Exercising Options

Russell Sinclair

In this month's installment of Working SQL, Russell Sinclair talks about optional parameters in SQL Server stored procedures, and how you can use them to do advanced searching of data.

I'VE been working with SQL Server for a number of years now, and one of the challenges that I consistently face is how to provide users with the most possible options to find data, without spending weeks developing search interfaces. However I plan my applications, there's always a user out there who doesn't think the way that I think he or she should think. Someone always wants to be able to search on data using fields that I don't think would return informative results. Fortunately, Access has always made dealing with these users easy by providing the ability to filter data on forms and reports using methods built into forms and reports. Unfortunately, when you look at what Access is doing in the background, the filtering that Access performs isn't the most efficient way of getting at data. When it comes to SQL Server, these inefficiencies can actually be a detriment to the performance of the application.

Parameters in SQL Server

Stored procedures and user-defined functions in SQL Server both have the ability to accept parameters. Parameters can be of any valid SQL Server datatype and, if you provide a default value, can be optional.

The syntax for declaring a parameter in SQL Server is simple:

```
@ParamName [AS] datatype [=defaultvalue]
```

Here's an example that declares a parameter called @CustName of type string with a default value of "Russell Sinclair":

```
@CustName AS char = 'Russell Sinclair'
```

When using parameters in SQL Server, specifying a default value for the parameter when you create the function or stored procedure means that you don't have to supply a value for the parameter when you use the function or stored procedure. For example, if I wanted to create a stored procedure that would return the customers in a country when I specify one, or return those customers in the United States if I didn't specify a country, I could use the following code:

```
CREATE PROCEDURE uspCustomersInCountry
  (@nvcCountry nvarchar(15) = 'USA')
AS
  SELECT * FROM Customers
  WHERE Country = @nvcCountry
```

When using the SQL Server tools, or in the RecordSource property of a form, I can call this procedure using this statement:

```
EXEC uspCustomersInCountry
```

By using the SQL Server DEFAULT keyword, I can return all customers in the USA:

```
EXEC uspCustomersInCountry DEFAULT
```

If I wanted to return all customers in Canada, I could use the statement:

```
EXEC uspCustomersInCountry 'Canada'
```

In my first example, I didn't specify value for the parameter of the query, so SQL Server uses the default value. This isn't too complicated because I provided a valid parameter value in my stored procedure. If I want my parameter to be truly optional, I need to do something else—I need to specify that the default value is Null. However, my query, as it exists now, won't work if I do this. I need to handle this situation differently.

The problem with Null

Null is typically defined as the absence of any value, either because it's unknown or because it doesn't apply to the instance of the object we're trying to define. A company might have a Null address because we don't know it, or an employee might have a Null SpouseName field because he isn't married. Either case might result in a Null value. However, Null can't be compared with other data.

In the database world, any value compared to Null results in a Null value. If I changed the previous example to accept Null as the default value for the stored procedure, I'd never get results from the query when I didn't provide a value. Unless I provide some value other than the default (even if I used a Like comparison with a wildcard), I won't get a result if I omit the "optional" parameter. Because of this problem, I need to figure out a way of handling Null in my stored procedure. The simplest way to do this is to use an If statement:

```
IF @nvcCountry IS NULL
    SELECT *
    FROM Customers
ELSE
    SELECT * FROM Customers
    WHERE Country = @nvcCountry
```

Although this solution will work, I start to run into problems if I have more than one optional parameter. With multiple optional parameters, I need to handle every possible combination of parameters being specified and not being specified. This result is a lot of T-SQL coding.

Handling Null gracefully

The solution to this problem requires that you do something you wouldn't normally do—include the parameter itself as the first part of a comparison within another portion of the WHERE clause. In order to handle my previous example, I need to change my SQL statement to this:

```
CREATE PROCEDURE uspCustomersInCountry
  (@nvcCountry nvarchar(15) = NULL)
AS
  SELECT * FROM Customers
  WHERE ((Country = @nvcCountry)
         OR (@nvcCountry IS NULL))
```

The logic for this statement is relatively simple—if the field matches the parameter (which it won't if the parameter is Null), it's considered a match. Otherwise, if the parameter itself is Null, then we also have a match. Since these two portions of the WHERE statement are matched with an OR comparison, either match type will result in records being returned. This same example could be extended to use a Like comparison by replacing the equal sign (=) on the first test with the LIKE keyword.

This simplifies matching multiple optional parameters. In order to determine whether I have a match, I simply create a matched pair of tests for each field. The same example, extended to include the City and Region, would be:

```
CREATE PROCEDURE uspCustomerMatch
  (@nvcCity nvarchar(15) = NULL,
   @nvcRegion nvarchar(15) = NULL,
   @nvcCountry nvarchar(15) = NULL)
AS
  SELECT * FROM Customers
  WHERE ((City = @nvcCity)
         OR (@nvcCity IS NULL))
      AND ((Region = @nvcRegion)
         OR (@nvcRegion IS NULL))
      AND ((Country = @nvcCountry)
         OR (@nvcCountry IS NULL))
```

The method described here can be extended to handle most simple comparisons, but what about other types of comparisons like BETWEEN and IN clauses?

Between options

BETWEEN clauses tend to be a bit more difficult to handle than simple comparisons. The reason for this is that BETWEEN requires two parameters, and both of them could be optional. If one parameter is supplied, but the other isn't, what can you do? One solution would be to set the default values of each parameter to the maximum and minimum values of the datatype you're dealing with. If you wanted to use this method to return all orders placed between two dates, you could use a stored procedure defined like this:

```
CREATE PROCEDURE uspOrderSearch
  (@dtmStart datetime = '1753 JAN 1',
   @dtmEnd datetime = '1999 DEC 31')
AS
  SELECT * FROM Orders
  WHERE OrderDate BETWEEN @dtmStart AND @dtmEnd
```

Although this method works, it's not the only option. Another option is to continue working as I did in the previous example, by providing extra comparisons to handle the Null:

```
CREATE PROCEDURE uspOrderSearch2
  (@dtmStart datetime = NULL,
   @dtmEnd datetime = NULL)
AS
  SELECT * FROM Orders
  WHERE ((OrderDate BETWEEN @dtmStart AND @dtmEnd)
         OR ((OrderDate >= @dtmStart) AND
                   (@dtmEnd IS NULL))
         OR ((OrderDate <= @dtmEnd) AND
                   (@dtmStart IS NULL))
         OR ((@dtmStart IS NULL) AND
                   (@dtmEnd IS NULL)))
```

In this example, I've handled each possible combination of the two parameters. Some of you may have assumed that the first method is faster than the second. However, this isn't true. In fact, when I timed both stored procedures, the difference between the execution time of each query was less than 100 milliseconds. This was true even when I used different combinations of supplying each of the parameters and accepting the defaults. However, since the first method is much easier to write and maintain, I suggest that you use that method for building your comparisons.

IN too deep

Now we come to the really challenging comparison: How can you possibly pass a list of items to SQL Server so that it can match a series of items? I'll look at a basic example just to get started. Suppose that the managers of Northwind Traders want to get a list of all orders sold by an employee. This is a simple enough request. All you have to do is retrieve all of the records in the Orders table with a specific EmployeeID:

```
SELECT Orders.*
FROM Orders
WHERE EmployeeID = 23
```

Now, suppose that the requirements change. The managers at Northwind each manage a group of salespeople, and they want to see and compare all of the salespeople in their group, and sometimes salespeople in

other groups.

If you wanted to satisfy this need on a one-time basis, you could create a SELECT statement with an IN clause:

```
SELECT Orders.*
FROM Orders
WHERE EmployeeID IN (1, 24, 34, 45)
```

The real problem that arises in this case is finding lists of people when you don't know what the list is ahead of time. You can't pass IN lists to SQL Server in the same way that you can pass individual variables. The solution to this problem requires that you use both VBA code and another specialized comparison in SQL Server.

Typically, the way in which you'd allow a user to select multiple employees would be to create a ListBox control on a form and set its MultiSelect property to a value other than "None." What you need to do at this point is create a string of the selected items, using a delimiter to separate them—my usual choice is the bar (|) character located above the Enter key on most keyboards (I'll explain the reason for this in a moment). The code I use is:

```
Dim lngRow As Long
Dim strList As String
Const strcDelim As String = "|"

strList = strcDelim
For lngRow = 0 To lstEmployees.ListCount - 1
    If lstEmployees.Selected(lngRow) Then
        strList = strList & _
            lstEmployees.Column(0, lngRow) & strcDelim
    End If
Next
```

If a user has selected employees 1, 2, 5, and 7, the variable strList will have the value "|1|2|5|7|" after the code has run. This is the value of a parameter that I supply to SQL Server.

In order to handle this parameter in SQL Server, I need to use a variation on an earlier trick—I need to compare a field to this parameter with the parameter on the left side of the comparison. Here's the code:

```
CREATE PROCEDURE uspOrderSearch3
  (@nvcList nvarchar(2000) = NULL)
AS
  SELECT *
  FROM Orders
  WHERE (@nvcList Like
  '%|' + CONVERT(nvarchar, EmployeeID) + '|%')
  OR (@nvcList IS NULL)
```

What this query does is convert the value of the EmployeeID field into a comparable entity by casting it to a string datatype (SQL Server doesn't support implicit conversions) and prepending and appending a bar character and SQL Server wildcard characters to each end. In cases where the field has a value of three, the comparison evaluates to "'|1|2|5|7|' Like '%|3|%.'" Since each EmployeeID is surrounded by the bar characters, you'll only ever get exact matches. If you took them out of the right-hand side of the comparison, it would evaluate to "'|1|2|5|7|' Like '%3%'" and any field with a 3 in it would be returned (30, 13, 463, and so forth).

Improving performance

The methods I've described here for creating advanced searches have one caveat—they're not very efficient. They force you to do a comparison on a field even if the user hasn't provided a value for that field as part of the search. For this reason, you should take some steps to optimize your SQL code. The best way to do this is to handle a few common search patterns differently, and for this you need to make educated guesses as to what fields users will provide search parameters for most often. You can then provide alternate, optimized searches if only a few (or no) parameters are provided.

Suppose that I wanted to extend my search against the Customers table. I'd think that the most common searches would be against the City field. Here's how I'd optimize an earlier stored procedure based on this assumption:

```
CREATE  PROCEDURE uspCustomerMatch2
    (@nvcCity nvarchar(15) = NULL,
     @nvcRegion nvarchar(15) = NULL,
     @nvcCountry nvarchar(15) = NULL)
AS

  IF (@nvcCity IS NULL) AND (@nvcRegion IS NULL)
    AND (@nvcCountry IS NULL)
```

Continues on page 24

Using the Graphical Designers

A word of warning about creating WHERE clauses that evaluate a parameter on the left side of a comparison—don't use the graphical designers. The graphical designers are great tools if you're creating simple SELECT statements. However, if you get deep into SQL and begin doing some unusual comparisons, switching to the graphical designer will completely mess up your code. If you want to see what I mean, simply enter the SQL statement for uspCustomerMatch as shown in the article into a stored procedure using SQL view. Save the stored procedure and switch to graphical view. Click the Save button while in graphical view. You'll notice that if you switch back to SQL view, the statement is virtually unreadable—there are 24 different tests taking place, instead of the six that you entered before.

If you're thinking of trying this in Access, my best advice is, "Don't." No matter what you do, Access will eventually reformat your SQL statement so that it's entirely unreadable. And, better yet, if you have more than one or two tests in your WHERE clause, it's quite likely that you'll crash Access. Believe me—I know this one from experience.

Exercising Options...

Continued from page 19

```
    SELECT * FROM Customers

ELSE IF (@nvcCity ISN'T NULL)
  AND (@nvcRegion IS NULL)
  AND (@nvcCountry IS NULL)

    SELECT * FROM Customers
    WHERE (City = @nvcCity)

ELSE

    SELECT * FROM Customers
    WHERE ((City = @nvcCity)
           OR (@nvcCity IS NULL))
      AND ((Region = @nvcRegion)
           OR (@nvcRegion IS NULL))
      AND ((Country = @nvcCountry)
           OR (@nvcCountry IS NULL))
```

This example is quite basic, but it shows you where you can start. In this case, I covered the first and most important case—where no parameters were supplied. In the next part of the IF structure, I check to see whether my most common parameter(s) are supplied and that the others weren't. Failing that second test, I do a full-blown search.

Happy searching. ▲

Russell Sinclair is an MSCD and is the owner of Synthesystems, a technology consulting firm specializing in Visual Basic, SQL Server, and Microsoft Access development. He's the author of *From Access to SQL Server*, an Access developer's guide to migrating to SQL Server, the senior programmer with Questica, Inc., a company that specializes in software for custom-design manufacturers, and is a *Smart Access* Contributing Editor. russell@synthesystems.com.

Downloads July 2002 Source Code

- OUTXML.ZIP—A sample database that passes task data between Access and Outlook. (Access 2000)
- USERRSTR.ZIP—This database reports on how many users are accessing your data (even for non-code MDBs). (Access 2000)
- TSTHRNSS.ZIP—Two databases: A test harness for creating automated testing packages and the sample "Stump the Expert" code in MyApp. (Access 97)
- GOOGLE.ZIP—This database accesses Google as a Web Service from Access VBA code. (Access 2002)

Editor Peter Vogel
Contributing Editors Andy Baron, Mary Chipman, Mike Gunderloy, Garry Robinson, Russell Sinclair
President Connie Austin
Executive Editor of IT Farion Grove
Editorial Assistant Micki Bailey

Direct all editorial, advertising, or subscription-related questions to Pinnacle Publishing, Inc.:
1-800-788-1900 or 770-992-9401
Fax: 770-993-4323
Pinnacle Publishing, Inc.
PO Box 769389
Roswell, GA 30076-8220
E-mail: **smartacc@pinpub.com**
Pinnacle Web Site:
www.pinnaclepublishing.com
Access technical support:
Call Microsoft at 425-635-7050

Smart Access (ISSN 1066-7911) is published monthly (12 times per year) by Pinnacle Publishing, Inc., 1000 Holcomb Woods Pkwy, Bldg 200, Suite 280, Roswell, GA 30076-2587.

POSTMASTER: Send address changes to *Smart Access*, PO Box 769389, Roswell, GA 30076-8220.

DOWNLOAD

The Source Code portion of the *Smart Access* Web site is available to paid subscribers only. Log in for access to all current and archive content and source code. For access to this issue only, go to www.smartaccessnewsletter.com, click on "Source Code," select the file(s) you want from this issue, and enter the User name and Password at right when prompted.

User name: monsoon
Password: novel