Paging and caching large record sets

Welcome to our next foray into database driven ASP development. Today we're going to be looking at viewing large record sets, and how to optimise this process both for the person doing the viewing and for the server processing the page. To handle the former, we'll use a technique known as paging, and for the latter, caching. This caching is not the client side browser caching we've all come to know and hate; this caching is done entirely on the server. Let's jump right in!

Creating the page with the recordset

First off, we're going to create a simple page using Dreamweaver, which will use a standard Dreamweaver connection to create a standard Dreamweaver recordset. We will then use a Server Behaviour to display the contents of this recordset.

1. Create the connection

As in the previous tutorial, we're going to use a DSN (Data Source Name):

Create the DSN

- Copy the database file to your web server folder (It's in the link "Code download" below the download buttons).
- Click Start > Programs > Administrative Tools > Data sources (ODBC)
- Select the System DSN tab, and click Add…
- Select the Microsoft Access Driver (*.mdb)
- Enter a name for your new data source. I've called mine **PagingSample**.
- Click the Select… button and browse to your copy of **PagingSample.mdb**.
- You should see the path to PagingSample.mdb above the select button. Click OK.
- Click OK to close the Data sources panel.

Before you can define a Connection with Dreamweaver, you will need to create a new Site first. If you have not done so, be sure to do so now. (Molly Holzschlag has a tutorial on setting up your site, if you're unsure how).

Create the Connection in Dreamweaver

- If your Application panel group is not visible, use Window > Databases to make it appear.
- Click the **+** icon in the Databases panel, and select Data Source Name (DSN) from the list.
- Enter a name for your new connection, I used **PagingSample**.
- Select your DSN from the drop-down list.
- Click the **Test** button. You should see a message: 'Connection was made successfully.'
- Click OK on the notification, and click OK again to save the new Connection.
- You should now see a Connection in the Databases panel.

And you're done! You should now be able to browse through the database's tables, views, and stored procedures. This database contains a single table, **Orders**. For the sake of clarity, I haven't normalised the data for this table.

2. Create a Recordset

Now that we have a connection to the database, let's retrieve some data through it. We do this by creating a recordset, which is a collection of rows or records, each containing a number of columns, or fields. These row/column combinations each contain one distinct piece of data.

- Using the Insert bar, switch to the Application sub-set. Click the first icon in this group, **Recordset**.
- A Recordset dialog appears.
- Name the recordset '**Orders**'.
- Select your **PagingSample** connection from the list.
- Select the **Orders** table from the Table list.
- For Columns, select 'Selected', and then holding down your Control key, click all but the Order ID columns.
- Set Sort to sort by **Order Date** in ascending order.
- Click Test to see whether the SQL query returns data.
- Click OK to close the Test SQL Statement dialog, and click OK again to save your new recordset.

Excellent! Now we're ready to display some data.

3. Use a Dynamic Table to display the Recordset

Let's use two SB's to display the data in our new **Orders** recordset, and add paging capability:

- Go to Design view by clicking **Design** on the **Document** toolbar.
- On the Application Insert bar, click the down arrow on the third icon from the left. Select the first option, **Dynamic Table** from the list.
- Name your new dynamic table '**OrdersView**'.
- For 'Show', set it to display 2 records at a time. Why 2? Well, if it can work for 2 records, it can work for 10 or 20 or 50!
- Click OK to save the dynamic table.
- Again on the Application Insert bar, click the down arrow on the sixth icon from the left. Select the first option, **Recordset Paging** from the list.
- Your Orders recordset should already be selected in the Recordset list. Set 'Display using' to Text.
- Click OK to save the recordset paging controls.

Right! We now have a paged recordset. The user won't have to wait minutes for the entire recordset load into the browser. This makes browsing this data a more pleasant experience. And what's more, we got to this point in less than 10 minutes! That's powerful.

However…

There is a catch. Dreamweaver inserted over 200 lines of code for this paging capability! What if this paging technique is to be used on a public website, where usage could conceivably grow to thousands of users at any given time? This code, however useful, is not optimal. Luckily, a lot can be done to fix this!

Apart from cleaning up this code, we'll also look at how to optimise this process from a logical standpoint (as opposed to a technical one). If this page is to be used often, it is quite conceivable that the very same recordset will be displayed to multiple users at a given time. If not, users who page back and forth between subsets of their recordset (using the above paging code) still retrieve a fresh instance of that record set every time they hit 'Next' or 'Previous'. Why not use one instance for the paging session instead? This will save precious database connectivity resources.

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

To make this work, we'd have to make sure that a couple things are true. Firstly, this would only really provide performance benefit if insertions and updates to this data are few, and views is high. Why? Well, every time that the data changes, the cached instance of the recordset in memory is no longer consistent with the data in the database. This means that we'd have to regenerate this cached copy if the data changes. Couple that fact with a high change rate and you start to come back to the recordset being generated on almost every page view.

Even so, if the change rate is moderate compared to the number of page views, there is still a performance benefit. Let's look at how to implement this caching technique for a single user first.

Session based recordset caching

Unfortunately, to be able to cache data in the Session store (the area of memory dedicated to the user's session), we need to transform the data in the recordset into intrinsic JavaScript objects, as we cannot store the recordset (with all its data) itself in the Session store. Although there is a very slight performance hit in doing this, implementing the caching which depends on doing so provides massive performance benefit.

FYI: JavaScript objects are basically named containers for variables. Objects can contain more objects, which can contain more objects, each with their own variables, arrays, and so on. They basically provide a way to structure information.

There is a useful side benefit too; the data becomes prepared for viewing only once. The catch: we will use almost none of Dreamweaver's generated code. The only code we do use is the bit that creates the recordset. The rest is hand-typed. As a consequence, you'll basically start with a new ASP JavaScript page. Simply copy the code below into your document.

Here's the solution. There's a fair bit to go through here. If it looks daunting, skip ahead to the walk-through! I urge you to read through it anyway, as this will make the walk-through an easier process for you.

```
<%@LANGUAGE="JAVASCRIPT" CODEPAGE="1252"%>
<!--#include file="../Connections/PagingSample.asp" -->
<%
function FormatShortDate ( dateObject ) {
 return dateObject.getDate() + "&middot;" +
 ( dateObject.getMonth() ) +
 "&middot;" + dateObject.getFullYear();
}

function FormatCurrency ( value ) {
 return "R" + value + ".00";
}

var orders, order;
var pageSize = 1, currentPage = 0, paging, nextPage, totalPages;
var orderIndex = 0, endPoint, recordCount;

if ( Session( "Orders" ) != null &&
 Application( "ordersChanged" ) != "changed" ) {

 orders = Session( "Orders" );

 if ( String( Request( "page" ) ) != "undefined" ) {
```

```
 currentPage = parseInt( Request( "page" ) );
 }

} else {

 var ordersRS = Server.CreateObject("ADODB.Recordset");
 ordersRS.ActiveConnection = MM_PagingSample_STRING;
 ordersRS.Source = "SELECT OrderDate, Client, Value, Products FROM Orders ORDER BY
OrderDate ASC";
 ordersRS.CursorType = 0;
 ordersRS.CursorLocation = 2;
 ordersRS.LockType = 1;
 ordersRS.Open();

 orders = new Array();

 while ( !ordersRS.EOF ) {

 order = new Object();

 order.OrderDate = FormatShortDate( new Date( ordersRS( "OrderDate" ) ) );
 order.Client = String( ordersRS( "Client" ) );
 order.Value = FormatCurrency( parseFloat( ordersRS( "Value" ) ) );
 order.Products = parseInt( ordersRS( "Products" ) );

 orders.push( order );

 ordersRS.moveNext;
 }

 ordersRS.Close();

 Session( "Orders" ) = orders;
}

recordCount = orders.length;
paging = recordCount > pageSize;

if ( paging ) {

 nextPage = currentPage + 1;
 totalPages = Math.ceil( recordCount / pageSize );

 orderIndex = currentPage * pageSize;
 endPoint = orderIndex + pageSize;

 if ( endPoint > recordCount ) {
 endPoint = recordCount;
 }

} else {
 endPoint = recordCount;
}

%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

```
<head>
 <title>Untitled Document</title>
 <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
 <table width="400" border="0">
<%
if ( paging ) {
%>
 <tr>
 <td colspan="4" align="center">
<%
 if ( currentPage > 0 ) {
%>
 <a href=" PagingSample.asp?page=<%= currentPage - 1 %>">Previous</a>
<%
 }
 if ( nextPage < totalPages ) {
%>
 <a href=" PagingSample.asp?page=<%= nextPage %>">Next</a>
<%
 }
%>
 </td>
 </tr>
<%
}
%>
 <tr>
 <th align="left">Order Date</th>
 <th align="left">Client</th>
 <th align="center">Value</th>
 <th align="center">Products</th>
 </tr>
<%
orderIndex--;
while ( ++orderIndex < endPoint ) {
 order = orders[ orderIndex ];
%>
 <tr>
 <td><%= order.OrderDate %></td>
 <td><%= order.Client %></td>
 <td align="right"><%= order.Value %></td>
 <td align="right"><%= order.Products %></td>
 </tr>
<%

%>
 </table>
</body>
</html>
```

Okay! Let's go through it step by step.

1. Workspace

First off, we define some formatting functions (for dates and currencies), and define all of the variables we'll need. Notably, we set up **currentPage** and **orderIndex** with their initial values (both 0), and we set up **pageSize** with the number of items we want to display per page.

Note - the formatting functions here aren't intended for production use, I include them merely to illustrate that these can exist, and can provide any formatting function, however simple or complex.

```
function FormatShortDate ( dateObject ) {
 return dateObject.getDate() + "&middot;" +
 ( dateObject.getMonth() ) +
 "&middot;" + dateObject.getFullYear();
}

function FormatCurrency ( value ) {
 return "R" + value + ".00";
}

var orders, order;
var pageSize = 2, currentPage = 0, paging, nextPage, totalPages;
var orderIndex = 0, endPoint, recordCount;
```

2. Determine where to get the data from

Next up, we need to determine in some way whether or not to retrieve the data from the database or from the cached copy in **Session**. Determining this is quite easy: if the cache doesn't exist, create it. Additionally, the cache could exist, but the data it contains could be out-of-date. To determine this, we check an Application variable (which is accessible from all sessions in the application) named **OrdersChanged** for the value 'changed' to see if some other process has modified this data. Obviously, you would have to set this variable to 'changed' in any script you write that modifies this data in the database.

```
if ( Session( "Orders" ) != null &&
 Application( "OrdersChanged" ) != "changed" ) {
```

3a. Retrieve cached data

If we find that the data in the cache is still good, we create a local (to the script) reference to it. Additionally, we check to see if we've received **page** via the query-string, and if so, we parse it as a number to our local **currentPage** variable. Doing the **page** check inside this branch, instead of as a part of the **if** statement above, allows users coming back to the page from another page to make use of the cached data.

```
 orders = Session( "Orders" );

 if ( String( Request( "page" ) ) != "undefined" ) {
 currentPage = parseInt( Request( "page" ) );
 }
```

3b. Retrieve and cache data from database

If the cache is nonexistent or invalid, we create our recordset as before.

```
var ordersRS = Server.CreateObject("ADODB.Recordset");
 ordersRS.ActiveConnection = MM_PagingSample_STRING;
 ordersRS.Source = "SELECT OrderDate, Client, Value, Products FROM Orders ORDER BY
OrderDate ASC";
 ordersRS.CursorType = 0;
 ordersRS.CursorLocation = 2;
 ordersRS.LockType = 1;
 ordersRS.Open();
```

We then define an empty array (or numbered list) container and loop through the recordset:

```
orders = new Array();

 while ( !ordersRS.EOF ) {
```

This next bit of code displays JavaScript's amazing flexibility when it comes to representing data. We are able to define arbitrary containers (objects) and assign values to named properties of these objects. Using the formatting functions defined above for the **Order Date** and **Value** columns, and **String** for the rest, we set named properties for the order object.

```
 order = new Object();

 order.OrderDate = FormatShortDate( new Date( ordersRS( "OrderDate" ) ) );
 order.Client = String( ordersRS( "Client" ) );
 order.Value = FormatCurrency( parseFloat( ordersRS( "Value" ) ) );
 order.Products = String( ordersRS( "Products" ) );
```

Doing this is vitally important! If we simply assigned from the recordset column directly to the object property, it would attempt to store a *reference* to that particular row/column combination in the recordset object. This means that once we've closed the recordset, that reference will be broken, which is not a good thing for a caching implementation!

We then use the Array.push() method to add the new order object to the end of the array, and move to the next record in the recordset. That finalises the loop. After the loop, now that we have transformed all the data, we close the recordset, and add a reference to the array to the **Session**:

```
 orders.push( order );

 ordersRS.moveNext;
 }

 ordersRS.Close();

 Session( "Orders" ) = orders;
```

And caching is done! Next, we have to determine whether or not we need to page this data.

4. Determine if paging is necessary

Now that we have an idea of how many records there are, we can determine whether or not to page the data. Firstly, we set **recordCount** to be the number of records in the set:

```
recordCount = orders.length;
```

Then we store whether or not we have to page the data as a Boolean or true/false value, named **paging**. We derive this value by comparing the relationship between our recently new **recordCount** and our constant **pageSize**. If **recordCount** is bigger than **pageSize**, or put differently, if the total number of records in the set exceeds the number of records displayed per page, **paging** will be set to **true**.

```
paging = recordCount > pageSize;
```

5. Configure render loop boundaries

Next, if **paging** is **true**, we need to configure the variables used to display Next and Previous links and set the range for the render loop.

Firstly, we store the index of the *next* page into **nextPage**. Next, we determine **totalPages**, the total number of pages for the data set. To get this we divide **recordCount** by **pageSize**, and round up to the nearest integer. That way, with a page size of 5 and a record set of 12, the third page will only display 2 records. If we rounded down, we wouldn't see these last 2 records.

```
if ( paging ) {

 nextPage = currentPage + 1;
 totalPages = Math.ceil( recordCount / pageSize );
```

Next we configure the initial value for **orderIndex**, the render loop's iterator (variable that increments by one every loop). We do so by setting it to the current page's index multiplied by the page size. It is for this reason that **currentPage** is initially set to 0; 0 x 5 is still 0, which is exactly where we want to be for the first 5 records.

Then we set **endPoint**, the upper boundary for the loop, which determines when the loop should end. We set it to be the current **orderIndex** plus one page's worth of records, thus, our page size.

```
 orderIndex = currentPage * pageSize;
 endPoint = orderIndex + pageSize;
```

Finally, just in case we happen to be on the last page, we make sure that **endPoint** isn't larger than **recordCount**. This could happen quite easily, as we are using multiples of **pageSize** to determine **endPoint**, and **recordCount** is not necessarily a multiple of **pageSize**.

If **paging** is **false**, however, we still need to configure **endPoint** so that the render loop can function correctly. We do not need to set **orderIndex** as it is already set to 0 (see the step 1).

```
} else {
 endPoint = recordCount;
}
```

Now that all of our data has been prepared, we're ready to begin rendering the HTML. There are two steps to this, rendering the Next and Previous links, and rendering the data itself.

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

6. Display paging links

This section would only render if **paging** is **true**.

```
<%
if ( paging ) {
%>
```

If so, we determine if we display the Previous link first. We display it if **currentPage** is bigger than 0, indicating that we are currently viewing page 2 or higher. We set the Previous link to open the same page with the **page** querystring variable to equal **currentPage** minus one.

```
<%
 if ( currentPage > 0 ) {
%>
 <a href="PagingSample.asp?page=<%= currentPage - 1 %>">Previous</a>
<%
 }
```

Similarly, we check if we need to display the Next link, by checking whether **nextPage** is smaller than **totalPages**. The logic here is, if the next page's index is smaller than the total number of pages, we are not currently on the last page. The reason for this is that our **currentPage** variable starts counting from 0 based whereas our **totalPage** starts counting from 1. If this is the case, we render the Next link in the same fashion as the Previous link, using **nextPage** as our **page** querystring variable.

```
if ( nextPage < totalPages ) {
%>
 <a href=" PagingSample.asp?page=<%= nextPage %>">Next</a>
<%
 }
%>
```

7. Render the data

Finally! After all that work, we can now display the data. Firstly, we decrement **orderIndex** by 1. You'll see why in a bit. Next, we start the loop:

```
<%
orderIndex--;
while ( ++orderIndex < endPoint ) {
```

As you can see, we increment **orderIndex** by one and then check to see if it is less than **endPoint**. The important bit here is that **orderIndex** increments *before* the comparison, not after, as with a traditional **for** loop. We do it this way to reduce the number of calculations in the comparison stage of the loop. Bear with me!

If we increment **orderIndex** after the comparison, we wouldn't see the last record in the range, because we'd reach our **endPoint** one loop too quickly. Ok, so we can increment **endPoint** by one before the loop. But have a look at the next line of code, the one which retrieves the record from our orders array:

```
 order = orders[ orderIndex ];
%>
```

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

The way we have it now, **orderIndex** is perfectly positioned to begin with the right record. If any of this doesn't make sense to you, I urge you to play around it and see how these variables interact. Anyhow, this is why we decrement once before we begin the loop.

To continue, as seen above, we create a local (to the loop) reference, **order**, to the current record, based on **orderIndex**. We then render out the HTML table row, using those nice easy to read names we assigned each field earlier.

```
<tr>
<td><%= order.OrderDate %></td>
<td><%= order.Client %></td>
<td align="right"><%= order.Value %></td>
<td align="right"><%= order.Products %></td>
</tr>
```

We close the loop, and render out the remainder of the HTML for the output. Voila! A cached, paged recordset viewer!

Conclusion

Today, you learnt how to use Dreamweaver Server Behaviours to achieve recordset paging in record time. You then learnt how that speed came at a cost; processing time.

Of course, if your Dreamweaver Server Behaviour page is to be used on a site which has 50 viewers a day, this is not a problem. 5000 viewers however will start to cause the web server some strain. Just for interest's sake, the hand-coded script is 124 lines, and the Dreamweaver page is 252 lines. The hand-coded script is half as big but provides easily 4 times the benefit!

You learnt about the benefits of caching data in server memory, and invalidating this data whenever updates happen to the database. Finally, you were led by the nose through a working example of hand-coded session based caching and paging. I hope you understood everything, as the techniques described above have helped me streamline growing web applications tremendously! Again, I urge you to grab the sample files, break them, make them work again, plug your own data in, go mad. The best teacher is play!

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com