CHAPTER 18

SQLXML

The key to everything is happiness. Do what you can to be happy in this world. Life is short—too short to do otherwise. The deferred gratification you mention so often is more deferred than gratifying.

—H. W. Kenton

NOTE: This chapter assumes that you're running, at a minimum, SQL Server 2000 with SQLXML 3.0. The SQLXML Web releases have changed and enhanced SQL Server's XML functionality significantly. For the sake of staying current with the technology, I'm covering the latest version of SQLXML rather than the version that shipped with the original release of SQL Server 2000.

This chapter updates the coverage of SQLXML in my last book, *The Guru's Guide to SQL Server Stored Procedures, XML, and HTML*. That book was written before Web Release 1 (the update to SQL Server 2000's original SQLXML functionality) had shipped. As of this writing, SQLXML 3.0 (which would be the equivalent of Web Release 3 had Microsoft not changed the naming scheme) has shipped, and Yukon, the next version of SQL Server, is about to go into beta test.

This chapter will also get more into how the SQLXML technologies are designed and how they fit together from an architectural standpoint. As with the rest of the book, my intent here is to get beyond the "how to" and into the "why" behind how SQL Server's technologies work.

I must confess that I was conflicted when I sat down to write this chapter. I wrestled with whether to update the SQLXML coverage in my last book, which was more focused on the practical application of SQLXML but which I felt really needed updating, or to write something completely new on just the architectural aspects of SQLXML, with little or no discussion of

how to apply them in practice. Ultimately, I decided to do both things. In keeping with the chief purpose of this book, I decided to cover the architectural aspects of SQLXML, and, in order to stay up with the current state of SQL Server's XML family of technologies, I decided to update the coverage of SQLXML in my last book from the standpoint of practical use. So, this chapter updates what I had to say previously about SQLXML and also delves into the SQLXML architecture in ways I've not done before.

Overview

With the popularity and ubiquity of XML, it's no surprise that SQL Server has extensive support for working with it. Like most modern DBMSs, SQL Server regularly needs to work with and store data that may have originated in XML. Without this built-in support, getting XML to and from SQL Server would require the application developer to translate XML data before sending it to SQL Server and again after receiving it back. Obviously, this could quickly become very tedious given the pervasiveness of the language.

SQL Server is an XML-enabled DBMS. This means that it can read and write XML data. It can return data from databases in XML format, and it can read and update data stored in XML documents. As Table 18.1 illustrates,

Table 18.1 SQL Server's XML Features

Feature	Purpose
FOR XML	An extension to the SELECT command that allows result sets to be returned as XML
OPENXML	Allows reading and writing of data in XML documents
XPath queries	Allows SQL Server databases to be queried using XPath syntax
Schemas	Supports XSD and XDR mapping schemas and XPath queries against them
SOAP support	Allows clients to access SQL Server's functionality as a Web service
Updategrams	XML templates through which data modifications can be applied to a database
Managed classes	Classes that expose the functionality of SQLXML inside the .NET Framework
XML Bulk Load	A high-speed facility for loading XML data into a SQL Server database

out of the box, SQL Server's XML features can be broken down into eight general categories.

We'll explore each of these in this chapter and discuss how they work and how they interoperate.

MSXML

SQL Server uses Microsoft's XML parser, MSXML, to load XML data, so we'll begin our discussion there. There are two basic ways to parse XML data using MSXML: using the Document Object Model (DOM) or using the Simple API for XML (SAX). Both DOM and SAX are W3C standards. The DOM method involves parsing the XML document and loading it into a tree structure in memory. The entire document is materialized and stored in memory when processed this way. An XML document parsed via DOM is known as a DOM document (or just "DOM" for short). XML parsers provide a variety of ways to manipulate DOM documents. Listing 18.1 shows a short Visual Basic app that demonstrates parsing an XML document via DOM and querying it for a particular node set. (You can find the source code to this app in the CH18\msxmltest subfolder on the CD accompanying this book.)

Listing 18.1

```
Private Sub Command1_Click()

  Dim bstrDoc As String

  bstrDoc = "<Songs> " & _
                "<Song>One More Day</Song>" & _
                "<Song>Hard Habit to Break</Song>" & _
                "<Song>Forever</Song>" & _
                "<Song>Boys of Summer</Song>" & _
                "<Song>Cherish</Song>" & _
                "<Song>Dance</Song>" & _
                "<Song>I Will Always Love You</Song>" & _
            "</Songs>"

  Dim xmlDoc As New DOMDocument30

  If Len(Text1.Text) = 0 Then
    Text1.Text = bstrDoc
  End If
```

```
  If Not xmlDoc.loadXML(Text1.Text) Then
    MsgBox "Error loading document"
  Else
    Dim oNodes As IXMLDOMNodeList
    Dim oNode As IXMLDOMNode

    If Len(Text2.Text) = 0 Then
      Text2.Text = "//Song"
    End If
    Set oNodes = xmlDoc.selectNodes(Text2.Text)

    For Each oNode In oNodes
      If Not (oNode Is Nothing) Then
        sName = oNode.nodeName
        sData = oNode.xml
        MsgBox "Node <" + sName + ">:" _
            + vbNewLine + vbTab + sData + vbNewLine
      End If
    Next

    Set xmlDoc = Nothing
  End If
End Sub
```

We begin by instantiating a DOMDocument object, then call its loadXML method to parse the XML document and load it into the DOM tree. We call its selectNodes method to query it via XPath. The selectNodes method returns a node list object, which we then iterate through using For Each. In this case, we display each node name followed by its contents via VB's MsgBox function. We're able to access and manipulate the document as though it were an object because that's exactly what it is—parsing an XML document via DOM turns the document into a memory object that you can then work with just as you would any other object.

SAX, by contrast, is an event-driven API. You process an XML document via SAX by configuring your application to respond to SAX events. As the SAX processor reads through an XML document, it raises events each time it encounters something the calling application should know about, such as an element starting or ending, an attribute starting or end-

ing, and so on. It passes the relevant data about the event to the application's handler for the event. The application can then decide what to do in response—it could store the event data in some type of tree structure, as is the case with DOM processing; it could ignore the event; it could search the event data for something in particular; or it could take some other action. Once the event is handled, the SAX processor continues reading the document. At no point does it persist the document in memory as DOM does. It's really just a parsing mechanism to which an application can attach its own functionality. In fact, SAX is the underlying parsing mechanism for MSXML's DOM processor. Microsoft's DOM implementation sets up SAX event handlers that simply store the data handed to them by the SAX engine in a DOM tree.

As you've probably surmised by now, SAX consumes far less memory than DOM does. That said, it's also much more trouble to set up and use. By persisting documents in memory, the DOM API makes working with XML documents as easy as working with any other kind of object.

SQL Server uses MSXML and the DOM to process documents you load via sp_xml_preparedocument. It restricts the virtual memory MSXML can use for DOM processing to one-eighth of the physical memory on the machine or 500MB, whichever is less. In actual practice, it's highly unlikely that MSXML would be able to access 500MB of virtual memory, even on a machine with 4GB of physical memory. The reason for this is that, by default, SQL Server reserves most of the user mode address space for use by its buffer pool. You'll recall that we talked about the MemToLeave space in Chapter 11 and noted that the non–thread stack portion defaults to 256MB on SQL Server 2000. This means that, by default, MSXML won't be able to use more than 256MB of memory—and probably considerably less given that other things are also allocated from this region—regardless of the amount of physical memory on the machine.

The reason MSXML is limited to no more than 500MB of virtual memory use regardless of the amount of memory on the machine is that SQL Server calls the GlobalMemoryStatus Win32 API function to determine the amount of available physical memory. GlobalMemoryStatus populates a MEMORYSTATUS structure with information about the status of memory use on the machine. On machines with more than 4GB of physical memory, GlobalMemoryStatus can return incorrect information, so Windows returns a -1 to indicate an overflow. The Win32 API function GlobalMemoryStatusEx exists to address this shortcoming, but SQLXML does not call it. You can see this for yourself by working through the following exercise.

Exercise 18.1 Determining How MSXML Computes Its Memory Ceiling

1. Restart your SQL Server, preferably from a console since we will be attaching to it with WinDbg. This should be a test or development system, and, ideally, you should be its only user.
2. Start Query Analyzer and connect to your SQL Server.
3. Attach to SQL Server using WinDbg. (Press F6 and select sqlservr.exe from the list of running tasks; if you have multiple instances, be sure to select the right one.)
4. At the WinDbg command prompt, add the following breakpoint:

```
bp kernel32!GlobalMemoryStatus
```

5. Once the breakpoint is added, type g and hit Enter to allow SQL Server to run.
6. Next, return to Query Analyzer and run the following query:

```
declare @doc varchar(8000)
set @doc='
<Songs>
  <Song name="She''s Like the Wind" artist="Patrick Swayze"/>
  <Song name="Hard to Say I''m Sorry" artist="Chicago"/>
  <Song name="She Loves Me" artist="Chicago"/>
  <Song name="I Can''t Make You Love Me" artist="Bonnie Raitt"/>
  <Song name="Heart of the Matter" artist="Don Henley"/>
  <Song name="Almost Like a Song" artist="Ronnie Milsap"/>
  <Song name="I''ll Be Over You" artist="Toto"/>
</Songs>
'
declare @hDoc int
exec sp_xml_preparedocument @hDoc OUT, @doc
```

7. The first time you parse an XML document using sp_xml_preparedocument, SQLXML calls GlobalMemoryStatus to retrieve the amount of physical memory in the machine, then calls an undocumented function exported by MSXML to restrict the amount of virtual memory it may allocate. (I had you restart your server so that we'd be sure to go down this code path.) This undocumented MSXML function is exported by ordinal rather than by name from the MSXML*n*.DLL and was added to MSXML expressly for use by SQL Server.
8. At this point, Query Analyzer should appear to be hung because your breakpoint has been hit in WinDbg and SQL Server has been stopped. Switch back to WinDbg and type kv at the command prompt to dump the call stack of the current thread. Your stack should look something like this (I've omitted everything but the function names):

```
KERNEL32!GlobalMemoryStatus (FPO: [Non-Fpo])
sqlservr!CXMLLoadLibrary::DoLoad+0x1b5
sqlservr!CXMLDocsList::Load+0x58
sqlservr!CXMLDocsList::LoadXMLDocument+0x1b
sqlservr!SpXmlPrepareDocument+0x423
sqlservr!CSpecProc::ExecuteSpecial+0x334
sqlservr!CXProc::Execute+0xa3
sqlservr!CSQLSource::Execute+0x3c0
sqlservr!CStmtExec::XretLocalExec+0x14d
sqlservr!CStmtExec::XretExecute+0x31a
sqlservr!CMsqlExecContext::ExecuteStmts+0x3b9
sqlservr!CMsqlExecContext::Execute+0x1b6
sqlservr!CSQLSource::Execute+0x357
sqlservr!language_exec+0x3e1
```

9. You'll recall from Chapter 3 that we discovered that the entry point for T-SQL batch execution within SQL Server is language_exec. You can see the call to language_exec at the bottom of this stack—this was called when you submitted the T-SQL batch to the server to run. Working upward from the bottom, we can see the call to SpXmlPrepareDocument, the internal "spec proc" (an extended procedure implemented internally by the server rather than in an external DLL) responsible for implementing the sp_xml_preparedocument xproc. We can see from there that SpXmlPrepareDocument calls LoadXMLDocument, LoadXMLDocument calls a method named Load, Load calls a method named DoLoad, and DoLoad calls GlobalMemoryStatus. So, that's how we know how MSXML computes the amount of physical memory in the machine, and, knowing the limitations of this function, that's how we know the maximum amount of virtual memory MSXML can use.
10. Type q and hit Enter to quit WinDbg. You will have to restart your SQL Server.

FOR XML

Despite MSXML's power and ease of use, SQL Server doesn't leverage MSXML in all of its XML features. It doesn't use it to implement server-side FOR XML queries, for example, even though it's trivial to construct a DOM document programmatically and return it as text. MSXML has facilities that make this quite easy. For example, Listing 18.2 presents a Visual Basic app that executes a query via ADO and constructs a DOM document on-the-fly based on the results it returns.

Listing 18.2

```
Private Sub Command1_Click()

  Dim xmlDoc As New DOMDocument30
  Dim oRootNode As IXMLDOMNode

  Set oRootNode = xmlDoc.createElement("Root")

  Set xmlDoc.documentElement = oRootNode

  Dim oAttr As IXMLDOMAttribute
  Dim oNode As IXMLDOMNode

  Dim oConn As New ADODB.Connection
  Dim oComm As New ADODB.Command
  Dim oRs As New ADODB.Recordset

  oConn.Open (Text3.Text)
  oComm.ActiveConnection = oConn

  oComm.CommandText = Text1.Text
  Set oRs = oComm.Execute

  Dim oField As ADODB.Field

  While Not oRs.EOF
    Set oNode = xmlDoc.createElement("Row")
    For Each oField In oRs.Fields
      Set oAttr = xmlDoc.createAttribute(oField.Name)
      oAttr.Value = oField.Value
      oNode.Attributes.setNamedItem oAttr
    Next
    oRootNode.appendChild oNode
    oRs.MoveNext
  Wend

  oConn.Close

  Text2.Text = xmlDoc.xml

  Set xmlDoc = Nothing
  Set oRs = Nothing
  Set oComm = Nothing
  Set oConn = Nothing
End Sub
```

As you can see, translating a result set to XML doesn't require much code. The ADO Recordset object even supports being streamed directly to an XML document (via its Save method), so if you don't need complete control over the conversion process, you might be able to get away with even less code than in my example.

As I've said, SQL Server doesn't use MSXML or build a DOM document in order to return a result set as XML. Why is that? And how do we know that it doesn't use MSXML to process server-side FOR XML queries? I'll answer both questions in just a moment.

The answer to the first question should be pretty obvious. Building a DOM from a result set before returning it as text would require SQL Server to persist the entire result set in memory. Given that the memory footprint of the DOM version of an XML document is roughly three to five times as large as the document itself, this doesn't paint a pretty resource usage picture. If they had to first be persisted entirely in memory before being returned to the client, even moderately large FOR XML result sets could use huge amounts of virtual memory (or run into the MSXML memory ceiling and therefore be too large to generate).

To answer the second question, let's again have a look at SQL Server under a debugger.

Exercise 18.2 Determining Whether Server-Side FOR XML Uses MSXML

1. Restart your SQL Server, preferably from a console since we will be attaching to it with WinDbg. This should be a test or development system, and, ideally, you should be its only user.
2. Start Query Analyzer and connect to your SQL Server.
3. Attach to SQL Server using WinDbg. (Press F6 and select sqlservr.exe from the list of running tasks; if you have multiple instances, be sure to select the right one.) Once the WinDbg command prompt appears, type g and press Enter so that SQL Server can continue to run.
4. Back in Query Analyzer, run a FOR XML query of some type:

```
SELECT * FROM (
SELECT 'Summer Dream' as Song
UNION
SELECT 'Summer Snow'
UNION
SELECT 'Crazy For You'
) s FOR XML AUTO
```

   This query unions some SELECT statements together, then queries the union as a derived table using a FOR XML clause.

5. After you run the query, switch back to WinDbg. You will likely see some ModLoad messages in the WinDbg command window. WinDbg displays a ModLoad message whenever a module is loaded into the process being debugged. If MSXML*n*.DLL were being used to service your FOR XML query, you'd see a ModLoad message for it. As you've noticed, there isn't one. MSXML isn't used to service FOR XML queries.
6. If you've done much debugging, you may be speculating that perhaps the MSXML DLL is already loaded; hence, we wouldn't see a ModLoad message for it when we ran our FOR XML query. That's easy enough to check. Hit Ctrl+Break in the debugger, then type lm in the command window and hit Enter. The lm command lists the modules currently loaded into the process space. Do you see MSXML*n*.DLL in the list? Unless you've been interacting with SQL Server's other XML features since you recycled your server, it should not be there. Type g in the command window and press Enter so that SQL Server can continue to run.
7. As a final test, let's force MSXML*n*.DLL to load by parsing an XML document. Reload the query from Exercise 18.1 above in Query Analyzer and run it. You should see a ModLoad message for MSXML's DLL in the WinDbg command window.
8. Hit Ctrl+Break again to stop WinDbg, then type q and hit Enter to stop debugging. You will need to restart your SQL Server.

So, based on all this, we can conclude that SQL Server generates its own XML when it processes a server-side FOR XML query. There is no memory-efficient mechanism in MSXML to assist with this, so it is not used.

Using FOR XML

As you saw in Exercise 18.2, you can append FOR XML AUTO to the end of a SELECT statement in order to cause the result to be returned as an XML document fragment. Transact-SQL's FOR XML syntax is much richer than this, though—it supports several options that extend its usefulness in numerous ways. In this section, we'll discuss a few of these and work through examples that illustrate them.

SELECT...FOR XML (Server-Side)

As I'm sure you've already surmised, you can retrieve XML data from SQL Server by using the FOR XML option of the SELECT command. FOR XML causes SELECT to return query results as an XML stream rather

than a traditional rowset. On the server-side, this stream can have one of three formats: RAW, AUTO, or EXPLICIT. The basic FOR XML syntax looks like this:

```
SELECT column list
FROM table list
WHERE filter criteria
FOR XML RAW | AUTO | EXPLICIT [, XMLDATA] [, ELEMENTS]
    [, BINARY BASE64]
```

RAW returns column values as attributes and wraps each row in a generic row element. AUTO returns column values as attributes and wraps each row in an element named after the table from which it came.[1] EXPLICIT lets you completely control the format of the XML returned by a query.

XMLDATA causes an XML-Data schema to be returned for the document being retrieved. ELEMENTS causes the columns in XML AUTO data to be returned as elements rather than attributes. BINARY BASE64 specifies that binary data is to be returned using BASE64 encoding.

I'll discuss these options in more detail in just a moment. Also note that there are client-side specific options available with FOR XML queries that aren't available in server-side queries. We'll talk about those in just a moment, too.

RAW Mode

RAW mode is the simplest of the three basic FOR XML modes. It performs a very basic translation of the result set into XML. Listing 18.3 shows an example.

Listing 18.3

```
SELECT CustomerId, CompanyName
FROM Customers FOR XML RAW
```

(Results abridged)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
```

1. There's actually more to this than simply naming each row after the table, view, or UDF that produced it. SQL Server uses a set of heuristics to decide what the actual element names are with FOR XML AUTO.

```
--------------------------------------------------------------------
<row CustomerId="ALFKI" CompanyName="Alfreds Futterkiste"/><row Cu
CompanyName="Ana Trujillo Emparedados y helados"/><row CustomerId=
CompanyName="Antonio Moreno Taquería"/><row CustomerId="AROUT" Com
Horn"/><row CustomerId="BERGS" CompanyName="Berglunds snabbköp"/><
CustomerId="BLAUS" CompanyName="Blauer See Delikatessen"/><row Cus
CompanyName="Blondesddsl p_re et fils"/><row CustomerId="WELLI"
CompanyName="Wellington Importadora"/><row CustomerId="WHITC" Comp
Clover Markets"/><row CustomerId="WILMK" CompanyName="Wilman Kala"
CustomerId="WOLZA"
CompanyName="Wolski Zajazd"/>
```

Each column becomes an attribute in the result set, and each row becomes an element with the generic name of row.

As I've mentioned before, the XML that's returned by FOR XML is not well formed because it lacks a root element. It's technically an XML fragment and must include a root element in order to be usable by an XML parser. From the client side, you can set an ADO Command object's xml root property in order to automatically generate a root node when you execute a FOR XML query.

AUTO Mode

FOR XML AUTO gives you more control than RAW mode over the XML fragment that's produced. To begin with, each row in the result set is named after the table, view, or table-valued UDF that produced it. For example, Listing 18.4 shows a basic FOR XML AUTO query.

Listing 18.4

```
SELECT CustomerId, CompanyName
FROM Customers FOR XML AUTO
```

(Results abridged)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
--------------------------------------------------------------------
<Customers CustomerId="ALFKI" CompanyName="Alfreds Futterkiste"/><
CustomerId="ANATR" CompanyName="Ana Trujillo Emparedados y helados
CustomerId="ANTON" CompanyName="Antonio Moreno Taquería"/><Custome
```

```
CustomerId="AROUT" CompanyName="Around the Horn"/><Customers Custo
CompanyName="Vins et alcools Chevalier"/><Customers CustomerId="WA
CompanyName="Wartian Herkku"/><Customers CustomerId="WELLI" Compan
Importadora"/><Customers CustomerId="WHITC" CompanyName="White Clo
Markets"/><Customers CustomerId="WILMK" CompanyName="Wilman Kala"/
CustomerId="WOLZA"
CompanyName="Wolski Zajazd"/>
```

Notice that each row is named after the table from whence it came: Customers. For results with more than one row, this amounts to having more than one top-level (root) element in the fragment, which isn't allowed in XML.

One big difference between AUTO and RAW mode is the way in which joins are handled. In RAW mode, a simple one-to-one translation occurs between columns in the result set and attributes in the XML fragment. Each row becomes an element in the fragment named row. These elements are technically empty themselves—they contain no values or subelements, only attributes. Think of attributes as specifying characteristics of an element, while data and subelements compose its contents. In AUTO mode, each row is named after the source from which it came, and the rows from joined tables are nested within one another. Listing 18.5 presents an example.

Listing 18.5

```
SELECT Customers.CustomerID, CompanyName, OrderId
FROM Customers JOIN Orders
ON (Customers.CustomerId=Orders.CustomerId)
FOR XML AUTO
```

(Results abridged and formatted)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
--------------------------------------------------------------------
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Orders OrderId="10643"/><Orders OrderId="10692"/>
  <Orders OrderId="10702"/><Orders OrderId="10835"/>
  <Orders OrderId="10952"/><Orders OrderId="11011"/>
</Customers>
<Customers CustomerID="ANATR" CompanyName="Ana Trujillo Emparedado
  <Orders OrderId="10308"/><Orders OrderId="10625"/>
  <Orders OrderId="10759"/><Orders OrderId="10926"/></Customers>
```

```
<Customers CustomerID="FRANR" CompanyName="France restauration">
  <Orders OrderId="10671"/><Orders OrderId="10860"/>
  <Orders OrderId="10971"/>
</Customers>
```

I've formatted the XML fragment to make it easier to read—if you run the query yourself from Query Analyzer, you'll see an unformatted stream of XML text.

Note the way in which the Orders for each customer are contained within each Customer element. As I said, AUTO mode nests the rows returned by joins. Note my use of the full table name in the join criterion. Why didn't I use a table alias? Because AUTO mode uses the table aliases you specify to name the elements it returns. If you use shortened monikers for a table, its elements will have that name in the resulting XML fragment. While useful in traditional Transact-SQL, this makes the fragment difficult to read if the alias isn't sufficiently descriptive.

ELEMENTS Option

The ELEMENTS option of the FOR XML AUTO clause causes AUTO mode to return nested elements instead of attributes. Depending on your business needs, element-centric mapping may be preferable to the default attribute-centric mapping. Listing 18.6 gives an example of a FOR XML query that returns elements instead of attributes.

Listing 18.6

```
SELECT CustomerID, CompanyName
FROM Customers
FOR XML AUTO, ELEMENTS
```

(Results abridged and formatted)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
--------------------------------------------------------------------
<Customers>
  <CustomerID>ALFKI</CustomerID>
  <CompanyName>Alfreds Futterkiste</CompanyName>
</Customers>
```

```
<Customers>
  <CustomerID>ANATR</CustomerID>
  <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
</Customers>
<Customers>
  <CustomerID>ANTON</CustomerID>
  <CompanyName>Antonio Moreno Taquería</CompanyName>
</Customers>
<Customers>
  <CustomerID>AROUT</CustomerID>
  <CompanyName>Around the Horn</CompanyName>
</Customers>
<Customers>
  <CustomerID>WILMK</CustomerID>
  <CompanyName>Wilman Kala</CompanyName>
</Customers>
<Customers>
  <CustomerID>WOLZA</CustomerID>
  <CompanyName>Wolski  Zajazd</CompanyName>
</Customers>
```

Notice that the ELEMENTS option has caused what were being returned as attributes of the Customers element to instead be returned as subelements. Each attribute is now a pair of element tags that enclose the value from a column in the table.

NOTE: Currently, AUTO mode does not support GROUP BY or aggregate functions. The heuristics it uses to determine element names are incompatible with these constructs, so you cannot use them in AUTO mode queries. Additionally, FOR XML itself is incompatible with COMPUTE, so you can't use it in FOR XML queries of any kind.

EXPLICIT Mode

If you need more control over the XML than FOR XML produces, EXPLICIT mode is more flexible (and therefore more complicated to use) than either RAW mode or AUTO mode. EXPLICIT mode queries define XML documents in terms of a "universal table"—a mechanism for returning a result set from SQL Server that *describes* what you want the document to look like, rather than composing the document itself. A universal table is just a

SQL Server result set with special column headings that tell the server how to produce an XML document from your data. Think of it as a set-oriented method of making an API call and passing parameters to it. You use the facilities available in Transact-SQL to make the call and pass it parameters.

A universal table consists of one column for each table column that you want to return in the XML fragment, plus two additional columns: Tag and Parent. Tag is a positive integer that uniquely identifies each tag that is to be returned by the document; Parent establishes parent-child relationships between tags.

The other columns in a universal table—the ones that correspond to the data you want to include in the XML fragment—have special names that actually consist of multiple segments delimited by exclamation points (!). These special column names pass muster with SQL Server's parser and provide specific instructions regarding the XML fragment to produce. They have the following format:

```
Element!Tag!Attribute!Directive
```

We'll see some examples of these shortly.

The first thing you need to do to build an EXPLICIT mode query is to determine the layout of the XML document you want to end up with. Once you know this, you can work backward from there to build a universal table that will produce the desired format. For example, let's say we want a simple customer list based on the Northwind Customers table that returns the customer ID as an attribute and the company name as an element. The XML fragment we're after might look like this:

```
<Customers CustomerId="ALFKI">Alfreds Futterkiste</Customers>
```

Listing 18.7 shows a Transact-SQL query that returns a universal table that specifies this layout.

Listing 18.7

```
SELECT 1 AS Tag,
NULL AS Parent,
CustomerId AS [Customers!1!CustomerId],
CompanyName AS [Customers!1]
FROM Customers
```

(Results abridged)

```
Tag    Parent   Customers!1!CustomerId Customers!1
------ -------- ---------------------- --------------------------
1      NULL     ALFKI                   Alfreds Futterkiste
1      NULL     ANATR                   Ana Trujillo Emparedados y
1      NULL     ANTON                   Antonio Moreno Taquería
```

The first two columns are the extra columns I mentioned earlier. Tag specifies an identifier for the tag we want to produce. Since we want to produce only one element per row, we hard-code this to 1. The same is true of Parent—there's only one element and a top-level element doesn't have a parent, so we return NULL for Parent in every row.

Since we want to return the customer ID as an attribute, we specify an attribute name in the heading of column 3 (bolded). And since we want to return CompanyName as an element rather than an attribute, we omit the attribute name in column 4.

By itself, this table accomplishes nothing. We have to add FOR XML EXPLICIT to the end of it in order for the odd column names to have any special meaning. Add FOR XML EXPLICIT to the query and run it from Query Analyzer. Listing 18.8 shows what you should see.

Listing 18.8

```
SELECT 1 AS Tag,
NULL AS Parent,
CustomerId AS [Customers!1!CustomerId],
CompanyName AS [Customers!1]
FROM Customers
FOR XML EXPLICIT
```

(Results abridged and formatted)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
----------------------------------------------------------------
<Customers CustomerId="ALFKI">Alfreds Futterkiste</Customers>
<Customers CustomerId="ANATR">Ana Trujillo Emparedados y helados
  </Customers>
<Customers CustomerId="WHITC">White Clover Markets</Customers>
<Customers CustomerId="WILMK">Wilman Kala</Customers>
<Customers CustomerId="WOLZA">Wolski Zajazd</Customers>
```

Table 18.2 EXPLICIT Mode Directives

Value	Function
element	Causes data in the column to be encoded and represented as a subelement
xml	Causes data to be represented as a subelement without encoding it
xmltext	Retrieves data from an overflow column and appends it to the document
cdata	Causes data in the column to be represented as a CDATA section in the resulting document
hide	Hides (omits) a column that appears in the universal table from the resulting XML fragment
id, idref, and idrefs	In conjunction with XMLDATA, can establish relationships between elements across multiple XML fragments

As you can see, each CustomerId value is returned as an attribute, and each CompanyName is returned as the element data for the Customers element, just as we specified.

Directives

The fourth part of the multivalued column headings supported by EXPLICIT mode queries is the directive segment. You use it to further control how data is represented in the resulting XML fragment. As Table 18.2 illustrates, the directive segment supports eight values.

Of these, element is the most frequently used. It causes data to be rendered as a subelement rather than an attribute. For example, let's say that, in addition to CustomerId and CompanyName, we wanted to return ContactName in our XML fragment and we wanted it to be a subelement rather than an attribute. Listing 18.9 shows how the query would look.

Listing 18.9

```
SELECT 1 AS Tag,
NULL AS Parent,
CustomerId AS [Customers!1!CustomerId],
```

```
CompanyName AS [Customers!1],
ContactName AS [Customers!1!ContactName!element]
FROM Customers
FOR XML EXPLICIT
```

(Results abridged and formatted)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
--------------------------------------------------------------------
<Customers CustomerId="ALFKI">Alfreds Futterkiste
  <ContactName>Maria Anders</ContactName>
</Customers>
<Customers CustomerId="ANATR">Ana Trujillo Emparedados y
  <ContactName>Ana Trujillo</ContactName>
</Customers>
<Customers CustomerId="ANTON">Antonio Moreno Taquería
  <ContactName>Antonio Moreno</ContactName>
</Customers>
<Customers CustomerId="AROUT">Around the Horn
  <ContactName>Thomas Hardy</ContactName>
</Customers>
<Customers CustomerId="BERGS">Berglunds snabbköp
  <ContactName>Christina Berglund</ContactName>
</Customers>
<Customers CustomerId="WILMK">Wilman Kala
  <ContactName>Matti Karttunen</ContactName>
</Customers>
<Customers CustomerId="WOLZA">Wolski  Zajazd
  <ContactName>Zbyszek Piestrzeniewicz</ContactName>
</Customers>
```

As you can see, ContactName is nested within each Customers element as a subelement. The elements directive encodes the data it returns. We can retrieve the same data by using the xml directive without encoding, as shown in Listing 18.10.

Listing 18.10

```
SELECT 1 AS Tag,
NULL AS Parent,
CustomerId AS [Customers!1!CustomerId],
CompanyName AS [Customers!1],
```

```
ContactName AS [Customers!1!ContactName!xml]
FROM Customers
FOR XML EXPLICIT
```

The xml directive (bolded) causes the column to be returned without encoding any special characters it contains.

Establishing Data Relationships

Thus far, we've been listing the data from a single table, so our EXPLICT queries haven't been terribly complex. That would still be true even if we queried multiple tables as long as we didn't mind repeating the data from each table in each top-level element in the XML fragment. Just as the column values from joined tables are often repeated in the result sets of Transact-SQL queries, we could create an XML fragment that contained data from multiple tables repeated in each element. However, that wouldn't be the most efficient way to represent the data in XML. Remember: XML supports hierarchical relationships between elements. You can establish these hierarchies by using EXPLICIT mode queries and T-SQL UNIONs. Listing 18.11 provides an example.

Listing 18.11

```
SELECT 1 AS Tag,
NULL AS Parent,
CustomerId AS [Customers!1!CustomerId],
CompanyName AS [Customers!1],
NULL AS [Orders!2!OrderId],
NULL AS [Orders!2!OrderDate!element]
FROM Customers
UNION
SELECT 2 AS Tag,
1 AS Parent,
CustomerId,
NULL,
OrderId,
OrderDate
FROM Orders
ORDER BY [Customers!1!CustomerId], [Orders!2!OrderDate!element]
FOR XML EXPLICIT
```

This query does several interesting things. First, it links the Customers and Orders tables using the CustomerId column they share. Notice the third column in each SELECT statement—it returns the CustomerId column from each table. The Tag and Parent columns establish the details of the relationship between the two tables. The Tag and Parent values in the second query link it to the first. They establish that Order records are children of Customer records. Lastly, note the ORDER BY clause. It arranges the elements in the table in a sensible fashion—first by CustomerId and second by the OrderDate of each Order. Listing 18.12 shows the result set.

Listing 18.12

(Results abridged and formatted)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
----------------------------------------------------------------
<Customers CustomerId="ALFKI">Alfreds Futterkiste
  <Orders OrderId="10643">
    <OrderDate>1997-08-25T00:00:00</OrderDate>
  </Orders>
  <Orders OrderId="10692">
    <OrderDate>1997-10-03T00:00:00</OrderDate>
  </Orders>
  <Orders OrderId="10702">
    <OrderDate>1997-10-13T00:00:00</OrderDate>
  </Orders>
  <Orders OrderId="10835">
    <OrderDate>1998-01-15T00:00:00</OrderDate>
  </Orders>
  <Orders OrderId="10952">
    <OrderDate>1998-03-16T00:00:00</OrderDate>
  </Orders>
  <Orders OrderId="11011">
    <OrderDate>1998-04-09T00:00:00</OrderDate>
  </Orders>
</Customers>
<Customers CustomerId="ANATR">Ana Trujillo Emparedados y helados
  <Orders OrderId="10308">
    <OrderDate>1996-09-18T00:00:00</OrderDate>
  </Orders>
  <Orders OrderId="10625">
    <OrderDate>1997-08-08T00:00:00</OrderDate>
```

```
  </Orders>
  <Orders OrderId="10759">
    <OrderDate>1997-11-28T00:00:00</OrderDate>
  </Orders>
  <Orders OrderId="10926">
    <OrderDate>1998-03-04T00:00:00</OrderDate>
  </Orders>
</Customers>
```

As you can see, each customer's orders are nested within its element.

The hide Directive

The hide directive omits a column you've included in the universal table from the resulting XML document. One use of this functionality is to order the result by a column that you don't want to include in the XML fragment. When you aren't using UNION to merge tables, this isn't a problem because you can order by any column you choose. However, the presence of UNION in a query requires order by columns to exist in the result set. The hide directive gives you a way to satisfy this requirement without being forced to return data you don't want to. Listing 18.13 shows an example.

Listing 18.13

```
SELECT 1 AS Tag,
NULL AS Parent,
CustomerId AS [Customers!1!CustomerId],
CompanyName AS [Customers!1],
PostalCode AS [Customers!1!PostalCode!hide],
NULL AS [Orders!2!OrderId],
NULL AS [Orders!2!OrderDate!element]
FROM Customers
UNION
SELECT 2 AS Tag,
1 AS Parent,
CustomerId,
NULL,
NULL,
OrderId,
OrderDate
FROM Orders
```

```
ORDER BY [Customers!1!CustomerId], [Orders!2!OrderDate!element],
[Customers!1!PostalCode!hide]
FOR XML EXPLICIT
```

Notice the hide directive (bolded) that's included in the column 5 heading. It allows the column to be specified in the ORDER BY clause without actually appearing in the resulting XML fragment.

The cdata Directive

CDATA sections may appear anywhere in an XML document that character data may appear. A CDATA section is used to escape characters that would otherwise be recognized as markup (e.g., <, >, /, and so on). Thus CDATA sections allow you to include sections in an XML document that might otherwise confuse the parser. To render a CDATA section from an EXPLICIT mode query, include the cdata directive, as demonstrated in Listing 18.14.

Listing 18.14

```
SELECT 1 AS Tag,
NULL AS Parent,
CustomerId AS [Customers!1!CustomerId],
CompanyName AS [Customers!1],
Fax AS [Customers!1!!cdata]
FROM Customers
FOR XML EXPLICIT
```

(Results abridged and formatted)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
--------------------------------------------------------------------
<Customers CustomerId="ALFKI">Alfreds Futterkiste
  <![CDATA[030-0076545]]>
</Customers>
<Customers CustomerId="ANATR">Ana Trujillo Emparedados y helados
  <![CDATA[(5) 555-3745]]>
</Customers>
<Customers CustomerId="ANTON">Antonio Moreno Taquería
</Customers>
<Customers CustomerId="AROUT">Around the Horn
  <![CDATA[(171) 555-6750]]>
```

```
</Customers>
<Customers CustomerId="BERGS">Berglunds snabbköp
  <![CDATA[0921-12 34 67]]>
</Customers>
```

As you can see, each value in the Fax column is returned as a CDATA section in the XML fragment. Note the omission of the attribute name in the cdata column heading (bolded). This is because attribute names aren't allowed for CDATA sections. Again, they represent escaped document segments, so the XML parser doesn't process any attribute or element names they may contain.

The id, idref, and idrefs Directives

The ID, IDREF, and IDFREFS data types can be used to represent relational data in an XML document. Set up in a DTD or XML-Data schema, they establish relationships between elements. They're handy in situations where you need to exchange complex data and want to minimize the amount of data duplication in the document.

EXPLICIT mode queries can use the id, idref, and idrefs directives to specify relational fields in an XML document. Naturally, this approach works only if a schema is used to define the document and identify the columns used to establish links between entities. FOR XML's XMLDATA option provides a means of generating an inline schema for its XML fragment. In conjunction with the id directives, it can identify relational fields in the XML fragment. Listing 18.15 gives an example.

Listing 18.15

```
SELECT 1 AS Tag,
       NULL AS Parent,
       CustomerId AS [Customers!1!CustomerId!id],
       CompanyName AS [Customers!1!CompanyName],
       NULL AS [Orders!2!OrderID],
       NULL AS [Orders!2!CustomerId!idref]
FROM Customers
UNION
SELECT 2,
       NULL,
       NULL,
       NULL,
```

```
       OrderID,
       CustomerId
FROM Orders
ORDER BY [Orders!2!OrderID]
FOR XML EXPLICIT, XMLDATA
```

(Results abridged and formatted)

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
--------------------------------------------------------------------
<Schema name="Schema2" xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <ElementType name="Customers" content="mixed" model="open">
    <AttributeType name="CustomerId" dt:type="id"/>
    <AttributeType name="CompanyName" dt:type="string"/>
    <attribute type="CustomerId"/>
    <attribute type="CompanyName"/>
  </ElementType>
  <ElementType name="Orders" content="mixed" model="open">
    <AttributeType name="OrderID" dt:type="i4"/>
    <AttributeType name="CustomerId" dt:type="idref"/>
    <attribute type="OrderID"/>
    <attribute type="CustomerId"/>
  </ElementType>
</Schema>
<Customers xmlns="x-schema:#Schema2" CustomerId="ALFKI"
  CompanyName="Alfreds Futterkiste"/>
<Customers xmlns="x-schema:#Schema2" CustomerId="ANATR"
  CompanyName="Ana Trujillo Emparedados y helados"/>
<Customers xmlns="x-schema:#Schema2" CustomerId="ANTON"
  CompanyName="Antonio Moreno Taquería"/>
<Customers xmlns="x-schema:#Schema2" CustomerId="AROUT"
  CompanyName="Around the Horn"/>
<Orders xmlns="x-schema:#Schema2" OrderID="10248"
    CustomerId="VINET"/>
<Orders xmlns="x-schema:#Schema2" OrderID="10249"
    CustomerId="TOMSP"/>
<Orders xmlns="x-schema:#Schema2" OrderID="10250"
    CustomerId="HANAR"/>
<Orders xmlns="x-schema:#Schema2" OrderID="10251"
    CustomerId="VICTE"/>
<Orders xmlns="x-schema:#Schema2" OrderID="10252"
    CustomerId="SUPRD"/>
<Orders xmlns="x-schema:#Schema2" OrderID="10253"
    CustomerId="HANAR"/>
```

```
<Orders xmlns="x-schema:#Schema2" OrderID="10254"
    CustomerId="CHOPS"/>
<Orders xmlns="x-schema:#Schema2" OrderID="10255"
    CustomerId="RICSU"/>
```

Note the use of the id and idref directives in the CustomerId columns of the Customers and Orders tables (bolded). These directives link the two tables by using the CustomerId column they share.

If you examine the XML fragment returned by the query, you'll see that it starts off with the XML-Data schema that the XMLDATA directive created. This schema is then referenced in the XML fragment that follows.

SELECT...FOR XML (Client-Side)

SQLXML also supports the notion of offloading to the client the work of translating a result set into XML. This functionality is accessible via the SQLXML managed classes, XML templates, a virtual directory configuration switch, and the SQLXMLOLEDB provider. Because it requires the least amount of setup, I'll cover client-side FOR XML using SQLXMLOLEDB here. The underlying technology is the same regardless of the mechanism used.

SQLXMLOLEDB serves as a layer between a client (or middle-tier) app and SQL Server's native SQLOLEDB provider. The Data Source property of the SQLXMLOLEDB provider specifies the OLE DB provider through which it executes queries; currently only SQLOLEDB is allowed.

SQLXMLOLEDB is not a rowset provider. In order to use it from ADO, you must access it via ADO's stream mode. I'll show you some code in just a minute that illustrates this.

You perform client-side FOR XML processing using SQLXMLOLEDB by following these general steps.

1. Connect using an ADO connection string that specifies SQLXMLOLEDB as the provider.
2. Set the ClientSideXML property of your ADO Command object to True.
3. Create and open an ADO stream object and associate it with your Command object's Output Stream property.
4. Execute a FOR XML EXPLICIT, FOR XML RAW, or FOR XML NESTED Transact-SQL query via your Command object, specifying the adExecuteStream option in your call to Execute.

Listing 18.16 illustrates. (You can find the source code for this app in the CH18\forxml_clientside subfolder on this book's CD.)

Listing 18.16

```
Private Sub Command1_Click()
  Dim oConn As New ADODB.Connection
  Dim oComm As New ADODB.Command

  Dim stOutput As New ADODB.Stream
  stOutput.Open

  oConn.Open (Text3.Text)
  oComm.ActiveConnection = oConn
  oComm.Properties("ClientSideXML") = "True"
  If Len(Text1.Text) = 0 Then
    Text1.Text = _
      "select * from pubs..authors FOR XML NESTED"
  End If
  oComm.CommandText = Text1.Text
  oComm.Properties("Output Stream") = stOutput
  oComm.Properties("xml root") = "Root"
  oComm.Execute , , adExecuteStream

  Text2.Text = stOutput.ReadText(adReadAll)

  stOutput.Close
  oConn.Close

  Set oComm = Nothing
  Set oConn = Nothing
End Sub
```

As you can see, most of the action here revolves around the ADO Command object. We set its ClientSideXML property to True and its Output Stream property to an ADO stream object we created before callings its Execute method.

Note the use of the FOR XML NESTED clause. The NESTED option is specific to client-side FOR XML processing—you can't use it in server-side queries. It's very much like FOR XML AUTO but has some minor differences. For example, when a FOR XML NESTED query references a

view, the names of the view's underlying base tables are used in the generated XML. The same is true for table aliases—their base names are used in the XML that's produced. Using FOR XML AUTO in a client-side FOR XML query causes the query to be processed on the server rather than the client, so use NESTED when you want similar functionality to FOR XML AUTO on the client.

Given our previous investigation into whether MSXML is involved in the production of server-side XML (Exercise 18.2), you might be wondering whether it's used by SQLXML's client-side FOR XML processing. It isn't. Again, you can attach a debugger (in this case, to the forxml_clientside app) to see this for yourself. You *will* see SQLXML*n*.DLL loaded into the app's process space the first time you run the query. This DLL is where the SQLXMLOLEDB provider resides and is where SQLXML's client-side FOR XML processing occurs.

OPENXML

OPENXML is a built-in Transact-SQL function that can return an XML document as a rowset. In conjunction with sp_xml_preparedocument and sp_xml_removedocument, OPENXML allows you to break down (or shred) nonrelational XML documents into relational pieces that can be inserted into tables.

I suppose we should begin the investigation of how OPENXML works by determining where it's implemented. Does it reside in a separate DLL (SQLXML*n*.DLL, perhaps?) or is it implemented completely within the SQL Server executable?

The most expedient way to determine this is to run SQL Server under a debugger, stop it in the middle of an OPENXML call, and inspect the call stack. That would tell us in what module it was implemented. Since we don't know the name of the classes or functions that implement OPENXML, we can't easily set a breakpoint to accomplish this. Instead, we will have to just be quick and/or lucky enough to stop the debugger in the right place if we want to use this approach to find out the module in which OPENXML is implemented. This is really easier said than done. Even with complicated documents, OPENXML returns fairly quickly, so breaking in with a debugger while it's in progress could prove pretty elusive.

Another way to accomplish the same thing would be to force OPENXML to error and have a breakpoint set up in advance to stop in SQL Server's standard error reporting routine. From years of working with the product and

seeing my share of access violations and stack dumps, I know that ex_raise is a central error-reporting routine for the server. Not all errors go through ex_raise, but many of them do, so it's worth setting a breakpoint in ex_raise and forcing OPENXML to error to see whether we can get a call stack and ascertain where OPENXML is implemented. Exercise 18.3 will take you through the process of doing exactly that.

Exercise 18.3 Determining Where OPENXML Is Implemented

1. Restart your SQL Server, preferably from a console since we will be attaching to it with WinDbg. This should be a test or development system, and, ideally, you should be its only user.
2. Start Query Analyzer and connect to your SQL Server.
3. Attach to SQL Server using WinDbg. (Press F6 and select sqlservr.exe from the list of running tasks; if you have multiple instances, be sure to select the right one.)
4. Once the WinDbg command prompt appears, set a breakpoint in ex_raise:

```
bp sqlservr!ex_raise
```

5. Type g and press Enter so that SQL Server can continue to run.
6. Back in Query Analyzer, run this query:

```
declare @hDoc int
set @hdoc=8675309  -- force a bogus handle
select * from openxml(@hdoc,'/',1)
```

7. Query Analyzer should appear to hang because the breakpoint you set in WinDbg has been hit. Switch back to WinDbg and type kv at the command prompt and press Enter. This will dump the call stack. Your stack should look something like this (I've removed everything but the function names):

```
sqlservr!ex_raise
sqlservr!CXMLDocsList::XMLMapFromHandle+0x3f
sqlservr!COpenXMLRange::GetRowset+0x14d
sqlservr!CQScanRmtScan::OpenConnection+0x141
sqlservr!CQScanRmtBase::Open+0x18
sqlservr!CQueryScan::Startup+0x10d
sqlservr!CStmtQuery::ErsqExecuteQuery+0x26b
sqlservr!CStmtSelect::XretExecute+0x229
sqlservr!CMsqlExecContext::ExecuteStmts+0x3b9
sqlservr!CMsqlExecContext::Execute+0x1b6
sqlservr!CSQLSource::Execute+0x357
sqlservr!language_exec+0x3e1
```

```
sqlservr!process_commands+0x10e
UMS!ProcessWorkRequests+0x272
UMS!ThreadStartRoutine+0x98 (FPO: [EBP 0x00bd6878] [1,0,4])
MSVCRT!_beginthread+0xce
KERNEL32!BaseThreadStart+0x52 (FPO: [Non-Fpo])
```

8. This call stack tells us a couple of things. First, it tells us that OPENXML is implemented directly by the server itself. It resides in sqlservr.exe, SQL Server's executable. Second, it tells us that a class named COpenXMLRange is responsible for producing the rowset that the T-SQL OPENXML function returns.
9. Type q and hit Enter to stop debugging. You will need to restart your SQL Server.

By reviewing this call stack, we can deduce how OPENXML works. It comes into the server via a language or RPC event (our code obviously came into the server as a language event—note the language_exec entry in the call stack) and eventually results in a call to the GetRowset method of the COpenXML-Range class. We can assume that GetRowset accesses the DOM document previously created via the call to sp_xml_preparedocument and turns it into a two-dimensional matrix that can be returned as a rowset, thus finishing up the work of the OPENXML function.

Now that we know the name of the class and method behind OPENXML, we could set a new breakpoint in COpenXMLRange::GetRowset, pass a valid document handle into OPENXML, and step through the disassembly for the method when the breakpoint is hit. However, we've got a pretty good idea of how OPENXML works; there's little to be learned about OPENXML's architecture from stepping through the disassembly at this point.

Using OPENXML

Books Online documents how to use OPENXML pretty well, so I'll try not to repeat that information here. Listing 18.17 shows a basic example of how to use OPENXML.

Listing 18.17

```
DECLARE @hDoc int
EXEC sp_xml_preparedocument @hDoc output,
'<songs>
  <song><name>Somebody to Love</name></song>
```

```
  <song><name>These Are the Days of Our Lives</name></song>
  <song><name>Bicycle Race</name></song>
  <song><name>Who Wants to Live Forever</name></song>
  <song><name>I Want to Break Free</name></song>
  <song><name>Friends Will Be Friends</name></song>
</songs>'
SELECT * FROM OPENXML(@hdoc, '/songs/song', 2) WITH
    (name varchar(80))
EXEC sp_xml_removedocument @hDoc
```

(Results)

```
name
----------------------------------------------------------------
Somebody to Love
These Are the Days of Our Lives
Bicycle Race
Who Wants to Live Forever
I Want to Break Free
Friends Will Be Friends
```

To use OPENXML, follow these basic steps.

1. Call sp_xml_preparedocument to load the XML document into memory. MSXML's DOM parser is called to translate the document into a tree of nodes that you can then access with an XPath query. A pointer to this tree is returned by the procedure as an integer.
2. Issue a SELECT statement from OPENXML, passing in the handle you received in step 1.
3. Include XPath syntax in the call to OPENXML in order to specify exactly which nodes you want to access.
4. Optionally include a WITH clause that maps the XML document into a specific table schema. This can be a full table schema as well as a reference to a table itself.

OPENXML is extremely flexible, so several of these steps have variations and alternatives, but this is the basic process you follow to shred and use an XML document with OPENXML.

Listing 18.18 presents a variation of the earlier query that employs a table to define the schema used to map the document.

Listing 18.18

```
USE tempdb
GO
create table songs (name varchar(80))
go
DECLARE @hDoc int
EXEC sp_xml_preparedocument @hDoc output,
'<songs>
  <song><name>Somebody to Love</name></song>
  <song><name>These Are the Days of Our Lives</name></song>
  <song><name>Bicycle Race</name></song>
  <song><name>Who Wants to Live Forever</name></song>
  <song><name>I Want to Break Free</name></song>
  <song><name>Friends Will Be Friends</name></song>
</songs>'
SELECT * FROM OPENXML(@hdoc, '/songs/song', 2) WITH songs
EXEC sp_xml_removedocument @hDoc
GO
DROP TABLE songs
```

(Results)

```
name
------------------------------------------------------------------
Somebody to Love
These Are the Days of Our Lives
Bicycle Race
Who Wants to Live Forever
I Want to Break Free
Friends Will Be Friends
```

You can also use the WITH clause to set up detailed mappings between the XML document and the tables in your database, as shown in Listing 18.19.

Listing 18.19

```
DECLARE @hDoc int
EXEC sp_xml_preparedocument @hDoc output,
'<songs>
  <artist name="Johnny Hartman">
  <song> <name>It Was Almost Like a Song</name></song>
```

```
  <song> <name>I See Your Face Before Me</name></song>
  <song> <name>For All We Know</name></song>
  <song> <name>Easy Living</name></song>
  </artist>
  <artist name="Harry Connick, Jr.">
  <song> <name>Sonny Cried</name></song>
  <song> <name>A Nightingale Sang in Berkeley Square</name></song>
  <song> <name>Heavenly</name></song>
  <song> <name>You Didn''t Know Me When</name></song>
  </artist>
</songs>'
SELECT * FROM OPENXML(@hdoc, '/songs/artist/song', 2)
WITH (artist varchar(30) '../@name',
      song varchar(50) 'name')
EXEC sp_xml_removedocument @hDoc
```

(Results)

```
artist                         song
------------------------------ ---------------------------------------
Johnny Hartman                 It Was Almost Like a Song
Johnny Hartman                 I See Your Face Before Me
Johnny Hartman                 For All We Know
Johnny Hartman                 Easy Living
Harry Connick, Jr.             Sonny Cried
Harry Connick, Jr.             A Nightingale Sang in Berkeley Square
Harry Connick, Jr.             Heavenly
Harry Connick, Jr.             You Didn't Know Me When
```

Note that attribute references are prefixed with the @ symbol. In Listing 18.19, we supply an XPath query that navigates the tree down to the song element, then reference an attribute called name in song's parent element, artist. For the second column, we retrieve a child element of song that's also called name.

Listing 18.20 offers another example.

Listing 18.20

```
DECLARE @hDoc int
EXEC sp_xml_preparedocument @hDoc output,
'<songs>
```

```
  <artist> <name>Johnny Hartman</name>
  <song> <name>It Was Almost Like a Song</name></song>
  <song> <name>I See Your Face Before Me</name></song>
  <song> <name>For All We Know</name></song>
  <song> <name>Easy Living</name></song>
  </artist>
  <artist> <name>Harry Connick, Jr.</name>
  <song> <name>Sonny Cried</name></song>
  <song> <name>A Nightingale Sang in Berkeley Square</name></song>
  <song> <name>Heavenly</name></song>
  <song> <name>You Didn''t Know Me When</name></song>
  </artist>
</songs>'
SELECT * FROM OPENXML(@hdoc, '/songs/artist/name', 2)
WITH (artist varchar(30) '.',
      song varchar(50) '../song/name')
EXEC sp_xml_removedocument @hDoc
```

(Results)

```
artist                         song
------------------------------ ---------------------------------------
Johnny Hartman                 It Was Almost Like a Song
Harry Connick, Jr.             Sonny Cried
```

Notice that we get only two rows. Why is that? It's due to the fact that our XPath pattern navigated to the artist/name node, of which there are only two. In addition to getting each artist's name element, we also grabbed the name of its first song element. In the previous query, the XPath pattern navigated us to the song element, of which there were eight, then referenced each song's parent node (its artist) via the XPath ".." designator.

Note the use in the above query of the XPath "." specifier. This merely references the current element. We need it here because we are changing the name of the current element from name to artist. Keep this technique in mind when you want to rename an element you're returning via OPENXML.

The flags Parameter

OPENXML's flags parameter allows you to specify whether OPENXML should process the document in an attribute-centric fashion, an element-

centric fashion, or some combination of the two. Thus far, we've been specifying 2 for the flags parameter, which specifies element-centric mapping. Listing 18.21 shows an example of attribute-centric mapping.

Listing 18.21

```
DECLARE @hDoc int
EXEC sp_xml_preparedocument @hDoc output,
'<songs>
  <artist name="Johnny Hartman">
  <song name="It Was Almost Like a Song"/>
  <song name="I See Your Face Before Me"/>
  <song name="For All We Know"/>
  <song name="Easy Living"/>
  </artist>
  <artist name="Harry Connick, Jr.">
  <song name="Sonny Cried"/>
  <song name="A Nightingale Sang in Berkeley Square"/>
  <song name="Heavenly"/>
  <song name="You Didn''t Know Me When"/>
  </artist>
</songs>'
SELECT * FROM OPENXML(@hdoc, '/songs/artist/song', 1)
WITH (artist varchar(30) '../@name',
      song varchar(50) '@name')
EXEC sp_xml_removedocument @hDoc
```

(Results)

```
artist                      song
--------------------------- -------------------------------------
Johnny Hartman              It Was Almost Like a Song
Johnny Hartman              I See Your Face Before Me
Johnny Hartman              For All We Know
Johnny Hartman              Easy Living
Harry Connick, Jr.          Sonny Cried
Harry Connick, Jr.          A Nightingale Sang in Berkeley Square
Harry Connick, Jr.          Heavenly
Harry Connick, Jr.          You Didn't Know Me When
```

Edge Table Format

You can completely omit OPENXML's WITH clause in order to retrieve a portion of an XML document in "edge table format"—essentially a two-dimensional representation of the XML tree. Listing 18.22 provides an example.

Listing 18.22

```
DECLARE @hDoc int
EXEC sp_xml_preparedocument @hDoc output,
'<songs>
  <artist name="Johnny Hartman">
  <song> <name>It Was Almost Like a Song</name></song>
  <song> <name>I See Your Face Before Me</name></song>
  <song> <name>For All We Know</name></song>
  <song> <name>Easy Living</name></song>
  </artist>
  <artist name="Harry Connick, Jr.">
  <song> <name>Sonny Cried</name></song>
  <song> <name>A Nightingale Sang in Berkeley Square</name></song>
  <song> <name>Heavenly</name></song>
  <song> <name>You Didn''t Know Me When</name></song>
  </artist>
</songs>'
SELECT * FROM OPENXML(@hdoc, '/songs/artist/song', 2)
EXEC sp_xml_removedocument @hDoc
```

(Results abridged)

```
id                  parentid            nodetype    localname
------------------- ------------------- ----------- -----------
4                   2                   1           song
5                   4                   1           name
22                  5                   3           #text
6                   2                   1           song
7                   6                   1           name
23                  7                   3           #text
8                   2                   1           song
9                   8                   1           name
24                  9                   3           #text
10                  2                   1           song
11                  10                  1           name
25                  11                  3           #text
14                  12                  1           song
```

```
15          14          1          name
26          15          3          #text
16          12          1          song
17          16          1          name
27          17          3          #text
18          12          1          song
19          18          1          name
28          19          3          #text
20          12          1          song
21          20          1          name
29          21          3          #text
```

Inserting Data with OPENXML

Given that it's a rowset function, it's natural that you'd want to insert the results of a SELECT against OPENXML into another table. There are a couple of ways to approach this. First, you could execute a separate pass against the XML document for each piece of it you wanted to extract. You would execute an INSERT…SELECT FROM OPENXML for each table you wanted to insert rows into, grabbing a different section of the XML document with each pass, as demonstrated in Listing 18.23.

Listing 18.23

```
USE tempdb
GO
CREATE TABLE Artists
(ArtistId varchar(5),
  Name varchar(30))
GO
CREATE TABLE Songs
(ArtistId varchar(5),
  SongId int,
  Name varchar(50))
GO

DECLARE @hDoc int
EXEC sp_xml_preparedocument @hDoc output,
'<songs>
  <artist id="JHART" name="Johnny Hartman">
  <song id="1" name="It Was Almost Like a Song"/>
  <song id="2" name="I See Your Face Before Me"/>
  <song id="3" name="For All We Know"/>
```

```
  <song id="4" name="Easy Living"/>
  </artist>
  <artist id="HCONN" name="Harry Connick, Jr.">
  <song id="1" name="Sonny Cried"/>
  <song id="2" name="A Nightingale Sang in Berkeley Square"/>
  <song id="3" name="Heavenly"/>
  <song id="4" name="You Didn''t Know Me When"/>
  </artist>
</songs>'
INSERT Artists (ArtistId, Name)
SELECT id,name
FROM OPENXML(@hdoc, '/songs/artist', 1)
WITH (id varchar(5) '@id',
      name varchar(30) '@name')

INSERT Songs (ArtistId, SongId, Name)
SELECT artistid, id,name
FROM OPENXML(@hdoc, '/songs/artist/song', 1)
WITH (artistid varchar(5) '../@id',
      id int '@id',
      name varchar(50) '@name')
EXEC sp_xml_removedocument @hDoc
GO
SELECT * FROM Artists
SELECT * FROM Songs
GO
DROP TABLE Artists, Songs
```

(Results)

```
ArtistId Name
-------- ------------------------------
JHART    Johnny Hartman
HCONN    Harry Connick, Jr.

ArtistId SongId      Name
-------- ----------- ---------------------------------------------
JHART    1           It Was Almost Like a Song
JHART    2           I See Your Face Before Me
JHART    3           For All We Know
JHART    4           Easy Living
HCONN    1           Sonny Cried
HCONN    2           A Nightingale Sang in Berkeley Square
HCONN    3           Heavenly
HCONN    4           You Didn't Know Me When
```

As you can see, we make a separate call to OPENXML for each table. The tables are normalized; the XML document is not, so we shred it into multiple tables. Listing 18.24 shows another way to accomplish the same thing that doesn't require multiple calls to OPENXML.

Listing 18.24

```
USE tempdb
GO
CREATE TABLE Artists
(ArtistId varchar(5),
  Name varchar(30))
GO
CREATE TABLE Songs
(ArtistId varchar(5),
  SongId int,
  Name varchar(50))
GO
CREATE VIEW ArtistSongs AS
SELECT a.ArtistId,
       a.Name AS ArtistName,
       s.SongId,
       s.Name as SongName
FROM Artists a JOIN Songs s
ON (a.ArtistId=s.ArtistId)
GO
CREATE TRIGGER ArtistSongsInsert ON ArtistSongs INSTEAD OF
    INSERT AS
INSERT Artists
SELECT DISTINCT ArtistId, ArtistName FROM inserted
INSERT Songs
SELECT ArtistId, SongId, SongName FROM inserted
GO

DECLARE @hDoc int
EXEC sp_xml_preparedocument @hDoc output,
'<songs>
  <artist id="JHART" name="Johnny Hartman">
  <song id="1" name="It Was Almost Like a Song"/>
  <song id="2" name="I See Your Face Before Me"/>
  <song id="3" name="For All We Know"/>
  <song id="4" name="Easy Living"/>
  </artist>
  <artist id="HCONN" name="Harry Connick, Jr.">
```

```
  <song id="1" name="Sonny Cried"/>
  <song id="2" name="A Nightingale Sang in Berkeley Square"/>
  <song id="3" name="Heavenly"/>
  <song id="4" name="You Didn''t Know Me When"/>
  </artist>
</songs>'
INSERT ArtistSongs (ArtistId, ArtistName, SongId, SongName)
SELECT artistid, artistname, songid, songname
FROM OPENXML(@hdoc, '/songs/artist/song', 1)
WITH (artistid varchar(5) '../@id',
      artistname varchar(30) '../@name',
      songid int '@id',
      songname varchar(50) '@name')

EXEC sp_xml_removedocument @hDoc
GO
SELECT * FROM Artists
SELECT * FROM Songs
GO
DROP VIEW ArtistSongs
GO
DROP TABLE Artists, Songs
```

(Results)

```
ArtistId Name
-------- ------------------------------
HCONN    Harry Connick, Jr.
JHART    Johnny Hartman

ArtistId SongId      Name
-------- ----------- ------------------------------------------
JHART    1           It Was Almost Like a Song
JHART    2           I See Your Face Before Me
JHART    3           For All We Know
JHART    4           Easy Living
HCONN    1           Sonny Cried
HCONN    2           A Nightingale Sang in Berkeley Square
HCONN    3           Heavenly
HCONN    4           You Didn't Know Me When
```

This technique uses a view and an INSTEAD OF trigger to alleviate the need for two passes with OPENXML. We use a view to simulate the denormalized layout of the XML document, then set up an INSTEAD OF trigger

to allow us to insert the data in the XML document "into" this view. The trigger performs the actual work of shredding, only it does so much more efficiently than calling OPENXML twice. It makes two passes over the logical inserted table and splits the columns contained therein (which mirror those of the view) into two separate tables.

Accessing SQL Server over HTTP

To get started accessing SQL Server via HTTP, you should set up an IIS virtual directory using the Configure IIS Support menu option in the SQLXML program folder. Of course, you can retrieve XML data from SQL Server without setting up a virtual directory (e.g., by using ADO or OLE DB); I'm referring exclusively to retrieving XML data from SQL Server via HTTP.

Configuring a virtual directory allows you to work with SQL Server's XML features via HTTP. You use a virtual directory to establish a link between a SQL Server database and a segment of a URL. It provides a navigation path from the root directory on your Web server to a database on your SQL Server.

SQL Server's ability to publish data over HTTP is made possible through SQLISAPI, an Internet Server API (ISAPI) extension that ships with the product. SQLISAPI uses SQLOLEDB, SQL Server's native OLE DB provider, to access the database associated with a virtual directory and return results to the client.

Client applications have four methods of requesting data from SQL Server over HTTP. These can be broken down into two broad types: those more suitable for private intranet access because of security concerns, and those safe to use on the public Internet.

Private Intranet

1. Post an XML query template to SQLISAPI.
2. Send a SELECT…FOR XML query string in a URL.

Public Internet

3. Specify a server-side XML schema in a virtual root.
4. Specify a server-side XML query template in a virtual root.

Due to their open-ended nature, methods 1 and 2 could pose security risks over the public Internet but are perfectly valid on corporate or private intranets. Normally, Web applications use server-side schemas and query

templates to make XML data accessible to the outside world in a controlled fashion.

Configuring a Virtual Directory

Load the Configure IIS Support utility in the SQLXML folder under Start | Programs. You should see the IIS servers configured on the current machine. Click the plus sign to the left of your server name to expand it. (If your server isn't listed—for example, if it's a remote server—right-click the IIS Virtual Directory Manager node and select Connect to connect to your server.) To add a new virtual directory, right-click the Default Web Site node and select New | Virtual Directory. You should then see the New Virtual Directory Properties dialog.

Specifying a Virtual Directory Name and Path

The Virtual Directory Name entry box is where you specify the name of the new virtual directory. This is the name that users will include in a URL to access the data exposed by the virtual directory, so it's important to make it descriptive. A common convention is to name virtual directories after the databases they reference. To work through the rest of the examples in the chapter, specify Northwind as the name of the new virtual directory.

Though Local Path will sometimes not be used, it's required nonetheless. In a normal ASP or HTML application, this would be the path where the source files themselves reside. In SQLISAPI applications, this folder does not necessarily need to contain anything, but it must exist nevertheless. On NTFS partitions, you must also make sure that users have at least read access to this folder in order to use the virtual directory. You configure which user account will be used to access the application (and thus will need access to the folder) in the dialog's Security page.

Click the Security tab to select the authentication mode you'd like to use. You can use a specific user account, Windows Integrated Authentication, or Basic (clear text) Authentication. Select the option that matches your usage scenario most closely; Windows Integrated Authentication will likely be the best choice for working through the demos in this chapter.

Next, click the Data Source page tab. This is where you set the SQL Server and the database that the virtual directory references. Select your SQL Server from the list and specify Northwind as the database name.

Go to the Virtual Names table and set up two virtual names, templates and schemas. Create two folders under Northwind named Templates and Schemas so that each of these virtual names can have its own local folder. Set the type for schemas to schema and the type for templates to template.

Each of these provides a navigation path from a URL to the files in its local folder. We'll use them later.

The last dialog page we're concerned with is the Settings page. Click it, then make sure every checkbox on it is checked. We want to allow all of these options so that we may test them later in the chapter. The subsections below provide brief descriptions of each of the options on the Settings page.

Allow sql=... or template=... or URL queries

When this option is enabled, you can execute queries posted to a URL (via an HTTP GET or POST command) as sql= or template= parameters. URL queries allow users to specify a complete Transact-SQL query via a URL. Special characters are replaced with placeholders, but, essentially, the query is sent to the server as is, and its results are returned over HTTP. Note that this option allows users to execute arbitrary queries against the virtual root and database, so you shouldn't enable it for anything but intranet use. Go ahead and enable it for now so that we can try it out later.

Selecting this option disables the Allow template=… containing updategrams only option because you can always post XML templates with updategrams when this option is selected. The Allow template=… containing updategrams only option permits XML templates (that contain only updategrams) to be posted to a URL. Since this disallows SQL and XPath queries from existing in a template, it provides some limited security.

Template queries are by far the most popular method of retrieving XML data from SQL Server over HTTP. XML documents that store query templates—generic parameterized queries with placeholders for parameters—reside on the server and provide a controlled access to the underlying data. The results from template queries are returned over HTTP to the user.

Allow XPath

When Allow XPath is enabled, users can use a subset of the XPath language to retrieve data from SQL Server based on an annotated schema. Annotated schemas are stored on a Web server as XML documents and map XML elements and attributes to the data in the database referenced by a virtual directory. XPath queries allow the user to specify the data defined in an annotated schema to return.

Allow POST

HTTP supports the notion of sending data to a Web server via its POST command. When Allow POST is enabled, you can post a query template (usually

implemented as a hidden form field on a Web page) to a Web server via HTTP. This causes the query to be executed and returns the results back to the client.

As I mentioned earlier, the open-endedness of this usually limits its use to private intranets. Malicious users could form their own templates and post them over HTTP to retrieve data to which they aren't supposed to have access or, worse yet, make changes to it.

Run on the client

This option specifies that XML formatting (e.g., FOR XML) is to be done on the client side. Enabling this option allows you to offload to the client the work of translating a rowset into XML for HTTP queries.

Expose runtime errors as HTTP error

This option controls whether query errors in an XML template are returned in the HTTP header or as part of the generated XML document. When this option is enabled and a query in a template fails, HTTP error 512 is returned and error descriptions are returned in the HTTP header. When it's disabled and a template query fails, the HTTP success code, 200, is returned, and the error descriptions are returned as processing instructions inside the XML document.

Enable all the options on the Settings page except the last two described above and click OK to create your new virtual directory.

TIP: A handy option on the Advanced tab is Disable caching of mapping schemas. Normally, mapping schemas are cached in memory the first time they're used and accessed from the cache thereafter. While developing a mapping schema, you'll likely want to disable this so that the schema will be reloaded each time you test it.

URL Queries

The facility that permits SQL Server to be queried via HTTP resides in SQLXML's ISAPI extension DLL, SQLIS*n*.DLL, commonly referred to as SQLISAPI. Although the Configure IIS Support tool provides a default, you can configure the exact extension DLL uses when you set up a virtual directory for use by HTTP queries.

If you attach to IIS (the executable name is inetinfo.exe) with WinDbg prior to running any HTTP queries, you'll see ModLoad messages for SQLIS*n*.DLL as well as one or two other DLLs. An ISAPI extension DLL is not loaded until the first time it's called.

Architecturally, here's what happens when you execute a basic URL query.

1. You supply the query as a URL in a Web browser.
2. It travels from your browser to the Web server as an HTTP GET request.
3. The virtual directory specified in your query indicates which extension DLL should be called to process the URL. IIS loads the appropriate extension and passes your query to it.
4. SQLIS*n*.DLL, the SQLISAPI extension DLL, gathers the connection, authentication, and database information from the specified virtual directory entry, connects to the appropriate SQL Server and database, and runs the specified query. If the query was passed as a plain T-SQL query, it comes into the server as a language event. If it was passed as a template query, it comes in as an RPC event.
5. The server gathers the requested data and returns it to SQLIS*n*.DLL.
6. The ISAPI extension returns the result data to the Web server, which then, in turn, sends it to the client browser that requested it. Thus, the original HTTP GET request is completed.

Using URL Queries

The easiest way to test the virtual directory you built earlier is to submit a URL query that uses it from an XML-enabled browser such as Internet Explorer. URL queries take this form:

```
http://localhost/Northwind?sql=SELECT+*+FROM+
Customers+FOR+XML+AUTO &root=Customers
```

NOTE: As with all URLs, the URL listed above should be typed on one line. Page width restrictions may force some of the URLs listed in this book to span multiple lines, but a URL should always be typed on a single line.

Here, localhost is the name of the Web server. It could just as easily be a fully qualified DNS domain name such as http://www.khen.com. Northwind is the virtual directory name we created earlier.

A question mark separates the URL from its parameters. Multiple parameters are separated by ampersands. The first parameter we pass here is named sql. It specifies the query to run. The second parameter specifies the name of the root element for the XML document that will be returned. By definition, you get just one of these per document. Failure to specify a root element results in an error if your query returns more than one top-level element.

To see how this works, submit the URL shown in Listing 18.25 from your Web browser. (Be sure to change localhost to the correct name of your Web server if it resides on a different machine).

Listing 18.25

```
http://localhost/Northwind?sql=SELECT+*+FROM+Customers+WHERE
+CustomerId='ALFKI'+FOR+XML+AUTO
```

(Results)

```
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"
ContactName="Maria Anders" ContactTitle="Sales Representative"
Address="Obere Str. 57" City="Berlin" PostalCode="12209"
Country="Germany" Phone="030-0074321" Fax="030-0076545" />
```

Notice that we left off the root element specification. Look at what happens when we bring back more than one row (Listing 18.26).

Listing 18.26

```
http://localhost/Northwind?sql=SELECT+*+FROM+Customers+
WHERE+CustomerId='ALFKI'+OR+CustomerId='ANATR'+FOR+XML+AUTO
```

(Results abridged)

```
The XML page cannot be displayed
Only one top level element is allowed in an XML document.
Line 1, Position 243
```

Since we're returning multiple top-level elements (two, to be exact), our XML document has two root elements named Customers, which, of course, isn't allowed since it isn't well-formed XML. To remedy the situation, we need to specify a root element. This element can be named anything—it serves only to wrap the rows returned by FOR XML so that we have a well-formed document. Listing 18.27 shows an example.

Listing 18.27

```
http://localhost/Northwind?sql=SELECT+*+FROM+Customers+WHERE
+CustomerId='ALFKI'+OR+CustomerId='ANATR'+FOR+XML+AUTO
&root=CustomerList
```

(Results)

```
<?xml version="1.0" encoding="utf-8" ?>
<CustomerList>
  <Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"
    ContactName="Maria Anders" ContactTitle="Sales Representative"
    Address="Obere Str. 57" City="Berlin" PostalCode="12209"
    Country="Germany" Phone="030-0074321" Fax="030-0076545" />
  <Customers CustomerID="ANATR" CompanyName=
    "Ana Trujillo Emparedados y helados" ContactName="Ana Trujillo"
    ContactTitle="Owner" Address="Avda. de la Constitución 2222"
    City="México D.F." PostalCode="05021" Country="Mexico"
    Phone="(5) 555-4729" Fax="(5) 555-3745" />
</CustomerList>
```

You can also supply the root element yourself as part of the sql parameter, as shown in Listing 18.28.

Listing 18.28

```
http://localhost/Northwind?sql=SELECT+'<CustomerList>';
SELECT+*+FROM+Customers+WHERE+CustomerId='ALFKI'+OR
+CustomerId='ANATR'+FOR+XML+AUTO;
SELECT+'</CustomerList>';
```

(Results formatted)

```
<CustomerList>
  <Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"
    ContactName="Maria Anders" ContactTitle="Sales Representative"
    Address="Obere Str. 57" City="Berlin" PostalCode="12209"
    Country="Germany" Phone="030-0074321" Fax="030-0076545" />
  <Customers CustomerID="ANATR" CompanyName=
    "Ana Trujillo Emparedados y helados" ContactName="Ana Trujillo"
    ContactTitle="Owner" Address="Avda. de la Constitución 2222"
    City="México D.F." PostalCode="05021" Country="Mexico"
    Phone="(5) 555-4729" Fax="(5) 555-3745" />
</CustomerList>
```

The sql parameter of this URL actually contains three queries. The first one generates an opening tag for the root element. The second is the query itself, and the third generates a closing tag for the root element. We separate the individual queries with semicolons.

As you can see, FOR XML returns XML document fragments, so you'll need to provide a root element in order to produce a well-formed document.

Special Characters

Certain characters that are perfectly valid in Transact-SQL can cause problems in URL queries because they have special meanings within a URL. You've already noticed that we're using the plus symbol (+) to signify a space character. Obviously, this precludes the direct use of + in the query itself. Instead, you must encode characters that have special meaning within a URL query so that SQLISAPI can properly translate them before passing on the query to SQL Server. Encoding a special character amounts to specifying a percent sign (%) followed by the character's ASCII value in hexadecimal. Table 18.3 lists the special characters recognized by SQLISAPI and their corresponding values.

Here's a URL query that illustrates how to encode special characters.

```
http://localhost/Northwind?sql=SELECT+'<CustomerList>';SELECT
  +*+FROM+Customers+ WHERE+CustomerId+LIKE+'A%25'+FOR+XML+AUTO;
  SELECT+'</CustomerList>';
```

This query specifies a LIKE predicate that includes an encoded percent sign (%), Transact-SQL's wildcard symbol. Hexadecimal 25 (decimal 37) is the ASCII value of the percent sign, so we encode it as %25.

Table 18.3 Special Characters and Their Hexadecimal Values

Character	Hexadecimal Value
+	2B
&	26
?	3F
%	25
/	2F
#	23

Style Sheets

In addition to the sql and root parameters, a URL query can also include the xsl parameter in order to specify an XML style sheet to use to translate the XML document that's returned by the query into a different format. The most common use of this feature is to translate the document into HTML. This allows you to view the document using browsers that aren't XML aware and gives you more control over the display of the document in those that are. Here's a URL query that includes the xsl parameter:

```
http://localhost/Northwind?sql=SELECT+CustomerId,+CompanyName+FROM
  +Customers+FOR+XML+AUTO&root=CustomerList&xsl=CustomerList.xsl
```

Listing 18.29 shows the XSL style sheet it references and the output produced.

Listing 18.29

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0">
  <xsl:template match="/">
    <HTML>
      <BODY>
        <TABLE border="1">
          <TR>
            <TD><B>Customer ID</B></TD>
            <TD><B>Company Name</B></TD>
```

```
          </TR>
          <xsl:for-each select="CustomerList/Customers">
            <TR>
              <TD>
              <xsl:value-of select="@CustomerId"/>
              </TD>
              <TD>
              <xsl:value-of select="@CompanyName"/>
              </TD>
            </TR>
          </xsl:for-each>
        </TABLE>
      </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

(Results abridged)

Customer ID	Company Name
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno TaquerÃa
AROUT	Around the Horn
BERGS	Berglunds snabbkÃ¶p
BLAUS	Blauer See Delikatessen
BLONP	Blondesddsl pÃ¨re et fils
WARTH	Wartian Herkku
WELLI	Wellington Importadora
WHITC	White Clover Markets
WILMK	Wilman Kala
WOLZA	Wolski Zajazd

Content Type

By default, SQLISAPI returns the results of a URL query with the appropriate type specified in the header so that a browser can properly render it. When FOR XML is used in the query, this is text/xml unless the xsl at-

tribute specifies a style sheet that translates the XML document into HTML. In that case, text/html is returned.

You can force the content type using the contenttype URL query parameter, like this:

```
http://localhost/Northwind?sql=SELECT+CustomerId,+CompanyName+FROM
  +Customers+FOR+XML+AUTO&root=CustomerList&xsl=CustomerList.xsl
  &contenttype=text/xml
```

Here, we've specified the style sheet from the previous example in order to cause the content type to default to text/html. Then we override this default by specifying a contenttype parameter of text/xml. The result is an XML document containing the translated result set, as shown in Listing 18.30.

Listing 18.30

```
<HTML>
  <BODY>
    <TABLE border="1">
      <TR>
        <TD>
          <B>Customer ID</B>
        </TD>
        <TD>
          <B>Company Name</B>
        </TD>
      </TR>
      <TR>
        <TD>ALFKI</TD>
        <TD>Alfreds Futterkiste</TD>
      </TR>
      <TR>
        <TD>ANATR</TD>
        <TD>Ana Trujillo Emparedados y helados</TD>
      </TR>
      <TR>
        <TD>WILMK</TD>
        <TD>Wilman Kala</TD>
      </TR>
      <TR>
        <TD>WOLZA</TD>
        <TD>Wolski Zajazd</TD>
      </TR>
```

```
    </TABLE>
  </BODY>
</HTML>
```

So, even though the document consists of well-formed HTML, it's rendered as an XML document because we've forced the content type.

Non-XML Results

Being able to specify the content type comes in particularly handy when working with XML fragments in an XML-aware browser. As I mentioned earlier, executing a FOR XML query with no root element results in an error. You can, however, work around this by forcing the content to HTML, like this:

```
http://localhost/Northwind?sql=SELECT+*+FROM+Customers+WHERE+
  CustomerId='ALFKI'+OR+CustomerId='ANATR'+FOR+XML+AUTO
  &contenttype=text/html
```

If you load this URL in a browser, you'll probably see a blank page because most browsers ignore tags that they don't understand. However, you can view the source of the Web page and you'll see an XML fragment returned as you'd expect. This would be handy in situations where you're communicating with SQLISAPI using HTTP from outside of a browser—from an application of some sort. You could return the XML fragment to the client, then use client-side logic to apply a root element and/or process the XML further.

SQLISAPI also allows you to omit the FOR XML clause in order to return a single column from a table, view, or table-valued function as a plain text stream, as shown in Listing 18.31.

Listing 18.31

```
http://localhost/Northwind?sql=SELECT+CAST(CustomerId+AS+
  char(10))+AS+CustomerId+FROM+Customers+ORDER+BY+CustomerId
  &contenttype=text/html
```

(Results)

```
ALFKI ANATR ANTON AROUT BERGS BLAUS BLONP BOLID BONAP BOTTM BSBEV
CACTU CENTC CHOPS COMMI CONSH DRACD DUMON EASTC ERNSH FAMIA FISSA
FOLIG FOLKO FRANK FRANR FRANS FURIB GALED GODOS GOURL GREAL GROSR
HANAR HILAA HUNGC HUNGO ISLAT KOENE LACOR LAMAI LAUGB LAZYK LEHMS
LETSS LILAS LINOD LONEP MAGAA MAISD MEREP MORGK NORTS OCEAN OLDWO
OTTIK PARIS PERIC PICCO PRINI QUEDE QUEEN QUICK RANCH RATTC REGGC
RICAR RICSU ROMEY SANTG SAVEA SEVES SIMOB SPECD SPLIR SUPRD THEBI
THECR TOMSP TORTU TRADH TRAIH VAFFE VICTE VINET WANDK WARTH WELLI
WHITC WILMK WOLZA
```

Note that SQLISAPI doesn't support returning multicolumn results this way. That said, this is still a handy way to quickly return a simple data list.

Stored Procedures

You can execute stored procedures via URL queries just as you can other types of Transact-SQL queries. Of course, this procedure needs to return its result using FOR XML if you intend to process it as XML in the browser or on the client side. The stored procedure in Listing 18.32 illustrates.

Listing 18.32

```
CREATE PROC ListCustomersXML
@CustomerId varchar(10)='%',
@CompanyName varchar(80)='%'
AS
SELECT CustomerId, CompanyName
FROM Customers
WHERE CustomerId LIKE @CustomerId
AND CompanyName LIKE @CompanyName
FOR XML AUTO
```

Once your procedure correctly returns results in XML format, you can call it from a URL query using the Transact-SQL EXEC command. Listing 18.33 shows an example of a URL query that calls a stored procedure using EXEC.

Listing 18.33

```
http://localhost/Northwind?sql=EXEC+ListCustomersXML
  +@CustomerId='A%25',@CompanyName='An%25'&root=CustomerList
```

(Results)

```
<?xml version="1.0" encoding="utf-8" ?>
<CustomerList>
  <Customers CustomerId="ANATR" CompanyName="Ana Trujillo
      Emparedados y helados" />
  <Customers CustomerId="ANTON" CompanyName="Antonio Moreno
      Taquería" />
</CustomerList>
```

Notice that we specify the Transact-SQL wildcard character "%" by using its encoded equivalent, %25. This is necessary, as I said earlier, because % has special meaning in a URL query.

TIP: You can also use the ODBC CALL syntax to call a stored procedure from a URL query. This executes the procedures via an RPC event on the server, which is generally faster and more efficient than normal T-SQL language events. On high-volume Web sites, the small difference in performance this makes can add up quickly.

Here are a couple of URL queries that use the ODBC CALL syntax:

```
http://localhost/Northwind?sql={CALL+ListCustomersXML}+
   &root=CustomerList

http://localhost/Northwind?sql={CALL+ListCustomersXML('ALFKI')}+
   &root=CustomerList
```

If you submit one of these URLs from your Web browser while you have a Profiler trace running that includes the RPC:Starting event, you should see an RPC:Starting event for the procedure. This indicates that the procedure is being called via the more efficient RPC mechanism rather than via a language event.

See the Template Queries section below for more information on making RPCs from SQLXML.

Template Queries

A safer and more widely used technique for retrieving data over HTTP is to use server-side XML templates that encapsulate Transact-SQL queries. Because these templates are stored on the Web server and referenced via a virtual name, the end users never see the source code. The templates are XML documents based on the XML-SQL namespace and function as a mechanism for translating a URL into a query that SQL Server can process. As with plain URL queries, results from template queries are returned as either XML or HTML.

Listing 18.34 shows a simple XML query template.

Listing 18.34

```
<?xml version='1.0' ?>
<CustomerList xmlns:sql='urn:schemas-microsoft-com:xml-sql'>
  <sql:query>
    SELECT CustomerId, CompanyName
    FROM Customers
    FOR XML AUTO
  </sql:query>
</CustomerList>
```

Note the use of the sql namespace prefix with the query itself. This is made possible by the namespace reference on the second line of the template (bolded).

Here we're merely returning two columns from the Northwind Customers table, as we've done several times in this chapter. We include FOR XML AUTO to return the data as XML. The URL shown in Listing 18.35 uses the template, along with the data it returns.

Listing 18.35

```
http://localhost/Northwind/templates/CustomerList.XML
```

(Results abridged)

```
<?xml version="1.0" ?>
<CustomerList xmlns:sql="urn:schemas-microsoft-com:xml-sql">
```

```
  <Customers CustomerId="ALFKI" CompanyName=
    "Alfreds Futterkiste" />
  <Customers CustomerId="VAFFE" CompanyName="Vaffeljernet" />
  <Customers CustomerId="VICTE" CompanyName=
    "Victuailles en stock" />
  <Customers CustomerId="VINET" CompanyName=
    "Vins et alcools Chevalier" />
  <Customers CustomerId="WARTH" CompanyName="Wartian Herkku" />
  <Customers CustomerId="WELLI" CompanyName=
    "Wellington Importadora" />
  <Customers CustomerId="WHITC" CompanyName=
    "White Clover Markets" />
  <Customers CustomerId="WILMK" CompanyName="Wilman Kala" />
  <Customers CustomerId="WOLZA" CompanyName="Wolski Zajazd" />
</CustomerList>
```

Notice that we're using the templates virtual name that we created under the Northwind virtual directory earlier.

Parameterized Templates

You can also create parameterized XML query templates that permit the user to supply parameters to the query when it's executed. You define parameters in the header of the template, which is contained in its sql:header element. Each parameter is defined using the sql:param tag and can include an optional default value. Listing 18.36 presents an example.

Listing 18.36

```
<?xml version='1.0' ?>
<CustomerList xmlns:sql='urn:schemas-microsoft-com:xml-sql'>
  <sql:header>
    <sql:param name='CustomerId'>%</sql:param>
  </sql:header>
  <sql:query>
    SELECT CustomerId, CompanyName
    FROM Customers
    WHERE CustomerId LIKE @CustomerId
    FOR XML AUTO
  </sql:query>
</CustomerList>
```

Note the use of sql:param to define the parameter. Here, we give the parameter a default value of % since we're using it in a LIKE predicate in the query. This means that we list all customers if no value is specified for the parameter.

Note that SQLISAPI is smart enough to submit a template query to the server as an RPC when you define query parameters. It binds the parameters you specify in the template as RPC parameters and sends the query to SQL Server using RPC API calls. This is more efficient than using T-SQL language events and should result in better performance, particularly on systems with high throughput.

Listing 18.37 gives an example of a URL that specifies a parameterized template query, along with its results.

Listing 18.37

```
http://localhost/Northwind/Templates/CustomerList2.XML?
  CustomerId=A%25
```

(Results)

```
<?xml version="1.0" ?>
<CustomerList xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Customers CustomerId="ALFKI" CompanyName=
    "Alfreds Futterkiste" />
  <Customers CustomerId="ANATR" CompanyName=
    "Ana Trujillo Emparedados y helados" />
  <Customers CustomerId="ANTON" CompanyName=
    "Antonio Moreno Taquería" />
  <Customers CustomerId="AROUT" CompanyName="Around the Horn" />
</CustomerList>
```

Style Sheets

As with regular URL queries, you can specify a style sheet to apply to a template query. You can do this in the template itself or in the URL that accesses it. Here's an example of a URL that applies a style sheet to a template query:

```
http://localhost/Northwind/Templates/CustomerList3.XML
  ?xsl=Templates/CustomerList3.xsl&contenttype=text/html
```

Note the use of the contenttype parameter to force the output to be treated as HTML (bolded). We do this because we know that the style sheet

we're applying translates the XML returned by SQL Server into an HTML table.

We include the relative path from the virtual directory to the style sheet because it's not automatically located in the Templates folder even though the XML document is located there. The path specifications for a template query and its parameters are separate from one another.

As I've mentioned, the XML-SQL namespace also supports specifying the style sheet in the template itself. Listing 18.38 shows a template that specifies a style sheet.

Listing 18.38

```
<?xml version='1.0' ?>
<CustomerList xmlns:sql='urn:schemas-microsoft-com:xml-sql'
    sql:xsl='CustomerList3.xsl'>
  <sql:query>
    SELECT CustomerId, CompanyName
    FROM Customers
    FOR XML AUTO
  </sql:query>
</CustomerList>
```

The style sheet referenced by the template appears in Listing 18.39.

Listing 18.39

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
  <xsl:template match="/">
    <HTML>
      <BODY>
        <TABLE border="1">
          <TR>
            <TD><I>Customer ID</I></TD>
            <TD><I>Company Name</I></TD>
          </TR>
          <xsl:for-each select="CustomerList/Customers">
            <TR>
              <TD><B>
```

```
            <xsl:value-of select="@CustomerId"/>
            </B></TD>
            <TD>
            <xsl:value-of select="@CompanyName"/>
            </TD>
          </TR>
        </xsl:for-each>
      </TABLE>
    </BODY>
  </HTML>
 </xsl:template>
</xsl:stylesheet>
```

Listing 18.40 shows a URL that uses the template and the style sheet shown in the previous two listings, along with the results it produces.

Listing 18.40

```
http://localhost/Northwind/Templates/CustomerList4.XML?
    contenttype=text/html
```

(Results abridged)

Customer ID	Company Name
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno TaquerÃa
AROUT	Around the Horn
VICTE	Victuailles en stock
VINET	Vins et alcools Chevalier
WARTH	Wartian Herkku
WELLI	Wellington Importadora
WHITC	White Clover Markets
WILMK	Wilman Kala
WOLZA	Wolski Zajazd

Note that, once again, we specify the contenttype parameter in order to force the output to be treated as HTML. This is necessary because XML-aware browsers such as Internet Explorer automatically treat the output returned by XML templates as text/xml. Since the HTML we're returning is also well-formed XML, the browser doesn't know to render it as HTML unless we tell it to. That's what the contenttype specification is for—it causes the browser to render the output of the template query as it would any other HTML document.

TIP: While developing XML templates and similar documents that you then test in a Web browser, you may run into problems with the browser caching old versions of documents, even when you click the Refresh button or hit the Refresh key (F5). In Internet Explorer, you can press Ctrl+F5 to cause a document to be completely reloaded, even if the browser doesn't think it needs to be. Usually, this resolves problems with an old version persisting in memory after you've changed the one on disk.

You can also disable the caching of templates for a given virtual directory by selecting the Disable caching of templates option on the Advanced page of the Properties dialog for the virtual directory. I almost always disable all caching while developing templates and other XML documents.

Applying Style Sheets on the Client

If the client is XML-enabled, you can also apply style sheets to template queries on the client side. This offloads a bit of the work of the server but requires a separate roundtrip to download the style sheet to the client. If the client is not XML-enabled, the style sheet will be ignored, making this approach more suitable to situations where you know for certain whether your clients are XML-enabled, such as with private intranet or corporate applications.

The template in Listing 18.41 specifies a client-side style sheet translation.

Listing 18.41

```
<?xml version='1.0' ?>
<?xml-stylesheet type='text/xsl' href='CustomerList3.xsl'?>
<CustomerList xmlns:sql='urn:schemas-microsoft-com:xml-sql'>
  <sql:query>
    SELECT CustomerId, CompanyName
    FROM Customers
    FOR XML AUTO
  </sql:query>
</CustomerList>
```

Note the xml-stylesheet specification at the top of the document (bolded). This tells the client-side XML processor to download the style sheet specified in the href attribute and apply it to the XML document rendered by the template. Listing 18.42 shows the URL and results.

Listing 18.42

```
http://localhost/Northwind/Templates/CustomerList5.XML?
    contenttype=text/html
```

(Results abridged)

Customer ID	Company Name
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno TaquerÃa
AROUT	Around the Horn
VICTE	Victuailles en stock
VINET	Vins et alcools Chevalier
WARTH	Wartian Herkku
WELLI	Wellington Importadora
WHITC	White Clover Markets
WILMK	Wilman Kala
WOLZA	Wolski Zajazd

Client-Side Templates

As I mentioned earlier, it's far more popular (and safer) to store templates on your Web server and route users to them via virtual names. That said, there are times when allowing the user the flexibility to specify templates on the client side is very useful. Specifying client-side templates in HTML or in an application alleviates the necessity to set up in advance the templates or the virtual names that reference them. While this is certainly easier from an administration standpoint, it's potentially unsafe on the public Internet because it allows clients to specify the code they run against your SQL Server. Use of this technique should probably be limited to private intranets and corporate networks.

Listing 18.43 presents a Web page that embeds a client-side template.

Listing 18.43

```
<HTML>
  <HEAD>
    <TITLE>Customer List</TITLE>
  </HEAD>
  <BODY>
    <FORM action='http://localhost/Northwind' method='POST'>
      <B>Customer ID Number</B>
      <INPUT type=text name=CustomerId value='AAAAA'>
      <INPUT type=hidden name=xsl value=Templates/CustomerList2.xsl>
      <INPUT type=hidden name=template value='
      <CustomerList xmlns:sql="urn:schemas-microsoft-com:xml-sql">
        <sql:header>
          <sql:param name="CustomerId">%</sql:param>
        </sql:header>
        <sql:query>
          SELECT CompanyName, ContactName
          FROM Customers
          WHERE CustomerId LIKE @CustomerId
          FOR XML AUTO
        </sql:query>
      </CustomerList>
      '>
      <P><input type='submit'>
    </FORM>
  </BODY>
</HTML>
```

The client-side template (bolded) is embedded as a hidden field in the Web page. If you open this page in a Web browser, you should see an entry box for a Customer ID and a submit button. Entering a customer ID or mask and clicking Submit Query will post the template to the Web server. SQLISAPI will then extract the query contained in the template and run it against SQL Server's Northwind database (because of the template's virtual directory reference). The CustomerList2.xsl style sheet will then be applied to translate the XML document that SQL Server returns into HTML, and the result will be returned to the client. Listing 18.44 shows an example.

Listing 18.44

Customer ID Number A%

Submit Query

(Results)

Company Name	Contact Name
Alfreds Futterkiste	Maria Anders
Ana Trujillo Emparedados y helados	Ana Trujillo
Antonio Moreno TaquerÃa	Antonio Moreno
Around the Horn	Thomas Hardy

As with server-side templates, client-side templates are sent to SQL Server using an RPC.

Mapping Schemas

XML schemas are XML documents that define the type of data that other XML documents may contain. They are a replacement for the old DTD technology originally employed for that purpose and are easier to use and more flexible because they consist of XML themselves.

By their very nature, schemas also define document exchange formats. Since they define what a document may and may not contain, companies wishing to exchange XML data need to agree on a common schema definition in order to do so. XML schemas allow companies with disparate business needs and cultures to exchange data seamlessly.

A mapping schema is a special type of schema that maps data between an XML document and a relational table. A mapping schema can be used to create an XML view of a SQL Server table. In that sense, a mapping schema is similar to a SQL Server view object that returns an XML-centric view of the underlying SQL Server table or view object.

Work on the final XML Schema standard was still under way when SQL Server 2000 shipped. At that time, Microsoft, along with several other companies, proposed that a subset of the W3C XML-Data syntax be used to define schemas for document interchange. SQL Server's original XML schema support was based on XML-Data Reduced (XDR), an XML-Data subset that can be used to define schemas. Since then, the XML Schema standard has been finalized, and SQLXML has been enhanced to support it. XML Schema is now the preferred method of building schemas for use by SQLXML. It is more flexible and has more features than the original XDR schema support in SQLXML. I'll cover SQLXML's XDR and XML Schema support in the next two sections.

XDR Mapping Schemas

Let's begin our coverage of XDR mapping schemas with an example (Listing 18.45).

Listing 18.45

```
<?xml version="1.0"?>
<Schema name="NorthwindProducts"
  xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">

  <ElementType name="Description" dt:type="string"/>
  <ElementType name="Price" dt:type="fixed.19.4"/>

  <ElementType name="Product" model="closed">
    <AttributeType name="ProductCode" dt:type="string"/>
    <attribute type="ProductCode" required="yes"/>
    <element type="Description" minOccurs="1" maxOccurs="1"/>
    <element type="Price" minOccurs="1" maxOccurs="1"/>
  </ElementType>

  <ElementType name="Category" model="closed">
    <AttributeType name="CategoryID" dt:type="string"/>
    <AttributeType name="CategoryName" dt:type="string"/>
    <attribute type="CategoryID" required="yes"/>
```

```
    <attribute type="CategoryName" required="yes"/>
    <element type="Product" minOccurs="1" maxOccurs="*"/>
  </ElementType>

  <ElementType name="Catalog" model="closed">
    <element type="Category" minOccurs="1" maxOccurs="1"/>
  </ElementType>

</Schema>
```

This schema defines how a product catalog might look. (We're using the sample tables and data from the Northwind database.) It uses the datatypes namespace (bolded) to define the valid data types for elements and attributes in the document. Every place you see dt: in the listing is a reference to the datatypes namespace. The use of the closed model guarantees that only elements that exist in the schema can be used in a document based on it.

Listing 18.46 shows an XML document that uses ProductCat.xdr.

Listing 18.46

```
<?xml version="1.0"?>
<Catalog xmlns=
  "x-schema:http://localhost/ProductsCat.xdr">
  <Category CategoryID="1" CategoryName="Beverages">
    <Product ProductCode="1">
      <Description>Chai</Description>
      <Price>18</Price>
    </Product>
    <Product ProductCode="2">
      <Description>Chang</Description>
      <Price>19</Price>
    </Product>
  </Category>
  <Category CategoryID="2" CategoryName="Condiments">
    <Product ProductCode="3">
      <Description>Aniseed Syrup</Description>
      <Price>10</Price>
    </Product>
  </Category>
</Catalog>
```

If you copy both of these files to the root folder of your Web server and type the following URL:

```
http://localhost/ProductsCat.xml
```

into your browser, you should see this output:

```
<?xml version="1.0" ?>
<Catalog xmlns="x-schema:http://localhost/ProductsCat.xdr">
<Category CategoryID="1" CategoryName="Beverages">
  <Product ProductCode="1">
    <Description>Chai</Description>
    <Price>18</Price>
  </Product>
  <Product ProductCode="2">
    <Description>Chang</Description>
    <Price>19</Price>
  </Product>
</Category>
<Category CategoryID="2" CategoryName="Condiments">
  <Product ProductCode="3">
    <Description>Aniseed Syrup</Description>
    <Price>10</Price>
  </Product>
</Category>
</Catalog>
```

You've already seen that XML data can be extracted and formatted in a variety of ways. One of the challenges in exchanging data using XML is this flexibility. Mapping schemas help overcome this challenge. They allow us to return data from a database in a particular format. They allow us to map columns and tables to attributes and elements.

The easiest way to use an XDR schema to map data returned by SQL Server into XML entities is to assume the default mapping returned by SQL Server. That is, every table becomes an element, and every column becomes an attribute. Listing 18.47 presents an XDR schema that does that.

Listing 18.47

```
<?xml version="1.0"?>
<Schema name="customers"
  xmlns="urn:schemas-microsoft-com:xml-data">
```

```
  <ElementType name="Customers">
    <AttributeType name="CustomerId"/>
    <AttributeType name="CompanyName"/>
  </ElementType>
</Schema>
```

Here, we retrieve only two columns, each of them from the Customers table. If you store this XDR schema under a virtual directory on your Web server and retrieve it via a URL, you'll see a simple XML document with the data from the Northwind Customers table in an attribute-centric mapping.

You use XML-Data's ElementType to map a column in a table to an element in the resulting XML document, as demonstrated in Listing 18.48.

Listing 18.48

```
<?xml version="1.0"?>
<Schema name="customers"
  xmlns="urn:schemas-microsoft-com:xml-data">
  <ElementType name="Customers">
    <ElementType name="CustomerId" content="textOnly"/>
    <ElementType name="CompanyName" content="textOnly"/>
  </ElementType>
</Schema>
```

Note the use of the content="textOnly" attribute with each element. In conjunction with the ElementType element, this maps a column to an element in the resulting XML document. Note that the elements corresponding to each column are actually empty—they contain attributes only, no data.

Annotated XDR Schemas

An annotated schema is a mapping schema with special annotations (from the XML-SQL namespace) that link elements and attributes with tables and columns. The code in Listing 18.49 uses our familiar Customer list example.

Listing 18.49

```
<?xml version="1.0"?>
<Schema name="customers"
```

```
  xmlns="urn:schemas-microsoft-com:xml-data">
  xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <ElementType name="Customer" sql:relation="Customers">
    <AttributeType name="CustomerNumber" sql:field="CustomerId"/>
    <AttributeType name="Name" sql:field="CompanyName"/>
  </ElementType>
</Schema>
```

First, note the reference to the XML-SQL namespace at the top of the schema. Since we'll be referencing it later in the schema, we begin with a reference to XML-SQL so that we can use the sql: namespace shorthand for it later. Next, notice the sql:relation attribute of the first ElementType element. It establishes that the Customer element in the resulting document relates to the Customers table in the database referenced by the virtual directory. This allows you to call the element whatever you want. Last, notice the sql:field references. They establish, for example, that the CustomerNumber element refers to the CustomerId column in the referenced table. Things get more complicated when multiple tables are involved, but you get the picture—an annotated schema allows you to establish granular mappings between document entities and database entities.

XSD Mapping Schemas

Similarly to XDR, you can also construct XML views using annotated XML Schema Definition (XSD) language. This is, in fact, the preferable way to build annotated schemas because XDR was an interim technology that preceded the finalization of the XML Schema standard, as I mentioned earlier. In this section, we'll talk about the various ways to construct annotated XSD mapping schemas and walk through a few examples.

Just as we did with XDR, let's begin our discussion of XSD mapping schemas with an example (Listing 18.50).

Listing 18.50

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
  <xsd:element name="Customers" >
    <xsd:complexType>
      <xsd:attribute name="CustomerID" type="xsd:string" />
      <xsd:attribute name="CompanyName" type="xsd:string" />
```

```
      <xsd:attribute name="ContactName" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Note the reference to the XSD namespace, http://www.w3.org/2001/XMLSchema. We alias this to xsd (the alias name is arbitrary—it serves merely as shorthand to distinguish XSD elements and attributes from those of other namespaces), then prefix XSD elements/attributes in the schema with xsd:.

SQLXML's mapping schema namespace is defined at urn:schemas-microsoft-com:mapping-schema. We use this namespace to map elements and attributes in the schema to tables and columns in a database. We've defined this namespace with an alias of sql, so we'll use a prefix of sql: when referring to elements and attributes in SQLXML's mapping schema namespace.

Default Mapping

The schema above uses default mapping to associate complex XSD types with tables/views of the same name and attributes with same-named columns. Note the absence of any reference to the sql namespace (once it's defined). We're not using it because we're not explicitly mapping any elements or attributes to tables or columns. You can construct a template like the following to query this XML view using an XPath expression:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:xpath-query mapping-schema="Customers.xsd">
    /Customers
  </sql:xpath-query>
</ROOT>
```

Follow these steps to query the XML view in Listing 18.50 by using the above template from your browser.

1. Save the XML view as Customers.XSD in the templates folder you created under the Northwind virtual directory earlier.
2. Save the template above as CustomersT.XML in the same folder.
3. Go to the following URL in your browser:

   ```
   http://localhost/Northwind/templates/CustomersT.XML
   ```

Explicit Mapping

A mapping schema can also specify explicit relationships between XSD elements and attributes and SQL Server tables and columns. This is done by using the SQLXML mapped schema namespace I mentioned above. Specifically, we'll make use of sql:field and sql:relation to establish these relationships, as shown in Listing 18.51.

Listing 18.51

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
  <xsd:element name="Cust" sql:relation="Customers" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustNo"
                     sql:field="CustomerId"
                     type="xsd:integer" />
        <xsd:element name="Contact"
                     sql:field="ContactName"
                     type="xsd:string" />
        <xsd:element name="Company"
                     sql:field="CompanyName"
                     type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Note the use of sql:relation to establish the mapping between the Cust document element and the Customers database table and the use of the sql:field notation to establish mappings between document elements and table columns. Because each table column is annotated as an element, each column in the Customers table will become a separate element in the resulting XML document. You can also map table columns to attributes, as demonstrated in Listing 18.52.

Listing 18.52

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
  <xsd:element name="Cust" sql:relation="Customers" >
```

```
    <xsd:complexType>
      <xsd:attribute name="CustNo" sql:field="CustomerId"
          type="xsd:integer" />
      <xsd:attribute name="Contact" sql:field="ContactName"
          type="xsd:string" />
      <xsd:attribute name="Company" sql:field="CompanyName"
          type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Here, we leave out the complexType element (because we don't need it—we're not defining nested elements) and simply map each table column to an attribute in the XSD using sql:field.

Relationships

You can use the sql:relationship annotation to establish a relationship between two elements. You define an empty sql:relationship element and include parent, parent-key, child, and child-key attributes to define the relationship between the two elements. Relationships defined this way can be named or unnamed. For elements mapped to tables and columns in a SQL Server database, this is similar to joining the tables; the parent/child and parent-key/child-key matchups supply the join criteria. Listing 18.53 shows an example (from EmpOrders.XSD in the CH18 subfolder on the CD accompanying this book).

Listing 18.53

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">

  <xsd:element name="Employee" sql:relation="Employees"
      type="EmployeeType" />
    <xsd:complexType name="EmployeeType" >
      <xsd:sequence>
        <xsd:element name="Order"
                     sql:relation="Orders">
          <xsd:annotation>
            <xsd:appinfo>
              <sql:relationship
                parent="Employees"
```

```
                parent-key="EmployeeID"
                child="Orders"
                child-key="EmployeeID" />
            </xsd:appinfo>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:attribute name="OrderID" type="xsd:integer" />
            <xsd:attribute name="EmployeeID" type="xsd:integer" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
        <xsd:attribute name="EmployeeID"   type="xsd:integer" />
        <xsd:attribute name="LastName"  type="xsd:string" />
    </xsd:complexType>
</xsd:schema>
```

In this schema, we establish a relationship between the Employee and Order elements using the EmployeeID attribute. Again, this is accomplished via the notational attributes provided by Microsoft's mapping-schema namespace.

sql:inverse

You can use the sql:inverse annotation to invert a relationship established with sql:relationship. Why would you want to do that? SQLXML's updategram logic interprets the schema in order to determine the tables being updated by an updategram. (We'll cover updategrams in the next section.) The parent-child relationships established with sql:relationship determine the order in which row deletions and inserts occur. If you specify the sql:relationship notation such that the parent-child relationship between the tables is the inverse of the underlying primary key/foreign key relationship, the attempted insert or delete operation will fail due to key violations. You can set the sql:inverse attribute to 1 (or true) in the sql:relationship element in order to flip the relationship so that this doesn't happen.

The usefulness of the sql:inverse notation is limited to updategrams. There's no point in inversing a regular mapping schema. Listing 18.54 presents an example of a mapping schema that puts the sql:inverse annotation attribute to good use. (You can find this in OrderDetails.XSD in the CH18 folder on the CD accompanying this book.)

Listing 18.54

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">

  <xsd:element name="OrderDetails" sql:relation="[Order Details]"
      type="OrderDetailsType" />
    <xsd:complexType name="OrderDetailsType" >
      <xsd:sequence>
        <xsd:element name="Order"
                     sql:relation="Orders">
          <xsd:annotation>
            <xsd:appinfo>
              <sql:relationship
                parent="[Order Details]"
                parent-key="OrderID"
                child="Orders"
                child-key="OrderID"
              inverse="true" />
            </xsd:appinfo>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:attribute name="OrderID" type="xsd:integer" />
            <xsd:attribute name="EmployeeID" type="xsd:integer" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="ProductID"   type="xsd:integer" />
      <xsd:attribute name="Qty" sql:field="Quantity" type="xsd:integer" />
    </xsd:complexType>
</xsd:schema>
```

Note the use of square brackets around the Order Details table name. These are required in the mapping schema for SQL Server table names that contain spaces.

sql:mapped

You can use the sql:mapped annotation to control whether an attribute or element is mapped to a database object. When the default mapping is used,

every element and attribute in a mapping schema maps to a database object. If you have a schema in which you have elements or attributes that you do not want to map to database objects, you can set the sql:mapped annotation to 0 (or false) in an XSD element or attribute specification. The sql:mapped annotation is especially useful in situations where the schema can't be changed or is being used to validate other XML data and contains elements or attributes that do not have analogues in your database. Listing 18.55 uses sql:mapped to include an element in a mapping schema that is not mapped to a database object.

Listing 18.55

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">

  <xsd:element name="Employee" sql:relation="Employees"
      type="EmployeeType" />
    <xsd:complexType name="EmployeeType" >
      <xsd:sequence>
        <xsd:element name="Order"
                     sql:relation="Orders">
          <xsd:annotation>
            <xsd:appinfo>
              <sql:relationship
                parent="Employees"
                parent-key="EmployeeID"
                child="Orders"
                child-key="EmployeeID" />
            </xsd:appinfo>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:attribute name="OrderID" type="xsd:integer" />
            <xsd:attribute name="EmployeeID" type="xsd:integer" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="EmployeeID"   type="xsd:integer" />
      <xsd:attribute name="LastName"  type="xsd:string" />
      <xsd:attribute name="Level" type="xsd:integer"
          sql:mapped="0" />
    </xsd:complexType>
</xsd:schema>
```

Note the inclusion of the Level attribute in the Employee element. Because it contains a sql:mapped annotation that is set to false, it is not mapped to a database object.

sql:limit-field and sql:limit-value

Similarly to the way you can filter XML views using XPath expressions, you can also filter them based on values returned from the database using the sql:limit-field and sql:limit-value annotations. The sql:limit-field annotation specifies the filter column from the database; sql:limit-value specifies the value to filter it by. Note that sql:limit-value is actually optional—if it isn't supplied, NULL is assumed. Listing 18.56 shows an example of a mapping schema that filters based on the value of a column in the database.

Listing 18.56

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">

  <xsd:element name="Employee" sql:relation="Employees"
      type="EmployeeType" />
    <xsd:complexType name="EmployeeType" >
      <xsd:sequence>
        <xsd:element name="Order"
                     sql:relation="Orders">
          <xsd:annotation>
            <xsd:appinfo>
              <sql:relationship
                parent="Employees"
                parent-key="EmployeeID"
                child="Orders"
                child-key="EmployeeID" />
            </xsd:appinfo>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:attribute name="OrderID" type="xsd:integer" />
            <xsd:attribute name="EmployeeID" type="xsd:integer" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
        <xsd:attribute name="EmployeeID"
          type="xsd:integer"
          sql:limit-field="EmployeeID"
```

```
          sql:limit-value="3"/>
        <xsd:attribute name="LastName"  type="xsd:string" />
    </xsd:complexType>

</xsd:schema>
```

This schema filters the XML document based on the EmployeeID column in the database. Only those rows with an EmployeeID of 3 are returned in the document. If you submit a URL query against this mapping schema using the following template:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:xpath-query mapping-schema="EmpOrders_Filtered.XSD">
    /Employee
  </sql:xpath-query>
</ROOT>
```

you'll see a document that looks something like this in your browser (results abridged):

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Employee EmployeeID="3" LastName="Leverling">
  <Order EmployeeID="3" OrderID="10251" />
  <Order EmployeeID="3" OrderID="10253" />
  <Order EmployeeID="3" OrderID="10256" />
  <Order EmployeeID="3" OrderID="10266" />
  <Order EmployeeID="3" OrderID="10273" />
  <Order EmployeeID="3" OrderID="10283" />
  <Order EmployeeID="3" OrderID="10309" />
  <Order EmployeeID="3" OrderID="10321" />
  <Order EmployeeID="3" OrderID="10330" />
  <Order EmployeeID="3" OrderID="10332" />
  <Order EmployeeID="3" OrderID="10346" />
  <Order EmployeeID="3" OrderID="10352" />
...
</ROOT>
```

sql:key-fields

You use the sql:key-fields annotation to identify the key columns in a table to which an XML view is mapped. The sql:key-fields annotation is usually required in mapping schemas in order to ensure that proper nesting occurs

in the resulting XML document. This is because the key columns of the underlying table are used to nest the document. This makes the XML that's produced sensitive to the order of the underlying data. If the key columns of the underlying data can't be determined, the generated XML might be formed incorrectly. You should always specify either sql:key-fields or elements that map directly to tables in the database. Listing 18.57 offers an example of a mapping schema that uses sql:key-fields (from EmpOrders_KeyFields.XSD in the CH18 folder on the CD accompanying this book).

Listing 18.57

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">

  <xsd:element name="Employee"
  sql:relation="Employees"
  type="EmployeeType"
  sql:key-fields="EmployeeID"/>
    <xsd:complexType name="EmployeeType" >
      <xsd:sequence>
        <xsd:element name="Order"
                     sql:relation="Orders">
          <xsd:annotation>
            <xsd:appinfo>
              <sql:relationship
                parent="Employees"
                parent-key="EmployeeID"
                child="Orders"
                child-key="EmployeeID" />
            </xsd:appinfo>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:attribute name="OrderID" type="xsd:integer" />
            <xsd:attribute name="EmployeeID" type="xsd:integer" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
        <xsd:attribute name="LastName"  type="xsd:string" />
        <xsd:attribute name="FirstName"  type="xsd:string" />
    </xsd:complexType>
</xsd:schema>
```

Note that we haven't mapped the EmployeeID column in the Employees table. Without this column, we don't have a column with which we can join the Orders table. Including it in the sql:key-fields annotation allows us to leave it unmapped but still establish the relationship between the two tables.

Updategrams

Thus far, we've looked at how data can be retrieved from SQL Server in XML format, but we haven't talked about how to update SQL Server data using XML. Updategrams provide an XML-based method of updating data in a SQL Server database. They are basically templates with special attributes and elements that allow you to specify the data you want to update and how you want to update it. An updategram contains a before image and an after image of the data you want to change. You submit updategrams to SQL Server in much the same way as you submit templates. All the execution mechanisms available with templates work equally well with updategrams. You can POST updategrams via HTTP, save updategrams to files and execute them via URLs, and execute updategrams directly via ADO and OLE DB.

How They Work

Updategrams are based on the xml-updategram namespace. You reference this namespace via the xmlns:updg qualifier. Each updategram contains at least one sync element. This sync element contains the data changes you wish to make in the form of before and after elements. The before element contains the before image of the data you wish to change. Normally, it will also contain a primary key or candidate key reference so that SQL Server will be able to locate the row you wish to change. Note that only one row can be selected for update by the before element. If the elements and attributes included in the before element identify more than one row, you'll receive an error message.

For row deletions, an updategram will have a before image but no after image. For insertions, it will have an after image but no before image. And, of course, for updates, an updategram will have both a before image and an after image. Listing 18.58 provides an example.

Listing 18.58

```
<?xml version="1.0"?>
<employeeupdate xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync>
    <updg:before>
      <Employees EmployeeID="4"/>
    </updg:before>
    <updg:after>
      <Employees City="Scotts Valley" Region="CA"/>
    </updg:after>
  </updg:sync>
</employeeupdate>
```

In this example, we change the City and Region columns for Employee 4 in the Northwind Employees table. The EmployeeID attribute in the before element identifies the row to change, and the City and Region attributes in the after element identify which columns to change and what values to assign them.

Each batch of updates within a sync element is considered a transaction. Either all the updates in the sync element succeed or none of them do. You can include multiple sync elements to break updates into multiple transactions.

Mapping Data

Of course, in sending data to the server for updates, deletions, and insertions via XML, we need a means of linking values in the XML document to columns in the target database table. SQL Server sports two facilities for doing this: default mapping and mapping schemas.

Default Mapping

Naturally, the easiest way to map data in an updategram to columns in the target table is to use the default mapping (also known as intrinsic mapping). With default mapping, a before or after element's top-level tag is assumed to refer to the target database table, and each subelement or attribute it contains refers to a column of the same name in the table.

Here's an example that shows how to map the OrderID column in the Orders table:

```
<Orders OrderID="10248"/>
```

This example maps XML attributes to table columns. You could also map subelements to table columns, like this:

```
<Orders>
  <OrderID>10248</OrderID>
</Orders>
```

You need not select either attribute-centric or element-centric mapping. You can freely mix them within a given before or after element, as shown below:

```
<Orders OrderID="10248">
  <ShipCity>Reims</ShipCity>
</Orders>
```

Use the four-digit hexadecimal UCS-2 code for characters in table names that are illegal in XML elements (e.g., spaces). For example, to reference the Northwind Order Details table, do this:

```
<Order_x0020_Details OrderID="10248"/>
```

Mapping Schemas

You can also use XDR and XSD mapping schemas to map data in an updategram to tables and columns in a database. You use a sync's updg:mapping-schema attribute to specify the mapping schema for an updategram. Listing 18.59 shows an example that specifies an updategram for the Orders table.

Listing 18.59

```
<?xml version="1.0"?>
<orderupdate xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync updg:mapping-schema="OrderSchema.xml">
    <updg:before>
```

```
      <Order OID="10248"/>
    </updg:before>
    <updg:after>
      <Order City="Reims"/>
    </updg:after>
  </updg:sync>
</orderupdate>
```

Listing 18.60 shows its XDR mapping schema.

Listing 18.60

```
<?xml version="1.0"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <ElementType name="Order" sql:relation="Orders">
    <AttributeType name="OID"/>
    <AttributeType name="City"/>
    <attribute type="OID" sql:field="OrderID"/>
    <attribute type="City" sql:field="ShipCity"/>
  </ElementType>
</Schema>
```

Listing 18.61 shows its XSD mapping schema.

Listing 18.61

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
  <xsd:element name="Order" sql:relation="Orders" >
    <xsd:complexType>
      <xsd:attribute name="OID" sql:field="OrderId"
          type="xsd:integer" />
      <xsd:attribute name="City" sql:field="ShipCity"
          type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

As you can see, a mapping schema maps the layout of the XML document to the Northwind Orders table. See the Mapping Schemas section earlier in the chapter for more information on building XML mapping schemas.

NULLs

It's common to represent missing or inapplicable data as NULL in a database. To represent or retrieve NULL data in an updategram, you use the sync element's nullvalue attribute to specify a placeholder for NULL. This placeholder is then used everywhere in the updategram that you need to specify a NULL value, as demonstrated in Listing 18.62.

Listing 18.62

```
<?xml version="1.0"?>
<employeeupdate xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync updg:nullvalue="NONE">
    <updg:before>
      <Orders OrderID="10248"/>
    </updg:before>
    <updg:after>
      <Orders ShipCity="Reims" ShipRegion="NONE"
        ShipName="NONE"/>
    </updg:after>
  </updg:sync>
</employeeupdate>
```

As you can see, we define a placeholder for NULL named NONE. We then use this placeholder to assign a NULL value to the ShipRegion and ShipName columns.

Parameters

Curiously, parameters work a little differently with updategrams than with templates. Rather than using at (@) symbols to denote updategram parameters, you use dollar ($) symbols, as shown in Listing 18.63.

Listing 18.63

```
<?xml version="1.0"?>
<orderupdate xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:header>
    <updg:param name="OrderID"/>
    <updg:param name="ShipCity"/>
  </updg:header>
  <updg:sync>
    <updg:before>
      <Orders OrderID="$OrderID"/>
    </updg:before>
    <updg:after>
      <Orders ShipCity="$ShipCity"/>
    </updg:after>
  </updg:sync>
</orderupdate>
```

This nuance has interesting implications for passing currency values as parameters. To pass a currency parameter value to a table column (e.g., the Freight column in the Orders table), you must map the data using a mapping schema.

NULL Parameters

In order to pass a parameter with a NULL value to an updategram, include the nullvalue placeholder attribute in the updategram's header element. You can then pass this placeholder value into the updategram to signify a NULL parameter value. This is similar to the way you specify a NULL value for a column in an updategram, the difference being that you specify nullvalue within the sync element for column values but within the header element for parameters. Listing 18.64 shows an example.

Listing 18.64

```
<?xml version="1.0"?>
<orderupdate xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:header nullvalue="NONE">
    <updg:param name="OrderID"/>
```

```
<updg:param name="ShipCity"/>
</updg:header>
  <updg:sync>
    <updg:before>
      <Orders OrderID="$OrderID"/>
    </updg:before>
    <updg:after>
      <Orders ShipCity="$ShipCity"/>
    </updg:after>
  </updg:sync>
</orderupdate>
```

This updategram accepts two parameters. Passing a value of NONE will cause the ShipCity column to be set to NULL for the specified order.

Note that we don't include the xml-updategram (updg:) qualifier when specifying the nullvalue placeholder for parameters in the updategram's header.

Multiple Rows

I mentioned earlier that each before element can identify at most one row. This means that to update multiple rows, you must include an element for each row you wish to change.

The id Attribute

When you specify multiple subelements within your before and after elements, SQL Server requires that you provide a means of matching each before element with its corresponding after element. One way to do this is through the id attribute. The id attribute allows you to specify a unique string value that you can use to match a before element with an after element. Listing 18.65 gives an example.

Listing 18.65

```
<?xml version="1.0"?>
<orderupdate xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync>
    <updg:before>
```

```
      <Orders updg:id="ID1" OrderID="10248"/>
      <Orders updg:id="ID2" OrderID="10249"/>
    </updg:before>
    <updg:after>
      <Orders updg:id="ID2" ShipCity="Munster"/>
      <Orders updg:id="ID1" ShipCity="Reims"/>
    </updg:after>
  </updg:sync>
</orderupdate>
```

Here, we use the updg:id attribute to match up subelements in the before and after elements. Even though these subelements are specified out of sequence, SQL Server is able to apply the updates to the correct rows.

Multiple before and after Elements

Another way to do this is to specify multiple before and after elements rather than multiple subelements. For each row you want to change, you specify a separate before/after element pair, as demonstrated in Listing 18.66.

Listing 18.66

```
<?xml version="1.0"?>
<orderupdate xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync>
    <updg:before>
      <Orders OrderID="10248"/>
    </updg:before>
    <updg:after>
      <Orders ShipCity="Reims"/>
    </updg:after>
    <updg:before>
      <Orders OrderID="10249"/>
    </updg:before>
    <updg:after>
      <Orders ShipCity="Munster"/>
    </updg:after>
  </updg:sync>
</orderupdate>
```

As you can see, this updategram updates two rows. It includes a separate before/after element pair for each update.

Results

The result returned to a client application that executes an updategram is normally an XML document containing the empty root element specified in the updategram. For example, we would expect to see this result returned by the orderupdate updategram:

```
<?xml version="1.0"?>
<orderupdate xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
</orderupdate>
```

Any errors that occur during updategram execution are returned as <?MSSQLError> elements within the updategram's root element.

Identity Column Values

In real applications, you often need to be able to retrieve an identity value that's generated by SQL Server for one table and insert it into another. This is especially true when you need to insert data into a table whose primary key is an identity column and a table that references this primary key via a foreign key constraint. Take the example of inserting orders in the Northwind Orders and Order Details tables. As its name suggests, Order Details stores detail information for the orders in the Orders table. Part of Order Details' primary key is the Orders table's OrderID column. When we insert a new row into the Orders table, we need to be able to retrieve that value and insert it into the Order Details table.

From Transact-SQL, we'd usually handle this situation with an INSTEAD OF insert trigger or a stored procedure. To handle it with an updategram, we use the at-identity attribute. Similarly to the id attribute, at-identity serves as a placeholder—everywhere we use its value in the updategram, SQL Server supplies the identity value for the corresponding table. (Each table can have just one identity column.) Listing 18.67 shows an example.

Listing 18.67

```
<?xml version="1.0"?>
<orderinsert xmlns:updg=
```

```
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync>
    <updg:before>
    </updg:before>
    <updg:after>
      <Orders updg:at-identity="ID" ShipCity="Reims"/>
      <Order_x0020_Details OrderID="ID" ProductID="11"
        UnitPrice="$16.00" Quantity="12"/>
      <Order_x0020_Details OrderID="ID" ProductID="42"
        UnitPrice="$9.80" Quantity="10"/>
    </updg:after>
  </updg:sync>
</orderinsert>
```

Here, we use the string "ID" to signify the identity column in the Orders table. Once the string is assigned, we can use it in the insertions for the Order Details table.

In addition to being able to use an identity column value elsewhere in an updategram, it's quite likely that you'll want to be able to return it to the client. To do this, use the after element's returnid attribute and specify the at-identity placeholder as its value, as shown in Listing 18.68.

Listing 18.68

```
<?xml version="1.0"?>
<orderinsert xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync>
    <updg:before>
    </updg:before>
    <updg:after updg:returnid="ID">
      <Orders updg:at-identity="ID" ShipCity="Reims"/>
      <Order_x0020_Details OrderID="ID" ProductID="11"
        UnitPrice="$16.00" Quantity="12"/>
      <Order_x0020_Details OrderID="ID" ProductID="42"
        UnitPrice="$9.80" Quantity="10"/>
    </updg:after>
  </updg:sync>
</orderinsert>
```

Executing this updategram will return an XML document that looks like this:

```
<?xml version="1.0"?>
<orderinsert xmlns:updg=
    "urn:schemas-microsoft-com:xml-updategram">
  <returnid>
    <ID>10248</ID>
  </returnid>
</orderinsert>
```

Globally Unique Identifiers

It's not unusual to see Globally Unique Identifiers (GUIDs) used as key values across a partitioned view or other distributed system. (These are stored in columns of type uniqueidentifier.) Normally, you use the Transact-SQL NEWID() function to generate new uniqueidentifiers. The updategram equivalent of NEWID() is the guid attribute. You can specify the guid attribute to generate a GUID for use elsewhere in a sync element. As with id, nullvalue, and the other attributes presented in this section, the guid attribute establishes a placeholder that you can then supply to other elements and attributes in the updategram in order to use the generated GUID. Listing 18.69 presents an example.

Listing 18.69

```
<orderinsert>
  xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
  <updg:sync>
    <updg:before>
    </updg:before>
    <updg:after>
      <Orders updg:guid="GUID">
        <OrderID>GUID</OrderID>
        <ShipCity>Reims</ShipCity>
      </Orders>
      <Order_x0020_Details OrderID="GUID" ProductID="11"
        UnitPrice="$16.00" Quantity="12"/>
      <Order_x0020_Details OrderID="GUID" ProductID="42"
        UnitPrice="$9.80" Quantity="10"/>
    </updg:after>
  </updg:sync>
</orderinsert>
```

XML Bulk Load

As we saw in the earlier discussions of updategrams and OPENXML, inserting XML data into a SQL Server database is relatively easy. However, both of these methods of loading data have one serious drawback: They're not suitable for loading large amounts of data. In the same way that using the Transact-SQL INSERT statement is suboptimal for loading large numbers of rows, using updategrams and OPENXML to load large volumes of XML data into SQL Server is slow and resource intensive.

SQLXML provides a facility intended specifically to address this problem. Called the XML Bulk Load component, it is a COM component you can call from OLE Automation–capable languages and tools such as Visual Basic, Delphi, and even Transact-SQL. It presents an object-oriented interface to loading XML data in bulk in a manner similar to the Transact-SQL BULK INSERT command.

Architecturally, XML Bulk Load is an in-process COM component named SQLXMLBulkLoad that resides in a DLL named XBLKLD*n*.DLL. When it bulk loads data to SQL Server, it does so via the bulk load interface of SQL Server's SQLOLEDB native OLE DB provider. If you have a Profiler trace running while the bulk load is occurring, you'll see an INSERT BULK language event show up in the trace. INSERT BULK is indicative of a special TDS packet type designed especially for bulk loading data. It's neither a true language event nor an RPC event; instead, it is a distinct type of data packet that bulk load facilities send to the server when they want to initiate a bulk copy operation.

Using the Component

The first step in using the XML Bulk Load component is to define a mapping schema that maps the XML data you're importing to tables and columns in your database. When the component loads your XML data, it will read it as a stream and use the mapping schema to decide where the data goes in the database.

The mapping schema determines the scope of each row added by the Bulk Load component. As the closing tag for each row is read, its corresponding data is written to the database.

You access the Bulk Load component itself via the SQLXMLBulkLoad interface on the SQLXMLBulkLoad COM object. The first step in using it is to connect to the database using an OLE DB connection string or by setting its ConnectionCommand property to an existing ADO Command object. The

second step is to call its Execute method. The VBScript code in Listing 18.70 illustrates.

Listing 18.70

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

You can also specify an XML stream (rather than a file) to load, making cross-DBMS data transfers (from platforms that feature XML support) fairly easy.

XML Fragments

Setting the XMLFragment property to True allows the Bulk Load component to load data from an XML fragment (an XML document with no root element, similar to the type returned by Transact-SQL's FOR XML extension). Listing 18.71 shows an example.

Listing 18.71

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.XMLFragment = True
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

Enforcing Constraints

By default, the XML Bulk Load component does not enforce check and referential integrity constraints. Enforcing constraints as data is loaded slows down the process significantly, so the component doesn't enforce them unless you tell it to. For example, you might want to do that when you're loading data directly into production tables and you want to ensure that the integrity of your data is not compromised. To cause the component to enforce your constraints as it loads data, set the CheckConstraints property to True, as shown in Listing 18.72.

Listing 18.72

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.CheckConstraints = True
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

Duplicate Keys

Normally you'd want to stop a bulk load process when you encounter a duplicate key. Usually this means you've got unexpected data values or data corruption of some type and you need to look at the source data before proceeding. There are, however, exceptions. Say, for example, that you get a daily data feed from an external source that contains the entirety of a table. Each day, a few new rows show up, but, for the most part, the data in the XML document already exists in your table. Your interest is in loading the new rows, but the external source that provides you the data may not know which rows you have and which ones you don't. They may provide data to lots of companies—what your particular database contains may be unknown to them.

In this situation, you can set the IgnoreDuplicateKeys property before the load, and the component will ignore the duplicate key values it encounters. The bulk load won't halt when it encounters a duplicate key—it will

simply ignore the row containing the duplicate key, and the rows with nonduplicate keys will be loaded as you'd expect. Listing 18.73 shows an example.

Listing 18.73

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.IgnoreDuplicateKeys = True
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

When IgnoreDuplicateKeys is set to True, inserts that would cause a duplicate key will still fail, but the bulk load process will not halt. The remainder of the rows will be processed as though no error occurred.

IDENTITY Columns

SQLXMLBulkLoad's KeepIdentity property is True by default. This means that values for identity columns in your XML data will be loaded into the database rather than being generated on-the-fly by SQL Server. Normally, this is what you'd want, but you can set KeepIdentity to False if you'd rather have SQL Server generate these values.

There are a couple of caveats regarding the KeepIdentity property. First, when KeepIdentity is set to True, SQL Server uses SET IDENTITY_ INSERT to enable identity value insertion into the target table. SET IDENTITY_ INSERT has specific permissions requirements—execute permission defaults to the sysadmin role, the db_owner and db_ddladmin fixed database roles, and the table owner. This means that a user who does not own the target table and who also is not a sysadmin, db_owner, or DDL administrator will likely have trouble loading data with the XML Bulk Load component. Merely having bulkadmin rights is not enough.

Another caveat is that you would normally want to preserve identity values when bulk loading data into a table with dependent tables. Allowing these values to be regenerated by the server could be disastrous—you could break parent-child relationships between tables with no hope of reconstructing them. If a parent table's primary key is its identity column and

KeepIdentity is set to False when you load it, you may not be able to resynchronize it with the data you load for its child table. Fortunately, KeepIdentity is enabled by default, so normally this isn't a concern, but be sure you know what you're doing if you choose to set it to False.

Listing 18.74 illustrates setting the KeepIdentity property.

Listing 18.74

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.KeepIdentity = False
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

Another thing to keep in mind is that KeepIdentity is a very binary option—either it's on or it's not. The value you give it affects every object into which XML Bulk Load inserts rows within a given bulk load. You can't retain identity values for some tables and allow SQL Server to generate them for others.

NULL Values

For a column not mapped in the schema, the column's default value is inserted. If the column doesn't have a default, NULL is inserted. If the column doesn't allow NULLs, the bulk load halts with an error message.

The KeepNulls property allows you to tell the bulk load facility to insert a NULL value rather than a column's default when the column is not mapped in the schema. Listing 18.75 demonstrates.

Listing 18.75

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.KeepNulls = True
```

```
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

Table Locks

As with SQL Server's other bulk load facilities, you can configure SQLXMLBulkLoad to lock the target table before it begins loading data into it. This is more efficient and faster than using more granular locks but has the disadvantage of preventing other users from accessing the table while the bulk load runs. To force a table lock during an XML bulk load, set the ForceTableLock property to True, as shown in Listing 18.76.

Listing 18.76

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.ForceTableLock = True
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

Transactions

By default, XML bulk load operations are not transactional—that is, if an error occurs during the load process, the rows loaded up to that point will remain in the database. This is the fastest way to do things, but it has the disadvantage of possibly leaving a table in a partially loaded state. To force a bulk load operation to be handled as a single transaction, set SQLXMLBulkLoad's Transaction property to True before calling Execute.

When Transaction is True, all inserts are cached in a temporary file before being loaded onto SQL Server. You can control where this file is written by setting the TempFilePath property. TempFilePath has no meaning unless Transaction is True. If TempFilePath is not otherwise set, it defaults to the folder specified by the TEMP environmental variable on the server.

I should point out that bulk loading data within a transaction is much slower than loading it outside of one. That's why the component doesn't

load data within a transaction by default. Also note that you can't bulk load binary XML data from within a transaction.

Listing 18.77 illustrates a transactional bulk load.

Listing 18.77

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.Transaction = True
objBulkLoad.TempFilePath = "c:\temp\xmlswap"
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

In this example, SQLXMLBulkLoad establishes its own connection to the server over OLE DB, so it operates within its own transaction context. If an error occurs during the bulk load, the component rolls back its own transaction.

When SQLXMLBulkLoad uses an existing OLE DB connection via its ConnectionCommand property, the transaction context belongs to that connection and is controlled by the client application. When the bulk load completes, the client application must explicitly commit or roll back the transaction. Listing 18.78 shows an example.

Listing 18.78

```
On Error Resume Next
Err.Clear
Set objCmd = CreateObject("ADODB.Command")
objCmd.ActiveConnection= _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.Transaction = True
objBulkLoad.ConnectionCommand = objCmd
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
```

```
If Err.Number = 0 Then
  objCmd.ActiveConnection.CommitTrans
Else
  objCmd.ActiveConnection.RollbackTrans
End If
Set objBulkLoad = Nothing
Set objCmd = Nothing
```

Note that when using the ConnectionCommand property, Transaction is required—it must be set to True.

Errors

The XML Bulk Copy component supports logging error messages to a file via its ErrorLogFile property. This file is an XML document itself that lists any errors that occurred during the bulk load. Listing 18.79 demonstrates how to use this property.

Listing 18.79

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.ErrorLogFile = "c:\temp\xmlswap\errors.xml"
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

The file you specify will contain a Record element for each error that occurred during the last bulk load. The most recent error message will be listed first.

Generating Database Schemas

In addition to loading data into existing tables, the XML Bulk Copy component can also create target tables for you if they do not already exist, or drop and recreate them if they do exist. To create nonexistent tables, set the component's SchemaGen property to True, as shown in Listing 18.80.

Listing 18.80

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.SchemaGen = True
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

Since SchemaGen is set to True, any tables in the schema that don't already exist will be created when the bulk load starts. For tables that already exist, data is simply loaded into them as it would normally be.

If you set the BulkLoad property of the component to False, no data is loaded. So, if SchemaGen is set to True but BulkLoad is False, you'll get empty tables for those in the mapping schema that did not already exist in the database, but you'll get no data. Listing 18.81 presents an example.

Listing 18.81

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.SchemaGen = True
objBulkLoad.BulkLoad = False
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

When XML Bulk Load creates tables, it uses the information in the mapping schema to define the columns in each table. The sql:datatype annotation defines column data types, and the dt:type attribute further defines column type information. To define a primary key within the mapping schema, set a column's dt:type attribute to id and set the SGUseID property of the XML Bulk Load component to True. The mapping schema in Listing 18.82 illustrates.

Listing 18.82

```
<ElementType name="Orders" sql:relation="Orders">
  <AttributeType name="OrderID" sql:datatype="int" dt:type="id"/>
  <AttributeType name="ShipCity" sql:datatype="nvarchar(30)"/>

  <attribute type="OrderID" sql:field="OrderID"/>
  <attribute type="ShipCity" sql:field="ShipCity"/>
</ElementType>
```

Listing 18.83 shows some VBScript code that sets the SGUseID property so that a primary key will automatically be defined for the table that's created on the server.

Listing 18.83

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.SchemaGen = True
objBulkLoad.SGUseID = True
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

Here's the Transact-SQL that results when the bulk load executes:

```
CREATE TABLE Orders
(
  OrderID int NOT NULL,
  ShipCity nvarchar(30) NULL,
  PRIMARY KEY CLUSTERED (OrderID)
)
```

In addition to being able to create new tables from those in the mapping schema, SQLXMLBulkLoad can also drop and recreate tables. Set the SGDropTables property to True to cause the component to drop and recreate the tables mapped in the schema, as shown in Listing 18.84.

Listing 18.84

```
Set objBulkLoad = CreateObject("SQLXMLBulkLoad.SQLXMLBulkLoad")
objBulkLoad.ConnectionString = _
  "provider=SQLOLEDB;data source=KUFNATHE;database=Northwind;" & _
  "Integrated Security=SSPI;"
objBulkLoad.SchemaGen = True
objBulkLoad.SGDropTables = True
objBulkLoad.Execute "d:\xml\OrdersSchema.xml",
    "d:\xml\OrdersData.xml"
Set objBulkLoad = Nothing
```

Managed Classes

SQLXML provides managed code classes that allow you to retrieve XML data from SQL Server (you can translate the data to XML on the server or at the client). These classes have analogues in the .NET Framework itself but are more geared toward SQLXML and exposing its unique functionality in managed code applications. The SQLXML classes reside in an assembly named Microsoft.Data.SqlXml, and, as with any managed code assembly, they can be accessed from apps written in any CLR-compliant language, including C#, VB.NET, Delphi.NET, and others.

The SqlXmlCommand, SqlXmlParameter, and SqlXmlAdapter classes are the key managed code classes in the SqlXml assembly. As I've mentioned, these are similar to their similarly named counterparts in the .NET Framework. SqlXmlCommand is used to execute T-SQL commands or SQL Server procedural objects and optionally return their results as XML. SqlXmlParameter is used to set up parameterized queries. SqlXmlAdapter is used to process the results from a SqlXmlCommand execution. If the underlying data source supports modification, changes can be made at the client and posted back to the server using diffgrams, specialized updategram-like templates used by the .NET Framework to encapsulate data modifications.

The best way to understand how these classes interoperate in a real application is to build one. The C# example code in the next example demonstrates how to use each of the main SQLXML managed classes to execute a stored procedure and process its result set. Let's begin with the source code for the stored procedure (Listing 18.85).

Listing 18.85

```
USE Northwind
GO
DROP PROC ListCustomers
GO
CREATE PROC ListCustomers @CustomerID nvarchar(10)='%'
AS
PRINT '@CustomerID = ' +@CustomerID

SELECT *
FROM Customers
WHERE CustomerID LIKE @CustomerID

RAISERROR('%d Customers', 1,1, @@ROWCOUNT)
GO
EXEC ListCustomers N'ALFKI'
```

This stored proc takes a single parameter, a customer ID mask, and lists all the rows from the Northwind Customers table that match it. Listing 18.86 shows the C# code that uses SQLXML managed classes to execute the stored proc. (You can find this code in the CH18\managed_classes subfolder on the CD accompanying this book.)

Listing 18.86

```
using System;
using Microsoft.Data.SqlXml;
using System.IO;
using System.Xml;
class CmdExample
{
  static string strConn = "Provider=SQLOLEDB;Data Source='(local)';
      database=Northwind; Integrated Security=SSPI";
  public static int CmdExampleWriteXML()
  {
    XmlReader Reader;
    SqlXmlParameter Param;
    XmlTextWriter TxtWriter;

    //Create a new SqlXmlCommand instance
    SqlXmlCommand Cmd = new SqlXmlCommand(strConn);
```

```
    //Set it up to call our stored proc
    Cmd.CommandText = "EXEC ListCustomersXML ?";

    //Create a parameter and give it a value
    Param = Cmd.CreateParameter();
    Param.Value = "ALFKI";

    //Execute the proc
    Reader = Cmd.ExecuteXmlReader();

    //Create a new XmlTextWriter instance
    //to write to the console
    TxtWriter = new XmlTextWriter(Console.Out);

    //Move to the root element
    Reader.MoveToContent();

    //Write the document to the console
    TxtWriter.WriteNode(Reader, false);

    //Flush the writer and close the reader
    TxtWriter.Flush();
    Reader.Close();

    return 0;
  }
  public static int Main(String[] args)
  {
    CmdExampleWriteXML();
    return 0;
  }
}
```

Note the reference to the Microsoft.Data.SqlXml assembly. You will have to add a reference to this assembly in the Visual Studio .NET IDE (or on the csc.exe command line) in order to compile and link this code.

Let's walk through how this code works. We begin by instantiating a new SqlXmlCommand and passing it our connection string. We then set its CommandText property to call a stored procedure with a replaceable parameter. Next, we create a SqlXmlParameter instance and assign its Value property in order to supply a value for the stored procedure's parameter.

Once the SqlXmlCommand object is properly set up, we call its ExecuteXmlReader method. This returns an XmlReader instance that we can use to process the stored proc's results. We then create an XmlTextWriter object so that we can write out the XML returned by the SqlXmlCommand object. We follow up by moving to the start of the document itself (via the MoveToContent call), then write the entire document to the console via the TxtWriter.WriteNode call. We then conclude by flushing the XmlTextWriter object and closing the XmlReader object that was originally returned by the call to SqlXmlCommand.ExecuteXmlReader.

If you've done much programming with the .NET Framework's ADO.NET and XML classes, this code probably looks very familiar to you. All three SQLXML managed classes have counterparts in the .NET Framework itself. The metaphors are the same. They return compatible types with the base .NET Framework classes where it makes sense and can be used interchangeably with them. Their purpose is to extend the ADO.NET classes to include functionality that's specific to SQLXML, not replace them or offer an alternative to them.

SQLXML Web Service (SOAP) Support

SQLXML's Web service support allows you to expose SQL Server as a Web service. This allows stored procedures, other procedural objects, and query templates to be executed as though they were methods exposed by a traditional SOAP-based Web service. SQLXML provides the plumbing necessary to access SQL Server data using SOAP from any platform or client that can make SOAP requests.

The advantage of this, of course, is that you don't need SQL Server client software to run queries and access SQL Server objects. This means that applications on client platforms not directly supported by SQL Server (e.g., Linux) can submit queries and retrieve results from SQL Server via SQLXML and its SOAP facility.

You set up SQL Server to masquerade as a Web service by configuring a SOAP virtual name in the IIS Virtual Directory Management tool. (You can find this under the SQLXML | Configure IIS menu option under Start | Programs.) A SOAP virtual name is simply a folder associated with an IIS virtual directory name whose type has been set to soap. You can specify whatever service name you like in the Web Service Name text box; the conventional name is soap. Once this virtual name is set up, you configure spe-

cific SQL Server objects to be exposed by the Web service by clicking the Configure button on the Virtual Names tab and selecting the object name, the format of the XML to produce on the middle tier (via SQLISAPI), and the manner in which to expose the object: as a collection of XML elements, as a single Dataset object, or as a collection of Datasets. As the exercise we'll go through in just a moment illustrates, you can expose a given server object multiple times and in multiple ways, providing client applications with a wealth of ways to communicate with SQL Server over SOAP.

Architecturally, SQLXML's SOAP capabilities are provided by its ISAPI extension, SQLISAPI. These capabilities are an extension of the virtual directory concept that you configure in order to access the server via URL queries and templates. The SOAP virtual name that you set up provides access to SQLXML's Web service facility via a URL. It allows any client application that can communicate over SOAP with this URL to access SQL Server objects just as it would any other Web service. Java applications, traditional ADO applications, and, of course, .NET applications can access SQL Server procedural objects and XML templates without using traditional SQL Server client software or communicating over TDS.

In this next exercise, we'll walk through exposing SQL Server as a Web service and then consuming that service in a C# application. We'll set up the SOAP virtual name, then we'll configure a SQL Server procedure object to be exposed as a collection of Web service methods. Finally, we'll build a small application to consume the service and demonstrate how to interact with it.

Exercise 18.4 Building and Consuming a SQLXML Web Service

1. Under the \inetpub\wwwroot\Northwind folder that you created earlier, create a folder named Soap.
2. Start the IIS Virtual Directory Management for SQLXML tool that you used to configure the Northwind virtual folder earlier.
3. Go to the Virtual Names tab and add a new virtual name with a Name, Type, and Web Service Name of soap. Set the path to the folder you created in step 1.
4. Save the virtual name configuration. At this point, the Configure button should be enabled. Click it to begin exposing specific procedural objects and templates via the Web service.
5. Click the ellipsis button to the right of the SP/Template text box and select the ListCustomers stored procedure from the list.
6. Name the method ListCustomers and set its row format to Raw and its output format to XML objects, then click OK.

7. Repeat the process and name the new method ListCustomersAsDataset (you will be referencing the ListCustomers stored procedure). Set its output type to Single dataset, then click OK.
8. Repeat the process again and name the new method ListCustomersAsDatasets. Set its output type to Dataset objects, then click OK. You've just exposed the ListCustomers stored procedure as three different Web service methods using three different output formats. Note that procedural objects you set up this way must not return XML themselves (i.e., they must not use the Transact-SQL FOR XML option) because XML formatting is handled exclusively at the middle tier by SQLISAPI when using the SQLXML Web service facility.
9. Start a new C# Windows application project in Visual Studio .NET. The app we'll build will allow you to invoke the SQLXML Web service facility to execute the ListCustomers stored proc using a specified CustomerID mask.
10. Add a single TextBox control to the upper-left corner of the default form to serve as the entry box for the CustomerID mask.
11. Add a Button control to the right of the TextBox control to be used to execute the Web service method.
12. Add three RadioButton controls to the right of the button to specify which Web method we want to execute. Name the first rbXMLElements, the second rbDataset, and the third rbDatasetObjects. Set the Text property of each control to a brief description of its corresponding Web method (e.g., the Text property for rbXMLElements should be something like "XML Elements").
13. Add a ListBox control below the other controls on the form. This will be used to display the output from the Web service methods we call. Dock the ListBox control to the bottom of the form and be sure it is sized to occupy most of the form.
14. Make sure your instance of IIS is running and accessible. As with the other Web-oriented examples in this chapter, I'm assuming that you have your own instance of IIS and that it's running on the local machine.
15. Right-click your solution in the Solution Explorer and select Add Web Reference. In the URL for the Web reference, type the following:

```
http://localhost/Northwind/soap?wsdl
```

    This URL refers by name to the virtual directory you created earlier, then to the soap virtual name you created under it, and finally to the Web Services Description Language (WSDL) functionality provided by SQLISAPI. As I mentioned earlier, a question mark in a URL denotes the start of the URL's parameters, so wsdl is being passed as a parameter into the SQLISAPI extension DLL. Like XML and SOAP, WSDL is its own W3C standard and describes, in XML, Web services as a set of end

points operating on messages containing either procedural or document-oriented information. You can learn more about WSDL by visiting this link on the W3C Web site: http://www.w3.org/TR/wsdl.

16. Once you've added the Web reference, the localhost Web service will be available for use within your application. A proxy class is created under your application folder that knows how to communicate with the Web service you referenced. To your code, this proxy class looks identical to the actual Web service. When you make calls to this class, they are transparently marshaled to the Web service itself, which might reside on some other machine located elsewhere on the local intranet or on the public Internet. You'll recall from Chapter 6 that I described Windows' RPC facility as working the very same way. Web services are really just an extension of this concept. You work and interoperate with local classes and methods; the plumbing behind the scenes handles getting data to and from the actual implementation of the service without your app even being aware of the fact that it is dealing with any sort of remote resource.
17. Double-click the Button control you added earlier and add to it the code in Listing 18.87.

Listing 18.87

```
int iReturn = 0;
object result;
object[] results;
System.Xml.XmlElement resultElement;
System.Data.DataSet resultDS;
localhost.soap proxy = new localhost.soap();
proxy.Credentials=System.Net.CredentialCache.DefaultCredentials;

// Return ListCustomers as XMLElements
if (rbXMLElements.Checked)
{
  listBox1.Items.Add("Executing ListCustomers...");
  listBox1.Items.Add("");

  results = proxy.ListCustomers(textBox1.Text);

  for (int j=0; j<results.Length; j++)
  {
    localhost.SqlMessage errorMessage;
    result= results[j];
```

```
    if (result.GetType().IsPrimitive)
    {
      listBox1.Items.Add(
        string.Format("ListCustomers return value: {0}", result));
    }
    if (result is System.Xml.XmlElement)
    {
      resultElement = (System.Xml.XmlElement) results[j];
      listBox1.Items.Add(resultElement.OuterXml);
    }
    else if (result is localhost.SqlMessage) {
      errorMessage = (localhost.SqlMessage) results[j];
      listBox1.Items.Add(errorMessage.Message);
      listBox1.Items.Add(errorMessage.Source);
    }
  }
  listBox1.Items.Add("");
}
// Return ListCustomers as Dataset objects
else if (rbDatasetObjects.Checked)
{
  listBox1.Items.Add("Executing ListCustomersAsDatasets...");
  listBox1.Items.Add("");
  results = proxy.ListCustomersAsDatasets(textBox1.Text);

  for (int j=0; j<results.Length; j++)
  {
    localhost.SqlMessage errorMessage;
    result= results[j];

    if (result.GetType().IsPrimitive)
    {
      listBox1.Items.Add(
        string.Format("ListCustomers return value: {0}", result));
    }
    if (result is System.Data.DataSet)
    {
      resultDS = (System.Data.DataSet) results[j];
      listBox1.Items.Add("DataSet " +resultDS.GetXml());
    }
    else if (result is localhost.SqlMessage)
    {
      errorMessage = (localhost.SqlMessage) results[j];
      listBox1.Items.Add("Message " +errorMessage.Message);
```

```
      listBox1.Items.Add(errorMessage.Source);
    }
  }
  listBox1.Items.Add("");
}
// Return ListCustomers as Dataset
else if (rbDataset.Checked)
{
  listBox1.Items.Add("Executing ListCustomersAsDataset...");
  listBox1.Items.Add("");
  resultDS = proxy.ListCustomersAsDataset(textBox1.Text,
      out iReturn);
  listBox1.Items.Add(resultDS.GetXml());
  listBox1.Items.Add(
    string.Format("ListCustomers return value: {0}", iReturn));
  listBox1.Items.Add("");
}
```

18. This code can be divided into three major routines—one each for the three Web service methods we call. Study the code for each type of output format and compare and contrast their similarities and differences. Note the use of reflection in the code to determine what type of object we receive back from Web service calls in situations where multiple types are possible.
19. Compile and run the app. Try all three output formats and try different CustomerID masks. Each time you click your Button control, the following things happen.
    a. Your code makes a method call to a proxy class Visual Studio .NET added to your project when you added the Web reference to the SQLXML SOAP Web service you set up for Northwind.
    b. The .NET Web service code translates your method call into a SOAP call and passes it across the network to the specified host. In this case, your Web service host probably resides on the same machine, but the architecture allows it to reside anywhere on the local intranet or public Internet.
    c. The SQLXML ISAPI extension receives your SOAP call and translates it into a call to the ListCustomers stored procedure in the database referenced by your IIS virtual directory, Northwind.
    d. SQL Server runs the procedure and returns its results as a rowset to SQLISAPI.
    e. SQLISAPI translates the rowset to the appropriate XML format and object based on the way the Web service method you called was configured, then returns it via SOAP to the .NET Framework Web service code running on your client machine.

f. The .NET Framework Web services code translates the SOAP it receives into the appropriate objects and result codes and returns them to your application.

g. Your app then uses additional method calls to extract the returned information as text and writes that text to the ListBox control.

So, there you have it, a basic runthrough of how to use SQLXML's SOAP facilities to access SQL Server via SOAP. As I've said, an obvious application of this technology is to permit SQL Server to play in the Web service space—to interoperate with other Web services without requiring the installation of proprietary client software or the use of supported operating systems. Thanks to SQLXML's Web service facility, anyone who can speak SOAP can access SQL Server. SQLXML's Web service support is a welcome and very powerful addition to the SQL Server technology family.

SQLXML Limitations

SQL Server's XML support has some fundamental limitations that make it difficult to use in certain situations. In this section, we'll explore a couple of these and look at ways to work around them.

sp_xml_concat

Given that sp_xml_preparedocument accepts document text of virtually any length (up to 2GB), you'd think that SQL Server's XML facilities would be able to handle long documents just fine—but that's not the case. Although sp_xml_preparedocument's xmltext parameter accepts text as well as varchar parameters, Transact-SQL doesn't support local text *variables*. About the closest you can get to a local text variable in Transact-SQL is to set up a procedure with a text *parameter*. However, this parameter cannot be assigned to nor can it be the recipient of the text data returned by the READTEXT command. About the only thing you can do with it is insert it into a table.

The problem is painfully obvious when you try to store a large XML document in a table and process it with sp_xml_preparedocument. Once the document is loaded into the table, how do you extract it in order to pass it into sp_xml_preparedocument? Unfortunately, there's no easy way to do so. Since we can't declare local text variables, about the only thing we can do is break the document into multiple 8,000-byte varchar variables and use parameter concatenation when we call sp_xml_preparedocument. This is a ridiculously difficult task, so I've written a stored procedure to do it for you.

It's called sp_xml_concat, and you can use it to process large XML documents stored in a table in a text, varchar, or char column.

The sp_xml_concat procedure takes three parameters: the names of the table and column in which the document resides and an output parameter that returns the document handle as generated by sp_xml_preparedocument. You can take the handle that's returned by sp_xml_concat and use it with OPENXML and sp_xml_unpreparedocument.

The table parameter can be either an actual table or view name or a Transact-SQL query wrapped in parentheses that will function as a derived table. The ability to specify a derived table allows you to filter the table that the procedure sees. So, if you want to process a specific row in the table or otherwise restrict the procedure's view of the table, you can do so using a derived table expression.

Listing 18.88 shows the full source code for sp_xml_concat.

Listing 18.88

```
USE master
GO
IF OBJECT_ID('sp_xml_concat','P') IS NOT NULL
  DROP PROC sp_xml_concat
GO
CREATE PROC sp_xml_concat
  @hdl int OUT,
  @table sysname,
  @column sysname
AS
EXEC('
SET TEXTSIZE 4000
DECLARE
  @cnt int,
  @c nvarchar(4000)
DECLARE
  @declare varchar(8000),
  @assign varchar(8000),
  @concat varchar(8000)

SELECT @c = CONVERT(nvarchar(4000),'+@column+') FROM '+@table+'

SELECT @declare = ''DECLARE'',
       @concat = '''''''''''''''',
       @assign = '''',
       @cnt = 0
```

```
WHILE (LEN(@c) > 0) BEGIN
  SELECT @declare = @declare + '' @c''+CAST(@cnt as nvarchar(15))
      +''nvarchar(4000),'',
    @assign = @assign + ''SELECT @c''+CONVERT(nvarchar(15),@cnt)
        +''= SUBSTRING(' + @column+','+ CONVERT(nvarchar(15),
        1+@cnt*4000)+ '', 4000) FROM '+@table+' '',
    @concat = @concat + ''+@c''+CONVERT(nvarchar(15),@cnt)
  SET @cnt = @cnt+1
  SELECT @c = CONVERT(nvarchar(4000),SUBSTRING('+@column+',
      1+@cnt*4000,4000)) FROM '+@table+'
END

IF (@cnt = 0) SET @declare = ''''
ELSE SET @declare = SUBSTRING(@declare,1,LEN(@declare)-1)

SET @concat = @concat + ''+''''''''''''''

EXEC(@declare+'' ''+@assign+'' ''+
''EXEC(
''''DECLARE @hdl_doc int
  EXEC sp_xml_preparedocument @hdl_doc OUT, ''+@concat+''
    DECLARE hdlcursor CURSOR GLOBAL FOR SELECT @hdl_doc AS
        DocHandle''''')''
)
')
OPEN hdlcursor
FETCH hdlcursor INTO @hdl
DEALLOCATE hdlcursor
GO
```

This procedure dynamically generates the necessary DECLARE and SELECT statements to break up a large text column into nvarchar(4000) pieces (e.g., DECLARE @c1 nvarchar(4000) SELECT @c1= …). As it does this, it also generates a concatenation expression that includes all of these variables (e.g., @c1+@c2+@c3, …). Since the EXEC() function supports concatenation of strings up to 2GB in size, we pass this concatenation expression into it dynamically and allow EXEC() to perform the concatenation on-the-fly. This basically reconstructs the document that we extracted from the table. This concatenated string is then passed into sp_xml_preparedocument for processing. The end result is a document handle that you can use with OPENXML. Listing 18.89 shows an example.

(You'll find the full test query in the CH18 subfolder on the CD accompanying this book.)

Listing 18.89

(Code abridged)

```
USE Northwind
GO
CREATE TABLE xmldoc
(id int identity,
  doc text)
INSERT xmldoc VALUES('<Customers>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
  <Order CustomerID="VINET" EmployeeID="5" OrderDate=
      "1996-07-04T00:00:00">
    <OrderDetail OrderID="10248" ProductID="11" Quantity="12"/>
    <OrderDetail OrderID="10248" ProductID="42" Quantity="10"/>
// More code lines here...
  </Order>
</Customer>
<Customer CustomerID="LILAS" ContactName="Carlos GOnzlez">
  <Order CustomerID="LILAS" EmployeeID="3" OrderDate=
      "1996-08-16T00:00:00">
    <OrderDetail OrderID="10283" ProductID="72" Quantity="3"/>
  </Order>
</Customer>
</Customers>')

DECLARE @hdl int
EXEC sp_xml_concat @hdl OUT, '(SELECT doc FROM xmldoc WHERE id=1)
    a', 'doc'

SELECT * FROM OPENXML(@hdl, '/Customers/Customer') WITH
    (CustomerID nvarchar(50))

EXEC sp_xml_removedocument @hdl
SELECT DATALENGTH(doc) from xmldoc
GO
DROP TABLE xmldoc
```

(Results)

```
CustomerID
--------------------------------------------------
VINET
LILAS

-----------
36061
```

Although I've abridged the XML document in the test query, the one on the CD is over 36,000 bytes in size, as you can see from the result of the DATALENGTH() query at the end of the test code.

We pass a derived table expression into sp_xml_concat along with the column name we want to extract, and the procedure does the rest. It's able to extract the nodes we're searching for, even though one of them is near the end of a fairly large document.

sp_run_xml_proc

Another limitation of SQL Server's XML support exists because XML results are not returned as traditional rowsets. Returning XML results as streams has many advantages, but one of the disadvantages is that you can't call a stored procedure that returns an XML result using a four-part name or OPENQUERY() and get a useful result. The result set you'll get will be an unrecognizable binary result set because SQL Server's linked server architecture doesn't support XML streams.

You'll run into similar limitations if you try to insert the result of a FOR XML query into a table or attempt to trap it in a variable—SQL Server simply won't let you do either of these. Why? Because the XML documents returned by SQL Server are not traditional rowsets.

To work around this, I've written a stored procedure named sp_run_xml_proc. You can use it to call linked server stored procedures (it needs to reside on the linked server) that return XML documents as well as local XML procedures whose results you'd like to store in a table or trap in a variable. This procedure does its magic by opening its own connection into the server (it assumes Windows Authentication is being used) and running your procedure. Once your procedure completes, sp_run_xml_proc processes the XML stream it returns using SQL-DMO calls, then translates it into a traditional rowset and returns that rowset. This result set can be inserted into a table or processed further just like any other result set. Listing 18.90 presents the source code for sp_run_xml_proc.

Listing 18.90

```
USE master
GO
IF OBJECT_ID('sp_run_xml_proc','P') IS NOT NULL
  DROP PROC sp_run_xml_proc
GO
CREATE PROC sp_run_xml_proc
  @procname sysname  -- Proc to run
AS

DECLARE @dbname sysname,
  @sqlobject int,   -- SQL Server object
  @object int,   -- Work variable for accessing COM objects
  @hr int,   -- Contains HRESULT returned by COM
  @results int,   -- QueryResults object
  @msgs varchar(8000)   -- Query messages

IF (@procname='/?') GOTO Help

-- Create a SQLServer object
EXEC @hr=sp_OACreate 'SQLDMO.SQLServer', @sqlobject OUT
IF (@hr <> 0) BEGIN
  EXEC sp_displayoaerrorinfo @sqlobject, @hr
  RETURN
END

-- Set SQLServer object to use a trusted connection
EXEC @hr = sp_OASetProperty @sqlobject, 'LoginSecure', 1
IF (@hr <> 0) BEGIN
  EXEC sp_displayoaerrorinfo @sqlobject, @hr
  RETURN
END

-- Turn off ODBC prefixes on messages
EXEC @hr = sp_OASetProperty @sqlobject, 'ODBCPrefix', 0
IF (@hr <> 0) BEGIN
  EXEC sp_displayoaerrorinfo @sqlobject, @hr
  RETURN
END

-- Open a new connection (assumes a trusted connection)
EXEC @hr = sp_OAMethod @sqlobject, 'Connect', NULL, @@SERVERNAME
IF (@hr <> 0) BEGIN
```

```
  EXEC sp_displayoaerrorinfo @sqlobject, @hr
  RETURN
END

-- Get a pointer to the SQLServer object's Databases collection
EXEC @hr = sp_OAGetProperty @sqlobject, 'Databases', @object OUT
IF @hr <> 0 BEGIN
  EXEC sp_displayoaerrorinfo @sqlobject, @hr
  RETURN
END

-- Get a pointer from the Databases collection for the
-- current database
SET @dbname=DB_NAME()
EXEC @hr = sp_OAMethod @object, 'Item', @object OUT, @dbname
IF @hr <> 0 BEGIN
  EXEC sp_displayoaerrorinfo @object, @hr
  RETURN
END

-- Call the Database object's ExecuteWithResultsAndMessages2
-- method to run the proc
EXEC @hr = sp_OAMethod @object, 'ExecuteWithResultsAndMessages2',
    @results OUT, @procname, @msgs OUT
IF @hr <> 0 BEGIN
  EXEC sp_displayoaerrorinfo @object, @hr
  RETURN
END

-- Display any messages returned by the proc
PRINT @msgs

DECLARE @rows int, @cols int, @x int, @y int, @col varchar(8000),
    @row varchar(8000)

-- Call the QueryResult object's Rows method to get the number of
-- rows in the result set
EXEC @hr = sp_OAMethod @results, 'Rows',@rows OUT
IF @hr <> 0 BEGIN
  EXEC sp_displayoaerrorinfo @object, @hr
  RETURN
END

-- Call the QueryResult object's Columns method to get the number
-- of columns in the result set
```

```
EXEC @hr = sp_OAMethod @results, 'Columns',@cols OUT
IF @hr <> 0 BEGIN
  EXEC sp_displayoaerrorinfo @object, @hr
  RETURN
END

DECLARE @table TABLE (XMLText varchar(8000))

-- Retrieve the result set column-by-column using the
-- GetColumnString method
SET @y=1
WHILE (@y<=@rows) BEGIN
  SET @x=1
  SET @row=''
  WHILE (@x<=@cols) BEGIN
      EXEC @hr = sp_OAMethod @results, 'GetColumnString',
          @col OUT, @y, @x
      IF @hr <> 0 BEGIN
        EXEC sp_displayoaerrorinfo @object, @hr
        RETURN
      END
    SET @row=@row+@col+' '
    SET @x=@x+1
  END
  INSERT @table VALUES (@row)
  SET @y=@y+1
END

SELECT * FROM @table

EXEC sp_OADestroy @sqlobject    -- For cleanliness

RETURN 0

Help:
PRINT 'You must specify a procedure name to run'
RETURN -1

GO
```

Although the prospect of having to open a separate connection into the server in order to translate the document is not particularly exciting, it is unfortunately the only way to do this without resorting to client-side

processing—at least for now. The test code in Listing 18.91 shows how to use sp_run_xml_proc.

Listing 18.91

```
USE pubs
GO
DROP PROC testxml
GO
CREATE PROC testxml as
PRINT 'a message here'
SELECT * FROM pubs..authors FOR XML AUTO
GO
EXEC [TUK\PHRIP].pubs.dbo.sp_run_xml_proc 'testxml'
```

(Results abridged)

```
a message here
XMLText
----------------------------------------------------------------
<pubs..authors au_id="172-32-1176" au_lname="White" au_fname="John
<pubs..authors au_id="672-71-3249" au_lname="Yokomoto" au_fname="A
```

Although I've clipped the resulting document considerably, if you run this code from Query Analyzer (replace the linked server reference in the example with your own), you'll see that the entire document is returned as a result set. You can then insert this result set into a table using INSERT…EXEC for further processing. For example, you could use this technique to assign the document that's returned to a variable (up to the first 8,000 bytes) or to change it in some way using Transact-SQL. And once the document is modified to your satisfaction, you could call sp_xml_concat (listed earlier in the chapter) to return a document handle for it so that you can query it with OPENXML. Listing 18.92 does just that.

Listing 18.92

```
SET NOCOUNT ON
GO
USE pubs
```

```
GO
DROP PROC testxml
GO
CREATE PROC testxml as
SELECT au_lname, au_fname FROM authors FOR XML AUTO
GO

CREATE TABLE #XMLText1
(XMLText varchar(8000))
GO

-- Insert the XML document into a table
-- using sp_run_xml_proc
INSERT #XMLText1
EXEC sp_run_xml_proc 'testxml'

-- Put the document in a variable
-- and add a root element
DECLARE @doc varchar(8000)
SET @doc=''
SELECT @doc=@doc+XMLText FROM #XMLText1
SET @doc='<root>'+@doc+'</root>'

-- Put the document back in a table
-- so that we can pass it into sp_xml_concat
SELECT @doc AS XMLText INTO #XMLText2

GO
DECLARE @hdl int
EXEC sp_xml_concat @hdl OUT, '#XMLText2', 'XMLText'
SELECT * FROM OPENXML(@hdl, '/root/authors') WITH
    (au_lname nvarchar(40))
EXEC sp_xml_removedocument @hdl
GO
DROP TABLE #XMLText1, #XMLText2
```

After the document is returned by sp_run_xml_proc and stored in a table, we load it into a variable, wrap it in a root element and store it in a second table so that we may pass it into sp_xml_concat. Once sp_xml_concat

returns, we pass the document handle it returns into OPENXML and extract part of the document:

(Results abridged)

```
au_lname
----------------------------------------
Bennet
Blotchet-Halls
Carson
DeFrance
...
Ringer
Ringer
Smith
Straight
Stringer
White
Yokomoto
```

So, using sp_xml_concat and sp_run_xml_proc in conjunction with SQL Server's built-in XML tools, we're able to run the entire XML processing gamut. We start with an XML fragment returned by FOR XML AUTO, then we store this in a table, retrieve it from the table, wrap it in a root node, and pass it into OPENXML in order to extract a small portion of the original document as a rowset. You should find that these two procedures enhance SQL Server's own XML abilities significantly.

Recap

SQLXML provides a veritable treasure trove of XML-enabled features for SQL Server. You can parse and load XML documents, query them using XPath syntax, query database objects using XPath, and construct templates and mapping schemas to query data. You can use OPENXML, updategrams, and XML Bulk Load to load data into SQL Server via XML, and you can use FOR XML to return SQL Server data as XML. You can access SQL Server via HTTP and SOAP, and you can return XML data to the client via both SQLOLEDB and SQLXMLOLEDB. You can translate a rowset to XML on the server as well as on the client, and you can control the format the generated XML takes through a variety of mechanisms. And when you run into a

couple of the more significant limitations in the SQLXML technologies, you can use the sp_xml_concat and sp_run_xml_proc stored procedures presented in this chapter to work around them.

Knowledge Measure

1. What XML parser does SQL Server's XML features use?
2. True or false: The NESTED option can be used only in client-side FOR XML.
3. What extended stored procedure is used to prepare an XML document for use by OPENXML?
4. What's the theoretical maximum amount of memory that SQLXML will allow MSXML to use from the SQL Server process space?
5. True or false: There is currently no way to disable template caching for a given SQLISAPI virtual directory.
6. Describe the use of the sql:mapping attribute from Microsoft's mapping-schema namespace.
7. Why is the maximum mentioned in question 4 only a theoretical maximum? What other factors could prevent MSXML from reaching its maximum memory allocation ceiling?
8. What XML support file must you first define before bulk loading an XML document into a SQL Server database?
9. What does sql:relationship establish for two tables?
10. Is it possible to change the name of the ISAPI extension DLL associated with a given virtual directory, or must all SQLISAPI-configured virtual directories use the same ISAPI extension?
11. Explain the way that URL queries are handled by SQLXML.
12. True or false: You can return traditional rowsets from SQLXMLOLEDB just as you can from any other OLE DB provider.
13. What Win32 API does SQLXML call in order to compute the amount of physical memory in the machine?
14. Name the two major APIs that MSXML provides for parsing XML documents.
15. Approximately how much larger in memory is a DOM document than the underlying XML document?
16. Describe what a "spec proc" is.
17. What internal spec proc is responsible for implementing the sp_xml_preparedocument extended procedure?

18. What two properties must be set on the ADO Command object in order to allow for client-side FOR XML processing?
19. What method of the ADO Recordset object can persist a recordset as XML?
20. What does the acronym "SAX" stand for in XML parlance?
21. When a standard Transact-SQL query is executed via a URL query, what type of event does it come into SQL Server as?
22. What's the name of the OLE DB provider that implements client-side FOR XML functionality and in what DLL does it reside?
23. Does SQLXML use MSXML to return XML results from server-side FOR XML queries?
24. True or false: SQLXML no longer supports XDR schemas.
25. What component should you use to load XML data into SQL Server in the fastest possible manner?
26. True or false: SQLISAPI does not support returning non-XML data from SQL Server.
27. Is it possible to configure a virtual directory such that FOR XML queries are processed on the client side by default?
28. Approximately how much larger than the actual document is the in-memory representation of an XML document stored by SQLXML for use with OPENXML?
29. True or false: SQLXML does not support inserting new data via OPENXML because OPENXML returns a read-only rowset.
30. What mapping-schema notational attribute should you use with the xsd:relationship attribute if you are using a mapping schema with an updategram and the mapping schema relates two tables in reverse order?
31. Name the central SQL Server error-reporting routine in which we set a breakpoint in this chapter.
32. Describe a scenario in which it would make sense to use a mapping schema with an updategram.
33. What lone value can SQLXMLOLEDB's Data Source parameter have?
34. True or false: The SAX parser is built around the notion of persisting a document in memory in a tree structure so that it is readily accessible to the rest of the application.