9

Improving XML Performance

Objectives

- Optimize XML processing design.
- Parse XML documents efficiently.
- Validate XML documents efficiently.
- Optimize your XML Path Language (XPath) queries.
- Write efficient XSL Transformations (XSLT).

Overview

When you build .NET applications, you use XML extensively. It is used to represent the message payload for Web services, and it is used by many Web applications to pass data across application layers. XML is platform-neutral, making it one of the best technologies for interoperability between disparate systems such as UNIX or mainframes integration with Windows.

While XML is extremely flexible and relatively easy to use, for some applications it may not be the best data representation. The text-based and verbose nature of XML, and the fact that it includes metadata (element and attribute names), means that it is not a compact data format. XML can require substantial processing effort.

The precise performance impact associated with processing XML depends on several factors that include the size of the data, the parsing effort required to process the data, the nature of any transformations that might be required, and the potential impact of validation. You should analyze the way your application processes XML because this area often accounts for a sizable portion of your application's per-request processing effort.

This chapter starts by providing a brief overview of XML in the Microsoft .NET Framework. It then highlights the main performance and scalability issues that tend to arise as a result of inefficient XML processing. The chapter then presents guidelines and recommendations that help you optimize the way you parse, validate, write, and transform XML.

How to Use This Chapter

Use this chapter to help design and implement effective XML processing in your applications. To get the most out of this chapter:

- **Jump to topics or read from beginning to end**. The main headings in this chapter help you locate the topics that interest you. Alternatively, you can read the chapter from beginning to end to gain a thorough appreciation of the issues that are related to designing for performance and scalability.
- **Use the checklist**. Use the "Checklist: XML Performance" checklist in the "Checklist" section of this guide to quickly view and evaluate the guidelines presented in this chapter.
- **Use the "Architecture" section of this chapter to understand how XML works**. By understanding the architecture, you can make better design and implementation choices.
- **Use the "Design Considerations" section of this chapter to understand the high-level decisions that affect implementation choices for XML performance**.
- **Read Chapter 13, "Code Review: .NET Application Performance."**
- **Measure your application performance**. Read the ".NET Framework Technologies" section of Chapter 15, "Measuring .NET Application Performance," to learn about the key metrics that you can use to measure application performance. It is important for you to measure application performance so that performance issues can be accurately identified and resolved.
- **Test your application performance**. Read Chapter 16, "Testing .NET Application Performance," to learn how to apply performance testing to your application. It is important to apply a coherent testing process and to analyze the results.
- **Tune your application performance**. Read Chapter 17, "Tuning .NET Application Performance," to learn how to resolve performance issues identified through the use of tuning metrics.

Architecture

The .NET Framework provides a comprehensive set of classes for XML manipulation. In addition to XML parsing and creation, these classes also support the World Wide Web Consortium (W3C) XML standards. These W3C XML standards include Document Object Model (DOM), XSLT, XPath 1.0, and XML Schema. The top-level namespace that contains XML-related classes is **System.Xml**.

The following list briefly describes the major XML-related classes:

- **XmlReader**. The **XmlReader** abstract base class provides an API for fast, forward-only, read-only parsing of an XML data stream. **XmlReader** is similar to Simple API for XML (SAX), although the SAX model is a "push" model where the parser pushes events to the application and notifies the application every time a new node is read. Applications that use **XmlReader** can pull nodes from the reader at will. The following are the three concrete implementations of **XmlReader:**
  - **XmlTextReader**. You use this class to read XML streams.
  - **XmlNodeReader**. You use this class to provide a reader over a given DOM-node subtree. It reads and then returns nodes from the subtree.
  - **XmlValidatingReader**. You use this class to read and validate XML data, according to predefined schemas that include document type definitions (DTD) and W3C XML schemas.
- **XmlWriter**. The **XmlWriter** abstract base class is used to generate a stream of XML. The .NET Framework provides an implementation in the form of the **XmlTextWriter** class. You use this class as a fast, noncached, forward-only way to generate streams or files that contain XML data.
- **XmlDocument**. The **XmlDocument** class is an in-memory or cached tree representation of an XML document. You can use it to edit and navigate the document. While **XmlDocument** is easy to use, it is very resource-intensive. You should generally use **XmlReader** and **XmlWriter** for better performance. **XmlDocument** implements the W3C DOM Level 1 and Level 2 recommendations, although it has been tailored to the .NET Framework. For example, in the .NET Framework, method names are capitalized, and common language runtime (CLR) types are used.
- **XslTransform**. You use the **XslTransform** class to perform XSLT transformations. It is located in the **System.Xml.Xsl** namespace.

- **XPathNavigator**. The **XPathNavigator** abstract base class provides random read-only access to data through XPath queries over any data store. Data stores include the **XmlDocument**, **DataSet**, and **XmlDataDocument** classes. To create an **XPathNavigator** object, use the **CreateNavigator** method. **XPathNavigator** also provides a cursor API for navigating a document. The **XPathNavigator** class is located in the **System.Xml.XPath** namespace.
- **XPathDocument**. The **XPathDocument** class is optimized for XSLT processing and for the XPath data model. You should always use **XPathDocument** with the **XslTransform** class. The **XPathDocument** class is located in the **System.Xml.XPath** namespace.
- **XmlSerializer**. You use the **XmlSerializer** class to perform object-to-XML serialization and vice versa. It is located in the **System.Xml.Serialization** namespace. You can use this class to convert the public properties and fields of a .NET Framework object to XML format.
- **XmlDataDocument**. The **XmlDataDocument** class extends **XmlDocument**. It provides both relational data representation of **DataSet** and hierarchical data representation of DOM. Data can be manipulated by using DOM or **DataSet** object. To load a **DataSet** with XML data, use the **ReadXmlSchema** class to build a relational mapping. The XML data can then be loaded by using the **ReadXml** method. To load a **DataSet** with relational data, specify the **DataSet** that contains the relational data as the parameter in the **XmlDataDocument** constructor.
- **XmlSchema**. The **XmlSchema** class contains the definition of a schema. All XML schema definition language (XSD) elements are children of the schema element. **XmlSchema** represents the W3C schema element.
- **XmlSchemaCollection**. The **XmlSchemaCollection** class is a library of **XmlSchema** objects that can be used with **XmlValidatingReader** to validate XML documents. Typically, this is loaded once at application startup and then reused across components.

The main XML namespaces and principal types are shown in Figure 9.1.

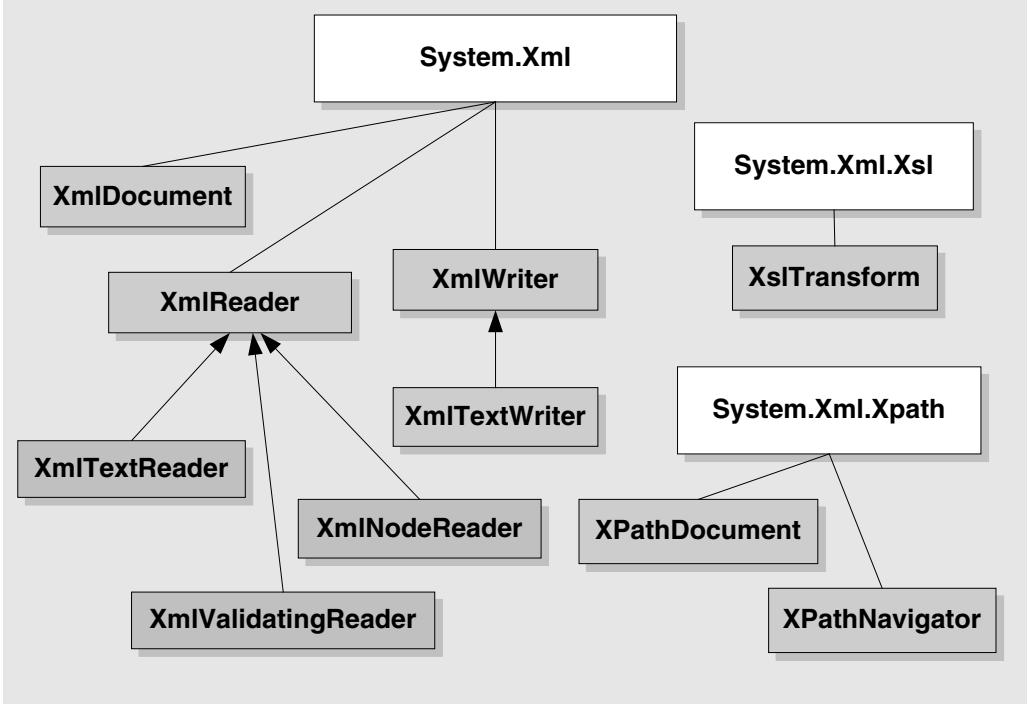

Figure 9.1
XML namespaces and principal types

Performance and Scalability Issues

The main XML-related issues that affect the performance and scalability of your application are summarized in the following list. Subsequent sections in this chapter provide strategies and technical implementation details to prevent or resolve each of these issues.

- **Resource consumption**. Processing large XML documents can cause high CPU, memory, and bandwidth utilization. XML is a verbose and text-based representation. Therefore, XML documents are larger than binary representations of the same data. These issues are magnified if users access your application over low-bandwidth networks, or if many users connect to the application at the same time.
- **Extreme memory consumption**. Using the DOM-model and the **XmlDocument** or **XPathDocument** classes to parse large XML documents can place significant demands on memory. These demands may severely limit the scalability of server-side Web applications.

- **Inefficient transformations**. Choosing the wrong transformation approach, building inefficient XPath queries, or processing large and poorly-structured source XML files can affect the performance of your application's transformation logic. You should transform application data to XML just before the data leaves the boundaries of the application. Keeping the data in binary format is the most efficient way to perform data manipulations.
- **Inappropriate use of the DataSet**. It is a common misconception that the **DataSet** is an XML store. It is not designed to be an XML store, although it does provide XML functionality. You cannot perform efficient XPath queries by using a **DataSet**, and it cannot represent all forms of XML. The **DataSet** provides a disconnected cache of data and is useful in some scenarios where you want to pass data across application layers. However, it should not be used for general-purpose XML manipulation of data.
- **Failure to cache and to precompile schemas, style sheets, and XPath queries**. XML schemas, XSLT, and XPath must be interpreted before the .NET Framework classes can process them. The .NET Framework classes that represent each of these items provide a mechanism for preprocessing and caching the resources. Preprocessing is also called compilation in some cases. For example, XSLT style sheets should be precompiled and cached for repeated use, as should XML schemas.
- **Inefficient data retrieval**. Retrieving XML data from a data source on a per-request basis, rather than caching the data, can cause performance bottlenecks.

Design Considerations

Appropriate design decisions can help you address many XML-related performance issues early in the application life cycle. The following recommendations help to ensure that XML processing in your application does not lead to performance and scalability issues:

- **Choose the appropriate XML class for the job**.
- **Consider validating large documents**.
- **Process large documents in chunks if possible**.
- **Use streaming interfaces**.
- **Consider hard-coded transformations**.
- **Consider element and attribute name lengths**.
- **Consider sharing the XmlNameTable**.

Choose the Appropriate XML Class for the Job

To help you choose the appropriate .NET Framework class to process XML, consider the following guidelines:

- Use **XmlTextReader** to process XML data quickly in a forward, read-only manner without using validation, XPath, and XSLT services.
- Use **XmlValidatingReader** to process and to validate XML data. Process and validate the XML data in a forward, read-only manner according to an XML schema or a DTD.
- Use **XPathNavigator** to obtain read-only, random access to XML data and to use XPath queries. To create an **XPathNavigator** object over an XML document, you must call the **XPathDocument.CreateNavigator** method.
- Use **XmlTextWriter** to write XML documents. You cannot use **XmlTextWriter** to update XML documents.
- Use the **XmlTextReader** and **XmlTextWriter,** in combination, for simple transformations rather than resorting to loading an **XmlDocument** or using XSLT. For example, updating all the price element values in a document can be achieved by reading with the **XmlTextReader**, updating the value and then writing to the **XmlTextWriter**, typically by using the **WriteNode** method.
- Use the **XmlDocument** class to update existing XML documents, or to perform XPath queries and updates in combination. To use XPath queries on the **XmlDocument,** use the **Select** method.
- If possible, use client-side XML processing to improve performance and to reduce bandwidth.

Consider Validating Large Documents

When you use the **XmlDocument** class to load a large document that contains errors because the format is not correct, you waste memory and CPU resources. Consider validating the input XML if there is a reasonable chance that the XML is invalid. In a closed environment, you might consider validation an unnecessary overhead, but the decision to use or not use validation is a design decision you need to consider.

You can perform the validation process and other operations at the same time because the validation class derives from **XmlReader**. For example, you can use **XmlValidatingReader** with **XmlSerializer** to deserialize and validate XML at the same time. The following code fragment shows how to use **XmlValidatingReader**.

```
// payload is the Xml data
StringReader stringReader = new StringReader( payload );
XmlReader xmlReader = new XmlTextReader( stringReader );
XmlValidatingReader vreader = new XmlValidatingReader( xmlReader );
vreader.Schemas.Add(XmlSchema.Read(
                        new XmlTextReader("xyz.xsd"), null ) );
vreader.ValidationType = ValidationType.Schema;
vreader.ValidationEventHandler += new ValidationEventHandler(ValidationCallBack);
```

You can also use a validating read with **XmlDocument** by passing the validating reader instance to the **XmlDocument.Load** method, as shown in the following code fragment.

```
XmlDocument doc = new XmlDocument();
doc.Load( xmlValidatingReaderInstance );
```

Validation comes at a performance cost and there is a tradeoff here between validating the XML documents early to catch invalid content as opposed to the additional processing time that validation takes even in a streaming scenario. Typically, using the **XmlValidatingReader** to validate an XML document is two to three times slower than using the **XmlTextReader** without validation and deciding on whether to perform validation depends on your particular application scenario.

Process Large Documents in Chunks If Possible

If you have very large XML documents to process, evaluate whether you can divide the documents and then process them in chunks. Dividing the documents makes processing them, by using XLST, more efficient.

Use Streaming Interfaces

Streaming interfaces, like the one provided by **XmlTextReader,** give better performance and scalability, compared to loading large XML documents into the **XmlDocument** or **XPathDocument** classes and then using DOM manipulation.

The DOM creates an in-memory representation of the entire XML document. The **XmlTextReader** is different from the DOM because **XmlTextReader** only loads 4-kilobyte (KB) buffers into memory. If you use the DOM to process large XML files, you can typically consume memory equivalent to three or four times the XML document size on disk.

Consider Hard-Coded Transformations

Using XSLT may be overly complicated for certain simple transformations such as changing a particular attribute value, replacing one node with another node, or appending or removing nodes from a document.

If using XSLT appears to be an overly-complicated approach for a simple transformation, you can use **XmlReader** and **XmlWriter** together to copy the document from **XmlReader** to **XmlWriter** and then modify the document while copying. The **XmlWriter.WriteNode** and **XmlWriter.WriteAttributes** methods receive an **XmlReader** instance, and the method copies the node and its child nodes to **XmlWriter**.

The disadvantage of using the classes to perform the transformation is that you can modify XSLT without having to recompile the code. However, in some situations, it might be better to hard code a transformation. A simple example of a hard-coded approach is shown in the following code fragment.

```
while( reader.Read() )
{
  if( reader.LocalName == "somethingToChange" )
  {
    writer.WriteStartElement( "somethingChanged" );
    writer.WriteAttributes( reader, false );
    //
  }
  else
  {
    writer.WriteNode( reader, false );
  }
}
```

Consider Element and Attribute Name Lengths

Consider the length of the element names and the length of the attribute names that you use. These names are included as metadata in your XML documents. Therefore, the length of an element or attribute name affects the document size. You need to balance size issues with ease of human interpretation and future maintenance. Try to use names that are short and meaningful.

Consider Sharing the XmlNameTable

Share the **XmlNameTable** class that is used to store element and attribute names across multiple XML documents of the same type to improve performance.

XML classes like **XmlTextReader** and **XmlDocument** use the **XmlNameTable** class to store elements and attribute names. When elements, attributes, or prefixes occur multiple times in the document, they are stored only once in the **XmlNameTable** and an atomized string is returned. When an element, attribute, or prefix is looked up, an object comparison of the strings is performed instead of a more expensive string operation.

The following code shows how to obtain access to and store the **XmlNameTable** object.

```
System.Xml.XmlTextReader reader = new System.Xml.XmlTextReader("small.xml");
System.Xml.XmlNameTable nt = reader.NameTable;
// Store XmlNameTable in Application scope and reuse it
System.Xml.XmlTextReader reader2 = new System.Xml.XmlTextReader("Test.xml", nt);
```

More Information

For more information about object comparisons, see MSDN article, "Object Comparison Using XmlNameTable," at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconobjectcomparisonusingxmlnametable.asp*.

Implementation Considerations

When you move from application design to development, you must carefully consider the implementation details of your XML code.

When you write XML processing code, it is important for you to use the most relevant .NET Framework classes. Do not use **DataSet** objects for general-purpose XML manipulation of data.

When you are working with large XML documents, consider validating them first if the cost of validation is less than the cost of redundant downstream processing. Also, consider working with large XML documents in chunks. Consider caching schema for repeated XML validation.

By following best-practice implementation guidelines, you can increase the performance of your XML processing code. The following sections highlight performance considerations for XML features and scenarios.

Parsing XML

The .NET Framework provides several ways to parse XML. The best approach depends on your scenario:

- If you want to read the document once, use **XmlTextReader**. This provides forward-only, read-only, and non-cached access to XML data. This model provides optimized performance and memory conservation.
- If you need to edit and to query the document, use **XmlDocument** with the **Select** command. This approach consumes large amounts of memory.
- If you want faster, read-only XPath query-based access to data, use **XPathDocument** and **XPathNavigator**.

The .NET Framework does not provide a SAX model. The **XmlReader** approach is similar and offers a number of advantages.

The following recommendations help you to ensure that XML parsing in your application is as efficient as possible:

- **Use XmlTextReader to parse large XML documents**.
- **Use XmlValidatingReader for validation**.
- **Consider combining XmlReader and XmlDocument**.
- **On the XmlReader use the MoveToContent and Skip methods to skip unwanted items**.

Use XmlTextReader to Parse Large XML Documents

Use the **XmlTextReader** class to process large XML documents in an efficient, forward-only manner. **XmlTextReader** uses small amounts of memory. Avoid using the DOM because the DOM reads the entire XML document into memory. If the entire XML document is read into memory, the scalability of your application is limited. Using **XmlTextReader** in combination with an **XmlTextWriter** class permits you to handle much larger documents than a DOM-based **XmlDocument** class.

The following code fragment shows how to use **XmlTextReader** to process large XML documents.

```
while (reader.Read())
{
    switch (reader.NodeType)
    {
      case System.Xml.XmlNodeType.Element :
      {
        if( reader.Name.Equals("patient")
        && reader.GetAttribute("number").Equals("25") )
      {
        doc = new System.Xml.XmlDocument();
        XmlNode node = doc.ReadNode( reader );
        doc.AppendChild( node );
      }
      break;
    }
  }
}
```

You can only use **XmlTextReader** and **XmlValidatingReader** to process files that are up to 2 gigabytes (GB) in size. If you need to process larger files, divide the source file into multiple smaller files or streams.

Use XmlValidatingReader for Validation

If you need to validate an XML document, use **XmlValidatingReader**. The **XmlValidatingReader** class adds XML Schema and DTD validation support to **XmlReader**. For more information, see "Validating XML" later in this chapter.

Consider Combining XmlReader and XmlDocument

In certain circumstances, the best solution may be to combine the pull model and the DOM model. For example, if you only need to manipulate part of a very large XML document, you can use **XmlReader** to read the document, and then you can construct a DOM that has only the data required for additional modification. This approach is shown in the following code fragment.

```
while (reader.Read())
{
  switch (reader.NodeType)
  {
    case System.Xml.XmlNodeType.Element :
    {
      if( reader.Name.Equals("patient")
          && reader.GetAttribute("number").Equals("25") )
      {
        doc = new System.Xml.XmlDocument();
        XmlNode node = doc.ReadNode( reader );
        doc.AppendChild( node );
      }
      break;
    }
  }
}
```

On the XmlReader, Use the MoveToContent and Skip Methods to Skip Unwanted Items

Use the **XmlReader.MoveToContent** method to skip white space, comments, and processing instructions, and to move to the next content element. **MoveToContent** skips to the next **Text**, **CDATA**, **Element**, **EndElement**, **EntityReference**, or **EndEntity** node. You can also skip the current element by using the **XmlReader.Skip** method.

For example, consider the following XML input.

```
<?xml version="1.0">
<!DOCTYPE price SYSTEM "abc">
<!--the price of the book -->
<price>123</price>
```

The following code finds the price element "123.4" and then converts the text content to a double:

```
if (readr.MoveToContent() == XmlNodeType.Element && readr.Name =="price")
{
    _price = XmlConvert.ToDouble(readr.ReadString());
}
```

For more information about how to use the **MoveToContent** method, see MSDN article, "Skipping Content with XmlReader" at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconSkippingContentWithXmlReader.asp*.

More Information

For more information about **XMLReader**, see MSDN article "Comparing XmlReader to SAX Reader," at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcomparingxmlreadertosaxreader.asp*.

For more information about how to parse XML, see the following Microsoft Knowledge Base articles:

- 301228, "HOW TO: Read XML Data from a Stream in .NET Framework SDK," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;301228*
- 301233, "HOW TO: Modify and Save XML with the XmlDocument Class in .NET Framework SDK" at *http://support.microsoft.com/default.aspx?scid=kb;en-us;301233*

Validating XML

You can validate XML to ensure that a document conforms to a schema definition. This involves verifying that the document includes the necessary elements and attributes in the correct sequence. This is often referred to as validating the content model of the document. Validating XML can also involve data type checking. The preferred approach is to validate XML documents against XML schema definitions (XSD schemas). However, you can also validate XML against Document Type Definitions (DTD) and XML-Data Reduced Schemas (XDR) schemas.

Validation introduces additional performance overhead. If there is a strong likelihood that clients will pass invalid XML to your application, you should validate and reject bad data early to minimize redundant processing effort. In a closed environment where you can make certain guarantees about the validity of input data, you might consider validation to be unnecessary overhead.

If you do use validation, consider the following to help minimize the validation overhead:

- **Use XmlValidatingReader**.
- **Do not validate the same document more than once**.
- **Consider caching the schema**.

Use XmlValidatingReader

A lot of specialized code is required to validate an XML document to ensure that the document matches the rules defined in a schema or a DTD. By using **XmlValidatingReader**, you avoid writing this code by hand. It also means that after validation, your application can make assumptions about the condition of the data. Permitting your application to make assumptions about the data can reduce the quantity of error-handling code that you would otherwise have to write.

Do Not Validate the Same Document More Than Once

Make sure that you do not waste processor cycles by validating the same source document multiple times.

Consider Caching the Schema

If you repeatedly validate input XML against the same schema on a per-request basis, consider loading the schema once and retaining it in memory for later requests. This avoids the overhead of parsing, loading, and compiling the schema multiple times. The following code fragment shows how to cache a schema in an **XmlSchemaCollection** object.

```
XmlTextReader tr = new XmlTextReader("Books.xml");
XmlValidatingReader vr = new XmlValidatingReader(tr);
XmlSchemaCollection xsc = new XmlSchemaCollection();
xsc.Add("urn:bookstore-schema", "Books.xsd");
vr.Schemas.Add(xsc);
```

Validation comes at a cost. Typically, using the **XmlValidatingReader** to validate a document is two to three times slower than using the **XmlTextReader** to simply parse the XML, so ensure that this is worth the cost in your particular application scenario.

More Information

For more information about XML validation, see Microsoft Knowledge Base article 307379, "HOW TO: Validate an XML Document by Using DTD, XDR, or XSD in Visual C# .NET," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;307379*.

Writing XML

If your application needs to generate XML, you can write XML by using the **XmlDocument** or the **XmlTextWriter** classes. The **XmlTextWriter** class performs better, but you should use **XmlDocument** if you need to manipulate the XML in memory before you write the XML to a byte stream.

Use XmlTextWriter

Using **XmlTextWriter** is the preferred way to write XML. The **XmlTextWriter** class creates XML in a forward-only cursor style. It also takes care of XML encoding, handling of special characters, adding quotes to attribute values, namespace declarations, and insertion of end tags. By performing these tasks, **XmlTextWriter** helps ensure the output is well-formed. The following code fragment shows how to use **XmlTextWriter** to create XML.

```
static void WriteQuote(XmlWriter writer, string symbol,
                double price, double change, long volume)
{
   writer.WriteStartElement("Stock");
   writer.WriteAttributeString("Symbol", symbol);
   writer.WriteElementString("Price", XmlConvert.ToString(price));
   writer.WriteElementString("Change", XmlConvert.ToString(change));
   writer.WriteElementString("Volume", XmlConvert.ToString(volume));
   writer.WriteEndElement();
}

public static void Main(){
    XmlTextWriter writer = new XmlTextWriter(Console.Out);
    writer.Formatting = Formatting.Indented;
    WriteQuote(writer, "MSFT", 74.125, 5.89, 69020000);
    writer.Close();
}
```

The previous code produces the following output.

```
<Stock Symbol="MSFT">
  <Price>74.125</Price>
  <Change>5.89</Change>
  <Volume>69020000</Volume>
</Stock>
```

XPath Queries

XML Path Language (XPath) provides a general-purpose query notation that you can use to search and to filter the elements and the text in an XML document. Query performance varies depending on the complexity of the query and the size of the source XML document. Use the following guidelines to optimize the way your application uses XPath:

- **Use XPathDocument to process XPath statements**.
- **Avoid the // operator by reducing the search scope**.
- **Compile both dynamic and static XPath expressions**.

Use XPathDocument to Process XPath Statements

If your application contains large amounts of intensive XPath or XSLT code, use **XPathDocument**, instead of **XmlDataDocument** or **XmlDocument**, to process XPath statements. However, a common scenario is to use XSLT to transform the default XML representation of a **DataSet**, in which case you can use the **XmlDataDocument** class for small-sized **DataSet** objects.

The **XPathDocument** class provides a fast, read-only cache for XML document processing by using XSLT. It provides an optimized in-memory tree structure that you can view by using the **XPathNavigator** interface. To move between a selected set of nodes by using an XPath query, use the **XPathNodeIterator** as shown in the following code.

```
XPathDocument Doc = new XPathDocument(FileName);
XPathNavigator nav = Doc.CreateNavigator();
XPathNodeIterator Iterator = nav.Select("/bookstore/book");
while (Iterator.MoveNext())
{
  Console.WriteLine(Iterator.Current.Name);
}
```

Avoid the // Operator by Reducing the Search Scope

Use path-specific XPath expressions, instead of the // operator, because the // operator performs a recursive descent and then searches the entire subtree for matches. Look for opportunities to reduce the search scope by restricting the search to specific portions of the XML subtree.

For example, if you know that a particular item only exists beneath a specific parent element, begin the search from that parent element and not from the root element. The following code fragment shows how to search an entire XML document and how to search beneath a specific element.

```
XPathDocument doc = new XPathDocument("books.xml");
XPathNavigator nav = doc.CreateNavigator();
// this will search entire XML for matches
XPathExpression Expr = nav.Compile("//price");
// this will reduce the search scope
XPathExpression Expr2 = nav.Compile("books/book/price");
```

Compile Both Dynamic and Static XPath Expressions

The **XPathNavigator** class provides a **Compile** method that you can use to compile a string that represents an XPath expression. If you use the **Select** method repeatedly instead of passing a string each time, use the **Compile** method to compile and then reuse the XPath expression. The **Compile** method returns an **XPathExpression** object. The following code fragment shows how to use the **Compile** method.

```
XPathDocument doc = new XPathDocument("one.xml");
XPathNavigator nav = doc.CreateNavigator();
XPathExpression Expr = nav.Compile("/invoices/invoice[number>20]");
// Save Expr in application scope and reuse it
XPathNodeIterator iterator = nav.Select(Expr);
while (iterator.MoveNext())
{
  str = iterator.Current.Name;
}
```

You can also compile dynamic expressions. For more information, see MSDN article, "Adding Custom Functions to XPath," at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml10212002.asp*.

More Information

For more information about using XPath expressions, see the following Microsoft Knowledge Base articles:

- 308333, "HOW TO: Query XML with an XPath Expression by Using Visual C# .NET," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;308333*
- 301111, "HOW TO: Navigate XML with the XPathNavigator Class by Using Visual Basic .NET," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;301111*
- 317069, "HOW TO: Execute XPath Queries by Using the System.Xml.XPath Classes," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;317069.*

XSLT Processing

Extensible Stylesheet Language Transformation (XSLT) specifies a transformation language for XML documents. You can use XSLT to transform the content of an XML document into another XML document that has a different structure. Or, you can use XSLT to transform an XML document into a different document format, such as HTML or comma-separated text.

The .NET Framework XSLT processor is implemented by the **XslTransform** class in the **System.Xml.Xsl** namespace. You typically perform XSLT processing by using either the DOM or the **XPathDocument** class. The **XPathDocument** class offers superior performance. Typically, transformations that use the **XPathDocument** class are 20 to 30 percent faster than transformations that use the **XmlDocument** class once the documents have been loaded. Actual percentages depend on your XSLT, input document, and computer.

The following recommendations help you optimize XSLT processing in your application. XSLT frequently uses XPath queries to select parts of an XML document. Therefore, the efficiency of your XPath queries directly affects XSLT performance.

- **Use XPathDocument for faster XSLT transformations**.
- **Consider caching compiled style sheets**.
- **Split complex transformations into several stages**.
- **Minimize the size of the output document**.
- **Write efficient XSLT**.

Use XPathDocument for Faster XSLT Transformations

The **XPathDocument** class provides a fast, read-only cache for XML document processing by using XSLT. Use this class for optimum performance. The following code fragment shows how to use this class.

```
XslTransform xslt = new XslTransform();
xslt.Load(someStylesheet);
XPathDocument doc = new XPathDocument("books.xml");
StringWriter fs = new StringWriter();
xslt.Transform(doc, null, fs, null);
```

Consider Caching Compiled Style Sheets

If your application performs a common transformation by using the same style sheet on a per-request basis, consider caching the style sheet between requests. This is a strong recommendation because it saves you having to recompile the style sheet every time you perform a transformation. In a .NET application, you compile the .NET application once into an executable file and then run it many times. The same applies to XSLT.

The following code fragment shows the **XslTransform** class being cached in the ASP.NET application state. Note that the **XslTransform** class is thread-safe.

```
protected void Application_Start(Object sender, EventArgs e)
{
  //Create the XslTransform and load the style sheet.
  XslTransform xslt = new XslTransform();
  xslt.Load(stylesheet);
  //Save it to ASP.NET application scope
  Application["XSLT"] = xslt;
}
private void Page_Load(object sender, System.EventArgs e)
{
  // Re-use the XslTransform stored in the application scope
  XslTransform xslt = Application["XSLT"];
}
```

Caching Extension Objects

You can use **Extension** objects to implement custom functions that are referenced in XPath query expressions that are used in an XSLT style sheet. The XSLT processor does not automatically cache **Extension** objects. However, you can cache **XsltArgumentList** objects that are used to supply **Extension** objects. This approach is shown in the following code fragment.

```
// Create the XslTransform and load the style sheet.
XslTransform xslt = new XslTransform();
xslt.Load(stylesheet);
// Load the XML data file.
XPathDocument doc = new XPathDocument(filename);
// Create an XsltArgumentList.
XsltArgumentList xslArgCache = new XsltArgumentList();
// Add an object to calculate the circumference of the circle.
Calculate obj = new Calculate();
xslArgCache.AddExtensionObject("urn:myObj", obj);
// Create an XmlTextWriter to output to the console.
XmlTextWriter writer = new XmlTextWriter(Console.Out);
// Transform the file.
xslt.Transform(doc, xslArgCache, writer);
writer.Close();
// Reuse xslArgCache
........
xslt.Transform(doc2, xslArgCache, writer2);
```

Split Complex Transformations into Several Stages

You can incrementally transform an XML document by using multiple XSLT style sheets to generate the final required output. This process is referred to as *pipelining* and is particularly beneficial for complex transformations over large XML documents.

More Information

For more information about how to split complex transformations into several stages, see Microsoft Knowledge Base article 320847, "HOW TO: Pipeline XSLT Transformations in .NET Applications," at *http://support.microsoft.com /default.aspx?scid=kb;en-us;320847*.

Minimize the Size of the Output Document

Try to keep the output document size to a minimum. If you are generating HTML, there are a couple of ways to do this.

First, use cascading style sheets to apply formatting instead of embedding formatting metadata in the HTML. Second, consider how your HTML is indented, and avoid unnecessary white space. To do so, set the indent setting to **no** as shown in the following XSLT fragment.

```
<xsl:output method="html" indent="no"/>
```

By default, the value of the **indent** attribute is **yes**.

Write Efficient XSLT

When you develop XLST style sheets, start by making sure that your XPath queries are efficient. For more information, see "XPath Queries" earlier in this chapter. Here are some common guidelines for writing efficient XSLT style sheets:

- Do not evaluate the same node set more than once. Save the node set in a <**xsl:variable**> declaration.
- Avoid using the <**xsl:number**> tag if you can. For example, use the **Position** method instead.
- Use the <**xsl:key**> tag to solve grouping problems.
- Avoid complex patterns in template rules. Instead, use the <**xsl:choose**> tag in the rule.
- Be careful when you use the preceding[-sibling] or the following[-sibling] axes. Use of these axes often involves algorithms that significantly affect performance.
- Do not sort the same node set more than once. If necessary, save it as a result tree fragment, and then access it by using the **node-set()** extension function.

- To output the text value of a simple #PCDATA element, use the <**xsl:value-of**> tag in preference to the <**xsl:apply-templates**> tag.
- Avoid using inline script. Use extensions written in Microsoft Visual C# or Microsoft Visual Basic .NET to pass it as a parameter to the **Transform** call, and then bind to it by using the <**xsl:param**> tag. However, if you cache the style sheet in your application as described earlier, this achieves the same result. It then is perfectly acceptable to use script in the style sheet. In other words, this is just a compile-time issue.
- Factor common queries into nested templates. For example, if you have two templates that match on "a/b/c" and "a/b/d," factor the templates into one common template that matches on "a/b." Have the common template call templates that match on "c" and "d."

More Information

For more information about XSLT processing, see the following Microsoft Knowledge Base articles:

- 325689, "INFO: Performance of XSLT Transformations in the .NET Framework," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;325689*
- 313997, "INFO: Roadmap for Executing XSLT Transformations in .NET Applications," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;313997*
- 307322, "HOW TO: Apply an XSL Transformation to an XML Document by Using Visual C# .NET," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;307322*
- 300929, "HOW TO: Apply an XSL Transformation from an XML Document to an XML Document by Using Visual Basic .NET," at *http://support.microsoft.com /default.aspx?scid=kb;en-us;300929*
- 320847, "HOW TO: Pipeline XSLT Transformations in .NET Applications," at *http://support.microsoft.com/default.aspx?scid=kb;en-us;320847*

Summary

When you build .NET applications, you use XML extensively. Whether you read XML from a simple configuration file, retrieve XML from a database, or access a Web service, knowing how to work with XML in the .NET Framework is essential. The performance guidelines presented in this chapter help you understand the necessary tradeoffs when you use **System.Xml**.

While the flexibility and power of XML are well documented, the text-based nature of XML and the metadata that is conveyed in an XML document mean that XML is not a compact data format. XML may require substantial processing effort. It is important for you to analyze how your application uses XML to ensure that XML processing does not create performance bottlenecks. The key factors that affect performance are parsing effort, XLST processing, and schema validation.

Additional Resources

For more information about XML performance, see the following resources:

- For a printable checklist, see the "Checklist: XML Performance" checklist in the "Checklists" section of this guide.
- Chapter 4, "Architecture and Design Review of a .NET Application for Performance and Scalability."
- Chapter 13, "Code Review: .NET Application Performance."
- Chapter 15, "Measuring .NET Application Performance."
- Chapter 16, "Testing .NET Application Performance."
- Chapter 17, "Tuning .NET Application Performance."
- For more information about XML, see the "Microsoft XML Developer Center" at *http://msdn.microsoft.com/xml*. This site regularly publishes articles on XML and the .NET Framework, including best practices for application development.

Checklist: XML Performance

How to Use This Checklist

This checklist is a companion to Chapter 9, "Improving XML Performance."

Design Considerations

Check	Description
☐	Choose the appropriate XML class for the job.
☐	Consider validating large documents.
☐	Process large documents in chunks, if possible.
☐	Use streaming interfaces.
☐	Consider hard-coded transformations.
☐	Consider element and attribute name lengths.
☐	Consider sharing the **XmlNameTable**.

Parsing XML

Check	Description
☐	Use **XmlTextReader** to parse large XML documents.
☐	Use **XmlValidatingReader** for validation.
☐	Consider combining **XmlReader** and **XmlDocument**.
☐	On the **XmlReader**, use the **MoveToContent** and **Skip** methods to skip unwanted items.

Validating XML

Check	Description
☐	Use **XmlValidatingReader**.
☐	Do not validate the same document more than once.
☐	Consider caching the schema.

Writing XML

Check	Description
☐	Use **XmlTextWriter**.

XPath Queries

Check	Description
☐	Use **XPathDocument** to process XPath statements.
☐	Avoid the // operator by reducing the search scope.
☐	Compile both dynamic and static XPath expressions.

XSLT Processing

Check	Description
☐	Use **XPathDocument** for faster XSLT transformations.
☐	Consider caching compiled style sheets.
☐	Consider splitting complex transformations into several stages.
☐	Minimize the size of the output document.
☐	Write efficient XSLT.