



CHAPTER

7

XML Integration

IN THIS CHAPTER:

The Native XML Data Type

XQuery Support

XML Indexes

FOR XML Enhancements

OPENXML Enhancements

XML Bulk Load

Native HTTP SOAP Access

XML Enhancements for Analysis Server

System XML Catalog Views

XML (eXtensible Markup Language) has emerged as one of the most important Internet technologies. XML's flexible text-based structure enables it to be used for an incredibly wide array of network tasks, including data/document transfer, web page rendering, and even as a transport for interapplication remote Procedure Calls (RPC) via SOAP (Simple Object Access Protocol). XML has truly become the lingua franca of computer languages.

Microsoft first added support for XML to SQL Server 2000, starting with support for the FOR XML clause as part of the SELECT statement and the OpenXML function. As XML continued to grow rapidly in enterprise acceptance and usage, Microsoft quickly provided additional functionality by producing a series of web releases. SQL for XML 1.0 added support for UpdateGrams, Templates, and BulkLoad to the base SQL Server 2000 release. The next two web releases, SQLXML 2.0 and SQLXML 3.0, further enhanced the SQL Server 2000 product by adding support for XML Views and SOAP in addition to several other new capabilities. While SQL Server 2000's support for XML provided a great starting point for integrating hierarchical XML documents with SQL Server's relational data, it had some limitations. Once the XML data was stored in a SQL Server database using either the Text or Image data type, there was little that you could do with it. SQL Server 2000 was unable to natively query the hierarchical data that made up the XML document without using complex T-SQL or client-side code.

SQL Server 2005 builds on this starting point by adding support for many new XML features. At a high level, SQL Server 2005 provides a new level of unified storage for XML and relational data. SQL Server 2005 adds a new XML data type that provides support for both native XML queries as well as strong data typing by associating the XML data type to an XSD (Extensible Schema Definition). In addition, it provides bidirectional mapping between relational data and XML data. The XML support is well integrated into the SQL Server 2005 relational database engine, as it provides support for triggers on XML, replication of XML data, and bulk load of XML data, as well as enhanced support for data access via SOAP and many other enhancements. In this chapter you'll get an introduction to the most important new XML features provided by SQL Server 2005.

The Native XML Data Type

Without a doubt the most important XML related enhancement that Microsoft has added to SQL Server 2005 is support for a new native XML data type. The XML data type, literally named XML can be used as a column in a table or a variable or

parameter in a stored procedure. It can be used to store both typed and untyped data. If the data stored in an XML column has no XSD schema, then it is considered untyped. If there is an associated XSD schema, then SQL Server 2005 will check the schema to make sure that the data store complies with the schema definition. In all cases, SQL Server 2005 checks data that is stored in the XML data type to ensure that the XML document is well formed. If the data is not well formed, SQL Server 2005 will raise an error and the data will not be stored. The XML data type can accept a maximum of 2GB of data and is stored like the varbinary(max) data type. The following listing illustrates creating a simple table that uses the new XML data type for one of its columns.

```
CREATE TABLE MyXMLDocs
  (DocID INT PRIMARY KEY IDENTITY,
   MyXmlDoc XML)
```

The most important thing to note in this example is the definition of the MyXmlDoc column, which uses the data type of XML to specify that the column will store XML data. You can store XML data into an XML column using the standard T-SQL INSERT statement. The following example shows how you can populate an XML column using a simple INSERT statement:

```
INSERT INTO MyXmlDocs Values
  ('<MyXMLDoc>
   <DocumentID>1</DocumentID>
   <DocumentText>Text</DocumentText>
  </MyXMLDoc>')
```



NOTE

One important point to notice here is that because the XML data is untyped, any valid XML document can be inserted into the XML data type.

Strongly Typed XML Data Types

The native XML data type checks to ensure that any data that's stored in an XML variable or column is a valid XML document. On its own, it doesn't check any more than that. However, Microsoft designed the XML data type to be able to support more sophisticated document validation using an XSD schema. When an XSD schema is defined for an XML data type column, the SQL Server engine will check to make sure that all of the data that is stored in the XML column complies with the definition that's contained in the XSD schema.

The following listing shows a sample XSD schema for the simple XML document that was used in the preceding example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="MyXMLDocSchema"
xmlns="MyXMLDocSchema">
  <xs:element name="MyXMLDoc">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="DocumentID" type="xs:string" />
        <xs:element name="DocumentBody" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This XSD schema uses the namespace of MyXMLDocSchema and defines an XML document that has a complex element named MyXMLDoc. The MyXMLDoc complex element contains two simple elements. The first simple element must be named DocumentID, and a second simple element is named DocumentBody. Both elements contain XML string-type data.

To create a strongly typed XML column or variable, you first need to register the XSD schema with SQL Server using the CREATE XMLSCHEMA T-SQL DDL statement. The following listing shows how you combine the CREATE XML SCHEMA COLLECTION statement with the sample MyXMLDocSchema to register the schema with the SQL Server 2005 database:

```
CREATE XML SCHEMA COLLECTION MyXMLDocSchema AS
N'<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="http://MyXMLDocSchema">
  <xs:element name="MyXMLDoc">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="DocumentID" type="xs:string" />
        <xs:element name="DocumentBody" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>'
```

The CREATE XML SCHEMA COLLECTION DDL statement takes a single argument that names the collection. Next, after the AS clause it expects a valid XSD

schema enclosed in single quotes. If the schema is not valid, an error will be issued when the statement is executed. The CREATE XML SCHEMA COLLECTION statement is database specific, and the schema that is registered can be accessed only in the database for which the schema is registered.

Once you've registered the XML schema with SQL Server 2005, you can go ahead and associate XML variables and columns with that schema. Doing so ensures that any XML documents that are contained in those variables or columns will adhere to the definition provided by the associated schema. The following example illustrates how you can create a table that uses a strongly typed XML column:

```
CREATE TABLE MyXMLDocs
  (DocID INT PRIMARY KEY IDENTITY,
   MyXmlDoc XML(MyXMLDocSchema))
```

Here you can see that the MyXMLDocs table is created using the CREATE TABLE statement much as in the preceding example. In this case, however, the MyXMLDoc column is created using an argument that specifies that name of the registered XSD schema definition. If you refer to the earlier listing, you can see that the schema was registered using the name MyXMLDocSchema. After the MyXMLDoc column has been associated with the schema that was registered, any data that's inserted into this column will be strongly typed according to the schema definition and any attempt to insert data that doesn't match the schema definition will be rejected. The following listing illustrates an INSERT statement that can add data to the strongly typed MyXMLDoc column:

```
INSERT INTO MyXMLDocs Values
  ('<MyXMLDoc xmlns="http://MyXMLDocSchema">
   <DocumentID>1</DocumentID>
   <DocumentBody>"My text"</DocumentBody>
  </MyXMLDoc>')
```



NOTE

Because this example uses a typed XML data type, the data must conform to the definition provided by the associated XSD schema.

In this case, the XML document must reference the associated XML namespace `http://MyXMLDocSchema`. And the XML document must contain a complex element named `MyXMLDoc`, which in turn contains the `DocumentID` and `DocumentBody` elements. The SQL Server engine will reject any attempt to insert any other XML documents into the `MyXMLDocs` column. If the data does not conform to the

supplied XSD schema, SQL Server will return an error message like the one shown in the following listing:

```
Msg 6965, Level 16, State 1, Line 1
XML Validation: Invalid content, expected
element(s):MyXMLDocSchema:DocumentID where element 'MyXMLDocSchema:Do' was
specified
```



NOTE

As you might expect from their dependent relationship, if you assign a schema to a column in a table, that table must be altered or dropped before that schema definition can be updated.

Retrieving a Registered XML Schema

Once you import a schema using CREATE XML SCHEMA COLLECTION, the schema components are stored in SQL Server's metadata. The stored schema can be listed by querying the sys.xml_namespaces system view, as you can see in the following example:

```
SELECT * FROM sys.xml_namespaces
```

This statement will return a result set showing all of the registered schemas in a database like the one that follows:

xml_collection_id	name	xml_namespace_id
1	http://www.w3.org/2001/XMLSchema	1
65540	http://MyXMLDocSchema	1

(2 row(s) affected)

You can also use the new XML_SCHEMA_NAMESPACE function to retrieve the XML schema. The following query retrieves a schema from the database for a given namespace.

```
SELECT XML_SCHEMA_NAMESPACE(N'dbo', N'MyXMLDocSchema')
```

This statement will return a result set showing all of the columns that use the registered schema, as you can see in the following listing:

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
targetNamespace="http://MyXMLDocSchema" xmlns:t=http://MyXMLDocSchema
```

```
elementFormDefault="qualified"><xsd:elementname="MyXMLDoc">
<xsd:complexType><xsd:complexContent><xsd:restriction base="xsd:any
```

```
(1 row(s) affected)
```

The `elementname` attribute lists the columns that use the typed XML data type.

XML Data Type Methods

SQL Server 2005 provides several new built-in methods that work much like user-defined types for working with the XML data type. These methods enable you to drill down into the content of XML documents that are stored using the XML data type. On its own, just being able to store XML data in SQL Server has limited value. To really facilitate deep XML integration, you also need a way to query and manipulated the data that's stored using the new XML data type, and that's just what the following XML data type methods enable.

Exists(XQuery, [node ref])

The XML data type's `Exists` method enables you to check the contents of an XML document for the existence of elements or attributes using an XQuery expression. (More information about the XQuery language is presented in the next section.) The following listing shows how to use the XML data type's `Exists` method:

```
SELECT * FROM MyXMLDocs
WHERE MyXmlDoc.exist('declare namespace xd=http://MyXMLDocSchema
/xd:MyXMLDoc[xd:DocumentID eq "1"]') = 1
```

The first parameter of the XML `Exists` method is required and takes an XQuery expression. The second parameter is optional and specifies a node reference within the XML document. Here the XQuery tests for a `DocumentID` element equal to a value of 1. A namespace is declared because the `MyXMLDoc` column has an associated schema. The `Exists` method can return the value of `TRUE` (1) if the XQuery expressions returns a node, `FALSE` (0) if the expression doesn't return an XML node, or `NULL` if the XML data type instance is null. You can see the results of the XML `Exists` method here:

```
DocID          MyXmlDoc
-----
1              <MyXMLDocxmlns="http://MyXMLDocSchema"><DocumentID>1
</DocumentID> <DocumentBody>"My text"</DocumentBody></MyXMLDoc>
(1 row(s) affected)
```

Modify(XML DML)

As you might guess, the XML data type's Modify method enables you to modify a stored XML document. You can use the Modify method either to update the entire XML document or to update just a selected part of the document. You can see an example of using the Modify method in the following listing:

```
UPDATE MyXMLDocs
SET MyXMLDoc.modify('declare namespace xd=http://MyXMLDocSchema
    replace value of (/xd:MyXMLDoc/xd:DocumentBody)[1] with "My New Body"')
WHERE DocID = 1
```

The XML data type's Modify method uses an XML Data Modification Language (XML DML) statement as its parameter. XML DML is a Microsoft extension to the XQuery language that enables modification of XML documents. The XQuery dialect supports the Replace value of, Insert, and Delete XML DML statements. In this example, since the MyXMLDoc XML column is typed, the XML DML statement must specify the namespace for the schema. Next, you can see where the Replace value of XML DML command is used to replace the value of the DocumentBody element with the new value of "My New Body" for the row where the relational DocID column is equal to one. The replace value of clause must identify only a single node, or it will fail. Therefore the first node is identified using the [1] notation.



NOTE

While this example illustrates performing a replace operation, the Modify method also supports insert and delete operations.

Query(XQuery, [node ref])

The XML data type's Query method can retrieve either the entire contents of an XML document or a selected section of the XML document. You can see an example of using the Query method in the following listing:

```
SELECT DocID, MyXMLDoc.query('declare namespace xd=http://MyXMLDocSchema
    /xd:MyXMLDoc/xd:DocumentBody') AS Body
FROM MyXMLDocs
```

This XQuery expression returns the values from the XML document's DocumentBody element. Again, the namespace is specified because the MyXMLDoc Data type has an associated schema named MyXMLDocSchema. In this example, you can see how SQL Server 2005 easily integrates relational column data with XML data. Here, DocID comes from a relational column, while the DocumentBody element is queried out of the XML column. The following listing shows the results of the XQuery:

```
DocID Body
-----
1      <xd:DocumentBody xmlns:xd="http://MyXMLDocSchema">My New Body</xd:DocumentBody>
2      <xd:DocumentBody xmlns:xd="http://MyXMLDocSchema">My 2nd text</xd:DocumentBody>

(2 row(s) affected)
```

Value(XQuery, [node ref])

The Value method enables the extraction of scalar values from an XML data type. You can see an example of how the XML data type's Value method is used in the following listing:

```
SELECT MyXMLDoc.value('declare namespace xd=http://MyXMLDocSchema
    (/xd:MyXMLDoc/xd:DocumentID)[1]', 'int') AS ID
FROM MyXMLDocs
```

Unlike the other XML data type methods, the XML Value method requires two parameters. The first parameter is an XQuery expression, and the second parameter specifies the SQL data type that will hold the scalar value returned by the Value method. This example returns all of the values contained in the DocumentID element and converts them to the int data type, as shown in the following results:

```
ID
-----
1
2

(2 row(s) affected)
```

XQuery Support

In the previous section you saw how XQuery is used in the new XML data type's methods. XQuery is based on the XPath language created by the W3C (www.w3c.org) for querying XML data. XQuery extends the XPath language by adding the ability to update data as well as support for better iteration and sorting of results. At the time of this writing, the XQuery language used by SQL Server 2005 is an early implementation based on a working draft of the XQuery standard submitted to the W3C, so it's possible that some implementation details could change before SQL Server 2005 is officially released. A description of the XQuery language is beyond the scope of this book, but for more details about the W3C XQuery standard you can refer to <http://www.w3.org/XML/Query>. The SQL Server 2005 Books Online also has an introduction to the XQuery language.

XML Indexes

The XML data type supports a maximum of 2GB of storage, which is quite large. The size of the XML data and its usage can have a big impact on the performance the system can achieve while querying the XML data. To improve the performance of XML queries, SQL Server 2005 provides the ability to create indexes over the columns that have the XML data type.

Primary XML Indexes

In order to create an XML index on an XML data type column, a clustered primary key must exist for the table. In addition, if you need to change the primary key for the table you must first delete the XML index. An XML index covers all the elements in the XML column, and you can have only one XML index per column. Because XML indexes use the same namespace as regular SQL Server relational indexes, XML indexes cannot have the same name as an existing index. XML indexes can be created only on XML data types in a table. They cannot be created on columns in views or on XML data type variables. A primary XML index consists of a persistent shredded representation of the data in the XML column. The code to create a primary XML index is shown in the following listing:

```
CREATE PRIMARY XML INDEX MyXMLDocsIdx ON MyXMLDocs(MyXMLDoc)
```

This example shows the creation of a primary XML index named `MyXMLDocsIdx`. This index is created on the `MyXMLDoc` XML data type column in the `MyXMLDocs` table. Just like regular SQL Server indexes, XML indexes can be viewed by querying the `sys.indexes` view.

```
SELECT * FROM sys.indexes WHERE name = 'MyXMLDocsIdx'
```

Secondary XML Indexes

In addition to the primary index, you can also build secondary XML indexes. Secondary indexes are built on one of the following document attributes:

- ▶ **Path** The document path is used to build the index.
- ▶ **Value** The document values are used to build the index
- ▶ **Property** The documents properties are used to build the index

Secondary indexes are always partitioned in the same way as the primary XML index. The following listing shows the creation of a secondary-path XML index:

```
CREATE XML INDEX My2ndXMLDocsIdx ON MyXMLDocs(MyXMLDoc)
USING XML INDEX MyXMLDocsIdx FOR PATH
```

FOR XML Enhancements

The FOR XML clause was first introduced to the T-SQL SELECT statement in SQL Server 2000. It has been enhanced in SQL Server 2005. Some of the new capabilities that are found in the FOR XML support in SQL Server 2005 include support for the XML data type, user-defined data types, the timestamp data type, and enhanced support for string data. In addition, the FOR XML enhancements also include support for a new Type directive, nested FOR XML queries, and inline XSD schema generation.

Type Directive

When XML data types are returned using the FOR XML clauses' Type directive, they are not serialized. Instead the results are returned as an XML data type. You can see an example of using the FOR XML clause with the XML Type directive here:

```
SELECT DocID, MyXMLDoc FROM MyXMLDocs
WHERE DocID=1 FOR XML AUTO, TYPE
```

This query returns the relational DocID column along with the MyXMLDoc XML data type column. It uses the FOR XML AUTO clause to return the results as XML. The TYPE directive specifies that the results will be returned as an XML data type. You can see the results of using the Type directive here:

```
<MyXMLDocs DocID="1" >
  <MyXMLDoc>
    <MyXMLDoc xmlns="MyXMLDocSchema">
      <DocumentID>1</DocumentID>
      <DocumentBody>My New Body</DocumentBody>
    </MyXMLDoc>
  </MyXMLDoc>
</MyXMLDocs>
```



NOTE

The Type directive returns the XML data type as a continuous stream. I added the formatting to the previous listing to make it more readable.

Nested FOR XML Queries

SQL Server 2000 was limited to using the FOR XML clause in the top level of a query. Subqueries couldn't make use of the FOR XML clause. SQL Server 2005 adds the ability to use nested FOR XML queries. Nested queries are useful for returning multiple items where there is a parent-child relationship. One example of this type of relationship might be order header and order details records; another might be product categories and subcategories. You can see an example of using a nested FOR XML clause in the following listing:

```
SELECT DocID, MyXMLDoc,
    (SELECT MyXMLDoc
     FROM MyXMLDocs2
     WHERE MyXMLDocs2.DocID = MyXMLDocs.DocID
     FOR XML AUTO, TYPE)
FROM MyXMLDocs Where DocID = 2 FOR XML AUTO, TYPE
```

In this example the outer query on table MyXMLDocs is combined with a subquery on the table MyXMLDocs2 (for this example, a simple duplicate of the MyXMLDocs table). The important thing to notice in this listing is SQL Server 2005's ability to use the FOR XML clause in the subquery. In this case the subquery is using the Type directive to return the results as a native XML data type. If the Type directive were not used, then the results would be returned as an nvarchar data type and the XML data would be entitized. You can see the results of the nested FOR XML query shown in the listing that follows:

```
<MyXMLDocs DocID="2">
  <MyXMLDoc>
    <MyXMLDoc xmlns="MyXMLDocSchema">
      <DocumentID>1</DocumentID>
      <DocumentBody>&quot;My text&quot;</DocumentBody>
    </MyXMLDoc>
  </MyXMLDoc>
</MyXMLDocs>
```



NOTE

I added the formatting to the previous listing to make it more readable.

Inline XSD Schema Generation

Another new feature in SQL Server 2005's FOR XML support is the ability to generate an XSD schema by adding the XMLSCHEMA directive to the FOR XML

clause. You can see an example of using the new XMLSCHEMA directive in the following listing:

```
SELECT MyXMLDoc FROM MyXMLDocs WHERE DocID=1 FOR XML AUTO, XMLSCHEMA
```

In this case, because the XMLSCHEMA directive has been added to the FOR XML clause the query will generate and return the schema that defines the specific XML column along with the XML result from the selected column. The XMLSCHEMA directive works only with the FOR XML AUTO and FOR XML RAW modes. It cannot be used with the FOR XML EXPLICIT mode. If the XMLSCHEMA directive is used with a nested query, it can be used only at the top level of the query. The XSD schema that's generated from this query is shown in the following listing:

```
<xsd:import namespace="http://MyXMLDocSchema" />
<xsd:element name="MyXMLDocs">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="MyXMLDoc" minOccurs="0">
        <xsd:complexType sqltypes:xmlSchemaCollection="[tecadb].[dbo].[MyXMLDocSchema]">
          <xsd:complexContent>
            <xsd:restriction base="sqltypes:xml">
              <xsd:sequence>
                <xsd:any processContents="strict" namespace="http://MyXMLDocSchema" />
              </xsd:sequence>
            </xsd:restriction>
          </xsd:complexContent>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
<MyXMLDocs xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1">
  <MyXMLDoc>
    <MyXMLDoc xmlns="http://MyXMLDocSchema">
      <DocumentID>1</DocumentID>
      <DocumentBody>My New Body</DocumentBody>
    </MyXMLDoc>
  </MyXMLDoc>
</MyXMLDocs>
```

The XMLSCHEMA directive can return multiple schemas, but it always returns at least two: one schema is returned for the SqlTypes namespace, and a second schema is returned that describes the results of the FOR XML query results. In the preceding listing you can see the schema description of the XML data type column beginning at: <xsd:element name="MyXMLDocs">. Next, the XML results can be seen at the line starting with <MyXMLDocs xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1">.

**NOTE**

You can also generate an XDR (XML Data Reduced) schema by using the XMLDATA directive in combination with the FOR XML clause.

OPENXML Enhancements

The FOR XML clause is great for retrieving XML from the SQL Server 2005 database. The FOR XML clause essentially creates an XML document from relational data. The OPENXML keyword is the counterpart to the FOR XML clause. The OPENXML function provides a relational rowset over an XML document. To use SQL Server's OPENXML functionality, you must first call the `sp_xml_preparedocument` stored procedure, which parses the XML document using the XML Document Object Model (DOM) and returns a handle to OPENXML. OPENXML then provides a rowset view of the parsed XML document. When you are finished working with the document, you then call the `sp_xml_removedocument` stored procedure to release the system resources consumed by OPENXML and the XML DOM. You can see an overview of this process in Figure 7-1.

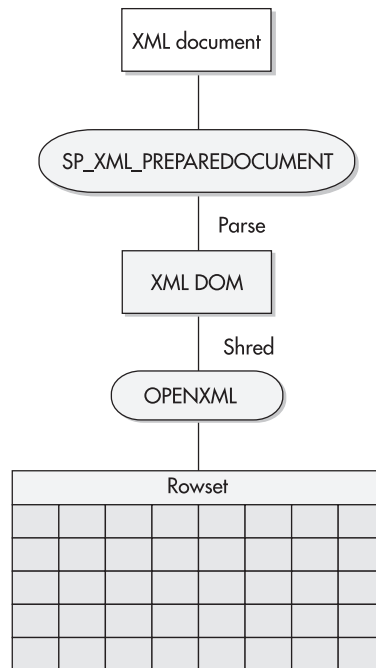


Figure 7-1 An overview of OPENXML

With SQL Server 2005 the OPENXML support has been extended to include support for the new XML data type, and the new user-defined data type. The following example shows how you can use OPENXML in conjunction with a WITH clause in conjunction with the new XML data type:

```

DECLARE @hdocument int
DECLARE @doc varchar(1000)
SET @doc = '<MyXMLDoc>
    <DocumentID>1</DocumentID>
    <DocumentBody>"OPENXML Example"</DocumentBody>
</MyXMLDoc>'
EXEC sp_xml_preparedocument @hdocument OUTPUT, @doc
SELECT * FROM OPENXML (@hdocument, '/MyXMLDoc', 10)
    WITH (DocumentID varchar(4),
        DocumentBody varchar(50))
EXEC sp_xml_removedocument @hdocument

```

At the top of this listing you can see where two variables are declared. The `@hdocument` variable will be used to store the XML document handle returned by the `sp_xml_preparedocument` stored procedure, while the `@doc` variable will contain the sample XML document itself. Next, the `sp_xml_preparedocument` stored procedure is executed and passed the two variables. The `sp_xml_preparedocument` stored procedure uses XML DOM to parse the XML document and then returns a handle to the parsed document in the `@hdocument` variable. That document handle is then passed to the OPENXML keyword used in the SELECT statement.

The first parameter used by OPENXML is the document handle contained in the `@hdocument` variable. The second parameter is an XPath pattern that specifies the nodes in the XML document that will construct the relational rowset. The third parameter specifies the type of XML-to-relational mapping that will be performed. The value of 2 indicates that element-centric mapping will be used. A value of 1 would indicate that attribute-centric mapping would be performed. The WITH clause provides the format of the rowset that's returned. In this example, the WITH clause specifies that the returned rowset will consist of two varchar columns named DocumentID and DocumentBody. While this example shows the rowset names matching the XML elements, that's not a requirement. Finally, the `sp_xml_removedocument` stored procedure is executed to release the system resources.

This SELECT statement using the OPENXML feature will return a rowset that consists of the element values from the XML document. You can see the results of using OPENXML in the following listing:

```

DocumentID DocumentBody
-----
1          "OPENXML Example"
(1 row(s) affected)

```

XML Bulk Load

One of the first things that you'll probably want to do to take advantage of the new XML data type is to load your XML documents into XML columns from disk. The new XML bulk load features make that task quite easy. This feature provides a high-speed mechanism for loading XML documents into SQL Server columns. You can see an example of using XML bulk load in the following listing:

```
INSERT into MyXMLDocs(MyXMLDoc) SELECT * FROM OPENROWSET
    (Bulk 'c:\temp\MyXMLDoc.xml', SINGLE_CLOB) as x
```

In this example the INSERT statement is used to insert the results of the SELECT * FROM OPENROWSET statement into the MyXMLDoc column in the MyXMLDocs table. The OPENROWSET function uses the Bulk rowset provider for OPENROWSET to read data in from the file 'C:\temp\MyXMLDoc.xml'. You can see the contents of the MyXMLDoc.xml file in the following listing:

```
<MyXMLDoc xmlns="http://MyXMLDocSchema">
    <DocumentID>3</DocumentID>
    <DocumentBody>"The Third Body"</DocumentBody>
</MyXMLDoc>
```

If you execute this command from the SQL Server Management Studio, you need to remember that this will be executed on the SQL Server system, and therefore the file and path references must be found on the local server system. The SINGLE_CLOB argument specifies that the data from the file will be inserted into a single row. If you omit the SINGLE_CLOB argument, then the data from the file can be inserted into multiple rows. By default, the Bulk provider for the OPENROWSET function will split the rows on the Carriage Return character, which is the default row delimiter. Alternatively, you can specify the field and row delimiters using the optional FIELDTERMINATOR and ROWTERMINATOR arguments of the OPENROWSET function.

Native HTTP SOAP Access

Another new XML-related feature found in SQL Server 2005 is native HTTP SOAP support. This new feature enables SQL Server to directly respond to the HTTP/SOAP requests that are issued by web services without requiring an IIS system to act as an intermediary. Using the native HTTP SOAP support, you can create web services

that are capable of executing T-SQL batches, stored procedures, and user-defined scalar functions. To ensure a high level of default security, native HTTP access is turned off by default. However, you can enable HTTP support by first creating an HTTP endpoint. You can see an example of the code to create an HTTP endpoint in the following listing:

```
CREATE ENDPOINT MyHTTPEndpoint
STATE = STARTED
AS HTTP(
    PATH = '/sql',
    AUTHENTICATION = (INTEGRATED ),
    PORTS = ( CLEAR ),
    SITE = 'server'
)
FOR SOAP (
    WEBMETHOD 'http://tempUri.org/'. 'GetProductName'
        (name='AdventureWorks.dbo.GetProductName',
        schema=STANDARD ),
    BATCHES = ENABLED,
    WSDL = DEFAULT,
    DATABASE = 'AdventureWorks',
    NAMESPACE = 'http://AdventureWorks/Products'
)
```

This example illustrates creating an HTTP endpoint named MyHTTPEndPoint for the stored procedure named GetProductName in the sample AdventureWorks database. Once the HTTP endpoint is created, it can be accessed via a SOAP request issued by an application. You can use the ALTER ENDPOINT and DROP ENDPOINT DDL statements to manage SQL Server's HTTP endpoints. The new HTTP endpoints are also able to provide data stream encryption using SSL. More information about SQL Server's new HTTP support can be found in Chapter 2.

The follow command shows how to list the HTTP endpoints that have been created:

```
select * from sys.http_endpoints
```

XML Enhancements for Analysis Server

While the majority of the XML enhancements in SQL Server 2005 have been implemented for the relational database engine, Analysis Services has also received several important new XML-related features, the most important of which is the

new XML for Analysis Services also known as XMLA. In this section you'll learn more about the new XML for Analysis Services feature.

XML for Analysis Services

XML for Analysis Services is an API that provides data access to Analysis Services data sources that reside on the web. XML for Analysis Services is modeled after OLE DB in that it is intended to provide a universal data access model for any multidimensional data source. However, unlike COM-based OLE DB, XML for Analysis Services is built on the XML-based SOAP protocol. Also unlike OLE DB, which was built with the client/server model in mind, XML for Analysis Services is optimized for use on the web.

XML for Analysis Services provides two publicly accessible methods: the Discover method and the Execute method. As its name implies, the Discover method gets information about a data source. The Discover method can list information about the available data sources, the data source providers, and the metadata that is available. The Execute method enables an application to run commands against XML for Analysis Services data sources.

System XML Catalog Views

SQL Server 2005 stores information about the XML that's used in the server in a number of new system views. Table 7-1 briefly describes the new system XML views.

XML Catalog View	Description
sys.xml_attributes	This view provides a row for each stored XML attribute.
sys.xml_components	This view provides a row for each component of an XML schema.
sys.xml_component_placements	This view provides a row for each placement of an XML component.
sys.xml_elements	This view provides a row for each XML component that is an XML element.
sys.xml_facets	This view provides a row for each facet (restriction) of an XML type.
sys.xml_model_groups	This view provides a list of all the XML component that are part of a Model-Group.
sys.xml_namespaces	This view provides a row for each XSD-defined namespace.
sys.xml_types	This view provides a row for each XML component that is an XML type.
sys.xml_wildcards	This view provides a row for each XML attribute or element wildcard.
sys.xml_wildcard_namespaces	This view provides a row for each XML wildcard namespace.

Table 7-1 SQL Server 2005 XML Catalog Views