
Creating and Managing Server-Side Cursors with ADO.NET

We told you it couldn't be done. We were wrong. Yes you can create and manage your own server-side, fully scrollable and updatable cursors on SQL Server—and you can do it all with ADO.NET.

Why are server-side cursors important?

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 any server-side cursor functionality in their new .NET Framework data access interface ADO.NET. This article discusses how you can 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. Server-side cursors are designed to meet this need. As illustrated by the examples in this article, I'll show you how to:

- Create a cursor based on a focused SELECT statement.
- Position a cursor to any designated row.
- Change the data in the currently selected cursor row.
- Adapt the cursor so other users' changes are visible (or not).

How does ADO.NET implement cursors?

ADO.NET is fully capable of running queries, but the “cursor” it knows how to create is not the same as DAO, RDO, or ADO “classic” (ADOC) developers might be used to. In the strictest sense, a cursor is a device used to browse through the rows that result from a query. Yes, a DataTable might be thought of as a cursor in that it exposes a Rows collection that can be browsed in any order. But this is static client-side data—really no different from simply taking a DB-Library data stream and dumping it into a local array. While server-side cursors can also be static data (that does not change as the database is updated), server-side cursors can also be a fixed set of keys (a keyset cursor) or a dynamically changing set of keys (a dynamic cursor) that points to live data. That’s the critical difference. Server-side cursors leave the data on the server (or they can)—returning only “pointers” in the form of keys that point back to live data. As other users change the data, the cursor member keys still point to fresh data.

The ADO.NET architects and Microsoft product managers think that static, client-side DataTables are enough for many customers. They’re right—up to a point. I think that quite a few applications designed to use server-side cursors could have been written using client-side static cursors. However, when they are, developers are sometimes forced to make extra round trips to the server to detect changes in the data or membership. I think there are plenty of viable situations where a server-side cursor makes sense. I keep hearing that Microsoft was concerned that developers were “misusing” server-side cursors. That’s understandable, as the default cursor library (CursorType) in ADO is server-side cursors (adUseServer). In addition, too many examples illustrated fundamental operations using (often expensive) server-side cursors. I think that developers are smart enough to make up their own minds and server-side cursors can increase developer and code performance if used wisely.

Note: This approach to data access is not particularly interesting in stateless Web-based applications. It makes the most sense in connected Windows Forms applications.

One of the downsides to server-side cursors is the burden they impose on SQL Server. When you create a server-side cursor, SQL Server writes a set of keys or entire rows to memory or TempDB. This process is repeated and its effects multiplied for each application that opens a server-side cursor. Using server-side cursors also assumes that the connection remains open as long as the cursor’s data is needed. This is definitely not a good idea for a Web-based (ASP/ASPX) application because the connection is usually closed, and its resources released, just after its queries are executed. On the other hand, for many traditional client/server applications, server-side cursors have been an effective way to manage data and simplify application design.

Sidebar: Cursor “membership” is simply the set of rows that qualify for the rowset resulting from a SELECT statement based on the WHERE clause (if any). If there is no WHERE clause in the query, all of the rows in the target table(s) are made members of the cursor. However, in most cases, cursor membership is limited to just those rows that qualify for inclusion. For example, if you ran a query on a Used Car database looking for all 1957 Chevy’s with less than 10,000 miles, you would find very few cars—thus the cursor membership would have very few rows. Membership is fixed in static and keyset cursors. That is, once the query is executed and SQL Server finishes constructing the cursor, changes to the database that might add or remove rows from the membership are not considered. In a dynamic cursor, the engine repeats the query each time the cursor is accessed so new members are added and existing members are removed as SQL Server processes changes to the database.

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

In my latest ADO.NET book I mention (repeatedly) that ADO.NET cannot create a server-side cursor. Strictly speaking this is true. However, it's true only in the sense that it can't create *and manage* a server-side cursor on its own in the same way that DAO and ADO classic (ADOC) could. As I'll show you in this article, you can create server-side cursors using some fairly easy ADO.NET methods.

Caution: It's 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 TSQL.

This technique 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 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).

Opening, closing, and reopening the connection

Unlike other ADO.NET examples, in this case I need to open a connection to the SQL Server 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 flushed. This also means that you might want to reset the connection when the connection is closed. Consider that a Windows Forms application creates a new connection pool for each unique connection string. If you create a cursor, it *won't* be destroyed (at least not immediately) when you close the connection—assuming you're using connection pooling (which is on by default). However, if the connection reset option is enabled (it is by default), then the cursor will be cleared when the connection is opened again. If you turn off the reset connection option, the cursor *should* remain viable. However, if the connection pool times out the connection and forces the connection closed, your cursor will be lost.

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.

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

Creating the Cursor

The next step is to create the cursor based on an SQL statement. This block of code uses the SQL statement passed from the user in a TextBox control. (No, it's not a good idea to just accept anything from a user, but this is a demonstration and test program so I'm pretty generous with what I permit in the SQL.) To be safe(er), I call a ValidateSQL routine to do some rough tests to make sure I'm not too sleepy and enter something destructive.

Note that the SQL used to create the cursor can be broken down into three parts:

- 1) The DECLARE cursor statement, which names the cursor and sets the options I use. In this case, I ask for a scrollable cursor (as opposed to a forward-only cursor) and name it MyCursor. Check out SQL Server Books Online 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. This also means you can't call a stored procedure to fetch the rowset.
- 3) The last part is a SELECT to return the number of rows in the cursor. If you select too many rows, this will return -1 until SQL Server has completed server-side rowset population (until it has found all of the rows and added them to the cursor).

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"
```

Next, I execute the SQL 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.

```
    RowsInCursor = CInt(.ExecuteScalar())
End With
```

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. Since ADO.NET won't do it for you, this task is up to you. You'll be using the TSQL FETCH operator to retrieve a specific row (and only one row at a time) from 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 simply by using the appropriate operand. 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 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"
```

Fetching the entire cursor

The TSQL spec does not support the ability to fetch all of the rows in the cursor at once. However, I think this is useful when you want to display the members of the cursor and let the user choose one of them to update (yes, it's possible to update through the cursor). While it's possible to loop through multiple FETCH NEXT calls to get each row one-at-a-time, it's far faster to get the rows in blocks. This means that instead of one round trip for each of 100 cursor rows, I only execute ten round trips—fetching ten rows at a time. I set the number of FETCH operations to the number of expected rows in the cursor, but not greater than about 50 or so.

To carry off this approach, I simply build a long string to contain a set of FETCH NEXT operators. This is done using the StringBuilder class to reduce string handling overhead. In this case, sbFetchNext is declared as a new StringBuilder and intBatchSize is declared as 10. I depend on the Fetch All function to position to the first row prior to running this batch.

```
' Create a block of FETCH operators to use in the Fetch All routine
For i = 1 To intBatchSize
    sbFetchNext.Append(" FETCH NEXT FROM MyCursor ")
Next i
sbFetchNext.Append(" SELECT @@FETCH_STATUS AS FetchStatus")
```

Executing this batch of FETCH operators causes SQL Server to return eleven resultsets—one for each row in the cursor and one to contain the @@FETCH_STATUS. No, I could not use the Fill method for this (I tried), as it creates eleven DataTables and I only want two (one for the rows and one for the status). To deal with this problem, I used the DataReader instead.

```
Dim intFetchStatus, i As Integer
Dim bolRows As Boolean
' Fetch the first row into the DataSet/DataTable(0).
' We need it to get the rowset data structure
da.SelectCommand.CommandText = " FETCH FIRST FROM Mycursor"
ds.Clear()
da.Fill(ds)          ' Get first row
' Get remaining rows 10 at a time
Try
    cmd.CommandText = sbFetchNext.ToString
    intFetchStatus = 0
    Do ' Loop until there are no more rows in the cursor
        dr = cmd.ExecuteReader ' Execute query script
        bolRows = dr.Read()    This is not needed post 1.1 we can use HasRows
        If bolRows = False Then Exit Do ' No rows then step to next result
        If dr.FieldCount > 1 Then ' Is this the @@FetchStatus?
            intFetchStatus = MoveRowsToTable()
        End If
        dr.Close() ' Note, we DON'T want to close the connection here
    Loop While intFetchStatus = 0
' Show the rows in the DataGrid
```

```

    DataGrid1.PreferredColumnWidth = intPCW
    DataGrid1.DataSource = ds.Tables(0)
' Handle the exceptions
Catch ex As Exception
    MsgBox(ex.ToString)
End Try

```

The MoveRowsToTables function is responsible for dealing with each of the resultsets returned by the FETCH operators, as well as the last resultset containing the @@FETCH_STATUS. The function uses multi-tiered Do Loops to work through each of the row-bearing resultsets one-at-a-time by constructing a new DataRow object to hold the column values and appending the new DataRow to the DataSet Tables(0). Note that if the remaining cursor contains fewer rows than the number of FETCH operations (it will eventually), a number of empty (rowless) resultsets are returned. The function returns the @@FETCH_STATUS captured from the last resultset.

```

Function MoveRowsToTable() As Integer
Try
    Dim i As Integer
    Dim bolMoreResults, bolRows As Boolean
    Dim drow As DataRow
    Do ' Loop through all resultsets (each one has one row)
        drow = ds.Tables(0).NewRow ' Populate DataRow
        Do ' Loop through all columns of row filling in data
            For i = 0 To ds.Tables(0).Columns.Count - 1
                If dr.Item(i) Is DBNull.Value Then
                    drow.Item(i) = DBNull.Value
                Else
                    drow.Item(i) = dr.GetValue(i)
                End If
            Next i
            ds.Tables(0).Rows.Add(drow) ' Add row to DataTable
            bolRows = dr.Read ' Read next row
        Loop While bolRows ' Loop through all rows
        ' No more rows in the resultset
        ' Process next resultset
        bolMoreResults = dr.NextResult
        bolRows = dr.Read ' Read first row
        Do While bolRows = False ' Skip over any rowless resultsets
            bolMoreResults = dr.NextResult
            bolRows = dr.Read
        Loop
        If dr.FieldCount = 1 And dr.GetName(0) = "FetchStatus" Then
            Return dr.GetInt32(0)
            Exit Do
        End If
    Loop While bolMoreResults
Catch ex As Exception
    MsgBox(ex.ToString)
End Try
End Function

```

Displaying the rowsets

Once the data has been returned, I place it (one way or another) in a DataTable object. For the single-line FETCH operations, I use the Fill method and for the Fetch All routine, I use a DataReader.

```

Sub FetchAndShowData(ByVal intMovements As Movements)

```

```

Dim intFetchStatus As Integer
Try
    If cbKeep.Checked Then Else ds.Clear()
    da.Fill(ds)
        ' Retrieve the @@FETCH_STATUS from the second resultset
        ' placed in Table1 (Tables(1)).
lblFetchInfo.Text = ds.Tables(1).Rows(0).Item(0).ToString
intFetchStatus = CInt(lblFetchInfo.Text)

    If @@FETCH_STATUS returns -1, then the cursor positioning just executed has hit either
    BOF or EOF (beginning of file or end of file). If this is the case, I need to disable the
    button(s) that caused this and enable buttons to move in the other direction.

    If intFetchStatus = -1 Then
        Select Case intMovements
            Case Movements.Backward, Movements.First
                btnFetchPrevious.Enabled = False
                btnFetchNext.Enabled = True
            Case Movements.Forward, Movements.Last
                btnFetchNext.Enabled = False
                btnFetchPrevious.Enabled = True
        End Select
    Else ' Status returned 0
        Display the

        DataGrid1.PreferredColumnWidth = intPCW
        DataGrid1.DataSource = ds.Tables(0)
    End If ' No rows
Catch ex As Exception
    MsgBox(ex.ToString)
End Try
End Sub

```

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 are no longer available. However, as we learn to adapt, it seems that easy work-arounds for some of our tried-and-true approaches have appeared. This article discussed one such work-around. Keep your eye out for another article where I discuss how to create and hold a row on the server using a pessimistic lock. In other cases, our well-worn techniques might not have been such good ideas in the first place. The

remaining viable techniques will either be picked up in post-Visual Studio .NET 2003 versions or left behind. As I discover these workable alternatives, I'll share them with you here.