# ASP Speed Tricks

## Table of Contents


- Table of Contents
- Synopsis
- Introduction
- Optimizing the Display of Simple Tables
    - Concatenating Database Output Into a String
    - Eliminating Concatenation From the Loop
    - Better Performance with Recordset Field References
    - Improving Browser Rendering Speed Using Fixed Size Tables
    - Using Minimal Formatting with the PRE tag
    - Using the GetString Recordset Function
    - Using the GetRows Recordset Function
    - Using a Native OLE DB Provider Instead of ODBC
    - Summary of Simple Table Techniques
- Optimizing the Display of Complex Tables
    - Introduction
    - Avoiding JOIN and SHAPE in Complex Tables
    - Creating a New Recordset on Each Iteration
    - Using a Prepared Connection and Recordset Object
    - Using a Prepared Command Object to Obtain a Recordset
    - PDF-Only Bonus Content
    - Summary of Complex Table Techniques
- Appendix
    - Testing Notes
    - Books
    - Acknowledgements


## Synopsis


**This article describes practical methods of optimizing the performance of ASP pages which retrieve and display tabular data from a database. Test results of each coding technique show the potential for dramatic speedups of dynamic web pages.**


## Introduction

The intended audience is intermediate to advanced ASP programmers. Beginners may read

this article and use the references to look up unfamiliar code. These techniques were developed for ASP 3.0, but should apply to ASP.Net as well. This is not intended to be a complete compendium of optimizations, but points out a few that result in great performance improvements for display of read-only data.

ADO is used to query the data, and the underlying database may be MySQL, SQL Server, Access, Oracle, or any other major RDBMS. Unlike many other articles, complete examples are presented, which may be copied into a file and run directly. Techniques for optimizing the output of tabular data are presented first, followed by techniques for optimizing more complex tabular data typical of reports. The appendix contains instructions for configuring the tests, links to references, and books suggestions.

# Optimizing the Display of Simple Tables

## Concatenating Database Output Into a String

Microsoft's ASP technology enables beginners to write dynamic web pages with little effort. The ADO object model hides the complexity of obtaining data from the database. However, hiding complexity under a simple interface also allows unsuspecting programmers to write wildly inefficient code. Consider the common task of querying the database and displaying the results in an HTML table.

One of the slowest methods is to loop through the recordset, and concatenate each row into a string. Once the loop is complete, the string is written to the response. Many novices may apply this technique due to its logical simplicity, or by following the bad example of others. However, for anything but very small data sets, this technique is highly innefficient. The next code example shows how this technique might be used.

**SIMPLETABLE1.ASP**

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string
Dim strTemp ' a temporary string

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

' Prepare a SQL query string
strsql = "SELECT * FROM tblData"
```

```
' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Write out the results in a table by concatenating into a string
Response.write "<table>"

Do While Not objRS.EOF
    strTemp = strTemp & "<tr><td>" & objRS("field1") & "</td>"
    strTemp = strTemp & "<td>" & objRS("field2") & "</td>"
    strTemp = strTemp & "<td>" & objRS("field3") & "</td>"
    strTemp = strTemp & "<td>" & objRS("field4") & "</td></tr>"
    objRS.MoveNext
Loop
Response.write strTemp
Response.write "</table>"

Set objCN = Nothing
Set objRS = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | |
|---|---|
| **Records** | **Time** |
| 1000 | 3.5 seconds |
| 2000 | 18.4 seconds |
| 10000 | 7.5 minutes (est.) |
| 20000 | 30 minutes (est.) |

The server processing time to display 1000 records from the table is about 3.5 seconds. Doubling the number of records to 2000 more than quadruples the time to 18.4 seconds. The script times out for the other tests, but some time estimates are given. In the code, the '&' concatenation operator is used heavily within the loop.

Concatenation in VBScript requires new memory to be allocated and the entire string to be copied. If the concatenation is *accumulating* in a single string, then an increasingly long string must be copied on each iteration. This is why the time increases as the square of the number of records. Therefore, the first optimization technique is to avoid accumulating the database results into a string.

## Eliminating Concatenation From the Loop

Concatenation may be removed easily by using Response.write directly in the loop. (In ASP.Net, the StringBuilder class can be used for creating long strings, but Response.write is fastest.) By eliminating accumulation, the processing time becomes proportional to the number of records being printed, rather than being exponential.

Each use of the concatenation operator results in unnecessary memory copying. With larger recordsets or high-load servers, this time can become significant. Therefore, instead of concatenating, programmers should simply write out the data with liberal use of Response.write. The code snippet below shows that even a few non-accumulative concatenations cause a noticeable time difference when run repeatedly.

```
' Using concatenation in a loop takes 1.93 seconds.
For i = 0 To 500000
        Response.write vbTab & "foo" & vbCrLf
Next
' Using multiple Response.write calls takes 1.62 seconds.
For i = 0 To 500000
        Response.write vbTab
        Response.write "foo"
        Response.write vbCrLf
Next
```

The following example eliminates accumulative concatenation from the loop and replaces it with direct calls to Response.write.

## SIMPLETABLE2.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

' Prepare a SQL query string
strsql = "SELECT * FROM tblData"

' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Write out the results directly without using concatenation operator
Response.write "<table>"
Do While Not objRS.EOF
    Response.write "<tr><td>"
    Response.write objRS("field1")
    Response.write "</td><td>"
    Response.write objRS("field2")
    Response.write "</td><td>"
    Response.write objRS("field3")
    Response.write "</td><td>"
    Response.write objRS("field4")
    Response.write "</td></tr>"
    objRS.MoveNext
Loop
Response.write "</table>"

objRS.Close
```

```
objCN.Close
Set objCN = Nothing
Set objRS = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | |
|---|---|
| **Records** | **Seconds** |
| 1000 | 0.145 |
| 2000 | 0.260 |
| 10000 | 0.980 |
| 20000 | 1.950 |

For 1000 records, this method runs an incredible 23 times faster. For more records, the difference is even greater. The processing time is now roughly proportional to the number of records being printed. This property is essential for large record sets.

## Better Performance with Recordset Field References

In the previous examples, the value of the field was retrieved from the Recordset object by specifying the field name directly. The Recordset's Fields collection supports this useful property to provide easy access to the fields. However, referencing the field value by using the textual field name causes a relatively slow string lookup to be performed in the Fields collection.

To avoid the string lookup, we could instead use a numeric index into the Fields collection, such as objRS(0), objRS(1), objRS(2), etc. But even better performance can be gained by saving a *pointer* to each field in the Recordset right after it is opened. This way, instead of looking up the field value using a string or number, direct access to the field value is obtained. When VBScript sees the pointer in the string context of the Response.write, it can quickly obtain the data from the field and convert it to a string. The next example shows how this can be done.

### SIMPLETABLE3.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string
Dim objField0, objField1, objField2, objField3

' Create a connection object
```

```
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

' Prepare a SQL query string
strsql = "SELECT Field1,Field2,Field3,Field4 FROM tblData"

' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Set up field references after opening recordset
Set objField0 = objRS(0)
Set objField1 = objRS(1)
Set objField2 = objRS(2)
Set objField3 = objRS(3)

' Write out the results using the field references
Response.write "<table>"
Do While Not objRS.EOF
    Response.write "<tr><td>"
    Response.write objField0
    Response.write "</td><td>"
    Response.write objField1
    Response.write "</td><td>"
    Response.write objField2
    Response.write "</td><td>"
    Response.write objField3
    Response.write "</td></tr>"
    objRS.MoveNext
Loop
Response.write "</table>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | |
|---|---|
| **Records** | **Seconds** |
| 1000 | 0.105 |
| 2000 | 0.190 |
| 10000 | 0.665 |
| 20000 | 1.350 |

This technique requires declaring each field as a variable, and then setting the pointers after the Recordset is opened. The need for these additional lines of code is a drawback to this technique. The benefit is a 30-40% speedup on this test.

## Improving Browser Rendering Speed Using Fixed Size Tables

While running the above example on Internet Explorer for 20,000 records, one may notice that the server finishes processing within two seconds, but the page does not display until much later. On the test system, the page displayed 12 seconds after the refresh button was pressed. For those 12 seconds, the browser window remained blank.

This delay is not due to the server, but it is due to the *browser rendering speed* on the client machine. When a web browser receives this huge data set, it must figure out a way to display it on the user's screen. This means calculating the width of the table columns and height of its rows. In fact, the browser has to receive and process all the data before it can finalize the layout of the table. On low bandwidth connections, the problem is compounded because the user must additionally wait for the data to be transferred.

Every browser takes a different amount of time to render a given page, but there are techniques that can be used to help the browser along. The goal is to give the browser as few calculations as possible. One minor technique is to print a new line (vbCrLf) after every 256 characters or so. This seems to help older browsers in particular.

The major technique is to make the table's columns fixed width, thereby eliminating the need for the browser to calculate the table's column widths. In fact, by making the table fixed width, *the browser can begin displaying the table before all the data has been received.* First, set the table style to "table-layout: fixed;". Second, right after the <table> tag, add a <colgroup> tag and define the column widths for each column with <col> tags. The next example shows this method.

## SIMPLETABLE4.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string
Dim objField0, objField1, objField2, objField3

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

' Prepare a SQL query string
strsql = "SELECT Field1,Field2,Field3,Field4 FROM tblData"

' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Set up field references after opening recordset
Set objField0 = objRS(0)
Set objField1 = objRS(1)
Set objField2 = objRS(2)
Set objField3 = objRS(3)
```

```
' Write out the results in a fixed size table
Response.write "<table style="&Chr(34)&"table-layout: fixed;"&Chr(34)&">"
Response.write "<colgroup>"
Response.write "<col width=100><col width=100><col width=100><col width=100>"
Response.write "</colgroup>"
Do While Not objRS.EOF
    Response.write "<tr><td>"
    Response.write objField0
    Response.write "</td><td>"
    Response.write objField1
    Response.write "</td><td>"
    Response.write objField2
    Response.write "</td><td>"
    Response.write objField3
    Response.write "</td></tr>"
    Response.write vbCrLf
    objRS.MoveNext
Loop
Response.write "</table>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

When this version of the code is refreshed, the top part of the table displays in about 3 seconds. The page is not scrollable, and the browser is still receiving and rendering the remaining data, but at least the user can begin reading the first part of table. The page completes loading in the same amount of time as before, about 12 seconds. While the *actual* speed is not faster, the *perceived* speed is significantly faster.

Note that different browsers render tables in different ways. For example, Mozilla actually renders the top part of the table before receiving the entire table, even if the table is not fixed in size. The Mozilla Layout Engine, on which Netscape 6+ is based, then resizes the columns again on the fly once all the data has been received. Rendering time is highly dependent upon browser choice, processor speed, memory speed and amount, and graphics card speed, which are aspects of the client's system. A web page programmer generally has no direct control over the client's system, but can generate HTML which requires less power to render.

## Using Minimal Formatting with the PRE tag

Rendering time can be drastically improved by eliminating the table and using pre-formatted text. Along the same lines as before, the trade-off is some loss in formatting flexibility for a gain in raw speed. The <PRE> tag can be used for text that displays in a fixed-width font using the formatting of the source code, like all the code examples in this document. Because there is no text wrapping, column sizing, or variable character sizes, the browser has minimal amounts of calculations to make for rendering.

To implement this, simply replace the table tags with the pre-formatted text tag, and then replace the table cell bounding tags with tabs and new lines. An additional benefit of this technique is that the raw amount of data is less because tabs and new lines are shorter than table cell bounding tags like <td> and <tr>. (It would be possible to do better formatting by using the VBScript Space and Len functions to place text in padded, fixed character width columns, but this is beyond the scope of this article.) With 20,000 records, the code example below rendered entirely within three seconds, which is at least four times faster than the fixed-width table formatted data.

## SIMPLETABLE5.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string
Dim objField0, objField1, objField2, objField3

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

' Prepare a SQL query string
strsql = "SELECT Field1,Field2,Field3,Field4 FROM tblData"

' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Set up field references after opening recordset
Set objField0 = objRS(0)
Set objField1 = objRS(1)
Set objField2 = objRS(2)
Set objField3 = objRS(3)

' Write out the results as pre-formatted text
Response.write "<pre>"
Do While Not objRS.EOF
    Response.write objField0
    Response.write vbTab
    Response.write objField1
    Response.write vbTab
    Response.write objField2
    Response.write vbTab
    Response.write objField3
    Response.write vbTab
    Response.write vbCrLf
    objRS.MoveNext
Loop
Response.write "</pre>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing
```

```
EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

As an aside, another way to reduce raw data size of the HTML is to simply eliminate the table row and column ending tags, </td> and </tr> This technique saves a signficant amount of space in a long query, and is also valid HTML according to the W3C specification. On the other hand, some older browsers choke if these ending tags are excluded. For newer applications in controlled settings, the end tags can be safely excluded. Removing these tags saves bandwidth and transfer time, but otherwise does not affect rendering time versus an ordinary table.

## Using the GetString Recordset Function

The Recordset object has two methods that allow retrieving data quickly, without requiring any looping. GetString returns a string from the Recordset, while GetRows returns an array.

Both functions can be used to quickly copy the RecordSet data into the web server's memory and then disconnect from the database server. This can improve performance where database scalability is an issue. GetString is considered first in order to develop a fast technique for using these functions.

The code snippets below show three ways of generating the same information. The first method is the "Optimized Looping" method used in the previous example. "Optimized Looping" is defined simply as looping with field references and eschewing the concatenation operator. The second method, "Full GetString", uses a single call to GetString to return and print the entire Recordset at once. The third method, "Partial GetString" uses a loop with multiple calls to GetString that return and print a small part of the Recordset on each iteration. Shown below are the code snippets and test results. The fastest times are highlighted.

```
' Optimized Looping (field references, no concatenation)
Response.write "<pre>"
Do While Not objRS.EOF
        Response.write objField0
        Response.write vbTab
        Response.write objField1
        Response.write vbTab
        Response.write objField2
        Response.write vbTab
        Response.write objField3
        Response.write vbTab
        Response.write vbCrLf
        objRS.MoveNext
Loop
Response.write "</pre>"
' Full GetString
Response.write "<pre>"
Response.write objRS.GetString(,,vbTab,vbCrLf)
Response.write "</pre>"
' Looping with Partial GetString (30 records per iteration)
Response.write "<pre>"
Do While Not objRS.EOF
```

```
        Response.write objRS.GetString(2,30,vbTab,vbCrLf)
Loop
Response.write "</pre>"
```

| Test Results | | | | |
|---|---|---|---|---|
| **Records** | **KBytes** | **Optimized Looping (s.)** | **Full GetString (s.)** | **Partial GetString (s.)** |
| 1000 | 25 | 0.10 | 0.08 | 0.08 |
| 2000 | 52 | 0.17 | 0.13 | 0.12 |
| 10000 | 262 | 0.67 | 0.50 | 0.39 |
| 20000 | 536 | 1.25 | 2.13 | 0.74 |

KBytes is the length of the generated HTML file, which is almost entirely the data from the Recordset.

First, the bad news. GetString is a non-intuitive way to print a table of data, requiring remembering the usage and order of the five parameters. The code for GetString is also much more condensed, making it harder to read. And finally, GetString joins every row and column using a fixed format, making customized formatting of the results difficult to impossible. Therefore, the downsides of GetString are that it decreases code maintainability, and it reduces the flexibility of formatting the data in HTML.

The upside, as you might have guessed, is that GetString is about 20-40% faster than "Optimized Looping". Using a single call to GetString is faster than "Optimized Looping" up until 20,000 records, at which point this method takes a significant performance hit. This hit could be explained by the need for the server to allocate additional memory for the lengthy 536KB string. The Pentium III processor that ran these tests has a 256 KB cache, and this is possibly why the performance hit occurs after 262 KB, when the processor may be forced to go to slower system memory.

In any case, this performance hit can be avoided and greater performance achieved by issuing multiple calls to GetString, each returning a small part of the Recordset. The second parameter of GetString limits the number of rows returned. By placing such a function call within a loop, the entire Recordset can be printed. The number of records per call in this test was thirty, and setting it to one hundred increased performance slightly. The optimal setting would vary based on the dataset and system specifications. The code using GetString is shown below.

## SIMPLETABLE6.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
```

```
Dim strsql ' SQL query string

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

' Prepare a SQL query string
strsql = "SELECT Field1,Field2,Field3,Field4 FROM tblData"

' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Write out the results using GetString in a loop
Response.write "<pre>"
Do While Not objRS.EOF
    Response.write objRS.GetString(2,30,vbTab,vbCrLf)
Loop
Response.write "</pre>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

## Using the GetRows Recordset Function

While GetString is a very fast way to print data from a Recordset, it suffers from a loss of code maintainability and formatting flexibility. It turns out using GetRows in the same way is nearly as fast, but has the added benefit of allowing ease of code maintainance and unlimited formatting flexibility. The following example replaces GetString with GetRows, and retrieves the array into a temporary variable. The array is then printed out in a simple loop.

**SIMPLETABLE7.ASP**

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string
Dim RecordsArray ' To hold the Array returned by GetRows
Dim i ' A counter variable

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
```

```
objCN.Open

' Prepare a SQL query string
strsql = "SELECT Field1,Field2,Field3,Field4 FROM tblData"

' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Write out the results using GetRows in a loop
Response.write "<pre>"
Do While Not objRS.EOF
    RecordsArray = objRS.GetRows(30)

    ' Print out the array
    For i = 0 To UBound(RecordsArray, 2)
        Response.write RecordsArray(0, i)
        Response.write vbTab
        Response.write RecordsArray(1, i)
        Response.write vbTab
        Response.write RecordsArray(2, i)
        Response.write vbTab
        Response.write RecordsArray(3, i)
        Response.write vbTab
        Response.write vbCrLf
    Next
Loop
Response.write "</pre>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | | |
|---|---|---|
| **Records** | **Partial GetString (s.)** | **Partial GetRows (s.)** |
| 1000 | 0.08 | 0.08 |
| 2000 | 0.12 | 0.12 |
| 10000 | 0.39 | 0.43 |
| 20000 | 0.74 | 0.82 |

The test results show that the GetRows technique is only about 10% slower than the fastest technique of using GetString. On the other hand, GetRows has the advantage that the code is easier to read and it allows any kind of additional formatting of the returned data. This benefit is achieved without the additional lines of code required by "Optimized Looping" for declaring and setting field references.

## Using a Native OLE DB Provider Instead of ODBC

The examples above use the Microsoft Access Driver for ODBC. The ODBC driver is generally slower than using a native OLE DB driver. It also has some differences in capabilities

and SQL syntax, which are described in the documentation. In order to use the OLE DB driver, you need to specify the OLE DB Provider for your database in your connection string. The connection string syntax is different for each provider, making it hard to remember. To create a connection string the easy way, follow these steps:

1. Use Notepad and save an empty file to your desktop called "mydsn.udl".
   (This file may be called anything but must have a .udl extension.)
2. Double click on the file and use the panels to configure your data source.
3. Use Notepad to open the "mydsn.udl" file, and copy the connection string to ASP.
4. Instead of step 3, use "File Name=c:\windows\desktop\mydsn.udl" as your connection string.

```
Using a native OLE DB Provider is generally faster than going through ODBC.
ADO <--> OLE DB Provider for ODBC  <--> ODBC <--> Your Database
ADO <--> Native OLE DB Provider <--> Your Database
```

The following test uses the previous "Partial GetRows" example and simply replaces "DSN=datasource" with a Native OLE DB connection string for the Jet 4.0 OLE DB Provider. This connects to the identical database file test.mdb. As you can see, the native OLE DB Provider is about 10-20% faster for the simple select query being tested.

## SIMPLETABLE8.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string
Dim RecordsArray ' To hold the Array returned by GetRows
Dim i ' A counter variable

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source using the native OLE DB Provider for Jet
objCN.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;"&_
    "Data Source=C:\Inetpub\wwwroot\data\test.mdb;"&_
    "Persist Security Info=False"
objCN.Open

' Prepare a SQL query string
strsql = "SELECT Field1,Field2,Field3,Field4 FROM tblData"

' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Write out the results using GetRows in a loop
Response.write "<pre>"
Do While Not objRS.EOF
    RecordsArray = objRS.GetRows(30)

    For i = 0 To UBound(RecordsArray, 2)
```

```
        Response.write RecordsArray(0, i)
        Response.write vbTab
        Response.write RecordsArray(1, i)
        Response.write vbTab
        Response.write RecordsArray(2, i)
        Response.write vbTab
        Response.write RecordsArray(3, i)
        Response.write vbTab
        Response.write vbCrLf
    Next
Loop
Response.write "</pre>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | | |
|---|---|---|
| **Records** | **ODBC (s.)** | **Jet OLE DB (s.)** |
| 1000 | 0.08 | 0.07 |
| 2000 | 0.12 | 0.10 |
| 10000 | 0.43 | 0.35 |
| 20000 | 0.82 | 0.65 |

## Summary of Simple Table Techniques

The seven techniques described have reduced the server processing time of the ASP page displaying 1000 records from 3.5 seconds down to 0.07 seconds. The client-side rendering time was also drastically reduced. Each technique increases performance at the cost of coding time, formatting flexibility, and code maintainability. The last method, "Partial GetRows", may cost the least for the amount of performance benefit it yields, because it is simple to code and maintain and has unlimited formatting flexibility. The seven techniques are listed below.

1. Replace the '&' concatenation operator with liberal use of 'Response.write' within the loop.
2. Set up pointers or references to the Recordset's fields and use these for printing the record data values.
3. Make the table and table columns fixed-width. Break lines with 'vbCrLf' after every 256 characters or so.
4. Eliminate the table completely, and instead use pre-formatted text.
5. Use GetString in a Loop
6. Use GetRows in a Loop
7. Use a Native OLE DB provider instead of ODBC

# Optimizing the Display of Complex Tables

## Introduction

A common task of ASP pages is to display complex reports or tables. This section generalizes one aspect of the problem and then provides techniques for increasing performance. Finally, test results of different methods and Providers is shown, and an optimal technique presented.

A complex table is defined here as a table that requires at least one sub-query to be performed for each record of a primary query. For example, suppose we have a table of public companies and a table of the officers of the public companies. If we wanted to display a table of public companies and their officers, we would have to display it in a complex report. An example of such a table is below.

**Public Companies and Their Officers in June 2003**

| Company Name | Stock Symbol | Exchange | Industry |
|---|---|---|---|
| Wal-Mart Stores, Inc. | WMT | NYSE | Retail |

| Officer Name | Officer Title |
|---|---|
| S. Robson Walton | Chairman of the board |
| David D. Glass | Chairman, executive committee of the board |
| H. Lee Scott | President and Chief Executive Officer |
| Thomas M. Coughlin | Vice Chairman |
| John B. Menzer | President and Chief Executive Officer International |

| Company Name | Stock Symbol | Exchange | Industry |
|---|---|---|---|
| General Electric Company | GE | NYSE | Conglomerates |

| Officer Name | Officer Title |
|---|---|
| Jeffrey R. Immelt | Chairman of the Board and CEO |
| Gary L. Rogers | Vice Chairman of the Board and Executive Officer |
| Robert A. Jeffe | Senior Vice President, Corporate Business Development |
| Keith S. Sherin | Senior Vice President, Finance and Chief Financial Officer |
| Ben W. Heineman, Jr. | Senior Vice President, General Counsel and Secretary |

The algorithm for printing this table involves querying the list of public companies, and then querying the list of officers for each public company. The companies may be thought of as the *primary records*, and the officers the *secondary records*. A complex table is defined as having this *primary-secondary heirarchy*, or nested queries. In practice, there may be more than two levels, and each level could have multiple queries.

## Avoiding JOIN and SHAPE in Complex Tables

Technically, it is possible to use a single query to retrieve all the necessary information for

both the companies and officers. This can be accomplished simply by joining the two tables together. The drawback of this method is that the *company information is repeated on each row of the record set.* If the company information is substantial, or the number of officers is large, this can have a significant performance hit. The perfomance hit will become prohibitive if there are multiple sub queries to be performed, such as pulling a list of the company's board members as well as officers.

Another method to avoid is the Microsoft SQL SHAPE clause. The SHAPE clause allows nesting sub-queries into a single query that returns a multi-dimensional Recordset. This method is to be avoided because it can make the initial query huge, awkward, and unmaintainable. Secondly, the SHAPE clause is not part of the SQL99 standard and is not widely supported by database engines. Finally, the SHAPE syntax is relatively advanced and obscure to beginners.

Therefore, it is most flexible and easiest logically to perform the main query, and then perform sub queries for each row of the main query. As before, the methods of doing this are described from slowest to fastest. These techniques build upon the previous techniques for simple tables. In the following examples, rather than printing out the multiple records of a sub-query, the sub-query will be a dummy query that returns a single result. This is because the simple table techniques have already shown ways to optimize printing the records. The intent is to optimize the method of issuing the sub-query from ADO, not to optimize the sub-query itself.

## Creating a New Recordset on Each Iteration

This first example shows a straight-forward method of performing the sub-query. The example builds on the method of using GetRows in a loop, but reverts back to the ODBC driver. A new recordset is created and initialized, used to perform the sub-query, and then destroyed. All these steps are performed right from within the loop.

**COMPLEXTABLE1.ASP**

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS, objRS2 ' ADO Recordset objects
Dim strsql ' SQL query string
Dim RecordsArray, i

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

' Prepare a SQL query string
strsql = "SELECT * FROM tblData"
```

```
' Execute the SQL query and set the implicitly created recordset
Set objRS = objCN.Execute(strsql)

' Write out the results using GetRows in a loop
Response.write "<pre>"
Do While Not objRS.EOF
    RecordsArray = objRS.GetRows(30)

    For i = 0 To UBound(RecordsArray, 2)
        Response.write RecordsArray(0, i)
        Response.write vbTab
        Response.write RecordsArray(1, i)
        Response.write vbTab
        Response.write RecordsArray(2, i)
        Response.write vbTab
        Response.write RecordsArray(3, i)
        Response.write vbTab
        Response.write vbCrLf

        ' Issue a dummy query and write out the result
        Set objRS2 = Server.CreateObject("ADODB.RecordSet")
        strsql = "SELECT COUNT(*) FROM tblData WHERE Field1="&RecordsArray(0, i)
        objRS2.Open strsql, "DSN=datasource", adOpenForwardOnly, adLockReadOnly
        Response.write "Dummy query result="
        Response.write objRS2(0)
        objRS2.Close
        Set objRS2 = Nothing
        Response.write vbCrLf
    Next
Loop
Response.write "</pre>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | |
|---|---|
| **Records** | **Seconds** |
| 1000 | 13.5 |

Simply adding a single query in the loop causes the time to jump from 0.08 seconds to over 13 seconds. One might think that the sub-query is a very slow query, but this is not the case. There are two non-obvious problems with this code, which are discussed in the following sections.

## Using a Prepared Connection and Recordset Object

A minor problem is that a new Recordset object is being created on each iteration. This can be fixed easily by initializing the Recordset once before the loop, and destroying it after the loop

The first major problem is that *a new database connection is being implicitly created and opened on each iteration*. This automatic creation is provided by ADO as a programming

convenience, but it sacrifices performance when used inappropriately. Once this problem is realized, the solution is obvious. As the following example shows, a Connection object is explicitly created and initialized before the loop. The Recordset performing the sub-query is then set to use this already opened Connection object. In fact, both the primary and secondary recordsets are set to use the same Connection object. The code follows.

## COMPLEXTABLE2.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string
Dim objRS2 ' Another ADO Recordset object
Dim RecordsArray, i

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

' Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

' Create the a recordset object, and initialize it
Set objRS = Server.CreateObject("ADODB.RecordSet")
With objRS
    .CursorType = adOpenForwardOnly
    .LockType = adLockReadOnly
    .CursorLocation = adUseServer
    Set .ActiveConnection = objCN
End With
' Create the second recordset object, and initialize it
Set objRS2 = Server.CreateObject("ADODB.RecordSet")
With objRS2
    .CursorType = adOpenForwardOnly
    .LockType = adLockReadOnly
    .CursorLocation = adUseServer
    Set .ActiveConnection = objCN
End With

' Execute the SQL query
objRS.Open "SELECT Field1,Field2,Field3,Field4 FROM tblData"

' Write out the results using GetRows in a loop
Response.write "<pre>"
Do While Not objRS.EOF
    RecordsArray = objRS.GetRows(30)

    For i = 0 To UBound(RecordsArray, 2)
        Response.write RecordsArray(0, i)
        Response.write vbTab
        Response.write RecordsArray(1, i)
        Response.write vbTab
        Response.write RecordsArray(2, i)
        Response.write vbTab
        Response.write RecordsArray(3, i)
        Response.write vbTab
        Response.write vbCrLf
```

```
        ' Use the pre-prepared Recordset object to issue the dummy query
        strsql = "SELECT COUNT(*) FROM tblData WHERE Field1="&RecordsArray(0, i)
        objRS2.Open strsql
        Response.write "Dummy query result="
        Response.write objRS2(0)
        objRS2.Close
        Response.write vbCrLf
    Next
Loop
Response.write "</pre>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing
Set objRS2 = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | |
|---|---|
| **Records** | **Seconds** |
| 1000 | 4.1 |

This simple change reduces the processing time down to about four seconds, for a three-fold improvement. Along with using an already opened Connection object, the Recordset properties are set for highest performance, i.e. using a forward-only cursor, read-only lock, and a server-side cursor. (A further discussion of cursors is outside the scope of this article.)

## Using a Prepared Command Object to Obtain a Recordset

The second major problem is that the "objRS2.Open" call is *implicitly creating a Command object on each iteration*. This may be eliminated by setting up a parameterized Command object in advance. To create a parameterized query, a question mark '?' is used in place of each parameter in the SQL statement. Then, the Command object's CreateParameter and Append functions set up the parameters. By setting the Command object's prepared property to True, the query's execution path only needs to be calculated once by the database engine. The dummy query in this test has a single input parameter in its WHERE clause, as shown in the next example.

### COMPLEXTABLE3.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
```

```
Dim strsql ’ SQL query string
Dim objRS2 ’ Another ADO Recordset object
Dim objCmd ’ ADO Command object
Dim RecordsArray, i

’ Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")

’ Connect to the data source
objCN.ConnectionString = "DSN=datasource"
objCN.Open

’ Create the a recordset object, and initialize it
Set objRS = Server.CreateObject("ADODB.RecordSet")
With objRS
    .CursorType = adOpenForwardOnly
    .LockType = adLockReadOnly
    .ActiveConnection = objCN
    .CursorLocation = adUseServer
    .Source = "SELECT Field1,Field2,Field3,Field4 FROM tblData"
End With
’ Create the second recordset object, and initialize it
Set objRS2 = Server.CreateObject("ADODB.RecordSet")
With objRS2
    .CursorType = adOpenForwardOnly
    .LockType = adLockReadOnly
    .CursorLocation = adUseServer
End With
’ Create command object
Set objCmd = Server.CreateObject("ADODB.Command")
With objCmd
    .ActiveConnection = objCN
    .CommandType = adCmdText
    .Prepared = True
    .CommandText = "SELECT COUNT(*) FROM tblData WHERE Field1=?"
End With
’ Create unnamed Parameter and append it to Parameters collection
objCmd.Parameters.Append _
    objCmd.CreateParameter(,adInteger,adParamInput,4)

’ Execute the SQL query
objRS.Open


’ Write out the results using GetRows in a loop
Response.write "<pre>"
Do While Not objRS.EOF
    RecordsArray = objRS.GetRows(30)

    For i = 0 To UBound(RecordsArray, 2)
        Response.write RecordsArray(0, i)
        Response.write vbTab
        Response.write RecordsArray(1, i)
        Response.write vbTab
        Response.write RecordsArray(2, i)
        Response.write vbTab
        Response.write RecordsArray(3, i)
        Response.write vbTab
        Response.write vbCrLf

        ’ Use prepared Command and Recordset to issue dummy query
        ’ Set the parameter for this iteration
        objCmd(0) = RecordsArray(0, i)
        ’ Run the prepared query
        objRS2.Open objCmd
        Response.write "Dummy query result="
        Response.write objRS2(0)
        objRS2.Close
```

```
        Response.write vbCrLf
    Next
Loop
Response.write "</pre>"

objRS.Close
objCN.Close
Set objCN = Nothing
Set objRS = Nothing
Set objRS2 = Nothing
Set objCmd = Nothing

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | |
|---|---|
| **Records** | **Seconds** |
| 1000 | 1.1 |

Being able to set the Prepared property of the Command object is the step that makes the nearly four-fold time difference. If set to False in the above code, the time jumps back to over four seconds.

## Comparision of Recordset Opening Methods in ODBC, Jet, and SQL Server

The above example opens the Recordset using a prepared Command object. In reality, ADO offers a multitude of methods for opening the Recordset. The purpose of these tests is to discover the optimal method of performing the database query from ASP. What follows is a comparison of seven such methods, tested in the context of the previous example. These methods use various combinations of the ADO Recordset, Connection, and Command objects, and stored versus inline queries. Each method is tested on three different OLE DB Providers: Jet, ODBC, and SQL Server.

Each test is further run with either one connection or two connections. The one connection test uses the same Connection for both the primary query and the sub-query. The two connection test uses a different Connection object for the sub-query. The source code for the tests is listed next, followed by tables of the test case code snippets and execution times.

### COMPLEXTABLE4.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim Method, DataSource, ConnectionString, NumberOfConnections

' Set up test case using the DataSource, Method, and NumberOfConnections
' Uncomment one at a time
'DataSource = "ODBC"
'DataSource = "Jet"
```

```
DataSource = "SQL Server"

'NumberOfConnections = 1
NumberOfConnections = 2

'Method = 1
'Method = 2
'Method = 3
'Method = 4
'Method = 5
'Method = 6
Method = 7

Dim StartTime, EndTime
Dim objCN ' ADO Connection object
Dim objRS ' ADO Recordset object
Dim strsql ' SQL query string
Dim objRS2 ' Another ADO Recordset object
Dim objCmd ' ADO Command object
Dim RecordsArray, i
Dim objCN2

' Start timer
StartTime = Timer

' Create a connection object
Set objCN = Server.CreateObject("ADODB.Connection")
Set objCN2 = Server.CreateObject("ADODB.Connection")

' Connect to the selected data source
If DataSource = "ODBC" Then
    ConnectionString = "DSN=datasource"
End If
If DataSource = "Jet" Then
    ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;"&_
        "Data Source=C:\Inetpub\wwwroot\data\test.mdb;"&_
        "Persist Security Info=False"
End If
If DataSource = "SQL Server" Then
    ConnectionString = "Provider=SQLOLEDB.1;"&_
        "User ID=sa;Data Source=localhost;"&_
        "Initial Catalog=sqltest;Password=;"
End If

' Initialize and open Connections
objCN.ConnectionString = ConnectionString
objCN.Open

If NumberOfConnections = 2 Then
    objCN2.ConnectionString = ConnectionString
    objCN2.Open
End If
If NumberOfConnections = 1 Then
    Set objCN2 = objCN
End If

' Create the a recordset object, and initialize it
Set objRS = Server.CreateObject("ADODB.RecordSet")
With objRS
    .CursorType = adOpenForwardOnly
    .LockType = adLockReadOnly
    .ActiveConnection = objCN
    .CursorLocation = adUseServer
    .Source = "SELECT Field1,Field2,Field3,Field4 FROM tblData"
End With
' Create the second recordset object, and initialize it
Set objRS2 = Server.CreateObject("ADODB.RecordSet")
With objRS2
```

```
    .CursorType = adOpenForwardOnly
    .LockType = adLockReadOnly
    .ActiveConnection = objCN2
    .CursorLocation = adUseServer
End With
' Create command object
Set objCmd = Server.CreateObject("ADODB.Command")
With objCmd
    .ActiveConnection = objCN2
    .Prepared = True
End With
If Method = 6 Then
    objCmd.CommandType = adCmdText
    objCmd.CommandText = "SELECT COUNT(*) FROM tblData WHERE Field1=?"
End If
If Method = 7 Then
    objCmd.CommandType = adCmdStoredProc
    objCmd.CommandText = "dummyquery"
End If

' Create Parameter object
objCmd.Parameters.Append _
    objCmd.CreateParameter(,adInteger,adParamInput,4)

' Execute the SQL query
objRS.Open

' Write out the results using GetRows in a loop
Response.write "<pre>"
Do While Not objRS.EOF
    RecordsArray = objRS.GetRows(30)

    For i = 0 To UBound(RecordsArray, 2)
        Response.write RecordsArray(0, i)
        Response.write vbTab
        Response.write RecordsArray(1, i)
        Response.write vbTab
        Response.write RecordsArray(2, i)
        Response.write vbTab
        Response.write RecordsArray(3, i)
        Response.write vbTab
        Response.write vbCrLf

        Response.write "Dummy query result="
        If Method = 1 Then
            strsql = "SELECT COUNT(*) FROM tblData WHERE Field1="&RecordsArray(0, i)
            objRS2.Open strsql
        End If
        If Method = 2 Then
            strsql = "SELECT COUNT(*) FROM tblData WHERE Field1="&RecordsArray(0, i)
            Set objRS2 = objCN2.Execute(strsql)
        End If
        If Method = 3 Then
            strsql = "EXECUTE dummyquery "&RecordsArray(0, i)
            Set objRS2 = objCN.Execute(strsql)
        End If
        If Method = 4 Then
            strsql = "{call dummyquery('"&RecordsArray(0, i)&"')}"
            Set objRS2 = objCN2.Execute(strsql)
        End If
        If Method = 5 Then
            objCN.dummyquery RecordsArray(0, i), objRS2
        End If
        If Method = 6 Or method = 7 Then
            objCmd(0) = RecordsArray(0, i)
            objRS2.Open objCmd
        End If
        Response.write objRS2(0)
```

```
        objRS2.Close
        Response.write vbCrLf

    Next
Loop
Response.write "</pre>"

objRS.Close
Set objRS = Nothing
Set objRS2 = Nothing
Set objCmd = Nothing
objCN.Close
Set objCN = Nothing
If NumberOfConnections = 2 Then
    objCN2.Close
    Set objCN2 = Nothing
End If

EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| **Method 1** | **Recordset.Open with unprepared query.** | | |
|---|---|---|---|
| `While ...`<br>`  strsql = "SELECT COUNT(*) FROM tblData WHERE Field1="&_`<br>`  RecordsArray(0, i)`<br>`  objRS2.Open strsql`<br>`Loop` | | | |
| | **ODBC** | **Jet** | **SQL Server** |
| **1 Connection** | 4.3 s. | 3.2 s. | 2.1 s. |
| **2 Connections** | 4.3 s. | 3.2 s. | 2.1 s. |
| SQL Server began at 7 seconds, then gradually declined over 20 tests levelling off at 2.1 seconds. | | | |
| **Method 2** | **Connection.Execute with unprepared query.** | | |
| `While ...`<br>`  strsql = "SELECT COUNT(*) FROM tblData WHERE Field1="&_`<br>`  RecordsArray(0, i)`<br>`  Set objRS2 = objCN2.Execute(strsql)`<br>`Loop` | | | |
| | **ODBC** | **Jet** | **SQL Server** |
| **1 Connection** | 4.7 s. | 3.5 s. | 5.3 s. |
| **2 Connections** | 4.8 s. | 3.7 s. | 1.3 s. |
| **Method 3** | **Connection.Execute of stored query.** | | |
| `While ...`<br>`  strsql = "EXECUTE dummyquery "&RecordsArray(0, i)`<br>`  Set objRS2 = objCN.Execute(strsql)`<br>`Loop` | | | |
| | **ODBC** | **Jet** | **SQL Server** |
| **1 Connection** | 5.3 s. | 3.4 s. | 5.4 s. |
| **2 Connections** | 5.3 s. | 3.4 s. | 5.5 s. |

SQL Server test sometimes took up to 20 seconds, and sometimes returned "SQL Server does not exist or access denied." error. Time shown is average of time when test was stable.

| Method 4 | Connection.Execute of stored query in ODBC syntax. | | |
|---|---|---|---|

```
While ...
        strsql = "{call dummyquery('"&RecordsArray(0, i)&"')}"
        Set objRS2 = objCN2.Execute(strsql)
Loop
```

| | ODBC | Jet | SQL Server |
|---|---|---|---|
| **1 Connection** | 5.4 s. | n/a | 5.3 s. |
| **2 Connections** | 5.7 s. | n/a | 1.4 s. |

Jet gave error "Invalid SQL statement."

| Method 5 | Stored query as method of Connection object. | | |
|---|---|---|---|

```
While ...
        objCN.dummyquery RecordsArray(0, i), objRS2
Loop
```

| | ODBC | Jet | SQL Server |
|---|---|---|---|
| **1 Connection** | 5.5 s. | 3.2 s. | 5.6 s. |
| **2 Connections** | 5.5 s. | 3.5 s. | 5.6 s. |

| Method 6 | Prepared Command object with unprepared query. | | |
|---|---|---|---|

```
objCmd.Prepared = True
objCmd.CommandType = adCmdText
objCmd.CommandText = "SELECT COUNT(*) FROM tblData WHERE Field1=?"
While ...
        objCmd(0) = RecordsArray(0, i)
        objRS2.Open objCmd
Loop
```

| | ODBC | Jet | SQL Server |
|---|---|---|---|
| **1 Connection** | 1.1 s. | 1.9 s. | 5.3 s. |
| **2 Connections** | 1.1 s. | 2.1 s. | 1.2 s. |

| Method 7 | Prepared Command object with stored query. | | |
|---|---|---|---|

```
objCmd.Prepared = True
objCmd.CommandType = adCmdStoredProc
objCmd.CommandText = "dummyquery"
While ...
        objCmd(0) = RecordsArray(0, i)
        objRS2.Open objCmd
Loop
```

| | ODBC | Jet | SQL Server |
|---|---|---|---|
| **1 Connection** | 1.1 s. | 1.7 s. | 5.6 s. |
| **2 Connections** | 1.1 s. | 2.0 s. | 1.2 s. |

## Provider-Specific Results of Recordset Opening Tests

With the ODBC Provider using one connection or two connections made no discernable difference in execution time. The first five methods had execution times of 4.3 to 5.7 seconds, with the first method of using a simple Recordset.Open being the fastest. The last two methods, which use a prepared Command object, both had an execution time of 1.1 seconds. This was the best result out of all the tests performed.

The Jet Provider was able to handle all the methods except the ODBC syntax stored procedure call. In 4 out of 6 working methods, the Provider performed about 10% slower when using two connections versus using one connection, and on the other two, there was no difference. In the first five methods, Jet had an execution time of 3.2 to 3.7 seconds. When using the Prepared Command object, the execution time fell to 1.7 to 2.1 seconds.

The SQL Server Provider executed in 2.1 seconds for the simple Recordset.Open method, but only after running the query numerous times. This reflects SQL Server's self-tuning mechanisms. For all the other methods, SQL Server performance was between 5.2 and 5.6 seconds, with a few interesting exceptions. In methods Two, Four, Six and Seven, using a different connection to perform the sub-query reduced the execution time down to 1.2 to 1.4 seconds.

## General Results of Recordset Opening Tests

From these test results alone, no general statement comparing the performance of the three Providers can be made, because the results were mixed. Using two connections made no difference for ODBC, was slightly slower for Jet, and was remarkably faster for SQL Server.

There was no significant difference between using a stored query versus using an inline query defined in ASP on any of the tests. However, these tests measured single-page execution time of a simple query on an otherwise idle server. Performance undoubtedly would be different in a high load environment or with more complex queries.

The general conclusion from these tests is that using a prepared Command object is the fastest way to perform a sub-query in a complex report. This method is approximately four times faster than any of the other methods. To gain this benefit with SQL Server, a second connection object *must* be used to perform the sub-query. The next section covers a SQL Server-specific optimization for even greater performance.

## Using a Parameterized Command Object Without a Recordset

The dummy sub-query only returns a single count, meaning that a full Recordset is not really needed. In fact, it is common for secondary queries in complex tables to either not return any data (such as an update, insert, or delete), or to only return a few pieces of information (such as a count, average, or single row.) In these circumstances, it is possible to optimize the speed even further by eliminating the Recordset.

The Command object supports both input and output parameters. Output parameters are not supported by ODBC or Jet, but will work under SQL Server and other major DBMS. The dummy

query will have one input parameter for the criteria of the WHERE clause, and one output parameter to return the count. The parameterized query is then executed with the special adExecuteNoRecords option, because no Recordset is needed. If no output parameters are needed, the adExecuteNoRecords option can also be used to realize performance benefits with ODBC or Jet.

In the example, the SQL Server query string is used in lieu of explicit Connection objects for both the main query and the sub-query. This is so that two Connections are implicitly opened for each object. As shown in the previous section, SQL Server performance suffers when the primary query and sub-query are executed on the same Connection.

## COMPLEXTABLE5.ASP

```
<%@ Language=VBScript %>
<% Option Explicit %>
<html>
<body>
<%
Dim StartTime, EndTime

StartTime = Timer

Dim objRS ' ADO Recordset object
Dim objCmd ' ADO Command object
Dim RecordsArray, i
Dim strConnectionString

strConnectionString = "Provider=SQLOLEDB.1;"&_
        "User ID=sa;Password=;"&_
        "Initial Catalog=sqltest;Data Source=localhost;"

' Create the a recordset object, and initialize it
Set objRS = Server.CreateObject("ADODB.RecordSet")
With objRS
    .CursorType = adOpenForwardOnly
    .LockType = adLockReadOnly
    .ActiveConnection = strConnectionString
    .CursorLocation = adUseServer
    .Source = "SELECT Field1,Field2,Field3,Field4 FROM tblData"
End With
' Create command object
Set objCmd = Server.CreateObject("ADODB.Command")
With objCmd
    .ActiveConnection = strConnectionString
    .Prepared = True
    .CommandType = adCmdText
    .CommandText = "SELECT ? = COUNT(*) FROM tblData WHERE Field1=?"
End With
' Append the parameter to the Command object's parameters collection
objCmd.Parameters.Append _
    objCmd.CreateParameter(,adInteger,adParamOutput,4)
objCmd.Parameters.Append _
    objCmd.CreateParameter(,adInteger,adParamInput,4)

' Execute the SQL query
objRS.Open

' Write out the results using GetRows in a loop
Response.write "<pre>"
Do While Not objRS.EOF
    RecordsArray = objRS.GetRows(30)

    For i = 0 To UBound(RecordsArray, 2)
```

```
        Response.write RecordsArray(0, i)
        Response.write vbTab
        Response.write RecordsArray(1, i)
        Response.write vbTab
        Response.write RecordsArray(2, i)
        Response.write vbTab
        Response.write RecordsArray(3, i)
        Response.write vbTab
        Response.write vbCrLf

        ' Use pre-prepared Command object without Recordset
        objCmd(1) = RecordsArray(0, i)
        objCmd.Execute ,,adExecuteNoRecords
        Response.write "Dummy query result="
        Response.write objCmd(0)
        Response.write vbCrLf

    Next
Loop
Response.write "</pre>"

objRS.Close
Set objRS = Nothing
Set objCmd = Nothing

Response.write "</table>"


EndTime = Timer
Response.write "<p>processing took "&(EndTime-StartTime)&" seconds<p> "
%>
</body>
</html>
```

| Test Results | |
|---|---|
| **Records** | **Seconds** |
| 1000 | 0.65 |

By parameterizing the query and using the adExecuteNoRecords option, the speed of this test query roughly doubles, and it runs in an impressive 0.65 seconds. If the same Connection object is used for both queries, the running time jumps to over five seconds.

## Summary of Complex Table Techniques

Complex tables are ASP reports where a sub-query must be performed for each record of a primary query. The nesting of the queries is what makes the report complex. To optimize such a report, the JOIN and SHAPE clauses are to be avoided because they have performance or maintainability drawbacks.

The time required to perform the sub-query can be reduced by adequate preparation. The first step is to use a single Connection object rather than creating one explicitly or implicitly each time. The second step is to use a parameterized and prepared Command object. These steps were shown to provide a 15-fold perfomance improvement in the example.

For complex table reports on low-load servers, there is no performance advantage to using

stored procedures versus defining the queries inline in ASP. The drawback is that the queries are separated from the ASP code, increasing the code maintenance burden. Under SQL Server, using a separate Connection object for the sub-query is the only way to realize the benefits of the prepared Command object. For some queries, further performance may be obtained by foregoing the Recordset and instead using a Command object with output parameters.

# Appendix

## Testing Notes

The tests were run by reloading the page five to ten times and estimating the average running time. The goal was not to be precise in the absolute running times, but to get an idea of the relative running times for each example. Here are specifications of the test system.

- Windows XP Professional, IIS 5.1, ADO version 2.8
- "datasource" is a System DSN using the Microsoft Access Driver linked to "test.mdb"
- Pentium-III 850Mhz, 640MB RAM, ATI Radeon 7200 32MB DDR
- Browsers in use were Mozilla 1.4 and Internet Explorer 6.0.
- For SQL Server tests, MSDE (SQL Server 2000 Desktop Edition) running on TCP/IP
- This machine is a workstation, and was not under any other processing load.

The script below was used to create the test data table for the MDB file and in SQL Server. The script creates a single table called "tblData" with four fields and 1000 rows. Obvious modifications to the script can be used to create tables with more rows.

**TESTDATA.ASP**

```
<%@ Language=VBScript %>
<% Option Explicit %>
<%

Dim objCN
Dim strsql, objRS
Dim i, FieldArray

Set objRS = Server.CreateObject("ADODB.Recordset")
Set objCN = Server.CreateObject("AdoDB.Connection")

' Create test data table
If True Then
    ' A Jet SQL Database
    objCN.ConnectionString = "DSN=datasource"
    strsql = "CREATE TABLE tblData "&_
        "(Field1 AUTOINCREMENT UNIQUE NOT NULL PRIMARY KEY,"&_
        "Field2 INTEGER, Field3 TEXT(50), Field4 TEXT(50))"
Else
    ' A SQL Server Database
    objCN.ConnectionString = "Provider=SQLOLEDB.1;"&_
        "User ID=sa;Data Source=localhost;"&_
        "Initial Catalog=sqltest;Password=;"
    strsql = "CREATE TABLE tblData "&_
        "(Field1 INTEGER IDENTITY PRIMARY KEY,"&_
        "Field2 INTEGER, Field3 NVARCHAR(50), Field4 NVARCHAR(50))"
End If
```

```
objCN.Open
objRS.Open strsql,objCN,adOpenForwardOnly,adLockReadOnly,adCmdText

' Populate the test data table with test data.
objRS.Open "tblData", objCN, adOpenStatic, adLockOptimistic, adCmdTable
FieldArray = Array("Field2","Field3","Field4")
For i = 1 To 100
    objRS.AddNew FieldArray, Array(879,"This is test data","abc")
    objRS.AddNew FieldArray, Array(458,"more test data.","def")
    objRS.AddNew FieldArray, Array(77,"another test","ghi")
    objRS.AddNew FieldArray, Array(66,"test test test","jkl")
    objRS.AddNew FieldArray, Array(54,"testing, 123","mno")
    objRS.AddNew FieldArray, Array(88,"test data test data","pqr")
    objRS.AddNew FieldArray, Array(498,"testing again","stu")
    objRS.AddNew FieldArray, Array(64,"test again","vwx")
    objRS.AddNew FieldArray, Array(8,"done testing","yz")
    objRS.AddNew FieldArray, Array(58,"again","xxx")
Next
objRS.Update

Response.write "added 1000 rows to new table tblData"
objRS.Close
objCN.Close
Set objRS = Nothing
Set objCN = Nothing

%>
```

## Books

- Professional Active Server Pages 3.0 - A good reference book.
- Microsoft Access Developer's Guide to SQL Server - Contains an excellent chapter on ADO, which unlike many other books, actually explains how to use it.
- Microsoft Jet Database Engine Programmer's Guide - Second Edition
- SQL In a Nutshell - O'Reilly
- Dynamic HTML - The Definitive Reference - O'Reilly

---

**Copyright © 2003 Shailesh N. Humbad. All Rights Reserved.**

No part of this PDF may be reproduced without the express written permission of Shailesh N. Humbad, except for brief quotations used in critical reviews or articles.

Whilst the author believes the information contained in this PDF to be accurate, the information is sold on an "as-is" basis without warranties of any kind, either express or implied, including but not limited to implied warranties of merchantability or fitness for a particular purpose.

Microsoft is a registered trademark of Microsoft Corporation. The author is not affiliated with Microsoft Corporation in any way.

Other trademarks are the property of their respective owners.

Shailesh N. Humbad, 22649 Foxmoor Dr., Novi, MI 48374, www.somacon.com

---