

Doing the Impossible with ADO.NET

William Vaughn

"They said it couldn't be done but, of course, they were wrong. Yes, you can create and manage your own server-side, fully scrollable and updatable cursors on SQL Server with ADO.NET." And in his inaugural column, Bill Vaughn shows you how. Please join me in welcoming Bill as *Hardcore Visual Basic's* new "data guy." If you've ever had the privilege of hearing bill speak, I think you'll recognize his "voice" in those first two sentences.

For over a decade now, SQL Server and other DBMS developers have been using server-side cursors to access their databases and scroll through updatable rowsets. This "connected" approach assumes that the database connection remains in place while the application runs a query and builds a server-side set of rows that can be retrieved and updated as needed. For a litany of reasons, Microsoft chose not to implement server-side cursor functionality in ADO.NET, but, in this article, I'll show you how to work around this limitation to create and manage your own server-side cursors.

Server-side cursors are especially useful when working with highly interactive applications—especially when the application cannot work with disconnected (static) data. This type of application needs a mechanism to work with a single row or a small set of rows at once, and server-side cursors are designed to meet this need. This month, I'll show you how to:

- Create a cursor based on a focused SELECT statement,
- Position a cursor to any designated row and return the data, and
- Change the data in the currently selected cursor row.

How can ADO.NET create a server-side cursor?

In my latest ADO.NET books (see bio at end), I mention (repeatedly) that ADO.NET cannot create server-side cursors. Strictly speaking this is true. However, it's true only in the sense that ADO.NET can't create and manage a server-side cursor on its own in the same way that DAO and ADO classic (ADOC) could.

Warning #1: Don't use with ADOC, ASP apps

It is not a good idea to use the following techniques when working with ADOC. That's because the mechanisms within ADOC can conflict with any cursors you create on your own using SQL Server's T-SQL. It's also not a good idea to use them when working with ASP-based applications, as the connection state can't be maintained between page invocations.

The technique described here is really pretty simple. As illustrated in the example code, I use the ADO.NET Command object to execute specific SQL scripts (two or more tightly coupled SQL statements). Sometimes I use the ExecuteNonQuery when I don't care about what comes back from the query, but, in most cases, I use the ExecuteScalar to execute the SQL script and return the cursor status or other interesting information. When it comes time to actually fetch the data row (and we only fetch one at a time), I use either the DataReader ExecuteReader or the DataAdapter Fill. I prefer the Fill method because it creates a bindable DataTable (we can't bind to a DataReader in a Windows Forms application). Nope, this is not particularly efficient as it makes a round trip for each row. ADOC cached N rows when it constructed its cursors and fetched from that cache when you positioned to each row. But yes, you *can* implement the same functionality if you need this degree of control.

Opening, closing, and reopening the connection

Unlike other ADO.NET examples, I need to open a SQL Server connection and keep it open. That's because the cursor is built and maintained by the server and "owned" by the connection. Once the connection closes, the cursor is automatically flushed by the server. This also means that *you* must make sure the connection is reset when the connection is reused. Consider that a Windows Forms application (where this approach makes the most sense) creates a new connection pool for each unique connection string. No, if you create a cursor, it *won't* be destroyed immediately when you close the connection—assuming you're using connection pooling (which is on by default). That's because the "real" database connection is left open and held dormant in the connection pool in anticipation of the connection being reused. However, if the connection reset option is enabled (it is by default), the cursor is cleared when the connection is opened again. If you turn *off* the reset connection option, the cursor should remain viable until the connection pool times out, closes and discards the connection. Just because the connection owns a cursor, this does not prevent you from using the connection for other tasks. You're free to execute any type of query (even creating additional cursors) against the connection. However, if

you use ADO.NET to open a DataReader, you'll be prevented from using the connection (and your cursor) until the DataReader is closed.

The following code opens a persistent application-wide connection. It also initializes the DataAdapter and Command object used to execute queries and manage the cursor:

```
' Opening a persistent connection
cn = New SqlConnection("data source=demo;" _
    & "integrated security=sspi;initial catalog=biblio")
cn.Open() 'Conn. must remain open to hold cursor state
da = New SqlDataAdapter(Nothing, cn)
cmd = New SqlCommand(Nothing, cn)
```

Listing 1. Opening a persistent connection.

Warning #2: Beware SQL injection

Yes, I realize it's not a good idea to blindly accept "anything" from a user, but this is simply a demonstration so I'm pretty generous with what I accept from the user. To be safe(er), I call a ValidateSQL routine to do some rough tests to make sure I'm not too sleepy and enter some truly destructive SQL—such as DROP TABLE Authors. [SQL Server Professional *subscribers can read columnist Ron Talmage's excellent description of SQL injection in the March 2002 issue. Ed.*]

Describing the Cursor

The next step is to create the cursor based on an SQL query. This example uses an SQL statement passed from the user in a TextBox control. We can break down the SQL, which is used to describe and create the cursor, into three parts:

- 1) The DECLARE cursor statement, which names and describes the cursor. In this case, I ask for a scrollable cursor (as opposed to a forward-only cursor), and name it MyCursor. This is where you can tell SQL Server to create a keyset, static, or dynamic cursor. Check out SQL Server's online help, *Books Online* (BOL) for more options.
- 2) The SQL SELECT statement, which is executed to build the cursor rowset, follows the CURSOR FOR argument. The SELECT should focus the cursor on a *few* manageable rows in the database—not an entire database table if you expect to scale the application. Sure, it can contain a JOIN or a VIEW, as long as it's a standard (simple) SELECT statement. (The keywords COMPUTE, COMPUTE BY, FOR BROWSE, and INTO are not allowed.) I also experimented with executing a stored procedure to create the cursor—passing in the required options as parameters. It worked fine.
- 3) The last part of the SQL is a SELECT statement that's used to return the number of rows in the cursor membership. If you select too many rows, this returns -1 until SQL Server has completed server-side rowset population (until it has found all of the rows and added them to the cursor).

The following code illustrates creating a named server-side cursor. Here, I execute the SQL that defines the cursor using the ADO.NET *SqlCommand.ExecuteScalar* method. This returns the @@CURSOR_ROWS global variable so I know how many rows are in the cursor. (Remember not to close the connection after having created the cursor—if you do, the cursor will be dropped by SQL Server.)

```
With cmd
If ValidateSQL(txtSQL.Text) Then
'Look for SQL Injection attacks
    .CommandText = "DECLARE mycursor SCROLL CURSOR _
        FOR " & txtSQL.Text & " OPEN myCursor _
        SELECT @@CURSOR_ROWS"
    RowsInCursor = CInt(.ExecuteScalar())
End With
```

Listing 2. Describing the cursor with an SQL statement.

Fetching data from the cursor

Once the cursor is open, you have lots of options. The data can be fetched once or any number of times and in any order. (Yes, as in ADO classic cursors, a "current position" must be maintained and since ADO.NET won't do it for you, this task is up to you.) To retrieve a specific row (and only one row at a time), use the T-SQL FETCH operator naming the server-side cursor. The FETCH operator can be set to fetch the FIRST, LAST, NEXT, or PRIOR rows based on the current row pointer by using the appropriate operand in another SQL statement. You can also fetch a specific row (the "nth" row) using the ABSOLUTE operand; or, if you want to fetch a row that's positioned in the cursor a specific number of rows forward or back, you can use the RELATIVE operand.

Let's go over these techniques and how they're implemented in code:

- To fetch the “first” or “next” row: When the cursor is first opened, the “current row” pointer is positioned before the first row, so either FETCH FIRST or FETCH NEXT will get this first row. In this case, I simply set the CommandText with the appropriate FETCH TSQL operator and execute it using the Fill method. Note that I’m also returning a second resultset that contains the @@FETCH_STATUS global variable. This tells me if I’ve fetched past the last row or before the first row, or if something else went wrong. If you position past either end of the cursor, @@FETCH_STATUS returns -1 and there is no “current” row.

```
da.SelectCommand.CommandText = "FETCH NEXT _
FROM mycursor" & " SELECT @@FETCH_STATUS"
```

- To fetch the previous row: You can use FETCH PRIOR to step backward through the rows until @@FETCHSTATUS returns -1, which indicates that you’ve stepped over the “BOF” line.

```
da.SelectCommand.CommandText = "FETCH PRIOR _
FROM mycursor" & " SELECT @@FETCH_STATUS"
```

- To fetch a specific (absolute) row: Use FETCH ABSOLUTE with a specific row number to fetch. (Note that cursor rows begin with 1 and end in @@CURSOR_ROWS.)

```
da.SelectCommand.CommandText = "FETCH ABSOLUTE " _
& txtRow.Text & " FROM mycursor _
SELECT @@FETCH_STATUS"
```

Updating with a server-side cursor

While the example program written for this article does not attempt to update the cursor data, it’s not that hard to do (though it does expose a much wider set of issues—fodder for another article). Suffice it to say that the cursor can be updated through use of the CURRENT OF expression in the UPDATE statement’s WHERE clause. For example, to position to a specific row (based on the cursor row (1 to @@CURSOR_ROWS)) and update that row you could code:

```
'Update fifth row
cmd.CommandText = "FETCH ABSOLUTE 5 FROM mycursor " _
& " UPDATE Authors SET Author = 'Fred' _
WHERE CURRENT OF mycursor"
cmd.ExecuteNonQuery()
```

Pulling it all together

When we first started working with ADO.NET, we found that many of the techniques we’ve used for years in DAO, RDO, and ADO were no longer available. However, as we learned to adapt, it seems that there *are* a few easy workarounds to replace some of our tried-and-true techniques. This article discussed one such workaround. Watch for a forthcoming column where I discuss how to create and hold a row on the server using a pessimistic lock. Frankly, it seems in hindsight some of our well-worn techniques might not have been such good ideas in the first place. As I discover workable alternatives, I’ll share them with you here.