Real World ASP.NET Best Practices

FARHAN MUHAMMAD AND MATT MILNER

Apress™

ISBN (pbk): 1-59059-100-3

Printed and bound in the United States of America 12345678910

Technical Reviewer: Scott Stabbert

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown Collins

Development Editor: Ami Knox

Copy Editor: Kristen Imler

Compositor and Proofreader: Kinetic Publishing Services, LLC

Indexer: Lynn Armstrong

Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 2

Cache, Session, and View State

THIS CHAPTER EXPLAINS the intricacies of the three features available in ASP.NET for maintaining state. Each of these features provides a unique solution that comes with its own sets of benefits and problems. Cache, for example, is a very useful mechanism for storing commonly used information in memory. However, cache is not built to support server farms and therefore is best used for nonsession-related information.

Session and view state, on the other hand, are built for session management and can be used across server farms. This chapter focuses on explaining the details of each feature and showing its behavior in a variety of scenarios. You need to understand the capabilities and limitations of each of these features and devise your own magical potion that contains their best combination.

Be Mindful of the Cache!

Caching is a mechanism for keeping content in memory for later use. It helps Web applications operate with higher performance by reducing the amount of work needed to obtain information from its data source, such as a database, Web service, mathematical computation, and many others. ASP.NET provides us with a variety of ways for caching information.

The output cache mechanism keeps rendered pages in memory and serves future requests with the in-memory image of the page. You can customize it and use it to cache different versions of the same page based on the query string values, browser cookies, or form variables.

The data cache mechanism provides us with the ability to store any object in memory and retrieve it at a later time. By using this mechanism, we can access the data source once and keep the result in the memory to serve future requests for the same information.

Cache Pros and Cons

Before you go too wild and start putting everything including the kitchen sink in cache, you need to consider the following benefits and risks.

The benefits include

- A reduced number of round trips to the data source, such as the database server, keeping the server resources more available for other operations.
- An increase in the number of users supported, due to a faster response time to each user's request.

The risks include

- Easily filling a computer's memory, which is relatively small, if you put a large amount of data in cache. As the memory gets full, the performance starts to decline, eventually leading to an unacceptable response time from the server.
- Problems in a server farm environment, when we cache information in the server's memory, where various Web pages for the same user session may be served by different Web servers.
- No guarantee of faster performance. It all depends on how effectively you manage objects in memory. We'll go into more detail on this topic later in this section.

In general, caching is useful when you have a large amount of relatively static information. A prime candidate for caching is product catalog information. There is little value in using SQL to search the database to retrieve the same list of products for each user who visits your Web site. It is a waste of database and network resources (assuming that the database is installed on a separate server than the Web site). You can easily store information like this in data cache. However, before you go wild and put your entire product catalog in one large XML DOM object (or DataSet object), consider this fact: Even though it is easier to get access to an object stored in memory, it is not necessarily faster to search that object.

A prime example of this fact is the DataSet object. The ADO.NET enthusiasts love to glorify this object by focusing on its ability to provide in-memory cache of the database. They often neglect to tell their innocent listeners about the slow performance of its search mechanism. We were surprised when we performance-tested the DataSet search capability and found it to be a much slower alternative to searching a SQL Server 2000 database by using embedded SQL. Let's put the in-memory data caching mechanism to the test. The examples in the following sections demonstrate various strengths and weaknesses related to this mechanism. The purpose of these exercises is merely to show you the realities of technologies involved, not to suggest any one method over another.

NOTE *As suggested at the beginning of this chapter, you should use your best judgment when selecting a data-caching mechanism for your next ASP.NET-based project.*

Performance Testing with Data Caching

Let's have some fun with caching a DataSet object. In this section, we will create a simple ASP.NET application that searches a data store that contains file and directory names. We will call it CacheBuster. This application will be capable of keeping the in-memory copy of the database and using it to search for required information. For comparison's sake, we will also create a mechanism for searching a SQL Server 2000 database by using embedded SQL to retrieve the same information.

The schema of the table contained in the SQL Server 2000 database is shown in Table 2-1. It contains roughly 50,000 records. It may seem like a lot of data, but it is not uncommon for a Web application to process this amount of information. A successful eCommerce site can easily have more than 50,000 customer records, order records, SKU records, etc.

Table 2-1. CacheBuster Database Table Schema

TABLE NAME	COLUMN NAME	DATA TYPE
Files	Id	Numeric
Files	FileName	Varchar(255)
Files	DirectoryName	Varchar(255)

The first thing we'll do is store the data in cache.

Storing Data in Cache

Use the Page_Load event to check whether the DataSet object is available in cache or not. If it is not, use embedded SQL and the ADO.NET framework to create a new DataSet object and store it in cache. Listing 2-1 shows C# code for storing data in cache.

Listing 2-1. C# Code for Storing Data in Cache

```
private void Page_Load(object sender, System.EventArgs e)
{
     // If the DataSet object is not in cache, get it
     // from the database.
     if (Cache["Result"] == null)
     {
          SqlCommand MyCommand = new SqlCommand();
          MyCommand.Connection =  new SqlConnection("your connection string");
          MyCommand.CommandText = " select * from files ";

          SqlDataAdapter MyAdapter = new SqlDataAdapter();
          MyAdapter.SelectCommand = MyCommand;

          DataSet MyDataSet = new DataSet();

          // Retreiving result set from the database and populating
          // DataSet with it.
          MyCommand.Connection.Open();
          MyAdapter.Fill(MyDataSet);
          MyConnection.Close();

          MyAdapter.SelectCommand.Dispose();
          MyAdapter.Dispose();

          // Placing the DataSet object in cache.
          Cache["Result"] = MyDataSet;
     }
}
```

Let's look at the user interface for the CacheBuster application, shown in Figure 2-1. You will see that it contains a text box that receives the name of the directory you want to search. By using this name, it can find all the files in that directory by using the in-memory cached DataSet object. Alternatively, it can also search for all files in the given directory by running a SQL statement on a SQL Server database.

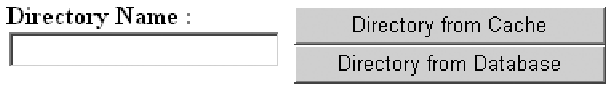


Figure 2-1. User interface for the CacheBuster application

Searching DataSet

Let's look at the code that uses the DataSet's built-in search mechanism. In Listing 2-2, we use the DataView object to filter rows contained in a DataTable object. The RowFilter property of the DataView object receives an expression similar to the WHERE clause of a SQL statement. By using this filter, the DataView object uses the Select method of the DataTable object to perform the search.

BEST PRACTICE *Why not use the Select method of the DataTable object directly? The problem with the Select method is that it returns an array of DataRow objects. It is not easy to bind this array to a server control because the System.Array class doesn't implement the IListSource and IList interfaces. It is better to use the DataView object to bind a set of DataRow objects to a server control.*

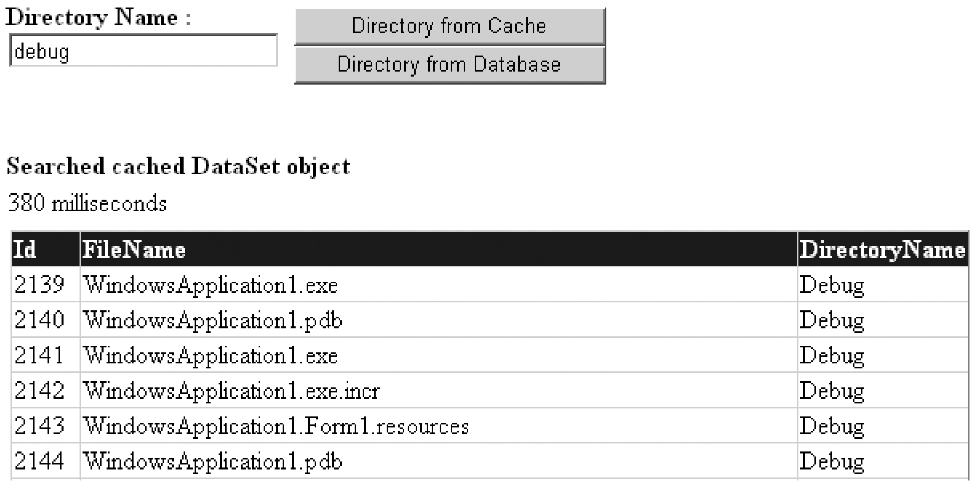
On the other hand, if you need to filter a data table for reasons other than binding to a control, you are better off using DataTable's Select method directly. This way, you don't incur the overhead associated with the DataView object. Listing 2-2 shows how to search a cached DataSet object.

Listing 2-2. Searching a Cached DataSet Object

```
private void DirectoryFromCacheButton_Click(object sender, System.EventArgs e)
{
     if (Cache["Result"] == null)
          return;
```

```
    DateTime StartTime, EndTime;
    TimeSpan Duration;

    // Retrieving DataSet object from cache.
    DataSet MyDataSet = Cache["Result"] as DataSet;

    // Starting to measure time.
    StartTime = DateTime.Now;

    // Creating DataView object, which will be filtered and bound
    // to the data grid control.
    DataView MyDataView = new DataView(MyDataSet.Tables[0]);
    MyDataView.RowFilter = " DirectoryName = '" + DirectoryTextBox.Text + "' ";

    ResultGrid.DataSource = MyDataView;
    ResultGrid.DataBind();

    // Stopping time measurement.
    EndTime = DateTime.Now;

    Duration = EndTime - StartTime;

    CacheLabel.Text = "Searched cached DataSet object";

    // Displaying elapsed time on the screen.
    DurationLabel.Text = Duration.Milliseconds.ToString() + " milliseconds ";
}
```

When we ran CacheBuster and measured the time it took for the DataSet to filter the desired rows, we got the result shown in Figure 2-2. The test was conducted on an 850 MHz PC with 256 MB of RAM.



Searched cached DataSet object
380 milliseconds

Id	FileName	DirectoryName
2139	WindowsApplication1.exe	Debug
2140	WindowsApplication1.pdb	Debug
2141	WindowsApplication1.exe	Debug
2142	WindowsApplication1.exe.incr	Debug
2143	WindowsApplication1.Form1.resources	Debug
2144	WindowsApplication1.pdb	Debug

Figure 2-2. The execution time for filtering the DataSet for the desired rows

Searching by Using a SQL Server Database

Now that we know that it takes 380 milliseconds to filter a data table of 50,000 records, let's perform the same operation by using a SQL Server 2000 database. Listing 2-1 showed how we used the ADO.NET framework to perform a Select query on a SQL Server database; Listing 2-3 shows the code to extract data from the database and to bind it to the data grid control.

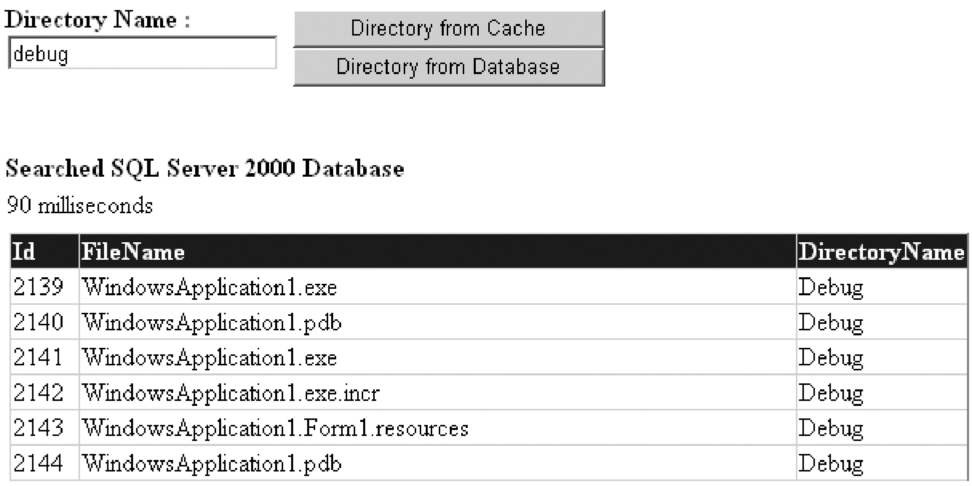
Listing 2-3. C# Code for Searching the SQL Server Database

```
private void DirectoryFromDBButton_Click(object sender, System.EventArgs e)
{
     DateTime StartTime, EndTime;
     TimeSpan Duration;

     DataSet MyDataSet = new DataSet();
     SqlDataAdapter MyAdapter = new SqlDataAdapter();
     SqlCommand MyCommand = new SqlCommand();
     MyCommand.Connection = new SqlConnection("your connection string");

     // Specifying SQL SELECT statement using WHERE clause to
     // filter the result by matching directory name.
     MyCommand.CommandText = " select * from files where DirectoryName = '" +
                             DirectoryTextBox.Text + "' ";
```

```
        // Starting to measure time.
        StartTime = DateTime.Now;

        MyAdapter.SelectCommand = MyCommand;
        MyCommand.Connection.Open();

        // Filling data set with results return from SQL query.
        MyAdapter.Fill(MyDataSet);

        ResultGrid.DataSource = MyDataSet.Tables[0].DefaultView;
        ResultGrid.DataBind();

        // Ending time measurement.
        EndTime = DateTime.Now;

        MyCommand.Connection.Close();
        MyCommand.Dispose();

        Duration = EndTime - StartTime;

        CacheLabel.Text = "Searched SQL Server 2000 Database";

        // Displaying elapsed time on the screen.
        DurationLabel.Text = Duration.Milliseconds.ToString() + " milliseconds ";
}
```

Let's run CacheBuster again and see the performance measurement. Figure 2-3 shows the execution time that results from performing the SQL Select query against a SQL Server 2000 database.

You can clearly see that searching the database only took 90 milliseconds, which is roughly four times faster than using a cached DataSet object. One would wonder why to bother with memory cache when it doesn't provide the performance increase for which one hoped. Before you throw away your data caching code and revert to writing SQL for all your data needs, let us explain why caching can still be a better option. We will also show you how to optimize memory cache for better performance.



Searched SQL Server 2000 Database

90 milliseconds

Id	FileName	DirectoryName
2139	WindowsApplication1.exe	Debug
2140	WindowsApplication1.pdb	Debug
2141	WindowsApplication1.exe	Debug
2142	WindowsApplication1.exe.incr	Debug
2143	WindowsApplication1.Form1.resources	Debug
2144	WindowsApplication1.pdb	Debug

Figure 2-3. The execution time for performing SQL Select statement

When Good Cache Goes Bad

The performance problems we just saw didn't occur because we chose to put a relatively large amount of data in memory. The problems also didn't occur because the server ran out of available memory and had to resort to using virtual memory. This performance problem arises because the DataView object doesn't seem to be optimized for performance. This problem becomes evident when you use this object on a data table that contains more than 20,000 records. Depending on the speed of your hardware, you might even see this problem with data table that contains fewer records.

Still, you can use a few tricks to get better performance while searching cached data. One such trick is to restructure the data to make it more fragmented.

Restructuring the Data Set

In our next example, we will modify the data set to include another data table with the name "Directories." This data table will contain only directory names. We will also create a relationship between two tables to link matching directory names. Figure 2-4 shows the data model that shows both the Directories and the Files tables.

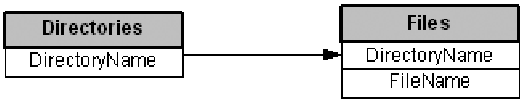


Figure 2-4. Modified data set structure

Including the Directories data table allows for a quick search for the directory name. Because this data table contains only unique directory names, it has much fewer records than the Files table. When a matching directory is found, we can use the relationship to retrieve all files contained in the directory from the Files data table.

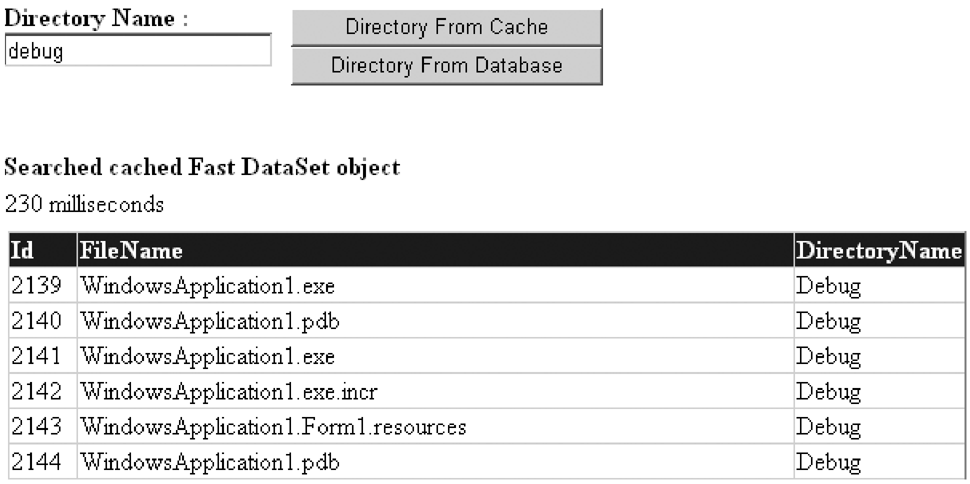
Listing 2-4. Making the Code Modifications to Show an Increase in Performance

```
private void Page_Load(object sender, System.EventArgs e)
{
     if (Cache["Result"] == null)
     {
          SqlCommand MyCommand = new SqlCommand();
          MyCommand.Connection =  new SqlConnection("your connection string");
          MyCommand.CommandText = " select * from files ";

          SqlDataAdapter MyAdapter = new SqlDataAdapter();
          MyAdapter.SelectCommand = MyCommand;

          DataSet MyFastDataSet = new DataSet();

          MyCommand.Connection.Open();

          // Adding Files table to the data set.
          MyAdapter.Fill(MyFastDataSet, "Files");

          // Selecting unique directory name from the database.
          MyCommand.CommandText = " select Distinct DirectoryName from Files ";

          // Adding Directories table to the data set.
          MyAdapter.Fill(MyFastDataSet, "Directories");

          DataColumn ParentColumn =
          MyFastDataSet.Tables["Directories"].Columns["DirectoryName"];
          DataColumn ChildColumn =
         MyFastDataSet.Tables["Files"].Columns["DirectoryName"];
```

```
            // Creating a relationship between Directories and Files
            // data tables.
            MyFastDataSet.Relations.Add("Directory_File_Relation",
                                        ParentColumn, ChildColumn);

            MyConnection.Close();
            MyAdapter.SelectCommand.Dispose();
            MyAdapter.Dispose();

            // Adding data set to cache
            Cache["FastResult"] = MyFastDataSet;
      }
}

// This method runs when user clicks Directory from Cache button.
private void DirectoryFromCacheButton_Click(object sender, System.EventArgs e)
{
      if (Cache["FastResult"] == null)
            return;

      DateTime StartTime, EndTime;
      TimeSpan Duration;

      DataSet MyFastDataSet = Cache["FastResult"] as DataSet;

      string DirectoryFilter = " DirectoryName = '" +  DirectoryTextBox.Text + "' ";

      // Starting measuring time.
      StartTime = DateTime.Now;

      DataView DirectoryView =  new
DataView(MyFastDataSet.Tables["Directories"]);

      // Filtering Directories table for the raw matching input
      // directory name.
      DirectoryView.RowFilter = DirectoryFilter;

      if (DirectoryView.Count == 0)
            return;

      // Using the relationship to find all files matching
      // directory name.
      DataView FilesView =
         DirectoryView[0].CreateChildView("Directory_File_Relation");
```

```
        ResultGrid.DataSource = FilesView;
        ResultGrid.DataBind();

        // Stopping time measurement.
        EndTime = DateTime.Now;

        Duration = EndTime - StartTime;

        CacheLabel.Text = "Searched cached Fast DataSet object";
        DurationLabel.Text = Duration.Milliseconds.ToString() +  " milliseconds ";
   }
```

Just by creating the Directories data table, we are able to reduce the time it took to search for files by roughly 30 percent, as shown by the snapshot view in Figure 2-5.



Searched cached Fast DataSet object
230 milliseconds

Id	FileName	DirectoryName
2139	WindowsApplication1.exe	Debug
2140	WindowsApplication1.pdb	Debug
2141	WindowsApplication1.exe	Debug
2142	WindowsApplication1.exe.incr	Debug
2143	WindowsApplication1.Form1.resources	Debug
2144	WindowsApplication1.pdb	Debug

Figure 2-5. The execution time for searching the data set by using the restructured data set

It's Still Too Slow!

You might not be a happy camper because all our work didn't provide a dramatic increase in performance. You might be curious why you should bother caching your data when you can retrieve it from the SQL Server within 90 milliseconds, as compared with 380 milliseconds from a regular data set cache or 230 milliseconds from the so-called fast data set cache.

There are, however, the following advantages of using the in-memory data cache:

- Keeping data in memory keeps your database available for more business-critical transactions. For example, an eCommerce site processes large amounts of data to display information to the users. Information such as product catalog, pricing, promotion, etc. doesn't change very often. Keeping the database available for critical business transactions, such as order processing, order fulfillment, and merchandising, contributes significantly to running business processes smoothly.

- Keeping data in memory also reduces network traffic on your network. Most Web sites use a separate server for keeping their database. If you make a trip to the database for all your data needs, you will generate a large amount of network traffic.

BEST PRACTICE *Keeping less-dynamic and often-used data in cache keeps the database and network available for other important processes. Architecturally, you will have to accept a slight degradation in performance for higher availability of these critical components.*

Refreshing Cached Data

The ASP.NET framework doesn't provide an elaborate mechanism for refreshing cached data if it is changed in the database. You can probably rig something to make it work. You can perhaps create an assembly and register it as a COM component by using the Regasm.exe tool. Once the assembly is registered as a COM component, you can fire a trigger on the database update to call a stored procedure that can invoke your assembly as a COM object. In your assembly, you can clear out the cache. The ASP.NET framework, however, does provide an elegant mechanism for receiving notification if the data is stored in a file instead of in the database. We would recommend that you consider keeping your cache candidate data in an XML file. While you develop designs for your system, you should determine which kinds of information make good candidates for caching. There are no hard and fast rules on finding such information. Each application is unique with its own set of data requirements. The general rule of thumb is to recognize the information that changes less frequently and needs to be presented consistently for every user.

You can export data from the database into an XML file fairly easily. All you have to do is to create a DataSet object by running one or many SQL statements. The data adapter object provides the flexibility of executing multiple SELECT commands and populating more than one data table from their result sets. Once you have filled all needed data tables, make sure to create appropriate relationships. These relationships will help you greatly when you attempt to filter the information contained in the data set. After creating the DataSet object, you can simply call its WriteXML method to save its content to an XML file.

Extracting Data from a Database and Saving It into an XML File

Listing 2-5 shows how to extract data from a database and save it to an XML file. It starts by creating a DataSet object by using multiple SQL statements and persistence of its content in an XML file.

Listing 2-5. Data Persistence in an XML File

```
string XmlFileName;
SqlCommand MyCommand = new SqlCommand();

MyCommand.Connection = new SqlConnection("your connection string");
MyCommand.CommandText = " select * from files ";

SqlDataAdapter MyAdapter = new SqlDataAdapter();
MyAdapter.SelectCommand = MyCommand;

DataSet MyFastDataSet = new DataSet();

MyCommand.Connection.Open();

// Filling one table in the data set.
MyAdapter.Fill(MyFastDataSet, "Files");

MyCommand.CommandText = " select Distinct DirectoryName from Files ";

// Filling another table in the data set.
MyAdapter.Fill(MyFastDataSet, "Directories");

DataColumn ParentColumn =
  MyFastDataSet.Tables["Directories"].Columns["DirectoryName"];
DataColumn ChildColumn = MyFastDataSet.Tables["Files"].Columns["DirectoryName"];

// Creating relationship between both tables.
MyFastDataSet.Relations.Add("Directory_File_Relation", ParentColumn,
ChildColumn);
```

```
MyConnection.Close();
MyAdapter.SelectCommand.Dispose();
MyAdapter.Dispose();


// Using MapPath method to get physical file path for virtual
// file location.
XmlFileName = Request.MapPath("CacheFile/MyFastDataSet.xml");

// Saving data set content to an XML file. The WriteSchema
// enumeration is used to cause the data set to write schema,
// as well as, content.
MyFastDataSet.WriteXml(XmlFileName, XmlWriteMode.WriteSchema);

// Disposing DataSet because we don't need it anymore.
MyFastDataSet.Dispose();
```

BEST PRACTICE *We recommend using the DataSet object to create the XML file if you are planning to use the data set to cache information.*

Yes, you can create the XML file by using a variety of mechanisms, including the SQL Server 2000 built-in XML generation features. However, in our experience, we have found it easier to convert an XML file to a data set if the file was originated from a DataSet object.

Also, make sure to write the schema to the file as well. Without writing the schema, you will lose important column-level information, such as data type. Without saving the schema, you will also lose relationships between tables, which is sure to cost you several hours of debugging to discover the real cause of the problem.

Losing a column's data type isn't much fun either. It will start to affect you when you try to sort the column by using the DataView object. Because the column won't know its data type, it will default to string, resulting in alphanumeric sorting for numeric information. Believe us, no customer likes to see alphanumeric sorting for phone numbers or dates. We have been burned on this one. Once you have saved data in an XML file by using the DataSet object, it is very easy to convert it back into a data set. In fact, it is as simple as calling the ReadXML method on the DataSet object and providing it with the name and the location of the XML file. The trick, however, is to update the cache correctly. Some of you may be thinking, what's so hard about updating cache? You simply get the data set from cache and call its ReadXML method.

CAUTION *Never ever, we repeat, never ever directly update the cached DataSet object from the XML file.*

When you are ready to refresh cache, make sure to create a new DataSet object and populate it by calling its ReadXML method. Once the new DataSet object is populated, throw away the current DataSet object that is residing in cache and insert the newly created DataSet object in its place.

The reason you shouldn't use the currently cached DataSet object to refresh data is that the ReadXML method can take up to several minutes if the XML file is large. Of course, you don't want to affect users who are innocently surfing your Web site by silently pulling the rug from under their feet (or updating their data as they are using it).

Populating a new DataSet object from an XML file doesn't affect Web site users. You should still be careful while overwriting cache with the new object. Even though it only takes a fraction of a second to place a new DataSet object in cache, it is best to synchronize access to the Cache object while performing this operation.

Refreshing a Cached DataSet Object

Make sure to read the code in Listing 2-6 to see an example of how to refresh cache appropriately. It uses the lock keyword to synchronize all threads that are trying to access the cache to make sure that only one thread gets to update it at a time. Because the ASP.NET runtime engine processes each Web request as a separate thread, this logic causes all Web requests that are trying to access cache to synchronize while it is being updated.

Listing 2-6. An Example of Refreshing Cache Appropriately

```
// Using MapPath method to convert virtual file path to physical path.
string XmlFileName = Request.MapPath("CacheFile/MyFastDataSet.xml");

// Creating new DataSet object.
DataSet MyFastDataSet = new DataSet();

// Populating newly created DataSet object from XML file.
// Make sure to use ReadSchema enumeration; otherwise,
// the DataSet object will not have data types and relations.
MyFastDataSet.ReadXml(XmlFileName, XmlReadMode.ReadSchema);
```

```
// Synchronize access to the Cache object by using the lock keyword.
// The lock keyword makes sure that no other thread can access the
// Cache object while it's being updated with new DataSet object.
lock(Cache)
{
Cache["Result"] = MyFastDataSet;
}
```

Expiring Cache

The caching mechanism provides us with the flexibility of expiring cached data by using a variety of methods. You can make a cached object dependent on a specific file. If the file changes, the caching mechanism will be notified and will expunge the cached object from the memory. Similarly, you can make a cached object dependent on a directory. If any change is made to that directory or any of its subdirectories, the object will be expunged from cache. You can make various cached objects dependent on each other, in which case, when a cached object is expired, all its dependent cached objects expire as well. The caching mechanism also provides us with the ability to make cached objects expire after a certain time interval. We are free to specify either a fixed time or a sliding duration.

The CacheItemRemovedCallback Delegate

Another nice feature provided by the caching mechanism is its ability to notify us when our objects are removed from cache. This notification is done by using the CacheItemRemovedCallback delegate, which is defined in the System.Web.Caching namespace. By using this delegate, we can receive the expired cached object, its key name, and a reason for the expiration.

BEST PRACTICE *If you use an XML file to load a data set in memory and keep it cached, we recommend that you set the dependency of your cached data set with that file.*

By using this mechanism, your code can receive notification when the file is changed. Once you receive such notification, you can read this recently updated file, create a new DataSet object, and replace the cached object with it.

This approach allows you to refresh the data simply by updating the underlying XML file and to let the ASP.NET runtime and its caching mechanism do the rest.

Listing 2-7 shows how you can set up dependency with a file, receive expiration notification, and set various expiration options.

Listing 2-7. Setting Up Dependency with a File

```
// Make sure to include System.Web.Caching namespace.
private void AddToCache()
{
     // Using MapPath method to convert virtual file path to physical path.
     string XmlFileName = Request.MapPath("CacheFile/MyFastDataSet.xml");

     // Creating new DataSet object.
     DataSet MyFastDataSet = new DataSet();

     // Populating newly created DataSet object
     // from XML file. Make sure to use ReadSchema
     // enumeration; otherwise, the DataSet object will not
     // have data types and relations.
     MyFastDataSet.ReadXml(XmlFileName, XmlReadMode.ReadSchema);

     CacheDependency MyDependency;
     CacheItemRemovedCallback onRemove;

     // Setting the dependency object to the XML file.
     MyDependency = new CacheDependency(XmlFileName);

     // Creating the delegate object and assigning it the
     // name of the method that should be called when cached
     // data set is expired.
     onRemove = new CacheItemRemovedCallback(RemoveResultCallback);

     // Inserting the newly created DataSet object in cache
     // and assigning it the dependency, the delegate, and expiration
     // values. In this example, the cached data set will expire
     // 24 hours after it is placed in cache.
     Cache.Insert("Result", MyFastDataSet, MyDependency,
       DateTime.Now.AddHours(24),
       TimeSpan.Zero, CacheItemPriority.Normal, onRemove);
}

// This method will be called when the cached data set is expunged.
// It receives the expired object, its key, and the reason for
// expiration as specified in CacheItemRemovedCallback delegate.
private void RemoveResultCallback(string key, object removedObject,
            CacheItemRemovedReason removeReason)
```

```
{
    // We simply call the AddToCache() method to reread the
    // XML file and refresh cached data set.
    AddToCache();
}
```

The Cache object provides a method called Add. We tend to stay away from using this method. We have yet to understand the reason for this method. Its existence seems redundant and, quite frankly, annoying. The Add method works in the same manner as the Insert method, but it returns the object you just inserted in cache, which is the part that we have yet to comprehend. Why would you want to receive the object you just provided to this method as a parameter? You already have it, don't you? Even if you receive the returned object, you lose all your type information because the returned object is of the generic type "object." The Add method also throws exceptions if you try to add an item with the key name that already exists. As a best practice, we recommend that you always check to see if another object uses the same key name before you use that key for your object. We recommend that you perform your own checks prior to using the Insert method instead of using the Add method, and handle exceptions as they happen.

Understanding the CacheDependency Class

The CacheDependency class is very versatile. With this class, you can set the expiration of your cached object by using a variety of mechanisms, from sensing file changes to the expiration of another cached object. This class doesn't expose many properties; rather, it relies on you to provide appropriate information in its constructor. By using one of its eight overloaded constructors, you can configure this object to create a dependency between your cached object and any number of files, directories, and other cached objects. You can even specify the date and time when you would like the dependency to take effect.

NOTE *When you use the delegate to receive cache expiration notification, the object representing the page stays in memory until the cache is purged. Normally, the ASP.NET Page objects are created for each request and destroyed after the request is processed. However, when the Cache object holds a reference to a method on the Page object, the Page object needs to stay instantiated. Otherwise, the Cache object may end up with a null reference in its delegate. The Page object, however, is not reused to process any more Web requests; it stays in memory, waiting to receive the callback from the Cache object.*

Losing Cached Information When Restarting the Application

It is important for you to know that the Cache object keeps its content in the memory of the Web application. If the Web application restarts for any reason, it loses its cached objects. A Web application can restart in many ways, some that you might already know and some that may surprise you. One way a Web application restarts is when the Web server is restarted, which can happen either accidentally or deliberately. However, the Web application also restarts if any change is made in either the Web.config or the Machine.config file. The ASP.NET runtime senses changes to its configuration files and starts a new application domain that loads the changed configuration file in memory. The new application domain creates its own Cache object, causing us to lose objects stored in the previous application domain's cache.

This behavior is the classic example of the compromises that are made in the ASP.NET framework for providing other, more needed, benefits. The ASP.NET framework designers wanted a mechanism for configuring Web applications that does not depend on Windows registry. The alternative was to use XML-based configuration files. However, reading configuration from file I/O can be fairly intense and may become the performance bottleneck in high-use scenarios. The ASP.NET runtime designers chose to read the configuration files once, when the Web application starts, and keep its information cached in memory. They also used the file dependency feature to sense changes made to the configuration files. Once a file change is detected, the ASP.NET runtime does not update the configuration information cached in the memory and instead creates a new application domain. The benefit of creating a new application domain is that all existing Web requests can continue to use the existing configuration and new Web requests are diverted to the new application domain. The disadvantage of creating a new application domain is that it doesn't bring the cached objects from the previous application domain and results in empty cache.

The Scalability Issue with Cached Objects

As we mentioned earlier, the cached objects are stored in the Web application memory, which can cause problems in the Web farm environment. It is not uncommon for high-traffic Web sites to use multiple Web servers. Often a load balancer is used to divert the incoming Web request to the least busy Web server. Because cached objects are stored in the Web application's memory, they are not accessible from another server in the farm, which results in unpredictable behavior, as there is no good way of knowing which server in the farm will be used to process the request. Usually when someone talks about scalability problems with a given technology, the conversation follows with bad publicity, expert opinions on what should have been, and the tendency of developers to stay away from using the technology. On the contrary, we are here to tell you that the scalability problem with the Cache object is a wonderful thing. But before we tell you why, let us also say that if you are experiencing a scalability problem with the Cache object, you built your application wrong.

BEST PRACTICE *The Cache object is not meant for keeping user-specific information.*

If you are keeping information pertaining to a specific user in the Cache object, you built your Web application wrong. The purpose of the Cache object is to store information available to all users of your Web application. If you need to keep user-specific information in memory, you should use the Session object instead.

If the information in the Cache object is not usercentric, then you don't have to worry about maintaining any specific user's session across the server farm.

You should always provide a mechanism for retrieving information from the physical store, such as the database, if the cache is found empty. Standardizing on this logic helps you to scale your application in a server farm environment. If a specific server in the farm doesn't have your cached information, your application will retrieve it from the physical store and add it to the cache. Eventually, all servers in the farm will have retrieved the cached information.

Some of you may wonder why the ASP.NET framework doesn't provide the ability to scale the Cache object by using either a state server or relational database, similar to the way the Session object can be scaled. The answer is quite simple: There are significant performance implications in keeping information in a state server or a database session. The performance degradation can be justified in the case of the Session object because of its ability to maintain usercentric information across multiple servers in the farm. On the other hand, the performance penalty can't be justified for information that does not pertain to a specific user of the Web application. If you are skeptical about the performance issues with the state server and database sessions, make sure to read the section "Understanding the Session Object" later in this chapter.

Turbo-Charging with Output Caching

Output caching can boost the performance of your Web application by multiple orders of magnitude. We highly recommend every one of you to pay close attention to the scenarios in which output caching can be a viable option.

The rule of thumb is to select pages that are used very often and don't change much. A really cool feature of the ASP.NET output caching mechanism is its ability to generate different cached versions of the same page based on a variety of input parameters, such as query string values, form post values, etc. This feature provides us with the ability to generate different cached outputs for different users of our site. You can keep the user ID either embedded in the query string or as a hidden field on the page. You can then use the VaryByParam attribute to cache different page outputs for each user. It is as simple as adding the following line of code at the beginning of the .aspx file.

```
<%@ OutputCache Duration="60" VaryByParam="UserId" %>
```

If you are not so inclined to keep the user ID in a query string or a hidden field, you can always keep it in a cookie and use the VaryByCustom attribute to accomplish the same purpose.

```
<%@ OutputCache Duration="60" VaryByCustom="UserId" %>
```

The UserId parameter doesn't exist, but the VaryByCustom attribute allows you create it by writing a few lines of code in the GetVaryByCustomString method in the Global.asax file. In this method, you can write the following line of code to cache different outputs for each user of your site.

```
public override string GetVaryByCustomString(HttpContext Context, string Args)
{
    return "UserId=" + Request.Cookies["UserIdCookie"].Value;
}
```

A good candidate for output caching is the home page of your Web application. In most applications, the home page is the most visited page. If the home page for your application is dynamically generated, we highly recommend that you consider enabling output caching on this page. If your home page doesn't show different content to different users, all you have to do is to write the following line of code at the beginning of the page:

```
<%@ OutputCache Duration="3600" VaryByParam="None" %>
```

The previously mentioned code will cause your home page to be cached for one hour (3,600 seconds). By setting the VaryByParam attribute to None, you can specify that this page should appear the same for all users of your site. If you need to show different versions of the home page to different users, you can use any of the techniques mentioned previously.

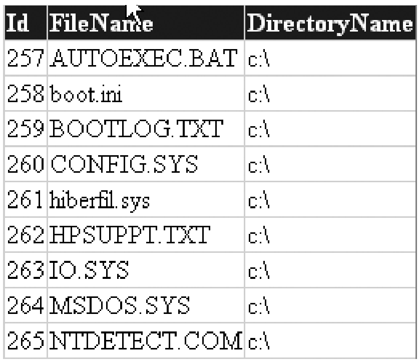
Another good candidate for output caching are pages that show product catalogs, search results, navigation links, etc. In fact, with a little creativity, you can even cache pictures and PDF files. Keep reading and you will find out how.

If you don't believe that output caching can turbo-boost your Web application, let me prove it to you with code examples and concrete performance test results. In Listing 2-8, we will create a simple page that retrieves some data from a SQL Server database and shows it on the screen. We will then use Application Center to performance-test this page with and without output caching. All you nonbelievers tag along with us and see for yourselves.

Listing 2-8. Retrieving Some Data from the SQL Server Database

```
private void Page_Load(object sender, System.EventArgs e
{
     SqlCommand MyCommand = new SqlCommand();
     MyCommand.Connection = new SqlConnection("Your Connection String");

     // Selecting 100 records from the database.
     MyCommand.CommandText = " select top 100 * from files ";

     SqlDataAdapter MyAdapter = new SqlDataAdapter();
     MyAdapter.SelectCommand = MyCommand;

     // Filling a data set with result from SQL query.
     DataSet MyDataSet = new DataSet();

     MyCommand.Connection.Open();
     MyAdapter.Fill(MyDataSet, "Files");
     MyCommand.Connection.Close();

     MyAdapter.SelectCommand.Dispose();
     MyAdapter.Dispose();

     // Showing the content from data set in the data grid control.
     ResultGrid.DataSource = MyDataSet.Tables[0].DefaultView;
     ResultGrid.DataBind();
}
```

When you compile and run this code, you will see the page shown in Figure 2-6.



Id	FileName	DirectoryName
257	AUTOEXEC.BAT	c:\
258	boot.ini	c:\
259	BOOTLOG.TXT	c:\
260	CONFIG.SYS	c:\
261	hiberfil.sys	c:\
262	HPSUPPT.TXT	c:\
263	IO.SYS	c:\
264	MSDOS.SYS	c:\
265	NTDETECT.COM	c:\

Figure 2-6. The page that demonstrates output caching

Let's create three versions of this page. The first version will not use any output caching, the second version will use client output caching, and the third version will use the server output caching.

The client output caching option caches the page at the browser, whereas the server output caching option caches it at the Web server. You will find the performance difference between these two options in Listing 2-9 and Figure 2-7.

Listing 2-9. The HTML Contained in the .aspx File for All Three Output Caching Options

```
<!-- HTML for no output caching -->
<%@ Page language="c#" Codebehind="OutputCaching-None.aspx.cs"
AutoEventWireup="false" Inherits="ASPNET.OutputCaching___None" %>
<HTML>
  <HEAD><title>OutputCaching - None</title></HEAD>
  <body MS_POSITIONING="GridLayout">
    <form id="OutputCaching_None" method="post" runat="server">
<asp:datagrid id=ResultGrid style="Z-INDEX: 103; LEFT: 8px;
POSITION: absolute; TOP: 8px" runat="server">
<HeaderStyle Font-Bold="True" ForeColor="White" BackColor="Blue">
</HeaderStyle></asp:datagrid>
     </form></body>
</HTML>

<!-- HTML for client output caching -->
<%@ OutputCache Duration="10" VaryByParam="None" Location="Client" %>
<%@ Page language="c#" Codebehind="OutputCaching-Client.aspx.cs"
AutoEventWireup="false" Inherits="ASPNET.OutputCaching" %>
```

```
<HTML>
  <HEAD><title>OutputCaching</title></HEAD>
  <body MS_POSITIONING="GridLayout">
    <form id="OutputCaching" method="post" runat="server">
<asp:datagrid id=ResultGrid style="Z-INDEX: 103; LEFT: 16px;
POSITION: absolute; TOP: 30px" runat="server">
<HeaderStyle Font-Bold="True" ForeColor="White" BackColor="Blue">
</HeaderStyle></asp:datagrid></form></body>
</HTML>

<!-- HTML for server output caching -->
<%@ OutputCache Duration="10" VaryByParam="None" Location="Server" %>
<%@ Page language="c#" Codebehind="OutputCaching-Server.aspx.cs"
AutoEventWireup="false" Inherits="ASPNET.OutputCaching___Server" %>
<HTML>
  <HEAD>
    <title>OutputCaching - Server</title>
  </HEAD>
  <body MS_POSITIONING="GridLayout">
    <form id="OutputCaching_Server" method="post" runat="server">
<asp:datagrid id=ResultGrid style="Z-INDEX: 103; LEFT: 8px;
POSITION: absolute; TOP: 8px" runat="server">
<HeaderStyle Font-Bold="True" ForeColor="White" BackColor="Blue">
</HeaderStyle></asp:datagrid></form></body>
</HTML>
```

We created Application Center test scripts that run each of these pages for 10 seconds simulating 10 simultaneous users. Application Center is a performance testing tool that ships with the enterprise architect version of Visual Studio .NET. It helps us to create test scripts to test Web sites for performance. The graph in Figure 2-7 shows the result.

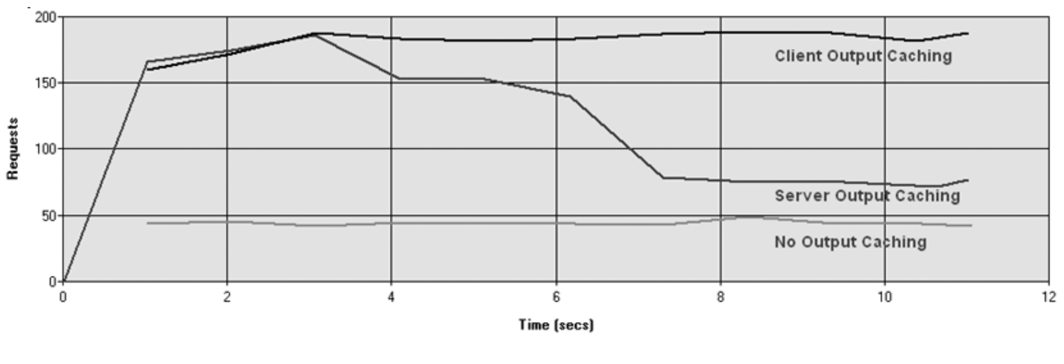


Figure 2-7. Performance test result from Application Center test scripts

Figure 2-7 clearly shows that the page that used output caching ran significantly faster than the page that didn't use it. Wondering why the page with the Server Output Caching option gradually slowed down as time went by? The slowdown happened because the Web server wasn't able to handle this many simultaneous users. This decline in performance is related to the capability of the hardware.

This is precisely the reason why the page with the Client Output Caching option demonstrated such good performance. It didn't need to make a trip back to the server. Instead, it used its local cached output.

BEST PRACTICE *Output caching is the turbo-charger for your Web application. Use it anywhere you possibly can.*

Deciding Whether to Maintain View State

In the good old days of ASP programming, most of us occasionally created a hidden field on the Web page for the sole purpose of retaining information during roundtrips to the server. The ASP.NET design team at Microsoft realized the importance of this mechanism and provided us with an easy to use and safe mechanism to accomplish the same purpose.

View state is simply an encoded piece of information that is written in a hidden field in the Web page. You can add any information you want to the view state by using the following syntax:

```
ViewState["MyName"] = "Farhan Muhammad";
```

Just like Cache and Session, ViewState is also a name/value collection. You can put any information in it and associate it with a key name. When the page is rendered, the information contained in the ViewState name/value collection is written in a hidden field named __VIEWSTATE. The structure and the content of this field are shown in the following code. Notice that the content of this field is encoded and therefore relatively safe from prying eyes.

```
<input type="hidden" name="__VIEWSTATE"
value="dDwtMTI3OTMzNDM4NDt0PHA8bDxNeU5hbWU7PjtsPEZhcmhhbiBNdWhhbW1hZ
Ds+Pjs7Pjs+hVkdc1OkDgvCSLOkCPMvlbKn5Yk=" />
```

The server controls also use view state to maintain their state. As you know, a key benefit of using ASP.NET server controls is their ability to retain their content during post-back events. This ability enables the programmers to focus their time and energy on providing business values and not in writing plumbing code.

As simple as it seems, the view state can quickly become evil and can tax your network to its limits.

The data bound server controls, such as drop-down lists, check box lists, data grid, repeater, etc., by default will persist their data to the view state hidden field. All that information ends up traveling from the browser to the server during each post-back event.

We would recommend using either Session or Cache object to store database-generated information and rebinding your controls on each post-back. The performance difference between maintaining view state and rebinding controls from cache is negligible. However, reduced view state size results in a lower amount of information travel from browser to the server, resulting in better throughput.

On the other hand, storing information in cache taxes memory use. However, we would prefer to be taxed on memory rather than network resources. Internet-enabled applications rely on third-party networks to facilitate communication between the browser PC and the server PC. The geographic distance between these computers could be significant, resulting in many hops before the information gets to its destination.

BEST PRACTICE *As a rule of thumb, we would recommend reducing view state size by setting the EnableViewState property of the server controls to false. You can store the information in either session or cache and rebind your server controls on each post-back.*

This approach results in higher memory use on the server but reduces the amount of information that travels across the Internet to reach its destination. In a high-traffic environment, you are sure to get better throughput by using this approach.

There is one caveat, though. If you choose to disable view state and use the server's memory to remember the information, then you won't be able to discover whether the user changed the information before the post-back event occurred. The ASP.NET runtime uses view state to discover if the user changed the value of the control and fired the OnChange event if necessary. In the absence of view state, ASP.NET runtime tries to compare the value of a control received during post-back with the value of the control declaratively assigned in the HTML, which could cause the OnChange event always to fire if the control's value is different than its default value. This method is more applicable to input type controls, such as text box. Needless to say, such behaviors can be very difficult to discover and debug. In light of such issues, we recommend the following caution associated with disabling view state.

CAUTION *If you need to capture a server-side OnChange event from a particular control, make sure to leave view state enabled. Otherwise, you may consider disabling it to reduce the page size.*

View State: Behind the Scenes

It is important for you to understand the mechanism that makes view state possible. Every time a page is requested from the Web server, the ASP.NET runtime creates a new object of the Page class. This object receives all input values from the request, reads the hidden __VIEWSTATE field, decrypts its content, and uses it to populate the values of appropriate server controls. This process is called server controls rehydration.

After the request is finished processing on the server, the Page object is destroyed. Before the Page object is destroyed, however, the server controls are given an opportunity to store their information in the hidden __VIEWSTATE field. This process is called server controls dehydration.

The amount of effort involved with dehydrating and rehydrating server controls can be significant, especially if the information is of a large size. In most cases, the amount of work involved is not much smaller than binding the controls to a data source.

To prove our point, we will create a Web page that runs several SQL statements on the Northwind database and uses their results to populate a number of drop-down lists and a data grid control. The results from the SQL statements are later cached by using a DataSet object.

We will then create an Application Center test script that conducts performance tests on this Web page by using two scenarios. In one scenario, we will enable view state for all controls and not rebind the controls with the cached data set. In the second scenario, we will disable view state for all drop-down lists and the data grid control and use the cached DataSet object to rebind to these controls. You will see that the Web page worked with almost identical performance in both scenarios.

Listing 2-10 shows the code for a Web page that contains several drop-down lists and a data grid control. These controls are populated from the Northwind database.

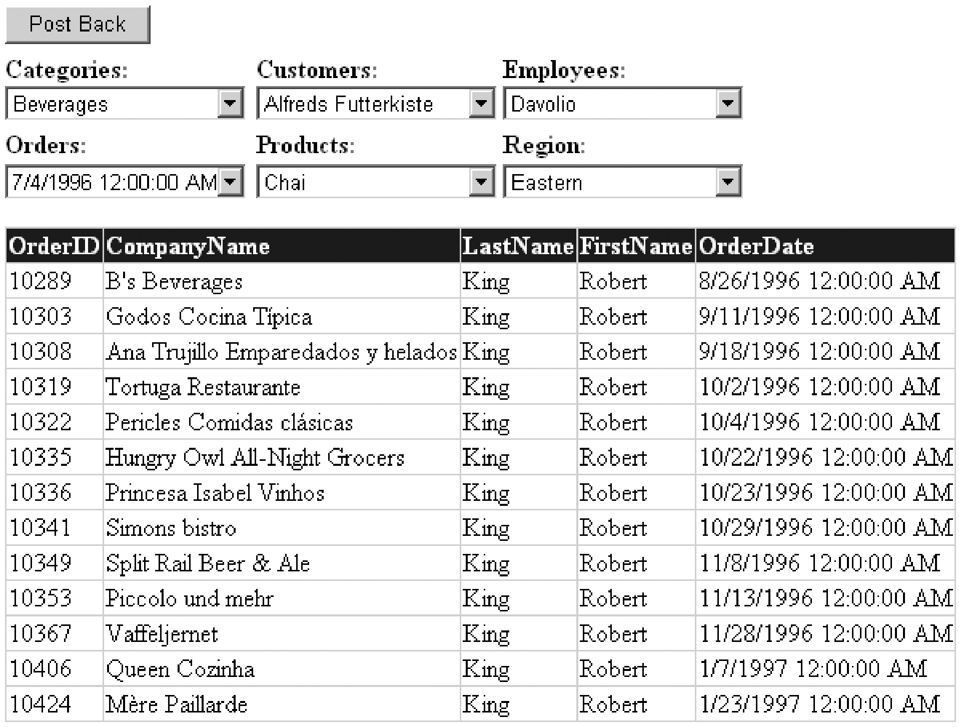
Listing 2-10. Web Page Containing Various Data-Bound Controls

```
private void Page_Load(object sender, System.EventArgs e)
{
     // To bind controls to data source on every request,
     // simply comment the following conditional statement.
     if (!IsPostBack)
     {
          DataSet MyDataSet;
```

```
// If cache is empty, run SQL and generate DataSet from
// the database.
if (Cache["Orders"] == null)
{
     MyDataSet = new DataSet();
     SqlDataAdapter MyAdapter = new SqlDataAdapter();

     SqlCommand MyCommand = new SqlCommand();
     MyCommand.Connection = new SqlConnection("Connection String");
     MyAdapter.SelectCommand = MyCommand;

     MyCommand.Connection.Open();

     // Creating six data tables. They will be used to
     // bind to drop-down lists.
     MyCommand.CommandText = " SELECT * FROM Categories ";
     MyAdapter.Fill(MyDataSet, "Categories");

     MyCommand.CommandText = " SELECT * FROM Customers ";
     MyAdapter.Fill(MyDataSet, "Customers");

     MyCommand.CommandText = " SELECT * FROM Employees ";
     MyAdapter.Fill(MyDataSet, "Employees");

     MyCommand.CommandText = " SELECT * FROM Orders ";
     MyAdapter.Fill(MyDataSet, "Orders");

     MyCommand.CommandText = " SELECT * FROM Products ";
     MyAdapter.Fill(MyDataSet, "Products");

     MyCommand.CommandText = " SELECT * FROM Region ";
     MyAdapter.Fill(MyDataSet, "Region");


     // Filling the data grid with order summary. This
     // SQL statement returns 72 records.
     MyCommand.CommandText = " select Orders.OrderID," +
     " Customers.CompanyName, Employees.LastName," +
     " Employees.FirstName, Orders.OrderDate," +
     " Orders.RequiredDate, Orders.ShippedDate" +
     " from Orders, Customers, Employees" +
     " where Orders.CustomerID = Customers.CustomerID" +
     " and Orders.EmployeeID = Employees.EmployeeID" +
     " and Employees.LastName = 'King' ";
```

```
                MyAdapter.Fill(MyDataSet, "OrderList");

                MyCommand.Connection.Close();
                MyCommand.Connection.Dispose();
                MyCommand.Dispose();

                // Adding newly created data set to cache.
                Cache["Orders"] = MyDataSet;
            }
            else
            {
                // Because cache is not empty, retrieving the data set
                // from it.
                MyDataSet = Cache["Orders"] as DataSet;
            }

                // Binding the data set to all six drop-down lists and
                // the data grid control.
                CategoriesDropDown.DataSource =
                      MyDataSet.Tables["Categories"].DefaultView;
                CustomersDropDown.DataSource =
                      MyDataSet.Tables["Customers"].DefaultView;
                EmployeesDropDown.DataSource =
                      MyDataSet.Tables["Employees"].DefaultView;
                OrdersDropDown.DataSource =
                   MyDataSet.Tables["Orders"].DefaultView;
                ProductsDropDown.DataSource =
                   MyDataSet.Tables["Products"].DefaultView;
                RegionDropDown.DataSource =
                   MyDataSet.Tables["Region"].DefaultView;
                ResultGrid.DataSource =
                   MyDataSet.Tables["OrderList"].DefaultView;

                CategoriesDropDown.DataBind();
                CustomersDropDown.DataBind();
                EmployeesDropDown.DataBind();
                OrdersDropDown.DataBind();
                ProductsDropDown.DataBind();
                RegionDropDown.DataBind();
                ResultGrid.DataBind();
            }
    }
```

When you compile and run the code shown in Listing 2-10, you will see the Web page shown in Figure 2-8.



Categories: Beverages
Customers: Alfreds Futterkiste
Employees: Davolio
Orders: 7/4/1996 12:00:00 AM
Products: Chai
Region: Eastern

OrderID	CompanyName	LastName	FirstName	OrderDate
10289	B's Beverages	King	Robert	8/26/1996 12:00:00 AM
10303	Godos Cocina Típica	King	Robert	9/11/1996 12:00:00 AM
10308	Ana Trujillo Emparedados y helados	King	Robert	9/18/1996 12:00:00 AM
10319	Tortuga Restaurante	King	Robert	10/2/1996 12:00:00 AM
10322	Pericles Comidas clásicas	King	Robert	10/4/1996 12:00:00 AM
10335	Hungry Owl All-Night Grocers	King	Robert	10/22/1996 12:00:00 AM
10336	Princesa Isabel Vinhos	King	Robert	10/23/1996 12:00:00 AM
10341	Simons bistro	King	Robert	10/29/1996 12:00:00 AM
10349	Split Rail Beer & Ale	King	Robert	11/8/1996 12:00:00 AM
10353	Piccolo und mehr	King	Robert	11/13/1996 12:00:00 AM
10367	Vaffeljernet	King	Robert	11/28/1996 12:00:00 AM
10406	Queen Cozinha	King	Robert	1/7/1997 12:00:00 AM
10424	Mère Paillarde	King	Robert	1/23/1997 12:00:00 AM

Figure 2-8. A partial snapshot of the Web page contains information from the Northwind database

Clicking the Post Back button resubmits the page. We deliberately didn't write any code in the event handler for the button because we didn't want to incur any overhead during post-back, other than the maintenance of view state for the server controls.

Now that you have seen the code and the user interface for this Web page, let us show you the results from the performance tests we conducted, as shown in Figure 2-9. As promised, we conducted the performance test with and without enabling view state for all drop-down lists and the data grid control.

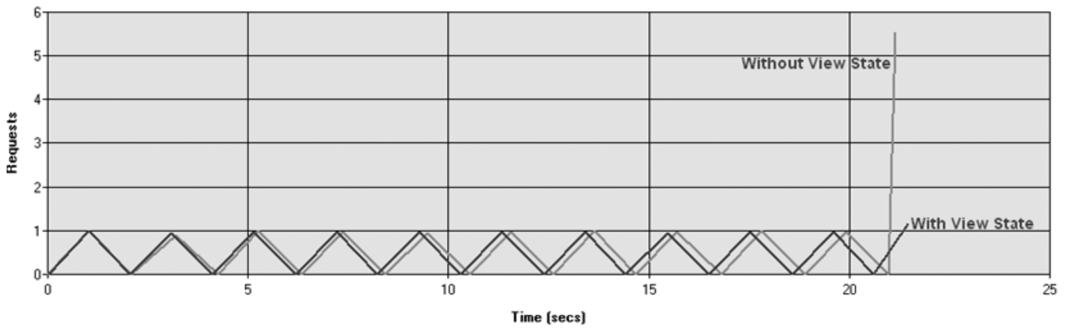


Figure 2-9. The result of Application Center performance test with and without using view state

There you have it! There is not much difference in performance whether you use view state to maintain information during post-back or rebind your server controls to a data source on each post-back. The only assumption made is that you must cache the data source to get comparable performance.

Understanding the Session Object

As you know, the Session object enables you to maintain user-specific information in memory cache. The ASP.NET runtime automatically creates this object when a new user visits your Web site, and it remains in memory for a specific period after the user stops using the Web site. You can alter the duration for which the Session object remains in memory by changing the session timeout setting in the Web.config file. The timeout is set to 20 minutes by default.

In this section, we will show you the impact on performance for different session configurations. The ASP.NET framework enables us to store session state either in Web server memory, in a Windows service, or in a SQL Server database. Each of you has to decide the configuration that best suits your environment. We will attempt to provide enough information to make this selection easy for you. You should remember that session configuration is simply the matter of setting a few switches in the Web.config file and therefore you can change them any time you feel a need to reconfigure your mechanism for maintaining session state.

In-Process Session

The default session state setting is InProc. When configured with this setting, the ASP.NET runtime keeps all session information in the Web application's memory. Because the session is stored in the application's memory, it is much faster to get the stored information. On the other hand, if your Web application restarts for any reason, you will lose every object stored in every user's session. You need to choose whether this is an acceptable compromise in your situation. Keeping session "in-process" also prohibits you from scaling your Web applications by using a *server farm*, which consists of multiple Web servers working in tandem. The incoming requests for Web pages are automatically routed to the least busy server in the farm. In this scenario, the in-process session is not a viable option as Web servers in the farm are unable to read each other's in-process memory.

State Server

The ASP.NET framework enables us to store session information by using a Windows service. Because the Windows service runs as an independent process, information stored in the service is not affected when the Web application is restarted. Also, other Web servers in the server farm can communicate with the Windows service to get access to the session information stored in it.

The bottom line is that the state server enables us to scale our Web application by using a server farm. It is also more robust than an in-process session because it does not depend on the Web application's memory. You can restart your Web application or Web server as many times as you please without losing session information. Rebooting the server, on the other hand, will cause you to lose the session information. If you are paranoid about losing session information during an unplanned server reboot, you need to use the database-enabled session instead.

SQL Server Session

We also have the ability to store session information in a SQL Server 7.0 or later database. This option also allows us to scale our Web site in the server farm environment because all servers in the farm can access the same SQL Server database. Common sense tells us that this option would demand the heaviest performance penalty, but you are about to discover that it isn't necessary so. The database-based session management may lose the performance battle if your

Web site handles hundreds of concurrent users. Otherwise, its performance is quite comparable to the state server.

However, you do get other benefits by not using your SQL Server database for session management. At the least, it keeps the database available for more business-critical processing. The choice between state server and the SQL Server state management needs to be made individually for each project. Depending on the application's needs, either choice could be the right choice for you.

BEST PRACTICE *The state server option can out-perform in the heaviest traffic area, but it allows you to keep your database resource available for critical business transactions, such as order processing, credit checking, reporting, etc. The SQL Server state management option allows you to maintain session information robustly, allowing for session recovery after a server reboot. On the other hand, it demands heavy use of your most valuable resource, the database server.*

Performance Testing the Session Object

It is more fun to see the Session object in action than just to talk about it. We conducted a series of performance tests on various session configurations and saw the results in this chapter. We conducted all tests by using the Application Center test and simulated 10 simultaneous Web users.

The tests are divided into two categories. The small session tests use the Session object to store a small piece of information, no more than a few characters. The large session tests store a data set that contains 100 rows retrieved from the Orders table of the Northwind database.

Each of these categories contains tests performed by using a combination of in-process, state server, and SQL Server settings. In the case of state server and SQL Server settings, tests were conducted by keeping the session on the same server as the Web site and on a remote server. Figure 2-10 shows performance from using the in-process session while session contains a small amount of information.

You can clearly see that the Web site was able to serve about 235 requests per second. The occasional drop in performance can be attributed to external influences generated by other applications running on the same server. On the average, the Web site was about able to serve more than 200 requests consistently.

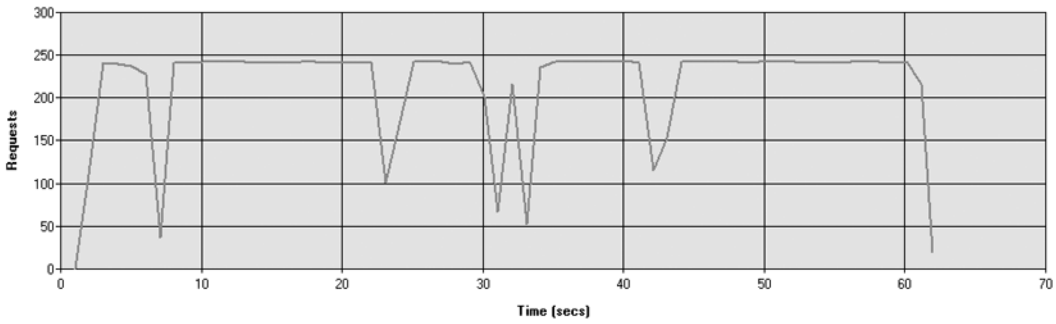


Figure 2-10. Performance using the in-process session containing a small amount of information

Figure 2-11 shows performance from using the state server session containing a small amount of information. The test was conducted in two scenarios. In one scenario, the Web server was used as state server, and in the second scenario, the state server was used on a remote computer.

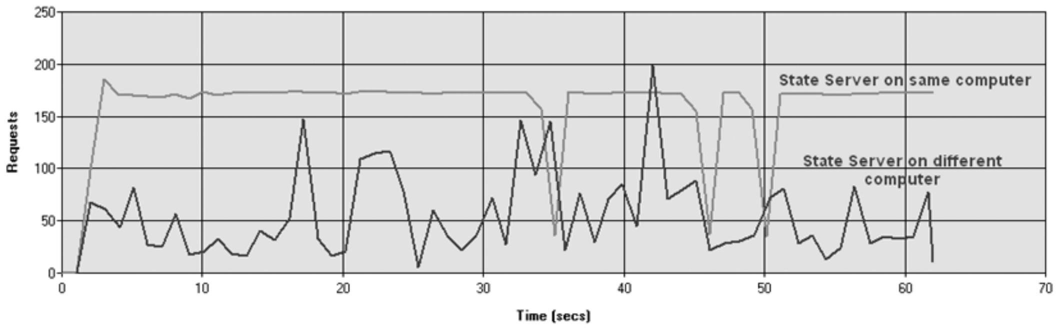


Figure 2-11. Performance using the state server

As you probably expected, the Web site performed better when the same computer that hosted the Web site was used for state server. The performance drops quite significantly when you host the state server on a remote computer. In either case, the performance is quite a bit slower than keeping in-process session.

Now that we have seen the performance from using in-process and state server options, let us show you the performance results from using SQL Server for state management. There are those who like to tell everyone that using a relational database engine for anything results in slower performance than storing the same content in memory. This theory used to be true years ago when the relational database engines were in their infancy. However, the newer versions of such systems are quite fast and can handle a much greater load than before. But why should you take our word? See for yourself!

Figure 2-12 shows performance from using a SQL Server session that contains a small amount of information. The test was conducted in two scenarios. In one scenario, the Web server was used for SQL Server state management, and in the second scenario, the SQL Server was used on a remote computer.

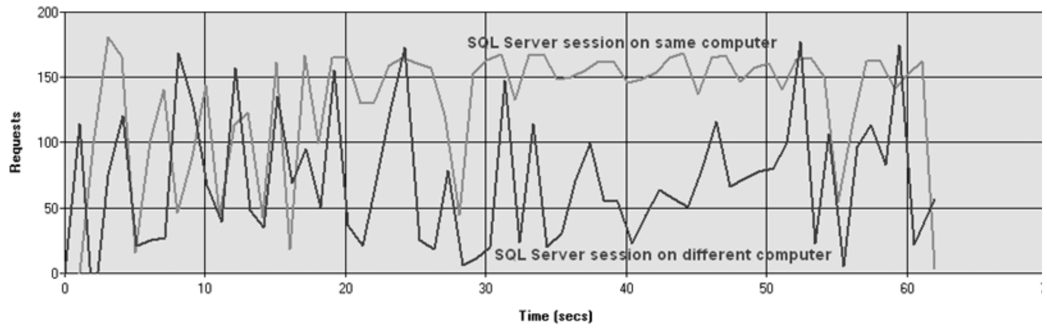


Figure 2-12. Performance using the SQL Server

Surprising, isn't it? Storing session information in a SQL Server database on the same computer provided comparable performance to the state server option. On the other hand, using the SQL Server database on a remote computer provided slightly better performance than using state server on a remote computer. The only strange thing about using SQL Server is that it seems to be quite erratic. The performance seems to change quite drastically from one request to another. We can't explain this phenomenon. Perhaps someone more knowledgeable about SQL Server can shed some light.

Table 2-2 summarizes the test results. Remember, these tests are conducted with a small amount of information in session. The performance slows down significantly as you start to put larger amounts of content in session.

Table 2-2. Performance Results from Using Various Session Configurations

SESSION TYPE	AVERAGE PERFORMANCE	EXPLANATION
In-Process	235 requests/second	Provides the fastest performance because it doesn't have the overhead of out-process marshalling.
State Server, Same Computer	170 requests/second	Provides 30 percent slower performance than the in-process session. However, the content stored in session is not lost when the Web application restarts.
State Server, Different Computer	55 requests/second	There is about 68 percent decrease in performance if you choose to use a remote server as state server. If you are building a Web application for a server farm, you have no choice but to use a remote server for session management. Otherwise, you won't be able to share the same session information across all servers in the farm.

Performance Testing Session with a Large Amount of Content

All our tests so far have used a small amount of content in the Session object. The session performance starts to decline significantly as we put larger amounts of content in it. The change in performance seems to affect only the state server and the SQL Server session management options. We found that the in-process session didn't show any effect on performance, whether we kept small or large quantities of information in it.

The next set of test results show the drastic decline in performance when we put a DataSet object that contains 100 rows from the Orders table of the Northwind database.

Figure 2-13 shows performance from using the SQL Server and state server options. Both options were used while keeping session on the same computer as the Web site. The Session object contained a large quantity of information.

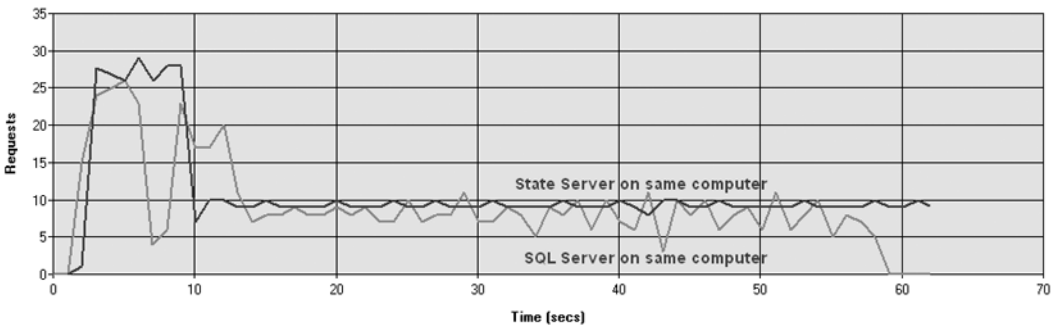


Figure 2-13. Performance using SQL Server and state server

Were we right, or were we right? Actually, we are quite surprised to see such a drastic decline in performance. The slow down is attributed to the fact that the large quantity of information needs to be sent across the process boundary. Regardless of the reason, it is quite amazing to see that the performance declined from 160 requests per second to a meager nine requests per second simply because we chose to store 100 records from the database. Keep this statistic in mind next time you choose to put a large quantity of data in session and still want to use the state server or SQL Server session option.

We repeated the same performance tests, keeping the state server and SQL server on a remote computer. As expected, the performance declined even further, averaging about six requests per second.

Someone Call the Session Doctor!

You will be quite surprised to learn that the decline in session performance with a large quantity of information is the result of a very common mistake. It's a mistake most of us make and never even realize its consequences. Even though the solution is very simple, we are willing to bet a dime that more than 90 percent of the readers of this book either don't know this solution or don't use it on a regular basis.

The solution is to use the ReadOnly session wherever possible. Most Web applications store information in session so that they can be accessed at a later time on a different Web page. We don't always need to modify this information. Often, we simply need to access it. Take our advice and use the following keyword in the @Page directive of every page that only needs read-only access to the session.

```
EnableSessionState=ReadOnly
```

This is it! This is all you need to do to increase the performance by at least 10 times. Figure 2-14 shows performance with the state server ReadOnly session containing a large quantity of information. The state server uses the same computer as the Web server.

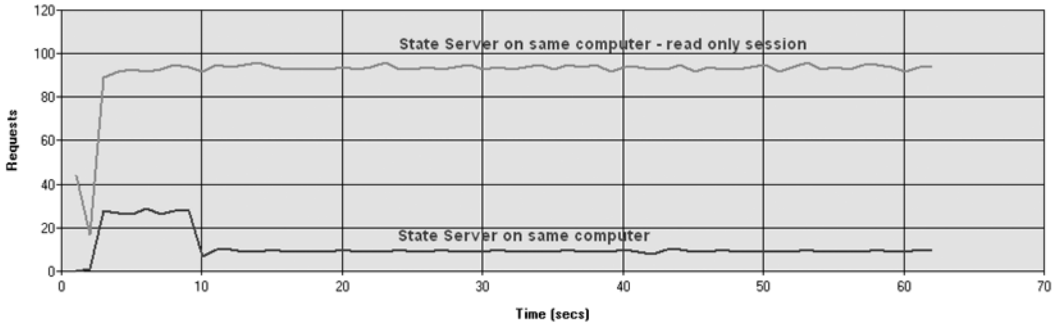


Figure 2-14. Performance using the ReadOnly session

The performance increases sharply during read-only access to session because the ASP.NET runtime doesn't attempt to persist the session information back to its original location when the page finishes rendering, resulting in fewer marshalling calls across process boundaries.

Summary

In this chapter, you learned the facts behind keeping information in server memory. It can be a rewarding experience, providing enormous performance and scalability gains. On the other hand, if used improperly, it can cost your application its performance and precious memory resources. Our intention was to show you the realities behind various cache, session, and view state options, hoping that you will use these features to their full capabilities.