BY** clause includes one or more column names that specify the sort order. If we return to one of our first **SELECT** statements, we can sort its results by City with the following statement:

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
ORDER BY City
```

By default, the sort order for a column is ascending (from lowest value to highest value), as shown below for the previous query:

EmployeeID	FirstName	LastName	HireDate	City
3	Janet	Leverling	1/4/1992 12:00:00 AM	Kirkland
5	Steven	Buchanan	17/10/1993 12:00:00 AM	London
6	Michael	Suyama	17/10/1993 12:00:00 AM	London
7	Robert	King	2/1/1994 12:00:00 AM	London
9	Anne	Dodsworth	15/11/1994 12:00:00 AM	London
4	Margaret	Peacock	3/5/1993 12:00:00 AM	Redmond
1	Nancy	Davolio	1/5/1992 12:00:00 AM	Seattle
8	Laura	Callahan	5/3/1994 12:00:00 AM	Seattle
2	Andrew	Fuller	14/8/1992 12:00:00 AM	Tacoma

If we want the sort order for a column to be descending, we can include the **DESC** keyword after the column name.

The **ORDER BY** clause is not limited to a single column. You can include a comma-delimited list of columns to sort by—the rows will all be sorted by the first column specified and then by the next column specified. If we add the Country field to the **SELECT** clause and want to sort by Country and City, we would write:

```
SELECT EmployeeID, FirstName, LastName, HireDate, Country, City FROM Employees
ORDER BY Country, City DESC
```

Note that to make it interesting, we have specified the sort order for the City column to be descending (from highest to lowest value). The sort order for the Country column is still ascending. We could be more explicit about this by writing

```
SELECT EmployeeID, FirstName, LastName, HireDate, Country, City FROM Employees
ORDER BY Country ASC, City DESC
```

but this is not necessary and is rarely done. The results returned by this query are

EmployeeID	FirstName	LastName	HireDate	Country	City
5	Steven	Buchanan	17/10/1993 12:00:00 AM	UK	London
6	Michael	Suyama	17/10/1993 12:00:00 AM	UK	London
7	Robert	King	2/1/1994 12:00:00 AM	UK	London
9	Anne	Dodsworth	15/11/1994 12:00:00 AM	UK	London
2	Andrew	Fuller	14/8/1992 12:00:00 AM	USA	Tacoma
1	Nancy	Davolio	1/5/1992 12:00:00 AM	USA	Seattle
8	Laura	Callahan	5/3/1994 12:00:00 AM	USA	Seattle
4	Margaret	Peacock	3/5/1993 12:00:00 AM	USA	Redmond
3	Janet	Leverling	1/4/1992 12:00:00 AM	USA	Kirkland

It is important to note that a column does not need to be included in the list of selected (returned) columns in order to be used in the **ORDER BY** clause. If we don't need to see/use the Country values, but are only interested in them as the primary sorting field we could write the query as

```
SELECT EmployeeID, FirstName, LastName, HireDate, City FROM Employees
ORDER BY Country ASC, City DESC
```

with the results being sorted in the same order as before:

EmployeeID	FirstName	LastName	HireDate	City
5	Steven	Buchanan	17/10/1993 12:00:00 AM	London
6	Michael	Suyama	17/10/1993 12:00:00 AM	London
7	Robert	King	2/1/1994 12:00:00 AM	London
9	Anne	Dodsworth	15/11/1994 12:00:00 AM	London
2	Andrew	Fuller	14/8/1992 12:00:00 AM	Tacoma
1	Nancy	Davolio	1/5/1992 12:00:00 AM	Seattle
8	Laura	Callahan	5/3/1994 12:00:00 AM	Seattle
4	Margaret	Peacock	3/5/1993 12:00:00 AM	Redmond
3	Janet	Leverling	1/4/1992 12:00:00 AM	Kirkland

Conclusion

In this article we have taken a look at the most basic elements of a SQL **SELECT** statement used for common database querying tasks. This includes how to specify and filter both the columns and the rows to be returned by the query. We also looked at how to control the order of rows that are returned.

Although the elements discussed here allow you to accomplish many data access / querying tasks, the SQL **SELECT** statement has many more options and additional functionality. This additional functionality includes grouping and aggregating data (summarizing, counting, and analyzing data, e.g. minimum, maximum, average values). This article has also not addressed another fundamental aspect of fetching data from a relational database—selecting data from multiple tables.

References

Additional and more detailed information on writing SQL queries and statements can be found in these two books:

McManus, Jeffrey P. and Goldstein, Jackie, Database Access with Visual Basic.NET (Third Edition), Addison-Wesley, 2003

Hernandez Michael J. and Viescas, John L., SQL Queries for Mere Mortals, Addison-Wesley, 2000.

Jackie Goldstein is the principal of Renaissance Computer Systems, specializing in consulting, training, and development with Microsoft tools and technologies. Jackie is a Microsoft Regional Director and MVP, founder of the Israel VB User Group, and a featured speaker at international developer events including TechEd, VSLive!, Developer Days, and Microsoft PDC. He is also the author of Database Access with Visual Basic.NET (Addison-Wesley, ISBN 0-67232-3435) and a member of the INETA Speakers Bureau. In December 2003, Microsoft designated Jackie as a .NET Software Legend.

© Microsoft Corporation. All rights reserved.

© 2016 Microsoft