

Sorting, Seeking, Filtering, and Finding

William Vaughn

With disconnected architectures, you're supposed to manipulate your data without that expensive SQL database. Here's how it's done.

I've been asked quite a few questions this month having to do with "client-side" data - how to get it arranged in the right order, locate a row that matches a criterion, show a set of rows that match a criterion, and generally perform other operations that one would traditionally pass off to the server for processing. Since ADO.NET expects us to fly "fetch and destroy" missions (fetch the data and destroy the connection), we can't simply reconnect and requery just because the user wants the data sorted by price instead of by location. Yes, we could requery, but connecting (while cheap), certainly isn't free and, frankly, it's usually overkill to ask the DBMS engine to repeat the query just to change the order or membership of the rowset.

Fetching data from the Server

So where does "rowset membership" fit in? Well, "membership" is a term that typically applies to rows that qualify to be included in a "cursor", but ADO.NET doesn't (currently) support cursors as such—it simply returns one or more sets of data rows (a "rowset") which are persisted and managed through `DataTable` objects and the `DataRow`s collection. Translation: a row becomes a member of your rowset when the server decides that the conditions in the SQL `WHERE` clause have been met. As you probably already know, when you execute a SQL query, your `SELECT` statement tells the DBMS engine which columns you want from which tables.

Because you usually include a `WHERE` clause, your rowset's membership is limited to just those rows of the table that qualify. Sure, there are queries that return *all* of the columns and rows of a table—the pubs database's infamous "`SELECT * FROM Authors`" query, for example—but this approach won't fly in a production system that you expect to scale to (support) more than a few users. In a serious system, your `WHERE` clause will often be far more complex. Here's another SQL example - one that's slightly more complex but a lot more scalable:

```
SELECT CID, Customer, City, Zip, Discount
FROM Customers
WHERE State IN ('CA', 'TX')
AND Discount > 20
```

In this case, the `WHERE` clause filters *in* just those rows from the `Customers` table where the customer is from California or Texas and has been given a discount of more than 20 percent. Customers from other states are ignored along with Customers whose discount is less than or equal to 20 percent.

Limiting membership

While some exclusive golf clubs are under attack because of their policy to limit their membership, I haven't heard of a DBMS that cares if you limit the members of a query rowset—in fact, they invariably work more efficiently if you do. Since a DBMS query engine is designed to fetch data quickly based on SQL queries such as the one shown above, it's no chore for a serious DBMS engine to crank out thousands of these rowsets a minute. This means that it's not all *that* terrible to requery when you have to. After all, there's a chance that the data to be fetched might be in the DBMS engine's data cache by virtue of a previous query.

Data freshness

Another factor you need to consider is data "freshness". When you execute a query, the data rows aren't actually *moved* from the DBMS to client memory—they're *copied*. And because ADO.NET doesn't support server-side cursors (at least not directly), the data row copies passed to your application reflect the data state at the time they were read—data that may have been changed an instant later by any other application¹. Unless you requery, you won't get a fresh (current) copy of the data row. And no, there's no built-in DBMS mechanism to have the server automatically tell you when the data changes.²

As you probably know, when you use COM-based ADO "classic" (ADOC), its default `Recordset` `Keyset`, `Static` and `Dynamic` cursors are implemented as a server-side cursors - meaning that the data remains in the database and your application is given a set of pointers to the "live" rows. When you navigate to a specific row of the

¹ To prevent changes by other users is difficult—even if you use pessimistic locking which is only supported in ADO.NET through use of "repeatable read" transactions.

² Microsoft Notification Services can inform your application when data changes, but that's another story.

Recordset that's not cached locally, ADOc fetches a copy of the current data to the client. If you revisit the row later, you might find different data—if some other application changed it since you last visited the row.

As I see it, if you want to obtain the same behavior in ADO.NET, you have two choices. First, you can simply requery periodically to get fresh data. The problem with this approach is that the requery process is expensive and time-consuming. It requires connection reactivation, query generation and execution, another rowset fetch, and some additional network traffic. Your other option is to create a server-side cursor on your own, but that's the subject of another article³.

Fetching “enough” members

If the data membership includes the rows you're looking for and you aren't concerned with the age of the data, it's a better idea to simply filter out those rows that don't qualify. This assumes, of course, that you fetched them in the first place. And therein lies to proverbial rub. On the one hand, you don't want to fetch more data than the customer can use, but if you don't, you'll have to get it with a subsequent requery - often changing the focus of the WHERE clause. It's up to *you* to find a balance between fetching too many rows (and paying the price in initial fetch performance and system overhead) and or fetching too few rows (and paying the price in subsequent fetch overhead and not having relevant data in your rowset).

Getting help from the DataAdapter

The DataAdapter and the DataSet and DataTable objects it populates can help fetch associated or related rowsets. When you execute the DataAdapter Fill method against a DataSet, ADO.NET opens the connection, executes the query and proceeds to fetch the rowset into a new DataTable *or* into an *existing* DataTable if ADO.NET senses that the root (database) table is the same. This means you can execute a query for customers from Texas in an initial query, and from California at a later time using another query by simply executing Fill twice and varying the SelectCommand.CommandText SQL WHERE clause (or a suitable Parameter) before each Fill. In this case, you end up with a DataTable containing rows from both queries - which can be somewhat disconcerting in cases where you only expect to see data from the most recent query. In this case, you'll need to use the DataSet.Clear method to flush out the DataSet Tables collection.

Tip: Yes, one trick is to use a second DataAdapter to populate DataTables in a common DataSet. For example, the first DataAdapter could extract rows from the Customers table and another from the Addresses table. This way you have an update path to both tables while managing them in a common DataSet.

Managing data with the DataView

Okay, let's assume that you have a fairly large but relevant - and current - set of rows in a DataTable. Once the rowset is loaded, you can get ADO.NET to manage the data in a variety of ways to handle a litany of common tasks. Most of these techniques use the DataView object to perform their magic. Each DataTable exposes a DefaultView property that points to a DataView class that you can leverage to count the rows, sort them, or filter “in” specific rows based on one or more criteria. Table 1 summarizes the DataView properties you're most likely to use.

DataView Properties	Purpose
Count	How many rows are “visible” in the DataView after the RowFilter and RowStateFilter have been applied.
Item	Addresses a row of data from the specified DataView.
RowFilter	Hides rows that do not qualify for the RowFilter string expression.
RowStateFilter	Hides rows that do not qualify for the RowStateFilter string expression. The string must be one of the valid DataViewRowState values
Sort	Gets or sets the sort column or columns, and sort order for the DataTable .
Table	Gets or sets the source DataTable .

Table 1. DataView properties.

Setting the RowFilter property

Once the rowset is loaded into the DataTable, you can use the RowFilter to limit the membership in the DataView just as you can use a WHERE clause to limit the membership in the initial rowset. However, in this

³ See “Doing the Impossible with ADO.NET” <http://www.hardcorevisualbasic.com/vb/VBMag.nsf/Index/3C1068081F83D2E285256CD9006BFDB0?opendocument>

case, since we're working with a DataView, changes in the membership are not permanent. If you clear the filter, the original rowset membership is restored.

Note: None of these DataView properties or methods trigger a round-trip to the server to refine or filter the DataTable's rowset. All of these operations are carried out by ADO.NET in client memory without requiring a connection to the server.

In the sample application accompanying this article, I use the RowFilter property to limit the number of rows in the DataGrid. Figure 1 shows the result of setting the RowFilter property to show only those rows where the title begins with the letters "Ha". Behind the scenes, the application adds the expression preamble so the RowFilter is set to "Title Like 'Ha%'".

Figure 1: DataView RowFilter set to simple expression.

The RowFilter property can also be set using a variety of simple or compound expressions—very much like the ones you use in a SQL WHERE clause. (See online help for details on actual expression syntax.) Figure 2 shows use of the sample application's "General Filter" tab which is used to set the RowFilter property directly. In this case, I set the RowFilter property to "Title like 'Hitch%' and Price > 60" to further reduce the membership of the DataView to 4 rows. Here the user is responsible for correctly formatting the RowFilter expression - which means the user has to know how to write a SQL WHERE clause—single quotes and all. Most of the grannies that use my applications aren't quite up to that task (not yet at least), so I don't usually make them learn SQL to use my applications—I anticipate their queries and provide tabs like "Filter on Title" or "Filter on Column" in the sample application. Listing 1 is a snippet of code from the sample application.

Listing 1. Key code from the sample application.

```
Sub SetFilter(ByVal strFilterExpression As String)
' This routine applies the filter expression
' generated by the button_click events (above)
' to the DataTable's DefaultView RowFilter property
' It's easy to get this "WHERE" clause wrong
' so there's lots of error traps.
    Try ' Trap Filter exceptions
        DataGrid1.DataSource = objDb
        ' Restore persisted DataSource
        With ds.Tables(0)
            .DefaultView.RowFilter = strFilterExpression
            If .DefaultView.Count > 0 Then
                ' If the RowFilter returned rows,
                ' set the DataSource to the filtered DataView
                DataGrid1.DataSource = _
                    ds.Tables(0).DefaultView
            Else
                MsgBox("No titles meet criteria.")
            End If
        End With
        ' Deal with the exceptions
    Catch exSEE As SyntaxErrorException
        MsgBox(exSEE.Message & " ADO could not _
            evaluate the ... ")
    Catch exDEE As System.Data.EvaluateException
        MsgBox(exDEE.Message & " Remember to use _
```

```

        single quotes ...")
Catch exg As Exception      ' Trap Filter exceptions
    MsgBox(exg.ToString)
End Try
End Sub

```

Listing 1. Setting the DataView RowFilter property.

Setting the RowStateFilter Property

As rows are made members of a DataTable Rows collection, ADO.NET sets each row's RowState to "Unchanged". However, as you add, change or delete rows, ADO.NET sets the RowState value to "Added", "ModifiedOriginal", or "Deleted" based on the action taken to add, change, or delete the row. These enumerations are shown in Table 2. When it comes time to execute the DataAdapter Update method, ADO.NET uses the RowStateFilter property to determine which one of the three action commands to execute: UpdateCommand, InsertCommand, or DeleteCommand. If you want to view a subset of the rows in a DataTable Rows collection based on the RowState value, simply set the DataView RowStateFilter to one (or more) of the RowState enumerations as shown in Table 2.

Figure 2. DataView RowFilter set to complex expression.

RowState Enumeration	What appears in the DataView
Added	All new rows.
Deleted	All deleted rows.
ModifiedCurrent	All current version rows which are modified versions of original data (see ModifiedOriginal).
ModifiedOriginal	All original version rows (although the rows have since been modified and are available as ModifiedCurrent).
None	None (No rows should be exposed in the DataView)
CurrentRows	All unchanged, new, and modified rows.
OriginalRows	Only the original rows including unchanged and deleted rows.
Unchanged	Only unchanged rows.

Table 2: DataView RowState enumeration values

Using DataView Methods

The

DataView also exposes a couple of methods that you can use to locate specific rows. The Find method expects you to set the DataView Sort property beforehand as ADO.NET uses the Sort property to determine which column to search based on the Find property argument. You can also use the FindRows method to locate Sort property column matches, but in this case FindRows returns an array of DataRowView objects instead of an integer pointer to the first matching row. Both methods require you to use the Sort property to specify one or more columns (separated by commas) to match against, but you also have to provide a matching number of arguments using an array of objects passed to the Find method. The code in Listing 2 illustrates how easy this is. Sure, you can skip all of the array logic if you simply pass a single argument value to the Find or FindRows methods that match up with the Sort column specified. This simply means that if you specify several Sort property columns, you have to pass a like-sized array of object values to the first Find or FindRows method argument.

DataView Methods	Purpose
Find	Returns the index of a row that contains specified values in its sort key columns.
FindRows	Returns an array of DataRowView objects whose columns match the specified sort key value

Table 3. DataView class methods.

The Find method returns an integer which can be used to index into the DataView DataRowView collection or to move the "current row" pointer in the DataGrid to point to the matching row. The FindRows method returns an array of DataRowView objects. The DataRowView collection contains pointers to the DataTable Rows collection for each member of the DataView. Each DataRowView object can be used to extract the specific DataRow as the example in Listing 2 illustrates.

```
Private Sub btnFind_Click(ByVal sender As _
```

```

System.Object, ByVal e As System.EventArgs) _
Handles btnFind.Click
Private Sub btnFind_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnFind.Click
    ' The user clicked on the Find Tab, Find button.
    ' Locate a row in the grid based on the
    ' Sort column value(s) supplied in a string.
    Try
        Dim intRow, intArgs As Integer
        Dim dv As DataView = ds.Tables(0).DefaultView
        Dim dr As DataRow
        If txtFind2.Text = "" Then
            intArgs = 0
        Else
            intArgs = 1
        End If
        Dim objFindArg(intArgs) As Object
        objFindArg(0) = txtFind1.Text 'Get search args
        If txtFind2.Text = "" Then
            Else
                objFindArg(1) = txtFind2.Text ' One/argument
            End If
            ' Pass a valid search argument, get back -1
            ' (can't find it) or a 0 to the rowcount-1.
            ' Set the Sort prop. with Col(s) to search
            dv.Sort = txtSortCols.Text
            intRow = dv.Find(objFindArg) ' Has search arg
            If intRow < 0 Or intRow > dv.Count Then
                MsgBox("Row not found based on the _
                    criteria provided.")
            Else
                dr = dv(i).Row ' Return the matching row
                ' Reset the DataControl's row pointer
                ' to the corresponding row in the DataTable.
                DataGrid1.CurrentRowIndex = intRow
            End If
        Catch exFE As FormatException
            MsgBox(exFE.Message & " ...Find _
                argument setting. " & vbCrLf & _
                txtFind1.Text & ", " & txtFind2.Text)
        Catch ex As Exception
            MsgBox(ex.ToString)
        End Try
    End Sub

```

Listing 2. Using the Find method with multiple arguments.

What happened to FindNext?

Well, there's no ADO.NET "FindNext" method as there is in ADO classic—but there's an easy workaround. One approach would be to code the RowFilter method to limit the number of members in the DataView just as you would in ADO classic's Recordset Find method. Get the DataView.Count property to determine the upper limit of the rows in the DataView. Next, to step through each "found" row, increment through the DataRowView collection exposed by the DataView using an integer ordinal between 0 and the DataView.Count.

Conclusion

I hope that this article will make it easier to find your rows if you lost them. No, I don't expect it can help much if you've lost your marbles, but if you've lost your temper working with ADO.NET as you tried to find replacements for ADO classic functionality, I'm sure I helped—at least to some extent.