ADO.NET Technical Articles

**Best Practices for Using ADO.NET**

Dennis Lu  
Doug Rothaus  
Microsoft Corporation

July 2002

Applies to:  
Developers with experience using Microsoft® ADO.NET  
Microsoft® .NET Framework

**Summary:** Best practices for writing Microsoft ADO.NET code and suggestions for developers on using the objects available in ADO.NET. (21 printed pages)

**Note** If you are not familiar with ADO.NET and the .NET Framework, see [Accessing Data with ADO.NET](http://msdn.microsoft.com/en-us/library/e80y5yhx.aspx) in the .NET Framework SDK. If you are an ADO programmer interested in migrating your application to ADO.NET, see [ADO.NET for the ADO Programmer](http://msdn.microsoft.com/en-us/library/ms973217.aspx) for more information.

**Contents**

[Introduction](http://msdn.microsoft.com/en-us/library/ms971481.aspx#adonetbest_topic1)  
[.NET Framework Data Providers](http://msdn.microsoft.com/en-us/library/ms971481.aspx#adonetbest_topic2)  
[Working with DataReaders, DataSets, DataAdapters, and DataViews](http://msdn.microsoft.com/en-us/library/ms971481.aspx#adonetbest_topic3)  
[Using Commands](http://msdn.microsoft.com/en-us/library/ms971481.aspx#adonetbest_topic4)  
[Using Connections](http://msdn.microsoft.com/en-us/library/ms971481.aspx#adonetbest_topic5)  
[Integration with XML](http://msdn.microsoft.com/en-us/library/ms971481.aspx#adonetbest_topic6)  
[More Useful Tips](http://msdn.microsoft.com/en-us/library/ms971481.aspx#adonetbest_topic7)

**Introduction**

This article provides you with the best solutions for implementing and achieving optimal performance, scalability, and functionality in your Microsoft ADO.NET applications; it also covers best practices when using objects available in ADO.NET and offers suggestions that can help you optimize the design of your ADO.NET application.

This article contains:

* Information about the .NET Framework data providers included with the .NET Framework.
* Comparisons between the **DataSet** and the **DataReader**, and an explanation of the best use for each of these objects.
* An explanation on how to use the **DataSet**, **Commands**, and **Connections**.
* Information about integrating with XML.
* General tips and issues.

For additional information on ADO.NET best practices, see .[NET Data Access Architecture Guide](http://msdn.microsoft.com/en-us/library/cc446443.aspx) available in the MSDN Library. Note that the .NET Data Access Architecture Guide focuses primarily on architectures that use Microsoft® SQL Server™ 7.0 or later.

The following list provides additional information about ADO.NET:

* Newsgroup: The BDA newsgroup is available through an NNTP newsreader at <news://msnews.microsoft.com/microsoft.public.dotnet.framework.adonet> or through your Web browser at [http://msdn.microsoft.com/newsgroups/loadframes.asp](http://msdn.microsoft.com/newsgroups/loadframes.asp?icp=msdn&slcid=us&newsgroup=microsoft.public.dotnet.framework.adonet).
* Discussion list:

<http://discuss.develop.com/dotnet.html>

**.NET Framework Data Providers**

A data provider in the .NET Framework serves as a bridge between an application and a data source. A .NET Framework data provider enables you to return query results from a data source, execute commands at a data source, and propagate changes in a **DataSet** to a data source. This article includes tips on which .NET Framework data provider is best suited for your needs.

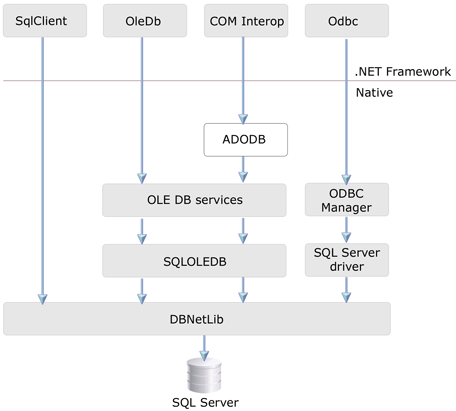
**Which .NET Framework Data Provider to Use?**

To achieve the best performance for your application, use the .NET Framework data provider that is most appropriate for your data source. There are a number of data provider options for use in your applications. The following table provides information about the available data providers and which data sources a data provider is most appropriate for.

|  |  |
| --- | --- |
| **Provider** | **Details** |
| SQL Server .NET Data Provider | Found in the **System.Data.SqlClient** namespace.  Recommended for middle-tier applications using Microsoft SQL Server version 7.0 or later.  Recommended for single-tier applications using the Microsoft Data Engine (MSDE) or Microsoft SQL Server 7.0 or later.  For Microsoft SQL Server version 6.5 and earlier, you must use the [OLE DB Provider for SQL Server](http://msdn.microsoft.com/en-us/library/aa213282.aspx) with the [OLE DB .NET Data Provider](http://msdn.microsoft.com/en-us/library/aa720361.aspx). |
| OLE DB .NET Data Provider | Found in the **System.Data.OleDb** namespace.  Recommended for middle-tier applications using Microsoft SQL Server 6.5 or earlier, or any OLE DB provider that supports the OLE DB interfaces listed in [OLE DB Interfaces Used by the OLE DB .NET Data Provider](http://msdn.microsoft.com/en-us/library/aa720361.aspx) in the .NET Framework SDK. (OLE DB 2.5 interfaces are not required.)  For Microsoft SQL Server 7.0 or later, the .NET Framework Data Provider for SQL Server is recommended.  Recommended for single-tier applications using a Microsoft® Access database. Use of an Access database for a middle-tier application is not recommended.  Support for the OLE DB Provider for ODBC (MSDASQL) is disabled. For access to Open Database Connectivity (ODBC) data sources, an ODBC .NET Data Provider [download](http://msdn.microsoft.com/library/default.asp?url=/downloads/list/netdevframework.asp) is available and will be included in the .NET Framework SDK version 1.1. |
| ODBC .NET Data Provider | The ODBC .NET Data Provider for is available for [download](http://msdn.microsoft.com/library/default.asp?url=/downloads/list/netdevframework.asp).  Found in the **Microsoft.Data.Odbc** namespace.  Provides access to data sources that are connected to using an ODBC driver.  **Note** The ODBC .NET Data Provider will be included in upcoming versions of the .NET Framework starting with version 1.1. The namespace for the included ODBC .NET Data Provider is **System.Data.Odbc**. |
| .NET Data Provider for Oracle | The Microsoft .NET Data Provider for Oracle is available for [download](http://msdn.microsoft.com/library/default.asp?url=/downloads/list/netdevframework.asp).  Found in the **System.Data.OracleClient** namespace.  Provides access to Oracle data sources (version 8.1.7 and later).  **Note** The .NET Data Provider for Oracle will be included in upcoming versions of the .NET Framework starting with version 1.1. |
| Custom .NET Data Provider | ADO.NET provides a minimal set of interfaces to enable you to implement your own .NET Framework data provider. For more information about creating a custom data provider, see [Implementing a .NET Data Provider](http://msdn.microsoft.com/en-us/library/aa720164.aspx) in the .NET Framework SDK. |
| SQLXML Managed Classes | The release of XML for Microsoft SQL Server 2000 (SQLXML 3.0) contains SQLXML Managed Classes that enable you to access the XML functionality of Microsoft SQL Server 2000 and later, from the .NET Framework. For example, these classes enable you to execute XML templates, perform XML Path Language (XPath) queries over data at the server, or perform updates to data using Updategrams or Diffgrams.  Building on the functionality from SQLXML 1.0 and 2.0, SQLXML 3.0 introduces Web Services to SQL Server 2000. With SQLXML 3.0, Stored Procedures and XML Templates can be exposed as a Web Service through SOAP.  SQLXML 3.0 is available for [download](http://msdn.microsoft.com/library/default.asp?url=/downloads/list/sqlserver.asp). |

**Connecting to SQL Server 7.0 or Later**

For best performance when connecting to Microsoft SQL Server 7.0 or later, use the SQL Server .NET Data Provider. The SQL Server .NET Data Provider is designed to access SQL Server directly without any additional technology layers. Figure 1 illustrates the difference between the various technologies available for accessing SQL Server 7.0 or later.



**Figure 1. Connectivity methods for accessing SQL Server 7.0 or later**

**Connecting to ODBC Data Sources**

The ODBC .NET Data Provider, found in the **Microsoft.Data.Odbc** namespace, has the same architecture as the .NET data providers for SQL Server and OLE DB. The ODBC .NET Data Provider (available for [download](http://msdn.microsoft.com/library/default.asp?url=/downloads/list/netdevframework.asp)), follows a naming convention that uses an "ODBC" prefix (for example **OdbcConnection**), and uses standard ODBC connection strings.

**Note** The ODBC .NET Data Provider will be included in future versions of the .NET Framework starting with version 1.1. The namespace for the included ODBC .NET Data Provider is **System.Data.Odbc**.

**Working with DataReaders, DataSets, DataAdapters, and DataViews**

ADO.NET provides two objects for retrieving relational data and storing it in memory: the **DataSet** and the **DataReader**. The **DataSet** provides an in-memory relational representation of data, a complete set of data that includes the tables that contain, order, and constrain the data, as well as the relationships between the tables. The **DataReader** provides a fast, forward-only, read-only stream of data from a database.

When using a **DataSet**, you will often make use of a **DataAdapter** (and possibly a **CommandBuilder**) to interact with your data source. Also, when using a **DataSet**, you may employ a **DataView** to apply sorting and filtering to the data in the **DataSet**. The **DataSet** can also be inherited to create a strongly typed **DataSet** in order to expose tables, rows, and columns as strongly typed object properties.

The following topics include information on when it is best to use a **DataSet** or a **DataReader** and how to optimize access to the data they contain, as well as tips on how to optimize the use of the **DataAdapter** (including the **CommandBuilder**) and **DataView**.

**DataSet vs. DataReader**

To determine whether to use the **DataSet** or the **DataReader** when you design your application, consider the level of functionality that is needed in the application.

Use the **DataSet** in order to do the following with your application:

* Navigate between multiple discrete tables of results.
* Manipulate data from multiple sources (for example, a mixture of data from more than one database, from an XML file, and from a spreadsheet).
* Exchange data between tiers or using an XML Web service. Unlike the **DataReader**, the **DataSet** can be passed to a remote client.
* Reuse the same set of rows to achieve a performance gain by caching them (such as for sorting, searching, or filtering the data).
* Perform a large amount of processing per row. Extended processing on each row returned using a **DataReader** ties up the connection serving the **DataReader** longer than necessary, impacting performance.
* Manipulate data using XML operations such as Extensible Stylesheet Language Transformations (XSLT transformations) or XPath queries.

Use the **DataReader** in your application if you:

* Do not need to cache the data.
* Are processing a set of results too large to fit into memory.
* Need to quickly access data once, in a forward-only and read-only manner.

**Note** The **DataAdapter** uses the **DataReader** when filling a **DataSet**. Therefore, the performance gained by using the **DataReader** instead of the **DataSet** is that you save on the memory that the **DataSet** would consume and the cycles it takes to populate the **DataSet**. This performance gain is, for the most part, nominal so you should base your design decisions on the functionality required.

**Benefits of Using a Strongly Typed DataSet**

Another benefit of the **DataSet** is that it can be inherited to create a strongly typed **DataSet**. The benefits of a strongly typed **DataSet** include type checking at design time and the advantage of Microsoft® Visual Studio® .NET statement completion for your strongly typed **DataSet**. When you have fixed schema or relational structure for your **DataSet**, you can create a strongly typed **DataSet** that exposes rows and columns as properties of an object rather than items in a collection. For example, instead of exposing the name column of a row from a table of customers, you expose a **Customer** object's **Name** property. A typed **DataSet** derives from the **DataSet** class, so that you do not sacrifice any of the **DataSet** functionality. That is, a typed **DataSet** can still be remoted and can be supplied as the data source of a data-bound control such as a **DataGrid**. If schema is not known in advance, you can still benefit from the functionality of a generic **DataSet**, but you forfeit the additional features of a strongly typed **DataSet**.

**Handling Nulls in a Strongly Typed DataSet**

When using a strongly typed **DataSet**, you can annotate the XML Schema definition language (XSD) schema of the **DataSet** to ensure that your strongly typed **DataSet** appropriately handles null references. The **nullValue** annotation enables you to replace **DBNull** with a specified value, **String.Empty**, persist the null reference, or throw an exception. Which option you choose depends on the context of your application. By default, an exception is thrown if a null reference is encountered.

For more information, see [Working with a Typed DataSet](http://msdn.microsoft.com/en-us/library/esbykkzb.aspx).

**Refreshing Data in a DataSet**

If you want to refresh the values in your **DataSet** with updated values from the server, use **DataAdapter.Fill**. If you have primary keys defined on your **DataTable**, **DataAdapter.Fill** matches new rows based on the primary keys, and applies the server values as it changes to the existing rows. The **RowState** of the refreshed rows are set to **Unchanged** even if they were modified before the refresh. Note that, if no primary key is defined for the **DataTable**, **DataAdapter.Fill** adds new rows with potentially duplicate primary key values.

If you want to refresh a table with the current values from the serve while retaining any changes made to the rows in the table, you must first populate it with **DataAdapter.Fill**, fill a new **DataTable**, and then **Merge** that **DataTable** into the **DataSet** with a **preserveChanges** value of **true**.

**Searching for Data in the DataSet**

When querying a **DataSet** for rows that match particular criteria, you can increase the performance of your searches by taking advantage of index-based lookups. When you assign a **PrimaryKey** value to a **DataTable**, an index is created. When you create a **DataView** for a **DataTable**, an index is also created. Here are a few tips for taking advantage of index-based lookups.

* If the query is against the columns that make up the **PrimaryKey** of the **DataTable**, use **DataTable.Rows.Find** instead of **DataTable.Select**.
* For queries involving non-primary key columns, you can improve performance for multiple queries of the data using a **DataView**. When you apply a sort order to a **DataView**, an index is built that is used when searching. The **DataView** exposes the **Find** and **FindRows** methods to query the data in the underlying **DataTable**.
* If you do not require a sorted view of a table, you can still take advantage of index-based lookups by creating a **DataView** for the **DataTable**. Note that this is only an advantage if you are performing multiple queries on the data. If you are only performing a single query, the processing required to create the index reduces the performance gained by using the index.

**DataView Construction**

The **DataView** builds an index for the data in the underlying **DataTable** when both the **DataView** is created, and when the **Sort**, **RowFilter** or **RowStateFilter** properties are modified. When creating a **DataView** object, use the **DataView** constructor that takes the **Sort**, **RowFilter**, and **RowStateFilter** values as constructor arguments (along with the underlying **DataTable**). The result is the index is built once. Creating an "empty" **DataView** and setting the **Sort**, **RowFilter** or **RowStateFilter** properties afterward results in the index being built at least twice.

**Paging**

ADO.NET gives you explicit control over what data is returned from your data source, as well as, how much of that data is cached locally in a **DataSet**. There is no single answer for paging through a query result, but here are some tips to consider when designing your application.

* Avoid the use of the **DataAdapter.Fill** overload that takes **startRecord** and **maxRecords** values. When filling a **DataSet** in this fashion, the **DataSet** is only filled with the number of records specified by the **maxRecords** parameter (starting from the record identified by the **startRecord** parameter), but the entire query is returned regardless. This incurs unnecessary processing to read past the "unwanted" records, as well as uses up unnecessary server resources to return the additional records.
* A technique used for returning only one page of records at a time is creating a SQL statement that combines a WHERE clause and an ORDER BY clause, with the TOP predicate. This technique relies on there being a way to identify each row uniquely. When navigating to the next page of records, modify the WHERE clause to include all records where the unique identifier is greater than the last unique identifier of the current page. When navigating to the previous page of records, modify the WHERE clause to return all the records where the unique identifier is less than the first unique identifier of the current page. For both queries, return only the TOP page of records. When navigating to the previous page, you need to order the results in descending order. This will, effectively, return the bottom page of the query (you will need to reorder the results before displaying them, if desired). For an example of this technique, see [Paging Through a Query Result](http://msdn.microsoft.com/en-us/library/tx1c9c2f.aspx).
* Another technique for returning only one page of records at a time is to create a SQL statement that combines the use of the TOP predicate and embedded SELECT statements. This technique does not rely on there being a way to identify each row uniquely. The first step using this technique is to multiply the page size with the number of the desired pages. You then pass this number to the TOP predicate of your SQL Query, ordered in ascending order. You then embed this query in another query that selects the TOP page-size from the embedded query results, ordered in descending order. Essentially, you return the bottom page of the embedded query. For example, to return the third page of a query result where the page size is 10, you would issue a command like the following:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl18_code');" \o "Copy Code)

SELECT TOP 10 \* FROM

(SELECT TOP 30 \* FROM Customers ORDER BY Id ASC) AS Table1

ORDER BY Id DESC

Note that the page of results returned from this query come in descending order. You will need to reorder them if desired.

* If your data does not change often, you can improve performance by maintaining a cache of records locally in a **DataSet**. For example, you can store 10 pages worth of data in a local **DataSet**, and only query the data source for new data when the user navigates beyond the first or last page in the cache.

For more information, see the [.NET Data Access Architecture Guide](http://msdn.microsoft.com/en-us/library/cc446443.aspx).

**Filling a DataSet with Schema**

When filling a **DataSet** with data, the **DataAdapter.Fill** method uses the existing schema of the **DataSet** and populates it with data returned from the **SelectCommand**. If there is no table name in the **DataSet** that matches the name of the table being filled, the **Fill** method creates a table. By default, **Fill** only defines columns and column types.

You can override the default behavior of **Fill** by setting the **MissingSchemaAction** property of the **DataAdapter**. For example, to have **Fill** create a table schema that also includes primary key information, unique constraints, column properties, whether nulls are allowed, the maximum length of the column, read-only columns, and auto-increment columns, specify that **DataAdapter.MissingSchemaAction** is **MissingSchemaAction.AddWithKey**. Alternatively, you can call **DataAdapter.FillSchema** before calling **DataAdapter.Fill** to ensure that the schema is in place when the **DataSet** is filled.

Calling **FillSchema** will result in an extra trip to the server to retrieve the additional schema information. For best performance, specify the schema of the **DataSet**, or set the **MissingSchemaAction** of the **DataAdapter** before calling **Fill**.

**Best Practices with the CommandBuilder**

The **CommandBuilder** automatically generates the **InsertCommand**, **UpdateCommand**, and **DeleteCommand** properties of a **DataAdapter** based on the **SelectCommand** property of the **DataAdapter**, provided that the **SelectCommand** performs a single table SELECT. Here are some tips for best performance using the **CommandBuilder**.

* Use of the **CommandBuilder** should be limited to design time or ad-hoc scenarios. The processing required to generate the **DataAdapter** command properties hinders performance. If you know the contents of your INSERT/UPDATE/DELETE statements beforehand, set them explicitly. A good design tip is to create stored procedures for your INSERT/UPDATE/DELETE commands and explicitly configure the **DataAdapter** command properties to use them.
* The **CommandBuilder** uses the **SelectCommand** property of the **DataAdapter** to determine the values for the other command properties. If the **SelectCommand** of the **DataAdapter** itself is ever changed, be sure to call **RefreshSchema** to update the command properties.
* The **CommandBuilder** only generates a command for a **DataAdapter** command property if that command property is null (the command properties are null by default). If you explicitly set a command property, the **CommandBuilder** does not overwrite it. If you want the **CommandBuilder** to generate a command for a command property that has been set previously, set the command property to null.

**Batch SQL Statements**

Many databases support combining, or batching, multiple commands together in a single command execution. For example, SQL Server enables you to separate commands using a semi colon (;). Combining multiple commands into a single command reduces the number of trips made to the server and can improve the performance of your application. You can, for example, store up all the intended deletes locally in your application, and then issue one batch command call to delete them from the data source.

Though this does improve performance, it may increase the complexity of your application when managing updates to data in the **DataSet**. To maintain simplicity, you may want to create a **DataAdapter** for each **DataTable** in your **DataSet**.

**Filling a DataSet with Multiple Tables**

If you use a batch SQL statement to retrieve multiple tables and fill a **DataSet**, the first table is named using the table name specified to the **Fill** method. Subsequent tables are named using the name specified to the **Fill** method plus a number starting from one and incrementing by one. For example, if you were to run the following code:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl20_code');" \o "Copy Code)

**'Visual Basic**

Dim da As SqlDataAdapter = New SqlDataAdapter("SELECT \* FROM Customers; SELECT \* FROM Orders;", myConnection)

Dim ds As DataSet = New DataSet()

da.Fill(ds, "Customers")

**//C#**

SqlDataAdapter da = new SqlDataAdapter("SELECT \* FROM Customers; SELECT \* FROM Orders;", myConnection);

DataSet ds = new DataSet();

da.Fill(ds, "Customers");

The data from the Customers table is placed in a **DataTable** named "Customers". The data from the Orders table is placed in a **DataTable** named "Customers1".

You can easily modify the **TableName** property of the "Customers1" table to be "Orders" after the **DataSet** has been filled. However, subsequent fills would result in the "Customers" table being re-filled, but the "Orders" table being ignored and another "Customers1" table being created. To remedy this situation, create a **DataTableMapping** that maps "Customers1" to "Orders" and additional table mappings for other subsequent tables. For example:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl21_code');" \o "Copy Code)

**'Visual Basic**

Dim da As SqlDataAdapter = New SqlDataAdapter("SELECT \* FROM Customers; SELECT \* FROM Orders;", myConnection)

da.TableMappings.Add("Customers1", "Orders")

Dim ds As DataSet = New DataSet()

da.Fill(ds, "Customers")

**//C#**

SqlDataAdapter da = new SqlDataAdapter("SELECT \* FROM Customers; SELECT \* FROM Orders;", myConnection);

da.TableMappings.Add("Customers1", "Orders");

DataSet ds = new DataSet();

da.Fill(ds, "Customers");

**Using the DataReader**

The following are some tips for best performance using a **DataReader**, as well as, answers to common questions regarding the use of the **DataReader**.

* The **DataReader** must be closed before accessing any output parameters for the associated **Command**.
* Always close the **DataReader** when you are finished reading the data. If the **Connection** you are using is only used to return the **DataReader**, close it immediately after closing the **DataReader**.

An alternative to explicitly closing the **Connection** is to pass **CommandBehavior.CloseConnection** to the **ExecuteReader** method to ensure that the associated connection is closed when the **DataReader** is closed. This is especially useful if you are returning a **DataReader** from a method and do not have control over the closing of the **DataReader** or associated connection.

* The **DataReader** cannot be remoted between tiers. The **DataReader** is designed for connected data access.
* When accessing column data use the typed accessors like **GetString**, **GetInt32**, and so on. This saves you the processing required to cast the **Object** returned from **GetValue** as a particular type.
* Only one **DataReader** can be open at a time, off of a single connection. In ADO, if you opened a single connection and requested two recordsets that used a forward-only, read-only cursor, ADO implicitly opens a second, unpooled connection to the data store for the life of that cursor, and then implicitly closes it. With ADO.NET, little is done for you "under-the-covers". If you want two **DataReaders** open at the same time, off the same data store, you have to explicitly create two connections, one for each **DataReader**. This is one way that ADO.NET gives you more control over the use of pooled connections.
* By default, the **DataReader** loads an entire row into memory with each **Read**. This allows for random access of columns within the current row. If this random access is not necessary, for increased performance, pass **CommandBehavior.SequentialAccess** to the call to **ExecuteReader**. This changes the default behavior of the **DataReader** to only load data into memory when it is requested. Note that, **CommandBehavior.SequentialAccess** requires you to access returned columns in order. That is, once you have read past a returned column, you can no longer read its value.
* If you are finished reading the data from a **DataReader**, but still have a large number of unread results pending, call **Cancel** on the **Command** prior to calling **Close** on the **DataReader**. Calling **Close** on the **DataReader** causes it to retrieve pending results and empty the stream prior to closing the cursor. Calling **Cancel** on the **Command** discards results on the server so that the **DataReader** does not have to read though them when it is closed. If you are returning output parameters from your **Command**, calling **Cancel** discards them as well. If you need to read any output parameters, do not call **Cancel** on the **Command**; just call **Close** on the **DataReader**.

**Binary Large Objects (BLOBs)**

When using the **DataReader** to retrieve a Binary Large Object (BLOB), you should pass **CommandBehavior.SequentialAccess** to the **ExecuteReader** method call. Because the default behavior of the **DataReader** is to load an entire row into memory with each **Read**, and because BLOB values can be very large, the result can be large amounts of memory being used up for a single BLOB. **SequentialAccess** sets the behavior of the **DataReader** to only load the data requested. You can then control how much data is loaded at a time using **GetBytes** or **GetChars**.

Remember that, when using **SequentialAccess**, you cannot access the different fields returned by the **DataReader** out of order. That is, if your query returns three columns, the third of which is a BLOB, and you want to access data in the first two columns, you must access the first column value, then the second column value before accessing the BLOB data. This is because the data is now returned in sequence and is not available once the **DataReader** has read past it.

For a detailed description of how to access a BLOB in ADO.NET, see [Obtaining BLOB Values from a Database](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguidnf/html/cpconobtainingblobvaluesfromdatabase.asp).

**Using Commands**

ADO.NET provides several different methods for command execution, as well as different options for optimizing the execution of a command. The following includes tips on choosing the best command execution and how to improve the performance of an executed command.

**Best Practices with OleDbCommand**

Command execution between the different .NET Framework data providers is standardized as much as possible. However, there are differences between the data providers. The following are some tips for fine-tuning command execution with the .NET Framework Data Provider for OLE DB.

* Use **CommandType.Text** with the ODBC CALL syntax to call stored procedures. Using **CommandType.StoredProcedure** just generates the ODBC CALL syntax under the covers.
* Be sure to set the **OleDbParameter** type, size (if applicable), as well as precision and scale (if the parameter is of type numeric or decimal). Note that, if you do not supply the parameter information explicitly, the **OleDbCommand** recreates the OLE DB parameter accessor with every command execution.

**Best Practices with SqlCommand**

A quick tip for executing stored procedures using the **SqlCommand**: If you are calling a stored procedure, specify a **CommandType** of **StoredProcedure** for the **CommandType** property of the **SqlCommand**. This removes the need to parse the command before execution, by explicitly identifying it as a stored procedure.

**Use of the Prepare Method**

The **Command.Prepare** method can improve the performance of parameterized commands that are repeated at your data source. **Prepare** instructs the data source to optimize the specified command for multiple calls. To use **Prepare** effectively, you need to thoroughly understand how your data source responds to the **Prepare** call. For some data sources such as SQL Server 2000, commands are implicitly optimized and a call to **Prepare** is unnecessary. For other data sources such as SQL Server 7.0, **Prepare** can be effective.

**Specify Schema and Metadata Explicitly**

Many objects in ADO.NET infer metadata information whenever the user does not specify it. Examples are:

* The **DataAdapter.Fill** method, which creates a table and columns in a **DataSet** if none exist
* The **CommandBuilder**, which generates **DataAdapter** command properties for single-table SELECT commands.
* **CommandBuilder.DeriveParameters**, which populates a **Command** object's **Parameters** collection.

Each time these features are used, however, there is a performance hit. It is recommended that these features be used primarily for design-time and ad-hoc applications. Whenever possible, specify the schema and metadata explicitly. This includes defining tables and columns in a **DataSet**, defining the **Command** properties of a **DataAdapter**, and defining **Parameter** information for a **Command**.

**ExecuteScalar and ExecuteNonQuery**

If you want to return a single value such as the result of Count(\*), Sum(Price), or Avg(Quantity), you can use **Command.ExecuteScalar**. **ExecuteScalar** returns the value of the first column of the first row, returning result set as a scalar value. **ExecuteScalar** both simplifies your code and improves performance by accomplishing in a single step, what would have been a two-step process using a **DataReader** (that is, **ExecuteReader** + Get the value).

When using SQL statements that do not return rows, like those that modify data (such as INSERT, UPDATE, or DELETE) or return only output parameters or return values, use **ExecuteNonQuery**. This removes any unnecessary processing to create an empty **DataReader**.

For more information, see Executing a Command.

**Testing for Null**

If a column in a table (in your database) allows nulls, you cannot test for a parameter value of "equal to" null. Instead, you need to write a WHERE clause to test whether both the column is null and the parameter is null. The following SQL statement returns rows where the LastName column equals the value assigned to the @LastName parameter, or whether both the LastName column and the @LastName parameter are null.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl24_code');" \o "Copy Code)

SELECT \* FROM Customers

WHERE ((LastName = @LastName) OR (LastName IS NULL AND @LastName IS NULL))

**Passing Null as a Parameter Value**

When sending a null value as a Parameter value in a command to the database, you cannot use **null** (**Nothing** in Visual Basic® .NET). Instead you need to use **DBNull.Value**. For example:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl25_code');" \o "Copy Code)

**'Visual Basic**

Dim param As SqlParameter = New SqlParameter("@Name", SqlDbType.NVarChar, 20)

param.Value = DBNull.Value

**//C#**

SqlParameter param = new SqlParameter("@Name", SqlDbType.NVarChar, 20);

param.Value = DBNull.Value;

**Performing Transactions**

The transaction model has changed for ADO.NET. In ADO, when **StartTransaction** was called, any update following the call is considered part of the transaction. However, in ADO.NET, when **Connection.BeginTransaction** is called, a **Transaction** object is returned that needs to be associated with the **Transaction** property of a **Command**. This design enables you to perform multiple root transactions off of a single connection. If the **Command.Transaction** property is not set to a **Transaction** that has been started for the associated **Connection**, the **Command** fails and an exception is thrown.

Upcoming releases of the .NET Framework will enable you to manually enlist in an existing distributed transaction. This is ideal for an object pooling scenario where a connection is opened once for a pooled object, but the object is involved in multiple separate transactions. This capability is not available in the .NET Framework 1.0 release.

For more information on transactions, see [Performing Transactions](http://msdn.microsoft.com/en-us/library/777e5ebh.aspx), as well as the [.NET Data Access Architecture Guide](http://msdn.microsoft.com/en-us/library/cc446443.aspx).

**Using Connections**

High performance applications keep connections to the data source in use for a minimal amount of time, as well as take advantage of performance enhancing technology such as connection pooling. The following topics provide you with tips to help you achieve greater performance when using ADO.NET to connect to your data source.

**Connection Pooling**

The SQL Server, OLE DB, and .NET Framework Data Provider for ODBC pool connections implicitly. You can control connection-pooling behavior by specifying different attribute values in the connection string. For details on how to control connection pooling behavior, see [Connection Pooling for the SQL Server .NET Data Provider](http://msdn.microsoft.com/en-us/library/8xx3tyca.aspx) and [Connection Pooling for the OLE DB .NET Data Provider](http://msdn.microsoft.com/en-us/library/aa719769.aspx).

**Optimizing Connections with the DataAdapter**

The **Fill** and **Update** methods, of the **DataAdapter**, automatically open the connection specified for the related command property if it is closed. If the **Fill** or **Update** method open the connection, **Fill** or **Update** will close it when the operation is complete. For best performance, keep connections to the database open only when required. Also, reduce the number of times you open and close a connection for multiple operations.

It is recommended that, if you are only performing a single **Fill** or **Update** method call, that you allow the **Fill** or **Update** method to open and close the connection implicitly. If you are making numerous calls to **Fill** and/or **Update**, it is recommended that you explicitly open the connection, make your calls to **Fill** and/or **Update**, and then explicitly close the connection.

Additionally, when performing transactions, explicitly open the connection before beginning the transaction and close the connection after you commit. For example:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl30_code');" \o "Copy Code)

**'Visual Basic**

Public Sub RunSqlTransaction(da As SqlDataAdapter, myConnection As SqlConnection, ds As DataSet)

myConnection.Open()

Dim myTrans As SqlTransaction = myConnection.BeginTransaction()

myCommand.Transaction = myTrans

Try

da.Update(ds)

myTrans.Commit()

Console.WriteLine("Update successful.")

Catch e As Exception

Try

myTrans.Rollback()

Catch ex As SqlException

If Not myTrans.Connection Is Nothing Then

Console.WriteLine("An exception of type " & ex.GetType().ToString() & \_

" was encountered while attempting to roll back the transaction.")

End If

End Try

Console.WriteLine("An exception of type " & e.GetType().ToString() & " was encountered.")

Console.WriteLine("Update failed.")

End Try

myConnection.Close()

End Sub

**//C#**

public void RunSqlTransaction(SqlDataAdapter da, SqlConnection myConnection, DataSet ds)

{

myConnection.Open();

SqlTransaction myTrans = myConnection.BeginTransaction();

myCommand.Transaction = myTrans;

try

{

da.Update(ds);

myCommand.Transaction.Commit();

Console.WriteLine("Update successful.");

}

catch(Exception e)

{

try

{

myTrans.Rollback();

}

catch (SqlException ex)

{

if (myTrans.Connection != null)

{

Console.WriteLine("An exception of type " + ex.GetType() +

" was encountered while attempting to roll back the transaction.");

}

}

Console.WriteLine(e.ToString());

Console.WriteLine("Update failed.");

}

myConnection.Close();

}

**Always Close Connections and DataReaders**

Always explicitly close your **Connection** or **DataReader** objects when you are finished using them. While garbage collection eventually cleans up objects and therefore releases connections and other managed resources, garbage collection only occurs when it is needed. Therefore, it is still your responsibility to make sure any expensive resources are explicitly released. Also, **Connections** that are not explicitly closed might not be returned to the pool. For example, a connection that has gone out of scope but that has not been explicitly closed will only be returned to the connection pool if the maximum pool size has been reached and the connection is still valid.

**Note** Do not call **Close** or **Dispose** on a **Connection**, a **DataReader**, or any other managed object in the **Finalize** method of your class. In a finalizer, only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a **Finalize** method in your class definition.

**Use the "Using" Statement in C#**

For C# programmers, a convenient way to ensure that you always close your **Connection** and **DataReader** objects is to use the **using** statement. The **using** statement automatically calls **Dispose** on the object being "used" when leaving the scope of the **using** statement. For example:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl31_code');" \o "Copy Code)

**//C#**

string connString = "Data Source=localhost;Integrated Security=SSPI;Initial Catalog=Northwind;";

using (SqlConnection conn = new SqlConnection(connString))

{

SqlCommand cmd = conn.CreateCommand();

cmd.CommandText = "SELECT CustomerId, CompanyName FROM Customers";

conn.Open();

using (SqlDataReader dr = cmd.ExecuteReader())

{

while (dr.Read())

Console.WriteLine("{0}\t{1}", dr.GetString(0), dr.GetString(1));

}

}

The using statement is not available for Microsoft® Visual Basic® .NET.

**Avoid Accessing the OleDbConnection.State Property**

If the connection has been opened, **OleDbConnection.State** property makes the native OLE DB call **IDBProperties.GetProperties** to the **DATASOURCEINFO** property set for the **DBPROP\_CONNECTIONSTATUS** property, which may result in a round trip to the data source. In other words, checking the **State** property can be expensive. So only check the **State** property when required. If you need to check this property often, your application may perform better if you listen for the **StateChange** event for your **OleDbConnection**. For details on the **StateChange** event, see [Working with Connection Events](http://msdn.microsoft.com/en-us/library/a0hee08w.aspx).

**Integration with XML**

ADO.NET provides extensive XML integration in the **DataSet**, and also exposes some of the XML functionality provided by SQL Server 2000 and later. You can also make use of SQLXML 3.0 for extensive access to the XML functionality in SQL Server 2000 and later. Here are tips and information when using XML and ADO.NET.

**The DataSet and XML**

The **DataSet** is tightly integrated with XML providing you with the ability to:

* Load the schema or relational structure of a **DataSet** from XSD Schema.
* Load the contents of a **DataSet** from XML.
* Infer the schema of a **DataSet** from the contents of an XML document when no schema is supplied.
* Write the schema of a **DataSet** as XSD Schema.
* Write the contents of a **DataSet** as XML.
* Have synchronous access to both the relational representation of your data using the **DataSet**, as well as the hierarchical representation of your data using the **XmlDataDocument**.

**Note** You can use this synchronization to apply XML functionality such as XPath queries and XSLT transformations to the data in your **DataSet**, or to provide a relational view of all, or a subset of the data in an XML document while preserving the fidelity of the original XML.

For detailed information on the XML functionality provided with the **DataSet**, see [XML and the DataSet](http://msdn.microsoft.com/en-us/library/84sxtbxh.aspx).

**Schema Inference**

When loading a **DataSet** from an XML file, you can load the schema of the **DataSet** from XSD Schema, or you can predefine the tables and columns before loading the data. If no XSD Schema is available and you do not know which tables and columns to define for the contents of an XML file, you can infer the schema based on the structure of the XML document.

Schema inference is useful as a migration tool, but should be limited to design-time applications only as the inference process has the following limitations.

* Inferring schema introduces additional processing that hinders the performance of an application.
* All inferred columns are of type string.
* The inference process is not deterministic. That is, it is based on the contents of the XML file, not the intended schema. As a result, you can have two XML files, with the same intended schema, that result in two entirely different inferred schemas because their contents differ.

For more information, see [Inferring DataSet Relational Structure from XML](http://msdn.microsoft.com/en-us/library/3b4194wc.aspx).

**SQL Server FOR XML Queries**

If you are returning the results of a SQL Server 2000 FOR XML query, you can have the .NET Framework Data Provider for SQL Server directly create an **XmlReader** using the **SqlCommand.ExecuteXmlReader** method.

**SQLXML Managed Classes**

There are classes available that expose the functionality of XML for SQL Server 2000 in the .NET Framework. These classes are found in the **Microsoft.Data.SqlXml** namespace and add the ability to execute XPath queries and XML Template files, as well as apply XSLT transformation to data.

SQLXML Managed Classes are included in the release of XML for Microsoft SQL Server 2000 (SQLXML 2.0), which is available at [XML for Microsoft SQL Server 2000 Web Release 2 (SQLXML 2.0)](http://www.microsoft.com/downloads/release.asp?ReleaseID=33055).

**More Useful Tips**

Here are some general tips for writing ADO.NET code.

**Avoiding Auto-Increment Value Conflicts**

Like most data sources, the **DataSet** enables you to identify columns that automatically increment their value when new rows are added. When using auto-increment columns in a **DataSet**, with auto-increment columns from a data source, avoid conflicts between the local numbering of rows added to the **DataSet** and rows added to the data source.

For example, consider a table with an auto-incrementing primary key column of CustomerID. Two new rows of customer information are added to the table and receive auto-incremented CustomerID values of 1 and 2. Then, only the second customer row is passed to the **Update** method of the **DataAdapter**, the newly added row receives an auto-incremented CustomerID value of 1 at the data source, which does not match the value 2, in the **DataSet**. When the **DataAdapter** fills the second row in the table with the returned value, a constraint violation occurs because the first customer row already has a CustomerID of 1.

To avoid this behavior, it is recommended that, when working with auto-incrementing columns at a data source and auto-incrementing columns in a **DataSet**, you create the column in the **DataSet** with an **AutoIncrementStep** of -1 and an **AutoIncrementSeed** of 0, as well as ensuring that your data source generates auto-incrementing identity values starting from 1 and incrementing with a positive step value. As a result, the **DataSet** generates negative numbers for auto-incremented values that do not conflict with the positive auto-increment values generated by the data source. Another option is to use columns of type **Guid** instead of auto-incrementing columns. The algorithm that generates **Guid** values should never generate the same **Guid** in the **DataSet** as is generated by the data source.

If your auto-incremented column is used simply as a unique value, and does not have any meaning, consider using Guids instead of auto-incrementing columns. They are unique and avoid the extra work necessary to work with auto-incremented columns.

For an example of retrieving auto-increment column values from a data source, see [Retrieving Identity or AutoNumber Values](http://msdn.microsoft.com/en-us/library/ks9f57t0.aspx).

**Check for Optimistic Concurrency Violations**

Because the **DataSet** is, by design, disconnected from the data source, you need to ensure that your application avoids conflicts when multiple clients update data at the data source, according to the optimistic concurrency model.

There are several techniques when testing for an optimistic concurrency violation. One involves including a timestamp column in the table. Another technique is to verify that all the original column values in a row still match those found in the database by testing using a WHERE clause in your SQL statement.

For a detailed discussion on this topic including a code example, see [Optimistic Concurrency](http://msdn.microsoft.com/en-us/library/aa0416cz.aspx).

**Multithreaded Programming**

ADO.NET is optimized for performance, throughput, and scalability. As a result, the ADO.NET objects do not lock resources and must only be used on a single thread. The one exception is the **DataSet**, which is thread-safe for multiple readers. However, you need to lock the **DataSet** during writes.

**Access ADO using COM Interop Only When Required**

ADO.NET is designed to be the best solution for a large number of applications. However, some applications require capabilities that are only available using ADO objects, for example ADO Multidimensional (ADOMD). In these situations, the application can access ADO using COM Interop. Note that using COM Interop to access data with ADO incurs a performance hit. When designing an application, first determine if ADO.NET meets your design needs before implementing a design that accesses ADO using COM Interop.

*The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.*

*This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.*

*Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.*

*Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.*

© *2002 Microsoft Corporation. All rights reserved.*

*Microsoft, Visual Basic, Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.*

*The names of actual companies and products mentioned herein may be the trademarks of their respective owners.*