

## SQLXML in .NET

by Christoph Schittko

We will investigate how we can build a business application that's completely XML-enabled. We will look at each tier of the three tier application model and discuss our options to leverage XML in the most efficient way; not only in terms of performance, but also with respect to the development effort and code maintenance.

The data tier is where we encapsulate all the details of data access. In our case we architect it to interface with the rest of the application only through XML. Our data layer is going to return query results in XML and accepts XML documents to insert them into a SQL Server 2000 database to avoid any unnecessary conversions in the business layer. But before we start studying different approaches to architect a data access layer we give a brief overview of SQL Server's built-in XML support, the SQLXML library and the managed .Net classes to access SQLXML.

### SQLXML Capabilities

The release of SQL Server 2000 started adding features to integrate XML with the relational world. The goal was to simplify development of XML-enabled applications and data exchange over the internet (what else?). Microsoft has been adding more features to this list, but will look at those separately. Initially SQL Server 2000 shipped with the following XML related features:

- SQL language extensions to retrieve XML formatted data directly from the database
- SQL language extensions to execute SQL queries against XML documents.
- Support for exposing data in SQL Server through IIS. You can return data from in XML format as a response to HTTP requests, sort of as the forerunner of SOAP web services.
- A SQLOLEDB OLE DB provider to improve performance when executing XML queries
- XML Views of relational data defined by schemas in the XDR format.
- XPath queries to retrieve XML formatted data from the database.
- Execution of XML query templates with optional XSL transformations.

Discussing all of these features in depth is beyond the scope of this book. We will explore how we can retrieve XML directly from SQL Server and insert data from XML documents in the following sections, but we have before we move on to the SQLXML web release and the managed classes to access data in SQL Server. We have to skip features enabling SQL Server access over HTTP because ADO.NET and the managed classes in SQLXML provide more flexible and re-usable means to access a database. We will also not discuss XML views defined by XDR schemas, since they are superceded by views defined by XSD schemas. Check the SQL Server Books online documentation on the MSDN web site, or install it from your SQL Server CD if you are interested in these features.

## Retrieving XML from SQL Server

SQL Server 2000 introduces support for returning data directly in an XML format. For that purpose Microsoft added an extension to its dialect of the Transact-SQL language. The XML clause, appended to a SELECT statement as shown below, indicates to SQL Server to return the matching rows as an XML document rather than the traditional rowset format.

### FOR XML mode

```
SELECT FOR XML mode [, XMLDATA] [, ELEMENTS][, BINARY BASE64]
```

The mode parameter controls the format of the returned XML document. Tables 15.1 lists the modes SQL Server 2000 understands. The optional parameters of the FOR XML clause (listed in table 15.2) refine the format further.

NOTE: FOR XML queries return XML fragments, not well-formed documents. The fragment does not group the results under a single root element to simplify adding the results into already existing XML documents.

#### 1.1 The Modes of SQL Server FOR XML queries control the generated XML format at a high level.

AUTO	Every row matching the query results in an element in the return XML document. Every selected column maps to an attribute or a child element, depending on the other parameters
EXPLICIT	The query is in a special format that describes how to map columns to XML elements and attributes.
RAW	Every field in the query result is returned as an attribute of an element named "row"
NESTED	Same as AUTO mode, but requires SQLXML client-side formatting.

The ExecuteXmlReader() on the SqlCommand class returns an XmlReader object populated with the results from the command execution. Consequently, ExecuteXmlReader() can only be used with statements returning XML formatted data – like the statements with the FOR XML clause, as in the snippet below:

```
SqlCommand cmd = conn.CreateCommand();  
cmd.Text = "select SupplierID, Contacts from " +  
    "Suppliers for XML AUTO";  
XmlReader reader = cmd.ExecuteXmlReader();  
DataSet dataset = new DataSet(reader);  
reader.Close();  
doc.Save(Console.Out);
```

Those of you familiar with ADO.NET will notice that this example uses the same SqlCommand class used to query relational data from SQL Server. In this example, we remember that the DataSet can load its data from an XML source, like an XmlReader. Note that the DataSet is can load an XML fragment. It does not require well-formed XML like the XmlDocument does.

Make sure to close the returned XmlReader before opening any new readers, just like you do with any other DataReader classes. If you forget to close it, your application will sooner or later run out of resources and quit functioning properly. Now let's take a closer look at the other modes of the FOR XML clause.

#### 1.2 Optional Parameters of the FOR XML clause. The parameters allow finer control of the format of the

#### generated XML.

Parameter	Description
XMLDATA	Adds an XSD schema describing the XML format of the returned resultset to output
ELEMENTS	Specifies that the columns are to be returned as child elements rather than attributes. This is only allowed with AUTO mode.
BINARY BASE64	Specifies that binary data is to be returned in base64-encoded format. We must provide this option to retrieve binary data in RAW or EXPLICIT mode.

## *Understanding the FOR XML clause modes*

Each mode of the FOR XML clause allows a different level of control over the format of the returned XML document, ranging from no control with FOR XML RAW to full control with FOR XML EXPLICIT. This section briefly describes each mode together with some examples to run against the Southrain example database we always refer to in this article. Once you downloaded the files from the book's web site and set up the database you can run these examples in SQL Query Analyzer to see the results.

### **Raw mode**

RAW mode queries result in a flat table-like XML format. The format does not preserve any information about the origin of the data or hierarchical relationships. SQL Server simply transforms each row of the result set into an XML element with the name *row*, very similar to the output format of the persist-to-XML option of classic ADO Recordsets. Every column that is not NULL is mapped to an attribute of the column's name. The example SQL statement below illustrates using XML RAW:

```
SELECT CompanyName, ProductName, UnitsInStock, UnitsOnOrder
from Products JOIN Suppliers ON Products.SupplierID =
Suppliers.SupplierID FOR XML RAW
```

A row returned from an FOR XML RAW query could look like this element:

```
<row CompanyName="Exotic Liquids" ProductName="Chai" UnitsInStock="39"
UnitsOnOrder="0"/>
```

The element does not reflect that CompanyName and ProductName came from different tables. In fact it does not even indicate anything about either table the results came from. It only lists the columns we selected.

### *Auto mode SQL XML queries*

AUTO mode returns the results in a more descriptive XML format. Each selected row results in an element named after the table from which it was selected. The selected columns result in attributes of the elements by default. If the SELECT statement joins multiple tables the results from the joined table are represented as child elements. AUTO mode also recognizes the ELEMENTS option. If we append “, ELEMENTS” to the query statement columns in the returned rowset are also mapped to child elements rather than attributes. The nesting of elements and their children is determined by the order of the tables in the SELECT clause. Thus the order of the columns in the SELECT clause is significant. We can see the difference if we replace RAW with AUTO in the above query:

```
SELECT CompanyName, ProductName, UnitsInStock, UnitsOnOrder
from Products JOIN Suppliers ON Products.SupplierID =
Suppliers.SupplierID Where CompanyName="Exotic Liquids"
```

## **FOR XML AUTO**

The results do show each column as attributes on elements named after the table from which they were selected. The result also reflects the parent-child relationship of the joined tables “Suppliers” and “Products” by nesting the Product children inside Suppliers parents:

```
<Suppliers CompanyName="Exotic Liquids">
  <Products ProductName="Chai" UnitsInStock="39" UnitsOnOrder="0"/>
  <Products ProductName="Chang" UnitsInStock="17" UnitsOnOrder="40"/>
  <Products ProductName="Aniseed Syrup" UnitsInStock="13"
    UnitsOnOrder="70"/
</Suppliers>
```

Note: You can change element and attribute names by defining aliases for tables and columns in the SQL query.

## **EXPLICIT Mode**

The EXPLICIT mode is, well, the most explicit. This mode allows you to control every aspect of the XML format returned by the query. In EXPLICIT mode, the SQL query must be written in a very specific way. Within the query we need to provide not only information about the data to query, but also the XML format in which to return the results. We are fully responsible for ensuring the XML is well formed and valid. The key is to setup the SQL statement as to result in a very specific rowset format called a “universal table”. EXPLICIT mode then transforms the universal table to an XML document. The easiest way to understand a universal table is to build one. Once you have mastered the first one, the next one is much easier, so let’s go ahead and look at the FOR XML EXPLICIT query below that returns the same format as the previous FOR XML AUTO query and examine how it works.

### **1 A FOR XML EXPLICIT query returning XML from SQL Server.**

```
SELECT 1 As Tag, NULL As Parent,
      CompanyName As [Suppliers!1!CompanyName],
      NULL As [Products!2!ProductName],
      NULL As [Products!2!UnitsInStock],
      NULL As [Products!2!UnitsOnOrder]
FROM Suppliers
UNION ALL
SELECT 2 As Tag, 1 As Parent,
      CompanyName,
      ProductName,
      UnitsInStock,
      UnitsOnOrder
from Products JOIN Suppliers ON Products.SupplierID =
Suppliers.SupplierID
ORDER BY [Suppliers!1!CompanyName], [Products!2!ProductName]
FOR XML EXPLICIT
```

Did you notice that the statement actually combines two individual SELECT statements to produce the universal table. The first column in each of the SELECT statements is a named column and its type is a number. This column is represented as the Tag column. We will see in just a minute what it does. The second column in the SELECT clause is also a named column and its type is also a number, I'll refer to this as the Parent column. These two columns together specify the parent-child relationship between the results of the two SELECT queries in the resulting XML document. The Tag column stores the tag number by which children can identify their parents in the parent column. Thus, if the Parent column is 0 or NULL, then the row is placed at the highest level of the XML tree. In our example query the Supplier data is the highest level because we selected NULL into the Parent column – not because its Tag is 1! According to this rule, the results from the second SELECT in our example make up the children of the first one because the parent column references the tag value from the first statement. Which children appear under which parent is determined by the JOIN clause of the second statement.

Even though you will never see or use a universal table directly, it is easier to work out EXPLICIT queries if you think in terms of the universal table concept. The table 8.13 below shows part of the universal table from our query for one supplier.

### 1.3 An example universal table

Tag	Parent	Supplier!1!CompanyName	Products!2!ProductName	Products!2!UnitsInStock
1	NULL	ExoticLiquids	NULL	NULL
2	1	ExoticLiquids	Aniseed Syrup	13
2	1	ExoticLiquids	Chai	39
2	1	ExoticLiquids	Chang	17

The output of the query above will produce results in the same XML format returned by the FOR XML AUTO query in the previous section:

```
<Suppliers CompanyName="Exotic Liquids">
  <Products ProductName="Aniseed Syrup" UnitsInStock="13"
    UnitsOnOrder="70"/>
  <Products ProductName="Chai" UnitsInStock="39" UnitsOnOrder="0"/>
  <Products ProductName="Chang" UnitsInStock="17"
    UnitsOnOrder="40"/>
</Suppliers>
```

Despite the tedious work setting up the universal table, we have not gained anything over querying with FOR XML AUTO yet. Now let's see why FOR XML EXPLICIT is much more powerful than XML AUTO. In the introduction to this article promised that EXPLICIT gives us full control over the returned XML format and here is how.

You probably noticed the strange-looking column names specified in the query statement. These names actually encode all the information SQL Server needs to format the query results. Each name contains up to four data fields separated by a "!":

ElementName!TagNumber!AttributeName!Directive

The meaning of the individual fields is:

- The ElementName field specifies the name of the XML to create in the XML document.
- The TagNumber identifies the nesting level of this column in the output document. The value must match a Tag value in the universal table.
- The AttributeName is the name of an XML attribute to create in the XML document for the specified column. If the AttributeName is missing the format will be determined by the Directive portion of the format.
- Directive is an optional directive that can be used to encode XML ID, IDREF, and IDREFS or it can indicate how to map string data to XML by using the keywords hide, element, xml, xmltext, and cdata. If Directive is not specified, the AttributeName must be specified. Table 15.4 lists the directives recognize by Sql Server 2000.

**1.4 The Directive of an FOR XML EXPLICIT query controls the XML format returned for a particular column in the rowset.**

Directive	Description
ID, IDREF, and IDREFS	Identifies XML ID columns and the IDREF columns referencing them. This clause has no effect on the returned format if XMLDATA is not specified anywhere in the FOR XML clause.
hide	Omits the field from the result XML document
element	Formats the field as an XML element rather than an attribute. If the field's data contains characters restricted in XML document, the characters are automatically converted to the corresponding entity references.
xml	Same as "element" directive without replacing restricted characters.
xmltext	Embeds an XML fragment from a column directly in the returned XML without a containing element. The column has to be of a text type.
cdata	Wraps the field data in a CDATA section. The field's data type must be character based. As of release 3.0 SP 1 FOR XML EXPLICIT cannot create CDATA sections inside a child element. In those cases you have to manually create the CDATA markers in the SQL.

I think by now you realize that EXPLICIT mode queries are very powerful. You probably also realize that they also can be quite cumbersome to develop, especially when your schema nests types more than two levels deep. Fortunately the SQLXML web releases provide a much easier-to-use wrapper around EXPLICIT mode queries. If you like the idea of returning arbitrarily formatted XML from SQL Server, be sure to read the section on mapping schemas and the SQLXML managed classes.

## **XMLDATA**

The last option we need to discuss with FOR XML is the XMLDATA option. XMLDATA's purpose is to generate an XDR that describes the format of the XML document returned for a query. The schema is generated and returned inline with the rest of the XML document. Care should be taken when using the XMLDATA option. Generating the schema takes server resources that could be used for other more "SQL-like" things. Since the XML that is returned is character data, you may just want to use the XML as it is without the XDR. This is especially true if the query is to be run hundreds or thousands of times a second.

## **Parameterized Templates**

Templates combine the output of multiple queries and construct complex XML documents. They outline an XML format together with placeholders to fill with the results of SQL and/or XPath queries that returns XML. That's already the whole idea of a template. Look at the template below for example:

```

<?xml version="1.0" encoding="utf-8" ?>
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:query>
    select CustomerID, ContactName
      from Customers
      FOR XML AUTO, ELEMENTS
  </sql:query>
</ROOT>

```

The results of the query replace the <query> tag in the XML frame when the template executes. One possible result could look like this:

```

<?xml version="1.0" encoding="utf-8" ?>
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Customer>
    <CustomerID>ALFKI</CustomerID>
    <ContactName>Maria Anders</ContactName>
  </Customer>
</ROOT>

```

Originally template queries were designed to run queries through HTTP requests. The template would be stored in file on a web server and clients retrieve query results simple by requesting the template file. By now templates can also be executed programmatically through COM objects or the SQLXML managed classes which we are going to discuss in section 15.2.4.

While executing the example above demonstrated the concept behind templates, it is not very useful for real world applications. The query above retrieves all records from the Customers table because we did not add any criteria restricting the query. How would we go about retrieving one specific customer record? Obviously templates would be useless if we had to code a different template for each customer record in the database, so that cannot be the right solution. The answer is: We add parameters to the template. Parameterized templates look similar to XSLT style sheets with parameters. If you think about it, they perform a similar function, too. We define the parameters in a separate header section and reference them throughout the template by their name. Template parameters are identified by a prefixing their name with a special character, just like parameters in an XSL style sheet. In the XML templates it's the '@' instead of the '\$', but the idea is the same. So let's see what the template looks like after we added a CusID parameter to restrict the query to records matching the value of the parameter.

```

<?xml version="1.0" encoding="utf-8" ?>
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:header>
    <sql:param name='CusID' />
  </sql:header>
  <sql:query>
    select CustomerID, ContactName
      from Customers Customer
      where CustomerID = @CusID
      FOR XML AUTO, ELEMENTS
  </sql:query>
</ROOT>

```

How we pass the parameters to the template depends on the way we are invoking it. If we invoke it through the HTTP interface, then we add the parameters to query string of the request. The managed classes have a different way to pass parameters which we will hear about in section 15.2.4.2.

There are more details about templates we have not discussed, how we pass null values, for example. You can look them up in the SQL Server Books Online documentation or the MSDN web site if you think about employing query templates in your applications.

## OPENXML

Besides returning results in XML format SQL Server 2000 also can insert data directly from an XML document – well almost. Actually we are still issuing an INSERT statement, but we select the data to insert directly from an XML document. SQL Server 2000 parses the XML document with a new system stored procedure, `sp_xml_preparedocument`. After we called that stored procedure, we can create a rowset view of the document with the OPENXML statement and then SELECT data from the rowset. Finally we need to clean up the server memory after we finished accessing the XML data. SQL Server 2000 provides another stored procedure `sp_xml_removedocument` we need to call let let SQL Server know when it can release all resources help to provide access to the XML document.

Does that sound too complicated? It's really not. Let's look at a concrete example and see how it works. Imagine you need to add a new customer record to your system. Currently you have the customer data in the following XML document:

```
<Customer>
  <CustomerID>NEWCUS1</CustomerID>
  <ContactName>Joe H. Buyer</ContactName>
</Customer>
```

To access the data in that document you have to follow the steps we outlined above: prepare the document and open it as a rowset:

```
-- @doc holds the XML document to process
DECLARE @idoc int
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc

SELECT CustomerID, ContactName
FROM OPENXML (@idoc, '/Customer', 2)
  WITH (CustomerID varchar(10),
        ContactName varchar(20))
EXEC sp_xml_removedocument @idoc
```

First we call the stored procedure to obtain a handle to the parsed XML document. The only thing we can do with this handle is pass it to the OPENXML statement to open a rowset like view on the data in the XML document.

NOTE: `sp_xml_preparedocument` requires well-formed XML. If we need to insert multiple records we have to group them under a common parent element.

The next parameter of the OPENXML statement is an XPath expression to tell SQL Server which part of the XML document we would like to view. The third parameter specifies if the rowset is made up from the attributes of the element(s) we selected, the immediate child elements or both. Table 15.5 shows the possible values and explains their effect on the results of the query in more detail. The final piece of the anatomy of an OPENXML statement is the WITH clause. This one is particularly interesting because it controls the columns of the returned rowset. Only columns listed in the WITH clause are part of the returned rowset. The third parameter of OPENXML specifies the default mappings available to the WITH clause. Table 15.5 lists the possible values.

**1.5 Conversion options of the OPENXML statement. The option specifies which XML nodes OPENXML maps by default.**

Conversion Option	Description
0	Same as 1
1	Attribute-centric mapping. Only attributes of the selected elements are mapped by default.
2	Element-centric mapping. Only immediate child elements are mapped by default.
3	Combines options 1 and 2. Attributes are resolved before immediate child elements.
8	Causes all mapped XML content that was not consumed in the WITH clause to be copied to the special meta property @mp:xmltext. We can combine this option with all previous options.

If we omit the WITH clause from the statement then the returned rowset will contain meta information like the node's local name, the namespace prefix and the namespace uri, about the matching nodes. The returned rowset format is called an "edge table". By default each row maps to a mapped attribute or element with the same name, but we can override the default by adding an XPath expression to map a column to any other attribute or element throughout the XML document or to the meta properties from the edge table.

The SQL Server Books online collection installed with SQL Server contains all the details about the edge table. You can read everything else there. We move on to use OPENXML to insert data into SQL Server tables. After all, this section is titled "Inserting Data with OPENXML" and we have not inserted anything so far.

Actually, doing the INSERT is a piece of cake – now that we know how to access the data in the XML document. We simply need to select the columns we want to insert from a rowset we obtain from a SELECT ... OPENXML query. To insert the customer data into a Customers table we would add an INSERT command like shown below:

```
INSERT Customers ( CustomerID, ContactName )
SELECT CustomerID, ContactName
FROM OPENXML (@idoc, '/Customer', 2)
    WITH (CustomerID varchar(10),
          ContactName varchar(20))
```

This concludes our overview of the original XML feature set of SQL Server 2000. Next we'll go over the features that have been added since then with several web releases.

## **SQLXML Web Release**

Since the initial release of SQL Server 2000, Microsoft released several XML feature packs – known as SQLXML – over the internet. These feature packs extend SQL Server's XML support

and offer seamless integration of the XML features with the .NET Framework. These additional features include:

- Client-side XML formatting to offload processing cycles from the database server. When you issue a FOR XML clause to SQL Server it must process the results and format them according to the FOR XML clause. This takes valuable processor time away from SQL Server. SQLXML has the ability to return the results to the client and format the XML client side thus reducing server load.
- Creating XML views of relational data using annotated XSD schemas instead of XDR schemas. The schema annotations define in detail how SQL Server tables map to the schema's XML data types.
- Support for XML-based syntax to insert, update and delete data from the database. The syntax is called Updategram
- Bulk loading of XML data from an XML document and an annotated XSD describing how the XML format maps to tables and columns in the database.
- A SQLXMLOLEDB OLEDB provider to support features like client-side formatting.
- Managed classes to seamlessly integrate SQLXML with pluggable XML architecture of the .NET Framework.
- Integration with ADO.NET via the direct execution of DiffGrams to insert, update and delete data from the database.
- Web Services support which allows a SQL Server to be exposed as a web service. SOAP messages can be sent and processed on SQL Server and returned as SOAP messages. Also a WSDL service definition is provided to help with the automatic generation of proxy classes to call procedures and user defined functions on the SQL Server.

Again, we try to stay with the focus of this book and only discuss the features that immediately help us to develop an XML enabled application using the .NET Framework. You can check out other features like the Web Service support, XML templates and bulk loading in the SQLXML documentation or web sites dedicated to SQLXML on the internet. The following three subsections introduce the different concepts to access SQL Server with SQLXML. After that, in section 15.2.4 we will specifically look at the managed classes we can call from our .NET applications to interact with SQLXML and SQL Server. Feel free to skip ahead if you are already familiar with Mapping Schemas and Updategrams and you want to find out how to integrate SQLXML into your .NET applications.

## Mapping Schemas

The first of the SQLXML features we are looking at are mapping schemas, because several other features depend on them. Mapping schemas are very handy when we have to build an XML enabled application on top of an existing SQL Server database. Imagine the following scenario: Your boss finally caught on that comma-separated files are no longer state-of-the-art to transfer data and XML is now the way to go. The next time a business partner requires a data feed, your assignment is to develop an application to send data from your existing relational data base to the business partner in an XML format. If you are lucky you get to define the XML for the feed yourself. In that case you remember SQL Server's built-in support for XML queries and chose a format that mirrors your database schema because you can create the feed with a simple FOR

XML AUTO select statement, as we have seen in 15.1.2.2. Then you write a component to issue the query and within two days you announce to your boss that your project is done.

Unfortunately, that scenario is hardly ever the happening. More than likely you wind up having to send data in an XML format that you cannot create with a simple FOR XML AUTO query. Now what are you going to do? In some simple cases you might be able to retrieve the data with a FOR XML AUTO query and employ XSL transformations to get the output from SQL Server into the XML format you need. If your format is structured as a hierarchy of types then XSLT will get very complex. You could resort to SQL queries with the FOR XML EXPLICIT clause (15.1.2.3), but we've already seen how cumbersome it is to develop these queries for XML types with more than a handful of elements. What you really need in that situation is an automated way to map your database schema to an XML schema – and that's exactly what mapping schemas do; in an XML-oriented fashion of course.

### Mapping Schema Annotations

A mapping schema is an XSD schema with special annotations describing how the XML data types map to tables and columns in the database. The annotations are special XML elements and attributes to define which type XML maps to which SQL Server table and whether a column maps to an element, an attribute or if it does not map to anything. Figure 15.1 illustrates relationship of types in an XSD schema and tables a SQL Server database and the role of the schema annotations.

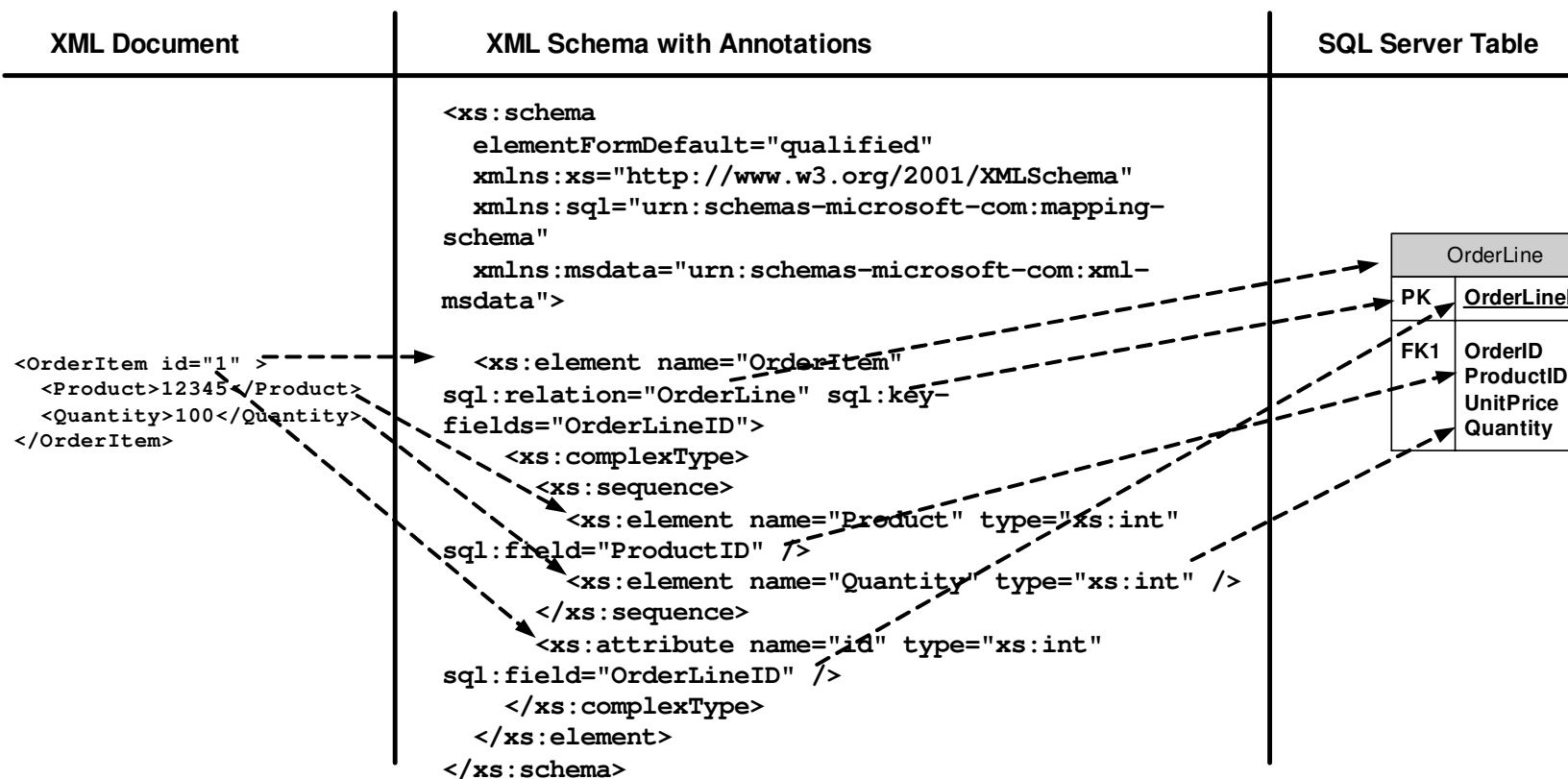


Figure 15.1: The annotations in a mapping schema define how an XML type maps to a SQL Server database.

SQLXML looks for these annotations when it analyzes an XSD schema for override the default mapping. Table 15.6 lists all the annotations SQLXML 3.0 recognizes in a mapping schema. By default SQLXML maps each top-level element in an XSD schema to a table with the same name. All attributes and immediate child elements of top level elements map to columns with the same name as the XML node. If element names in the schema do not match column names in the database for example, we can add the `field` annotation to the element definition in the schema. This annotation tells SQLXML which database column maps to this XML element. Other annotations provide additional mapping rules map from SQL Server to XML types. One class of annotations helps SQLXML to tune the way to it accesses the data in SQL Server to the constraints on the database. The `relationship` annotation for example identifies parent-child relationships between tables. We will examine this particular annotation shortly.

**1.6 Schema Annotations defined in the `urn:schemas-microsoft-com:mapping-schema` namespace. The annotations provide detailed information how an XML schema type maps to tables in a SQL Server database. Annotated schemas allow XPath queries returning XML data structures defined in the schema and allow insert, update and delete operations in the database directly from schema valid documents.**

XSD annotations	Description
<code>sql:datatype</code>	Maps an XSD type to a SQL Server type when type or format conversions are required, for example the XML format <code>dateTime</code> is not compatible with SQL Server's <code>datetime</code> . We need to add the <code>datatype="datetime"</code> annotation to the schema for SQLXML perform a format conversion to insert XML <code>datetime</code> values.
<code>sql:encode</code>	SQL Server can return data from image type columns either as base64 encoded data embedded in the returned XML document or as an expression by which the data can be queried (dobject queries) over the SQL Server HTTP interface. The <code>encode</code> annotation tells SQL Server which of the two to return. Possible values are "url" for a dobject expression or "default" for embedded base64 encoded data. You cannot annotate elements annotated with <code>sql:use-cdata</code> or on the ID, IDREF, IDREFS, NMTOKEN, or NMTOKENS attribute types with <code>sql:encode</code> . Check the SQL Server Books Online documentation for more details on dobject queries.
<code>sql:field</code>	Maps an XML element or attribute to a database column. You can only map leaf nodes to database columns. SQLXML does not support mixed content models
<code>sql:guid</code>	Specifies whether SQL Server will insert a GUID value from an Updategram or automatically generate a new value when inserting a new row. Possible values for the <code>guid</code> annotation are: "generate" and "useValue".
<code>sql:hide</code>	Excludes the annotation element or attribute from the returned XML fragment. However, the excluded node can be part of an XPath query. Modifications to the excluded node with updategrams also succeed.
<code>sql:identity</code>	Identifies a table's identity column. Possible values are: "ignore" and "useValue". When the annotation specifies "useValue" SQL Server will insert the value from an updategram into the database. "ignore" causes SQL Server to automatically generate an identity value. Check 15.3.4.4 for more details.
<code>sql:inverse</code>	Indicates that a parent-child relationship between the XML type is the opposite of the relationship between the corresponding database tables. The <code>inverse</code> annotation is only valid on a relationship annotation. You can find more information on the relationship annotation in 15.2.1.2.
<code>sql:is-constant</code>	Indicates that the annotated XML element does not map to the database. SQLXML will generate the element when executing XPath queries and ignore it in Updategrams. You cannot add this annotation to attributes.
<code>sql:key-fields</code>	Identifies the column(s) that uniquely identify a row in a table. Leaving this annotation out can cause an unexpected order of XML elements.
<code>sql:limit-field</code> <code>sql:limit-value</code>	<code>limit-field</code> and <code>limit-value</code> together restrict the results returned by XPath queries to the rows where the column identified by <code>limit-field</code> has the value specified in <code>limit-value</code> . SQLXML ignores these annotations for updategrams.
<code>sql:mapped</code>	Indicates SQLXML to ignore the annotated schema item. The item is not generated in returned XML elements and cannot be part of an XPath query.
<code>sql:max-depth</code>	Specifies the maximum depth of recursive relationships. SQLXML will not return results beyond the maximum depth when executing XPath queries.
<code>sql:overflow-field</code>	Identifies a database column that receives all XML content from an updategram that did not map to the database.
<code>sql:prefix</code>	Specifies a character sequence to prefix the value from the database. This allows generating valid XML ID and IDREFS, since these have to be unique across an entire XML document. SQLXML will append the prefix when it generates XML documents and strips the sequence when it is present in an Updategram.
<code>sql:relation</code>	Specifies the SQL Server table mapping to an element or attribute. Attributes can map to a table to realize ID/IDREFs as m:n relationships in the database.
<code>sql:relationship</code>	A relationship annotation describes a parent-child relationship between two tables mapping to nested elements or ID/IDREF/IDREFS references in an XML document. The attributes: <code>parent</code> , <code>child</code> , <code>parent-key</code> , and <code>child-key</code> identify the database column(s) which realize the relationship between the tables. The columns identified as the child key do not always have to be part of the XML type related to the child table.
<code>sql:use-cdata</code>	SQLXML will always return the content of the annotated element wrapped in a CDATA section when <code>use-cdata</code> is set to

	"1" or "true". The annotation is meaningless for updategrams since the underlying XML parser always handles CDATA sections correctly. You cannot combine the use-cdata and encode annotations on the same element.
--	--

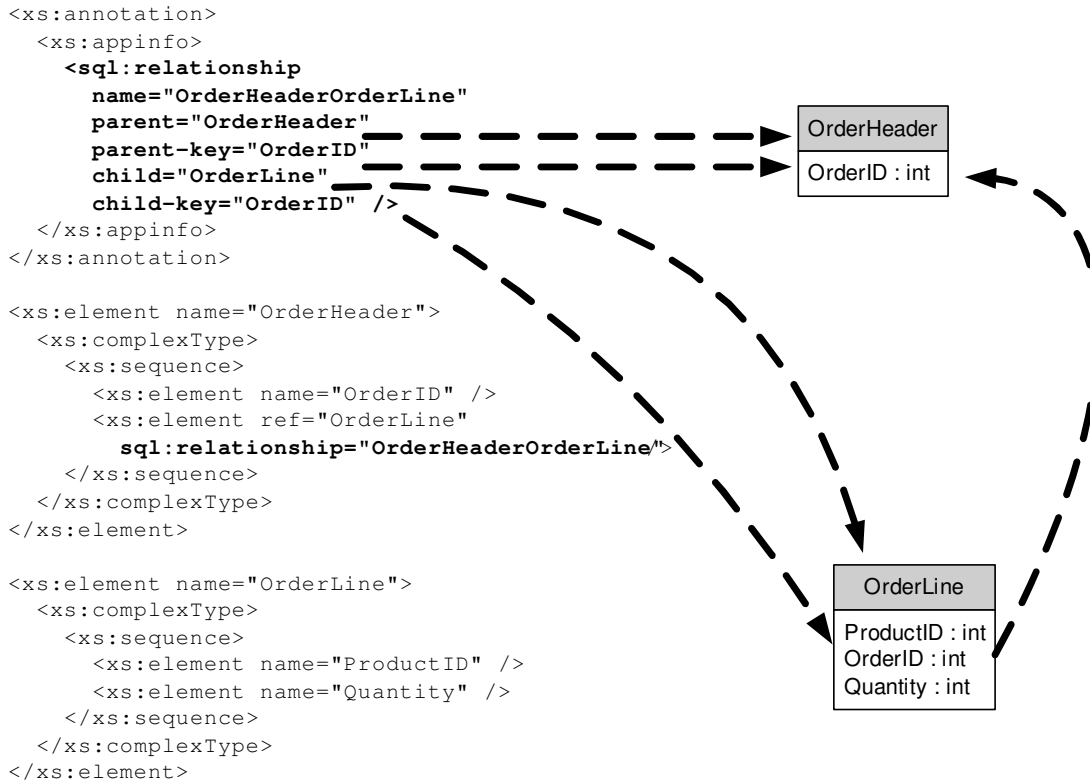
If your application has to support an already existing XML schema then you can simply add these annotations to that schema. If there is no schema describing your XML data types you can very easily create one using the XML Schema Designer in Visual Studio.NET. You need simply drag a database or a table from a SQL Server instance in the Server Explorer window onto the schema designer surface and watch the definitions for the data types appear in the schema. Now you have an XML schema which you can tailor to the needs of your application and add the annotations for SQLXML. Once we created an annotated schema we can use it in several ways:

- Issue XPath queries against the database, either directly from the client application or from within a Template.
- Insert, update and delete data from the database by sending Updategrams containing the changes to execute.
- Insert, update and delete data from the database by sending DiffGrams containing the changes to execute.

TIP: The easiest way to test mapping schemas during development is to set up an HTTP interface to your database as described in the SQLXML documentation and issue XPath queries for every XML type in the schema directly from a web browser. The browser will display the query results or the error messages returned by SQLXML. Don't forget to turn off the "Show friendly HTTP error messages" option if you are using Internet Explorer. Once you validated your schema by querying for each XML type in the schema, you can go on to build Templates and Updategrams and build an application around them.

## Mapping Relationships To XML

It is very important that we specify all parent-child relationships between the tables and designate key columns in a mapping schema. SQLXML needs this information to be able to create the correct SQL statements when we issue XPath queries and Updategrams. To map a parent-child relationship between XML types to SQL Server tables we first have to add a `relationship` annotation to the mapping schema. Then we add a reference to this relationship definition to the element where the actual nesting occurs as depicted in the following diagram:



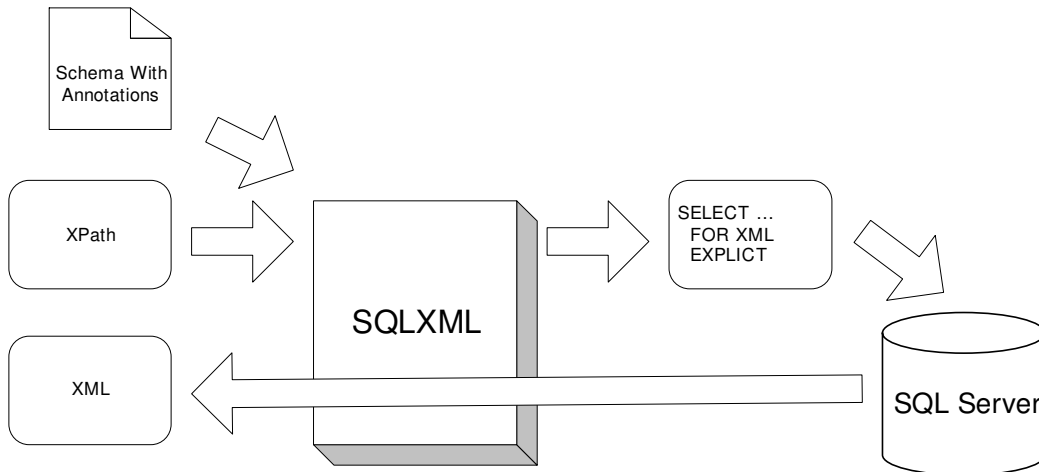
**Figure 15.xx: The relationship annotation defines how nested XML types map to database tables and which XML nodes map to primary and foreign key columns.**

Note that SQLXML can also handle situations where parent-child relationships between XML elements and SQL Server are opposites of each other. You can add the *inverse* annotation to the relationship definition and SQLXML will swapping parent and child when transforming XML to data in SQL Server.

Also, note that the key fields referenced in the relationship definition do not need to be part of the XML type definition. SQLXML will automatically generate the foreign key entry in the table identified as the child. Likewise, SQLXML will appropriately generate entries for M:N relationships if they are properly set up in the mapping schema.

## XPath Queries

With SQL Server and SQLXML we can query data with XPath queries to navigate around XML documents. Together with the query we have to specify a mapping schema to define how the format of the query maps to the database schema. SQLXML first inspects the mapping schema and transforms the XPath expression into a `SELECT ... FOR XML EXPLICIT` statement. It will then send the `FOR XML EXPLICIT` query to SQL Server to retrieve the data in the XML format defined in the mapping schema.



There are several advantages to define your DB-to-XML mappings using annotated schemas when we need to retrieve data from the database and convert it to a fixed format. First of all, they relieve us from writing any code to convert data returned from the database into the XML format. We simply point SQLXML to the mapping schema and the conversion happens behind the scenes. Furthermore, all the information about how your XML maps to your database resides in one place outside of the compiled code. This helps to further reduce development time as you can quickly propagate changes to the mapping during development or if you find any bugs after you deployed your application to your production environment. Also, querying object hierarchies with XPath expressions is much simpler and more readable than using SQL statements joining several tables. Don't you agree that querying the database for an XML structure of Orders with nested OrderItem elements like this

```

<OrderHeader>
  <OrderID />
  <OrderDate />
  <OrderItem>
    <ProductID>1<ProductID>
    <Quantity>10</Quantity>
  </OrderItem>
</OrderHeader>

```

with the XPath expression

```
/OrderHeader/OrderLine[ProductID=1]
```

is much easier to read than:

```

SELECT      OrderLine.*, OrderHeader.*
FROM        OrderHeader INNER JOIN
           OrderLine ON OrderHeader.OrderID = OrderLine.OrderID
WHERE       OrderLine.ProductID = 1

```

Finally, I prefer XPath queries over SQL statements, especially when it comes to writing reusable components, because the fields returned by the query are defined by the type definition of

the type you are retrieving, not by the query itself. So much for retrieving XML from the database, now let's see how we can modify a SQL Server database when have an XML document and a mapping schema.

WARNING: SQLXML 3.0 does not support the XPath functions and operators: xmlns, descendant-or-self, preceding-sibling, preceding, namespace, following-sibling, followingdescendant, ancestor-or-self, ancestor, ceiling(), mod, concat(), floor(), count(), contains(), id(), translate(), sum(), substring-before(), substring-after(), substring(), string-length(), starts-with(), round(), normalize-space(), namespace-uri(), name(), local-name(), position(), last(), lang() as well as node tests with the \* wildcard.

## Updategrams

SQLXML exposes a fully XML-based data access API allowing to INSERT, UPDATE and DELETE data from the database. The API is not based on a W3C standard and it is very different from SQL, but it is not hard to understand. The concept of the API is to send specially formatted query templates to SQL Server. The templates define the modifications to execute by identifying the items to modify and their state after the modification. The template format is called an Updategram and its structure is outlined below.

### 2 Updategram Format. Optional elements are shown in square brackets.

```
<ROOT xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
  [<updg:header>
    <updg:param name="" />
  </updg:header>]
  <updg:sync [mapping-schema= "MappingSchema.xsd"] >
    <updg:before>
      <elem1 [updg:id="x"]>
        ...
      </elem1>
      <elem2 [updg:id="y"]>
        ...
      </elem2>
    </updg:before>
    <updg:after>
      <elem1 [updg:id="x"]>
        ...
      </elem1>
      <elem2 [updg:id="y"]>
        ...
      </elem2>
    </updg:after>
  </updg:sync>
</ROOT>
```

Each <sync> block inside an Updategram defines an atomic set of changes, i.e. all modifications succeed or fail together. An Updategram can contain multiple <sync> sections if necessary. A <sync> block identifies the rows we need to manipulate in the <before> section. The <after> section lists the changes to carry out. Similar to regular batch updates, the <before> and <after> sections can list more than one XML element to modify. In order for SQLXML to match up the elements in the <before> and the <after> sections we have identify the matching ones

either explicitly through the id attribute or include the key-fields annotation in the mapping schema.

## How Do Updategrams Work?

Expressing insert and delete operations with the Updategram syntax is very simple. When we insert data we can simply omit the before section or leave it empty, while the after section contains the data we want to insert. Likewise, we can delete data from the database by omitting the after section or leaving it empty.

Behind the scenes, SQLXML does nothing but convert the Updategrams to INSERT, UPDATE and DELETE statements. When performing an insert operation the Updategram results in a SQL INSERT ... VALUES statement containing all items from the <after> section. For update and delete operations, SQLXML will compose UPDATE or DELETE statements with WHERE clauses containing the elements and attributes specified in the <before> section of the Updategram, as shown in the following diagram:

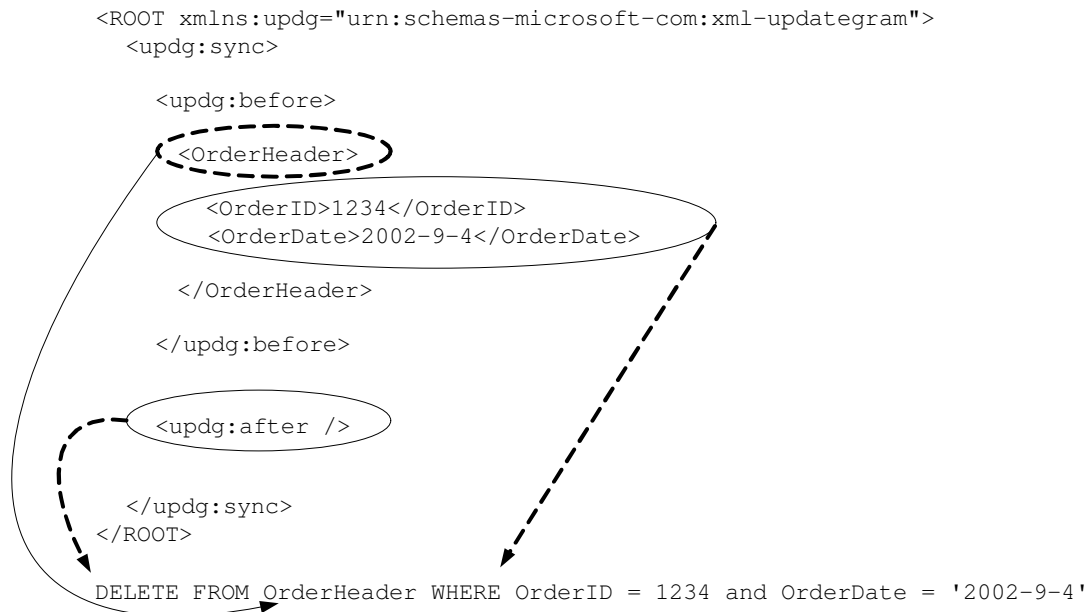


Figure 15.3: SQLXML turns the components of an Updategram into a SQL statement.

The more details you include in the Updategram the more specific the generated where clause is going to be. It's important to remember this because we cannot explicitly leverage SQL Server's locking mechanisms when we query data with SQLXML. Since the query results are not associated with a persistent connection, we cannot hold any locks while our application modifies the data and writes it back to the database. That is a good architecture with respect to throughput and database scalability, because nobody has to wait for outstanding, but it poses certain challenges to ensure that two users do not overwrite each others changes. The only way to guarantee that this does not happen is to include every column in the <before> section, i.e. the WHERE clause of an update (or delete) statement. In terms of an Updategram this means we have to decide which elements we need to include in the <before> section when sending

Updategrams to the database. If SQLXML does not find a record matching the data in the <before> section of an Updategram it will always return an error to help us develop applications that can properly handle these concurrency issues.

Of course Updategrams are not limited to operations on a single SQL Server table. We can specify entire XML trees in the <before> and <after> sections and SQLXML will translate them into an SQL statement that spawns all the tables referenced by the XML types in the tree. In these cases it is extra important to define the relationships between the tables with the <relationship> annotation and identify key columns with the <key-fields> annotation in the mapping schema. SQLXML determines the order in which the changes to the database need to occur to avoid violating any constraints in the database. Imagine you need to insert an XML document with nested elements OrderHeader and OrderItem with the following Updategram:

```
<ROOT xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
  <updg:sync >
    <updg:after>
      <OrderHeader>
        <OrderID>1234</OrderID>
        <OrderDate>2002-9-4</OrderDate>
        <OrderItem>
          <ProductID>1</ProductID>
          <Quantity>10</Quantity>
        </OrderItem>
      </OrderHeader>
    </updg:after>
  </updg:sync>
</ROOT>
```

If you have a foreign key constraint on the OrderID from the OrderItem to the OrderHeader table then the correct order of the two inserts is very important. SQLXML needs to insert the row in the OrderHeader table before it inserts the row in the OrderItem table or SQL Server will abort with a constraint violation because the foreign key is not valid. With the <relationship> annotation SQLXML executes the SQL statements in the correct order. But that is not the only where we need to make sure the key-fields annotations are in place. SQLXML also requires the key-fields annotation to match up the corresponding elements when the <before> and <after> sections contain multiple elements of the same type. Matching up the elements on their key values is difficult though, if we are trying to change the key value with the updategram. In the next section show a solution to this problem.

## ***Annotating Updategrams***

In certain cases, we have to provide more information than just the elements in the <before> and <after> sections for the SQLXML engine to compose the correct SQL statement. When we need to set a database field to NULL for example, simply omitting the corresponding element or attribute from the Updategram will not set the field to NULL. Instead the value in the database remains unchanged because leaving the field we need to set out of the Updategram only causes the value to be excluded from the generated SQL statement.

To let SQLXML know that we want to set a field to NULL, we need to define which value in the Updategram represents NULL in the database and include it in the updategram. We define how NULL is represented in the updategram, by adding the nullvalue annotation to the <sync> element. Now we can reference the defined nullvalue in the updategram anywhere we need compare against or set something to NULL. The following snippet demonstrates the use of the nullvalue attribute to assign a value to WebSite field that was previously NULL:

```
<ROOT xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
  <updg:sync >
    <updg:before updg:nullvalue="(null)">
      <Supplier>
        <SupplierID>12345</SupplierID>
        <WebSite>(null)</WebSite>
      </Supplier>
    </updg:before>
    <updg:after>
      <Supplier>
        <SupplierID>12345</SupplierID>
        <WebSite>http://www.yoursupplier.com</WebSite>
      </Supplier>
    </updg:after>
  </updg:sync>
</ROOT>
```

Other situations where we need to annotate Updategrams, are retrieving automatically generated IDENTITY values and GUIDs and updates of designated key fields. Table 15.7 lists all annotation attributes for recognized by SQLXML. Section 15.3.4.4 discusses all issues around identity inserts in more detail.

**1.7 Annotation to updategrams provide meta information about the XML document beyond what is in the mapping schema. The updategram processing engine reads these annotations to generate the translate the updategrams to SQL statements correctly.**

Annotation	Valid On	Description
at-identity	<i>Inserted element or attribute</i>	Defines a place holder to receive SQL Server generated IDENTITY values. Other elements and attributes can reference the value by the placeholder.
guid	<i>Inserted element</i>	Defines a place holder for a GUID generated when the updategram executes.
id	<i>Inserted element</i>	Helps the updategram engine to track corresponding elements in the before and after sections of a sync block. Corresponding elements must carry the same id if the mapping schema does not identify the key-fields or if the updategram modifies a key field.
nullvalue	<sync>, <header>	Identifies the character sequence the update processor will treat as NULL.
returnid	<after>	Lists the SQL Server generated identities to return. The listed identifiers must match identity placeholders defined in at-identity annotations.

## Parameterized Updategrams

I mentioned that an Updategram is nothing but a special Template format. This implies that we can also work with parameters in Updategrams just like we can in Templates. The parameters

are defined in the optional <header> section and can be referenced throughout the rest of the Updategram, as shown in the following example:

```
<ROOT xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
  <updg:header>
    <updg:param name="ID" />
    <updg:param name="Name" />
  </updg:header>
  <updg:sync>
    <updg:after>
      <Supplier>
        <SupplierID>$ID</SupplierID>
        <SupplierName>$Name</SupplierName>
      </Supplier>
    </updg:after>
  </updg:sync>
</ROOT>
```

Parameterized Updategrams are useful in situations where we do not want to compose an Updategram on the fly or when we want to maintain a higher level of control over an Updategram's content. Another benefit of parameterized templates is the fact that we can keep the templates outside the compiled code, for example in resource DLLs or regular files, so we can perform minor bug fixes or simple updates without the need to recompile any code.

## .NET Managed Classes

Now that we know about the concepts of data access in SQLXML it is time to actually look at the managed classes to leverage them. SQLXML ships with COM objects as well as with managed code classes to interact with SQL Server in an XML-centric fashion. In the next four sections we will investigate the managed classes and see how we can issue XPath queries, send Updategrams, how we can interact with the DataSet class. Last but not least we will also take a look how we need to process errors occurring during SQLXML operations. Table 15.8 lists the managed classes of SQLXML.

### 1.8 The Microsoft.Data.SqlXml namespace contains four classes to access the SQLXML functionality from .NET applications.

Managed Class	Purpose
SqlXmlCommand	The command class of the SQLXML provider. Can perform queries using queries stored in XML documents called XML templates. Can be used to update data using an Updategram or a Diffgram. Also provides support for client side XML processing.
SqlXmlParameter	Used with a SqlXmlCommand object to provide parameter values for parameterized SQL statements, Templates and Updategrams.
SqlXmlAdapter	The Adapter class for the SQLXML provider. Allows you to fill a DataSet with data from a SqlXmlCommand.
SqlXmlException	The SqlXmlException encapsulates error information returned by SQL Server.

Notice that the SQLXML library does not expose a connection class. Data access with SQLXL never maintain any connections, which forces all your queries onto separate connections and isolates all your database updates. Consequently, there is no concept of distributed transactions spawning across multiple commands or databases that can be committed and rolled back with SQLXML. The only transacted operations are the changes defined by a single <sync> block of an Updategram. These only run in the context of a local SQL Server transaction. Local

transactions offer much better performance than the distributed transactions managed by COM+/.NET ServicedComponents, but they only run inside a single command execution and within a single database instance. Furthermore, you need to carefully design your Updategrams to tailor the transaction boundaries to the needs of your application. So much for the class that is not there, now let's look at the classes that are.

## **.NET SqlXmlCommand Class**

The `SqlXmlCommand` class is your one-stop shop to access the XML features of SQL Server and SQLXML. Similar to the `System.Data.SqlClient.SqlCommand` class or the `Command` class in classic ADO, we can issue commands to the database and add parameter values for parameterized queries. With the `SqlXmlCommand` class the actual command can be in one of the five different SQLXML formats: SQL statements returning XML, Updategrams, DiffGrams, XPath queries and Templates. We tell the `SqlXmlCommand` object which one of the formats to execute by setting the `CommandType` property to the corresponding value from the `SqlXmlCommandType` enumeration before we execute the command.

We have two options to specify the command text: the `CommandText` or the `CommandStream` property. The former is of type `string`, the latter of type `Stream`. `CommandText` helps us to avoid any conversions or memory copies when we dynamically create a SQL query or an XPath expression with a `System.Text.StringBuilder` object or a call to the `String.Format()`. `CommandStream` on the other hand reads the command from `Stream` object, which could be a `FileStream` to read from a file or a `MemoryStream` if we compose an Updategram dynamically in memory.

Once we have set the command text and its type we can execute the command. `SqlXmlCommand` gives us several options to retrieve the results. We can call the `ExecuteXmlReader()` method to get all the results nicely packaged in an `XmlReader` to traverse over the results. Alternatively, SQLXML will return the results as a raw stream. If we call the `ExecuteStream()` method the `SqlXmlCommand` will return a new stream and return it to us, or, if we want to inject the results into an existing stream, we supply the target stream to the `ExecuteToStream()` method. The next code example shows how we set up a `SqlXmlCommand` to execute a SQL query returning XML and then write the results directly to the Console with the `ExecuteToStream()` method.

```
static string connString =
"Provider=SQLOLEDB;Server=(local);database=XMLBookDB;";
public static void TestSqlXml ()
{
    Stream outputStream = Console.OpenStandardOutput();
    SqlXmlCommand cmd = new SqlXmlCommand(connString);
    cmd.Root = "Suppliers";
    cmd.CommandType = SqlXmlCommandType.Sql;
    cmd.CommandText = "SELECT SupplierName, WebSite FROM Suppliers
FOR XML AUTO";
    strm = cmd.ExecuteToStream( outputStream );
    outputStream.Close();
}
```

The example above also sets the Root property to instruct the SqlXmlCommand object to return well-formed XML. Without setting the Root property SQLXML would return the XML fragment returned by SQL Server without a common root node, which could cause exceptions if we processed the results with an XmlTextWriter or an XmlDocument for example. Having the SqlXmlCommand generating the root tag is not only convenient, it also avoids an extra, potentially expensive processing step if we had to add a root element around the returned results.

The SqlXmlCommand class exposes more properties we haven't discussed yet. The SchemaPath property, for example, specifies the location of the mapping schema for an XPath query or an Updategram which we will learn how to use shortly. Another property allows us to handle the transformation of the resultset into XML on the application server rather than using up processing cycles on SQL Server. We will discuss client-side XML generation later on. The SqlXmlCommand also does XSLT transformations as part of the command execution, but we will not examine them closer because we should use that feature only in a rather small application without well-defined application layers. Mapping schemas are sufficient to handle the initial transformation from the database schema into an XML format. The application layer should handle more complex transformations to keep the data access layer as generic as possible. Furthermore, the SqlXmlCommand class can only load XSL stylesheets from the file system, whereas the System.Xml.Xslt namespace allows processing of stylesheets from any type of stream source. Table 15.9 lists all methods and properties for quick reference purposes.

**1.9 The SqlXmlCommand class executes the difference command types in SQLXML: SQL Statements returning XML, XPath queries, Query Templates, Updategrams and Diffgrams.**

<b>Constructor</b>	
SqlXmlCommand( string connectionString )	Instantiates a new SqlXmlCommand from the specified connection string. The connection string is in OLE DB format and must reference the SQLOLEDB provider as in: Provider=SQLOLEDB; Server=(local); database=Northwind; user id=<UserLogin>; password=<UserPassword>.
<b>Properties</b>	
public bool ClientSideXml { get; set; }	With the ClientSideXml property set to true SQLXML, not SQL Server, handles the XML conversion of the query results. Converting the results outside SQL Server helps to offload conversion load to the application requesting the data. The property also allows you to wrap the existing stored procedures with FOR XML to get XML output. Check 15.3.3.1 for more details.
public string SchemaPath { get; set; }	Sets the mapping schema to reference for execution of this command. The path must be a local file system path or a UNC path, not a URL. It can be absolute or relative. Relative paths are relative to the path specified by the BasePath property or the executing process' current directory if BasePath is empty.
public string XsltPath { get; set; }	Sets the XSLT stylesheet to apply to the results of the command. The path must be a local file system path or a UNC path, not a URL. It can be absolute or relative. Relative paths are relative to the path specified by the BasePath property or the executing process' current directory if BasePath is empty.
public string BasePath { get; set; }	Sets the base path to resolve relative path references to schemas, XSLT stylesheets or a schema reference inside a query template or an updategram.
public string OutputEncoding { get; set; }	Sets the encoding for the returned stream. The encoding name must be recognized by SQLXML. The default is "UTF-8". Other options are "ANSI" and "Unicode".

public string Namespaces { get; set; }	Defines prefix declarations namespaces to use in XPath queries. Multiple declarations are separated by spaces.
public string RootTag { get; set; }	Sets the name of the root element for query results. If this properties is empty the command will return the XML fragment returned by SQL Server, regardless if it is well-formed XML.
public string CommandText { get; set; }	Sets the command text to execute. The command text can be any one of the types in the SqlXmlCommandType enumeration. Set the CommandType property to indicate the type.
public System.IO.Stream CommandStream { get; set; }	Passes a stream with the command text to execute. The stream can only contain commands of type Template, UpdateGram and DiffGram.
public SqlXmlCommandType CommandType { get; set; }	Specifies the command type in the CommandText or CommandStream property, which can be one of the following:  SqlXmlCommandType.Sql SqlXmlCommandType.XPath SqlXmlCommandType.Template SqlXmlCommandType.TemplateFile SqlXmlCommandType.UpdateGram SqlXmlCommandType.Diffgram  The default is Sql.
<b>Methods</b>	
public void ExecuteNonQuery()	Executes a command without returning any results.
public System.IO.Stream ExecuteStream()	Executes a command and returns the results in a stream..
public void ExecuteToStream(System.IO.Stream outputStream)	Executes a command and writes the results to the current position of the passed in stream. For example you can write the output of several command executions to the same stream.
public System.Xml.XmlReader ExecuteXmlReader()	Executes the command and returns an XmlReader to access the results.
public SqlXmlParameter CreateParameter()	Returns a SqlXmlParameter object that is automatically linked to the command. You can set name and value of the parameter to execute parameterize SQL statements or parameterized templates and updategrams.
public void ClearParameters()	Clears all parameters associated with the command.

## .NET SqlXmlParameter Class

The SqlXmlParameter class encapsulates the data for a query parameter in a SQL statement or a parameterized Template or Updategram. We have to create a SqlXmlParameter with the SqlXmlCommand's CreateParameter() method, since it does not expose a public constructor. The SqlXmlParameter class exposes a public property Value to set the parameter's value. The following code snippet demonstrates how to set up a parameterized SQL FOR XML AUTO query.

```
SqlXmlCommand cmd = new SqlXmlCommand(SouthRainConnString);
cmd.CommandText = "select SupplierName, WebSite from Suppliers where
    SupplierID = ? For XML Auto";
SqlXmlParameter parameter = cmd.CreateParameter();
parameter.Value = 1;
```

NOTE: We do not have to explicitly append a SqlXmlParameter object to a parameters collection like we do in classic ADO or with the SqlCommand class in ADO.NET. Parameters are automatically added to the SqlXmlCommand when we call CreateParameter().

The `SqlXmlParameter` class also exposes a `Name` property when we add named parameters to Templates or Updategrams. Table 15.10 lists the properties of the `SqlXmlParameter` class.

**1.10 Table 15.10: The `SqlXmlParameter` class encapsulates parameters passed to a SQLXML query.**

<b>Constructors</b>	
<i>None</i>	
<b>Properties</b>	
string Name { get; set; }	Sets the name of a named parameter for templates and updategrams.
object Value { get; set; }	Sets the value of a parameter.
<b>Methods</b>	
<i>None</i>	

## .NET `SqlXmlAdapter` Classes

The central piece of the ADO.NET architecture for data access is the `DataSet` class. It is intended to allow a unified access model to data from all kinds of data sources. Together with the `XmlDataDocument` class it also serves as a bridge between the world of relational data and hierarchical XML data structures. Many UI controls for ASP.NET and Windows Forms applications support the `DataSet` as their data source for data binding, which greatly simplifies programming for rather rich user interfaces.

Because of the `DataSet`'s central role, SQLXML added seamless interoperability with the `DataSet` with release 2.0. SQLXML provides a `SqlXmlAdapter` to fill a `DataSet` with the results of the execution of an `SqlXmlCommand`. The `SqlXmlAdapter` class also allows propagating updates made to a `DataSet` back to the original data source. The updates are passed to SQLXML directly in the `DataSet`'s Diffgram format, which was introduced to efficiently manage updates to a `DataSet`. The `SqlXmlAdapter`'s two methods, `Fill()` and `Update()`, allow updating a database with very few lines of code as we can see in the next example. When we work with a `DataSet` we don't have to write code to insert, update or delete data. All this functionality is already coded into the `SqlXmlAdapter` and the `DataSet`.

```
static string = "Provider=SQLOLEDB;Server=(local);database=XMLBookDB;";
public static void TestSqlXml ()
{
    Stream outputStream = Console.OpenStandardOutput();
    SqlXmlAdapter adapter = new SqlXmlAdapter(
        "SELECT SupplierName, WebSite FROM Suppliers
        FOR XML AUTO",
        SqlXmlCommandType.Sql,
        connectionString);

    DataSet ds = new DataSet();
    adapter.Fill( ds );
    // change something in the dataset
    ds.Tables[0].Rows[0][1] = "http://www.manning.com";
    // write the changes back to the data base
    adapter.Update( ds );
    outputStream.Close();
}
```

A typical use-case for this kind of architecture is an n-tier application where the business tier requires the XML format, but we want to build leverage the data binding capabilities of the ASP.NET web controls on the presentation tier. Table 15.11 gives a quick summary of the of the SqlXmlAdapter.

**1.11 The SqlXmlAdapter class enables seamless integration of the ADO.NET DataSet with SQLXML.**

<b>Constructors</b>	
public SqlXmlAdapter(SqlXmlCommand cmd)	Creates a new SqlXmlAdapter to fill a DataSet with the specified command.
public SqlXmlAdapter( string commandText, SqlXmlCommandType cmdType, string connectionString )	Creates a new SqlXmlAdapter to fill a DataSet from the database identified in the connectionString with the specified commandText of type cmdType.
public SqlXmlAdapter( Stream commandStream, SqlXmlCommandType cmdType, string connectionString )	Creates a new SqlXmlAdapter to fill a DataSet from the database identified in the connectionString with the specified command of type cmdType in commandStream.
<b>Properties</b>	
<i>None</i>	
<b>Methods</b>	
void Fill(DataSet ds)	Fills a DataSet with the command supplied to the constructor.
void Update(DataSet ds)	Applies the pending changes since the DataSet was filled or last updated back to the records in SQL Server.

## .NET SqlXmlException Class

The managed SQLXML classes throw a SqlXmlException to report any errors that occurred during the execution of a method. This class extends the System.Excpetion class and thus provides all the properties you are used to examine when you catch exceptions in a .NET application.

However, there are some cases where the HResult and the Message property do not provide enough information. Especially errors occurring during the execution of Updategrams often return an ambiguous HResult with the hexadecimal value 0x80040e21 and insufficient information in the Message property. In these cases we have to parse at the SqlXmlException's ErrorStream property. The ErrorStream contains the execution results – of course also in XML format where execution errors are reported as processing instructions.

One way to get to the details of the exception is to parse the ErrorStream with an XmlTextReader as outlined in the next code fragment:

```
try
{
    // do some SQLXML stuff
}
catch( SqlXmlException ex )
{
    XmlTextReader xmlReader = new XmlTextReader( _Ex.ErrorStream );
    while( xmlReader.Read() )
    {
```

```

    if( ( XmlNodeType.ProcessingInstruction == xmlReader.NodeType )
        && ( "MSSQLError" == xmlReader.Name ) )
    {
        Console.WriteLine( xmlReader.Value );
    }
}
}

```

Of course this example is not very useful because it writes the error to the console. I have taken a more useful approach in an small class you can find on the web site. It parses the `ErrorMessage` property and return all errors in a collection similar to the `SqlErrors` collection in the `System.Data` namespace.

**TIP:** In cases where even the `SqlXmlException` still does not provide enough details your best bet is to spy on SQLXML with SQL Server's profiler tool. This tool will show you exactly what queries SQLXML sends to the database. Make sure you run the profiler in a test environment, not in a production environment because it does occupy a fair amount of processing cycles on your database server.

## XML-enabling the Data Layer

So far, so good, we learned about the available features in SQL Server, we saw how we can access them from a .NET application, but what can we do with all this? Now let's explore how useful these SQLXML classes are really are, compared to ADO.NET for example. We will study different data access and conversion strategies for the data access layer of an XML-centric application. You can find a data access class based on the findings as part of the sample application on the web site.

We will work through a number of examples that all will take an XML document with order data and store the information in the document in the database. They will also query the database for the order data and return the results in an XML document. Each example will employ a different combination of techniques to access SQL Server and transform the results into XML. We will explore the pros and cons of the different approach with respect to performance, flexibility, extensibility and re-usability.

First let's see what tools we have in our XML toolbox by now. To read and write XML formatted data we have:

- The `XmlDocument` class, which is a little clunky to use if we are manually creating an XML document, but it allows random navigation and XPath queries against the data in the document. The big caveat of using an `XmlDocument` object is that it always holds the complete document in memory. This may not be acceptable if an application has to handle multi-megabyte documents on a regular basis.
- The `XmlTextReader`/`XmlTextWriter` classes. These classes do not require keeping an entire XML document in memory to read or write it, but they do not offer any navigation or query capabilities. Both are light-weight, but forward-only for high performance and small memory footprints.
- The `XmlSerializer` and serialization classes, which offer similar performance to the `XmlTextReader`/`XmlTextWriter` classes, but also require that all data is in memory. We can

access the data as an object hierarchy rather than a DOM after we deserialized an XML document.

- The DataSet class allows reading XML data into a relational table structure. We can also run queries against a DataSet or its alter ego, the synced XmlDocument. Like the XmlDocument and serialization classes the DataSet also keeps all data in memory.

To read and write XML from and to SQL Server we have:

- SQL Server's built-in XML features: FOR XML queries and inserts and updates using OPENXML executed from either the .NET Framework's native SQL Server client library or the SQLXML OLE DB data provider.
- XPath queries against annotated schemas executed through SQLXML.
- Updategrams and Diffgrams to perform insert, update and delete operations from SQLXML
- Templates to combine multiple FOR XML or XPath queries for more complex XML structures. We are not going to examine this approach in more detail since it merely combines the characteristics of executing SQL statements and XPath queries.
- Plain SQL to fill a DataSet with the results from a SqlCommand. We can save the DataSet as XML or an XmlDocument.

## ***Turning data into XML***

The first round of comparisons will evaluate several different approaches to query data from SQL Server and return the results as XML. We have a wide range of choices to obtain SQL Server data as XML, beginning with straight SQL queries and conversion by hand, over SQL Server's built-in SQL extensions to directly retrieve results in an XML format to executing XPath queries against an annotated mapping schema with SQLXML.

## ***The Traditional Method – Manual Conversion***

Before SQL Server 2000, SQLXML and the .NET Framework, there was exactly one solution to convert data from a SQL Server database to an XML format: issuing a SQL query and creating an XML document by looping through the returned recordset. Granted, ADO 2.x introduced some capability to persist an ADO Recordset object to XML, but you had to stick with the rather flat, table-like structure ADO generated for you; you could not customize the XML format.

We can still try this approach today, by executing a SQL statement over a SqlCommand object into a DataReader as outlined in the fragment below.

```
public static void WriteAllOrderData( SqlConnection connection,
    string rootName, XmlTextWriter writer )
{
    SqlCommand cmd = new SqlCommand("SELECT * from Orders", connection );
    SqlDataReader reader = cmd.ExecuteReader();
    writer.WriteStartElement( rootName ); // make sure we have a root
    while( reader.Read() )
    {
        writer.WriteStartElement( "Order" );
        writer.WriteElementString( "OrderID", reader.GetValue(0).ToString()
    }
}
```

```

);
    writer.WriteElementString( "CustomerID",
reader.GetValue(1).ToString() );

// write more XML by calling the appropriate WriteXXX method
}

```

While this approach may render the highest performance and allow for the highest level of customization of the generated XML format, it bears a huge development and maintenance liability. We have to tightly couple the data retrieval code and the XML transformation code and we have to write many, many lines of code for each XML type we introduce into our system. The more complex our XML format becomes the more complex become our data access classes as this approach relies on custom code to query and write each XML type.

## **XmlTextWriter**

SQL Server 2000 introduced the FOR XML extensions to SQL. We no longer have to write large amounts custom code for each XML type. All we need to do know is to carefully craft our SQL queries and SQL Server would do the conversion for us:

```

public static void GetSQLResultsAsXml( string query,
    SqlConnection connection, ref XmlTextWriter writer )
{
    SqlCommand cmd = new SqlCommand( query, connection );
    XmlReader reader = cmd.ExecuteXmlReader();
    writer.WriteNode( reader );
}

```

It is quite obvious that this approach results in far less code, than transforming the results by hand. Still we have to custom develop queries for each data type, but at least we only have to supply the query statements. The data access code can remain in one generic method and SQL Server and does the XML transformation for us. The downside is that we lose some of the flexibility as to what we can map from the database to XML. We are constrained by what SQL Server can return with one single query. In the case of FOR XML EXPLICIT mode queries we can create quite complex XML structures with one single query, but they are quite complicated to develop and maintain. From the performance perspective, the combination of SQL Server and the XmlTextWriter also delivers the XML data quite speedy.

## **SQLXML XPath Queries**

One slight change to the last approach to buy us an extra edge on the performance side would be to replace the SqlCommand with the SqlXmlCommand from the SQLXML library. The SqlXmlCommand uses the SQLXML OLE DB provider which performs better than the SQL Server library used by the SqlCommand class when it comes to retrieving XML. Combined with the either the StreamReader or the XmlTextWriter this combination will outperform all other approaches we examine in this section. Switching to the SqlXmlCommand also enables client-side XML formatting, i.e. we can move the load associated with the transformation of the results to XML from the data base server to the application server. We'll examine client-side formatting more detailed in section 15.3.3.1.

In the next approach we replace SQL queries with XPath expressions and annotated XML schemas. This approach is a great compromise between ease-of-development, speed, flexibility and maintainable code. The code is easy to maintain because all the mapping logic is maintained in the annotated schema outside the compiled code reducing the cost of changes to the mapping logic to a minimum. XPath queries are also significantly simpler to develop than FOR XML EXPLICIT queries. Sounds great already, but there is more. The annotations allow a great deal of flexibility how the SQL Server table structure maps to XML and we are still using the fast SQLXML OLE DB provider under the hood. If you are concerned about the impact of interpreting the schema and transforming the XPath expression to a FOR XML EXPLICIT query, you might feel better knowing that SQLXML caches the result of a schema's analysis. After the initial hit parsing the schema, XPath queries are similar in performance to the first approach, manually transforming results of a "regular" SQL query from the returned DataReader. The code example below shows that executing an XPath query is just as simple as in the previous example.

```
public static void ExecuteXPathQuery( string xpathQuery, string
    connectionString, string schemaPath, string rootName,
    ref Stream stream )
{
    SqlXmlCommand cmd = new SqlXmlCommand(connectionString);
    cmd.CommandText = xpathQuery;
    cmd.CommandType = SqlXmlCommandType.XPath;
    cmd.RootTag = rootName;
    cmd.SchemaPath = schemaPath;
    cmd.ExecuteToStream( stream );
}
```

With only a few generic lines we can execute the XPath expression and return the results in XML. The example below makes use of the `SqlXmlCommand` class' `ExecuteToStream()` method, which can speed up returning the results another notch. The XML transformation of the results happens now straight into the stream that is returned to the application, whereas the first two techniques required copying the query results to a separate output stream (wrapped by an `XmlTextWriter`). Just like the previous approach the `SqlXmlCommand` allows client-side formatting, which reduces the load we put on the database server.

## **.NET DataSet Class**

The last approach is a little bit different: We will not directly return XML from SQL Server, instead we store the query results in a `DataSet`. I added this technique to the lineup of this section a) because it's the most commonly used way to access SQL Server in .NET applications, b) it enables data binding to UI controls in Windows Forms and ASP.NET applications and c) we can produce XML from the `DataSet` either by calling `GetXml()`, `WriteXml()` or by linking an `XmlDataDocument` object to the `DataSet`. However, in the context of an XML-driven application this approach faces quite a few challenges to make it work.

The first challenge is tell the `DataSet` object about the XML format in which you would like to see the data when you call `GetXml()`. Elsewhere we have explained how we can either load an XML schema into the `DataSet` or how we can set the `ColumnMapping` property on each `DataColumn` object to define the format. We can also do all this work up front at development time and create a typed `DataSet`, which already contains the correct property settings for all XML

types, from an XML schema. OK, that was not a real challenge, but it is something to keep in mind.

The real challenge is to fill the DataSet correctly. There are no problems to correctly fill the typed DataSet with traditional SQL queries as long as each type in the XML schema has its counterpart in the database. With this approach we can benefit from the highly performance optimized classes built on top of native SQL Server access libraries instead of OLE DB and fill the DataSet with a method like the one shown below.

```
public static void FillDataSetFromSql( string[] sqlQueries,
    string[] tableNames, SqlConnection connection, ref DataSet ds )
{
    System.Diagnostics.Debug.Assert( sqlQueries.Length == tableNames.Length
);
    int count = queries.Length;
    for( int i = 0; i < count; i++ )
    {
        using( SqlCommand cmd = new SqlCommand( sqlQueries[i], connection ) )
        {
            using( SqlDataAdapter adapter = new SqlDataAdapter() )
            {
                adapter.SelectCommand = cmd;
                adapter.Fill( ds, tableNames [i] );
            }
        }
    }
}
```

Now there are cases where we do not have a 1:1 relationship between XML types and database tables. Sometimes XML formats introduce a common parent element around a group of elements of the same type, like the <OrderLines> element around the <OrderLine> elements.

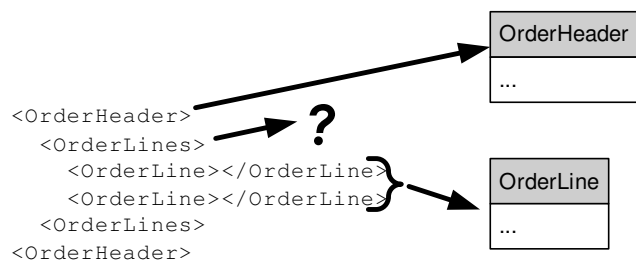


Figure 15.4: Example of an XML document with nested elements that do not map to database tables.

To create the OrderLines element the DataSet needs an OrderLines table, but there is no such thing in the database. Without the correct values in the OrderLines table the DataSet will not be able to create the correct nesting and render a sequence of OrderHeader and OrderLine elements.

In this case we have to forgo the performance advantage we gained from not converting the results to XML and fill the DataSet from a properly constructed XML string using the SqlXmlCommand instead. We could just as easy load data into the DataSet using the ExecuteXPathQuery from the previous example.

```

public void FillDataSetWithXPath( string query, string schemaPath,
    string connectionString, DataSet ds )
{
    SqlXmlCommand cmd = new SqlXmlCommand( connectionString );
    cmd.CommandText = query;
    cmd.CommandType = SqlXmlCommandType.XPath;
    cmd.SchemaPath = schemaPath;
    SqlXmlAdapter adapter = new SqlXmlAdapter( cmd );
    adapter.Fill( ds );
}

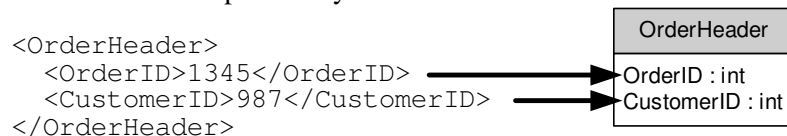
```

Once we filled the DataSet object we can get an XML representation of the data inside, or we can take advantage of the great data-binding capabilities of the UI controls in the .NET Framework.

So much for an overview or the possible options of querying XML from SQL Server, now let's move on to inserting data into SQL Server.

## Manually inserting SQLXML data

In this section we will examine different approaches to inserting data from an XML document into the database. The tables used in some of the examples are very simplified versions of tables in the database for the Southrain sample application, which you can download from the book's website at URL. Specifically we will insert data as shown in the following diagram:



Insert operation performed by the examples in this section

Note that this table does not define an automatically generated identity column. Identities add a whole order of complexity when we are inserting XML data. 15.3.4.4 will discuss the limitations and show some possible solutions.

Handling update and delete operations with updategrams is very similar to the inserts we are about to examine. In fact, INSERTs and DELETEs are special cases of an UPDATE as we have already seen in section 15.3.2.3. Hence we do not need separate sections for updates and deletes. The characteristics of the approaches we discuss in the section are equally true for update and delete operations.

The first approach we examine is again the conversion of XML to SQL by hand, just like you would if you were not using SQL Server 2000. In order to access the data in the XML document we first have to parse it. After reading your way up to this point you've seen quite a few different ways how to accomplish this with the classes in the .NET Framework.

Parsing with the XmlTextReader is not a good choice in this particular situation because of the forward-only access to the document. While accessing the XML data node-by-node is sufficient in simple cases like this example, it makes composing a SQL statement very hard once the document contains nested types that map to multiple database tables. You would have create

multiple SQL statements in parallel, which gets pretty tricky, or you would have to parse over your XML document multiple times, which would impact the performance of your insert method. I think you agree that the `XmlTextReader` is not a good choice here.

The other two options for parsing and XML document for random access are `Xml` serialization and, of course, the `XmlDocument`. The performance of the two approaches will vary with the complexity of the XML document you are parsing, for simple documents the `XmlDocument` has a slight edge over `Xml` Serialization. Both techniques do have similar memory requirements, too. You need available memory about twice the size of the `Xml` document to convert the raw document into the parsed representation. Keep this in mind if your application frequently inserts data from large XML documents.

The example below shows how we would parse the XML document with an `XmlDocument` object to create a SQL statement:

```
public static void InsertWithXmlDocument( string xml,
    string connectionString )
{
    XmlDocument doc = new XmlDocument();
    doc.LoadXml( xml );

    XmlNode orderNode = doc.FirstChild;
    string statement = String.Format( "INSERT OrderHeader ( "
        +"OrderID, CustomerID"
        +" ) VALUES ( " {0}, {1} )"
        , orderNode.ChildNodes[0].InnerXml
        , orderNode.ChildNodes[1].InnerXml );

    ExecuteNonQuerySQL( statement, connectionString )
}

public static void ExecuteNonQuerySQL( statement, ConnectionString )
{
    SqlConnection cn = new SqlConnection( connectionString );
    cn.Open();
    SqlCommand cmd = new SqlCommand( statement );
    cmd.Connection = cn;
    cmd.ExecuteNonQuery();
    cmd.Dispose();
    cn.Close();
}
```

Looking at the first method in this example you can imagine that generating the SQL INSERT for a large XML type quickly gets tedious. If the XML document we are trying to insert is made up from a heavily nested structure we will have to write code to create one SQL statement for each table into which we are inserting data. To make matters worse, none of the code in the first method is reusable. We will have to write more methods like this for every XML type we have to insert into our database and we can only hope that our schemas, both XML and the database rarely change, because changes might impact several methods in our data access component.

On the upside, inserting data this way is one of the fastest techniques in terms of overall performance. Not putting additional load on the database server because it never sees an XML document is another benefit of this approach. Finally, because we write custom code for every type anyway, there are no limitations as to what XML structures we can map to the database and in which format they are coming in. We can even map data from the XML document to different databases altogether.

## Using OPENXML to insert data

The second approach is a slight evolution of the first one. Instead of writing .NET code to extract the data from the XML document we write SQL code containing a SELECT statement combined with the OPENXML clause as shown in the next method.

```
public static void InsertOrderHeaderWithOPENXML( string xml,
    string connectionString )
{
    string statement = String.Format(
        @"DECLARE @idoc int
        DECLARE @doc varchar(1000)

        SET @doc='{0}'
        EXEC sp_xml_preparedocument @idoc OUTPUT, @doc

        INSERT OrderHeader ( OrderID, CustomerID )
        SELECT OrderID, CustomerID
        FROM OPENXML (@idoc, '/OrderHeader',2)
        WITH (
            OrderID int,
            CustomerID int)

        EXEC sp_xml_removedocument @idoc", xml );

    ExecuteNonQuerySQL( statement, connectionString );
}
```

Coding effort and maintenance liability for this technique are very similar to the previous one. The SQL code in this method is just as much special purpose as the C# code that composed the SQL from the content of an XmlDocument. The opportunity for re-using this function in any other context than inserting data from this particular XML format is zero. Handling more complex XML formats is also possible. We would simply add more INSERT...SELECT ... OPENXML ...WITH statements to insert data into more than one table.

The performance characteristics are slightly different than the last one. On the plus side, you don't need as much memory on the application server because you are not parsing the XML document to compose a SQL statement. On the other hand, you need the extra memory and the processing cycles on your database server, which lowers the overall scalability of your application. Of course this point is moot if your application is not deployed to separate servers.

Depending on your company's database philosophy you may want to move the SQL code into a stored procedure, which would yield better separation of XML conversion and database access code. In some workplaces, however access to databases is much more restricted than

access to an application server, in which case you might rather want to create SQL in your .NET components than executing stored procedures.

## Using UpdateGrams to insert data

Updategrams offer a different approach to inserting data from XML documents into the database. An annotated XML schema describes the mappings from the XML types in the schema to the tables in the database. We simply reference the schema when we execute the insert and in return we do not have to write a single line of code to handle the conversion. Take a look at the next three methods and see for yourself.

```
public static InsertOrderWithUpdategram( string xml,
    string connectionString, string schemaPath )
{
    ExecuteUpdategram( GetUpdategram( xml, null ),
        connectionString, schemaPath );
}

public static string GetUpdategram( string before, string after )
{
    const String UPDATEGRAM_NAMESPACE =
        "urn:schemas-microsoft-com:xml-updategram";
    const String MS_SQL_NAMESPACE =
        "urn:schemas-microsoft-com:xml-sql";
    StringWriter sw = new StringWriter();

    XmlTextWriter updateGramWriter = new XmlTextWriter(sw);
    updateGramWriter.WriteStartElement("ROOT");
    updateGramWriter.WriteAttributeString( "xmlns", "updg", null,
        UPDATEGRAM_NAMESPACE );
    updateGramWriter.WriteAttributeString( "xmlns", "sql", null,
        MS_SQL_NAMESPACE );
    updateGramWriter.WriteStartElement( "sync", UPDATEGRAM_NAMESPACE );
    updateGramWriter.WriteStartElement( "updg", "before",
        UPDATEGRAM_NAMESPACE );
    updateGramWriter.WriteRaw( before );
    updateGramWriter.WriteEndElement(); // before
    updateGramWriter.WriteStartElement( "after", UPDATEGRAM_NAMESPACE );
    updateGramWriter.WriteRaw( after );
    updateGramWriter.WriteEndElement(); // after
    updateGramWriter.WriteEndElement(); // sync
    updateGramWriter.WriteEndElement(); // ROOT

    updateGramWriter.Close();
    sw.Close();

    return sw.ToString();
}

public static void ExecuteUpdategram( string updategram,
    string connectionString, string schemaPath )
{
    SqlXmlCommand cmd = new SqlXmlCommand( connectionString );
```

```

cmd.CommandText = updategram;
cmd.CommandType = SqlXmlCommandType.UpdateGram;
cmd.SchemaPath = schemaPath;
cmd.ExecuteNonQuery();
}

```

There is not a single line in these three methods that is specific to the XML type we are inserting. We could just rename `InsertOrderWithUpdategram()` to `InsertWithUpdategram()` and run every insert and every update and delete through that single method. Now that is re-useable code! Moreover, keeping all the mapping definition in the schema makes a lot of sense, too, because you have to modify the schema if your XML type definitions change. Otherwise you could no longer validate incoming documents, regenerate the serialization classes or typed `DataSets` you are using in your application. Now you might as well keep your database mappings in there as well.

Now there has to be a catch somewhere, right? And of course there is: The performance of this approach is not quite as good as fully customized code, but it is within reasonable range. After all, `SQLXML` does cache the mapping schema after it first analyzed it and it does issue `INSERT` statements very similar to the ones we are creating by hand. `SQLXML` may only add some conversions, depending on the data types in the database and datatype annotations in the mapping schema.

The real catch, however, is the flexibility we lose when we switch to updategrams. The example above is so simple because we chose not to work with SQL Server's `IDENTITY` columns. If we are trying to insert data from into a table with an identity column and need to reference the generated identity to insert child element then things get quite a bit more complicated as we will see in 15.3.4.4. For now let's just note that updategrams are great for update and delete operations, but do require some extra work for certain inserts.

## Using `SqlXmlAdapter` to insert data

The last technique we examine in more detail is the insert (and update) capability of the `DataSet` in combination with the `SqlXmlAdapter`. Of all the insert techniques this one requires by far the least amount of code:

```

public static InsertWithDataSet( string xml, string connectionString,
    string schemaPath )
{
    DataSet ds = new DataSet();
    SqlXmlCommand cmd = new SqlXmlCommand( connectionString );
    cmd.SchemaPath = schemaPath;
    SqlXmlAdapter adapter = new SqlXmlAdapter( cmd );
    ds.ReadXml( xml );
    adapter.Update( ds );
}

```

The code to let a `DataSet` object and a `SqlXmlAdapter` object collaborate to insert data into SQL Server is very re-usable, since it does not require any XML specific code. Also, the `DataSet` has built-in support to process more than root object at a time. Again the mappings from XML to the database are defined in a mapping schema so we do not have to write and maintain any code to

parse XML and compose a SQL statement. The `SqlXmlAdapter()` calls `get GetChanges()` on the passed in `DataSet` to get one of those `DiffGrams` which it then hands off to the `SqlXmlCommand` for execution. The `SqlXmlCommand` then transforms the `DiffGram` into an `Updategram` and transforms that into SQL. Now you can already guess that this doesn't perform as well as directly composing an `updategram` with the extra transformation step in the chain of execution.

Another disadvantage is that we now have the three representations of the data in the XML document in memory: The original document, the `DataSet` object and the `DiffGram`, but the real show stopper is that you cannot insert nested XML types into the database if children require a foreign key from an automatically generated `IDENTITY`. The `DiffGram` does not know how to propagate the parent's `IDENTITY` to the children and the complete insert fails. In `SQLXML` release 3.0 there is no work-around for this problem. The only way to avoid this problem is to not handle those types of inserts from a `DataSet`.

### ***Also Ran ... Bulk Load***

`SQLXML` offers one more technique to insert data into SQL Server: XML Bulk Load. I want to mention it here because it has some nice features that are not available through any other upload technique: Fully transacted upload, for example, and automatic table generation if a table referenced in the source document is not present in the database. We're not going into detail here because it is not accessible through a managed .NET class, only through COM interop.

As you can derive from the name, the XML Bulk Load COM component is designed for loading large feeds of XML data. In contrast to all the other techniques we examined so far, `BulkLoad` does not require loading the complete source document into memory. It also relies on an annotated schema to map the XML to the database, i.e. we can build a very generic component to handle bulk uploads. On the negative side `Bulk Load` also cannot handle nested types where children require the `IDENTITY` of the parent as a foreign key.

## **SQLXML Performance**

We yet have to discuss what performance impact `SQLXML` will have on our applications. We get a lot of powerful, easy-to-use functionality out of `XPath` queries, `updategrams`, `templates` (and the `xsl` transformations that we have not discussed) but there is quite some processing going behind the scenes. Take an `XPath` query for example. First, `SQLXML` validates the query against the schema, which of course requires loading and parsing the schema. Then it converts the `XPath` query into a `FOR XML EXPLICIT` query which combines several `SELECT` queries to a `UNION`. Finally it transforms the resulting rowset to XML. All this processing has to create more load on the application than just reading a rowset and converting it to XML – if it wasn't for a few performance enhancing features in XML. To reduce the performance impact of the parsing the mapping schemas `SQLXML` does by default cache the parsing results, i.e. the major performance hit is out of the way once every schema in the system is loaded. Also to reduce total load on the database, `SQLXML` does allow to move the load of the XML conversion away from SQL Server. This section does examine both of these features and then show some benchmark results for the different transformations from the previous section.

## ***Client-Side XML formatting***

The first performance-enhancing feature in SQLXML is geared more towards maintaining scalability rather than increasing overall performance. Instead of having SQL Server converting query results, SQLXML lets us chose to do the conversion on the database client, hence the name client-side formatting.

SQLXML allows client-side XML formatting, which we can activate through a property on the `SqlXmlCommand` class. When the property is set to true SQLXML will strip the FOR XML clause from the query and send the plain SQL statement to retrieve a rowset. The SQLXML engine running on application server instead of the database server generates the XML. This feature is very important if your application is set up in the n-tier model, where multiple application servers access data from a shared database server. If all application servers query results in XML format, through either FOR XML or XPath queries, SQL Server has to convert the results for all application servers accessing the database. This lowers your application's overall scalability, especially if it is architected to scale horizontally, by adding more applications servers to increase capacity. Each application server not only increases the number of queries against the single database, it also adds an extra the XML processing liability. If you make use of SQLXML client-side formatting an XPath query only results in a UNION of one or more simple select statements, i.e. barely any more load than we would have without SQLXML.

Another application of client-side formatting that comes in very handy is XML-ifying existing stored procedures. Imagine you have a few hundred existing stored procedures that you would like to re-use in an XML-driven application. You could change each stored procedure to return the XML from the server, but then you would have to also change all the places that call the stored procedure. That's not a good solution, next you contemplate to make copies of all the stored procedures and change these to return XML. That's not a good solution either because then you have to remember to modify two procedures when changes are needed. The best solution might be to not change the stored procedures at all and let SQLXML do the XML formatting as shown in the example below:

```
SqlXmlCommand cmd = new SqlXmlCommand(connString);  
cmd.ClientSideXml = true;  
cmd.CommandText = "Exec GetAllOrders FOR XML NESTED, ELEMENTS";  
return cmd.ExecuteXmlReader();
```

The NESTED option on the FOR XML clause is similar(!) to the AUTO option used on the server, but we must only use it in conjunction with client-side processing. Specifically, NESTED is the only mode that we can apply to transform the output of a stored procedure. There are also a few little differences in the XML format AUTO mode and NESTED mode generate when it comes to querying views and when you define table aliases. AUTO names the returned elements after the view, while NESTED names them after the view's base tables. NESTED does not recognize table aliases, while AUTO uses the alias for element names. You need to keep in mind that SQLXML will ignore the `ClientSideXml` property if you execute an AUTO mode query and submit the FOR XML AUTO clause to SQL Server anyway. SQLXML can only apply client-side formatting to results from queries in RAW, NESTED and EXPLICIT mode.

Another difference between client-side formatting and server-side formatting is that you can execute statement prohibited by server generated XML. The FOR XML clause is not allowed in several situations like in a nested SELECT statement, in conjunction with a COMPUTE BY clause, or with GROUP BY and aggregate functions. But since SQLXML only sends the “regular” SQL part of the query to the server, you can avoid these limitations of server generated XML.

## **Schema Caching**

The second performance-enhancing feature we examine is schema caching. The first time we execute an XPath or an Updategram against an annotated schema, SQLXML will parse the schema and store the parsed schema in an internal cache. The next time we reference the schema SQLXML can skip loading and parsing the schema, which makes the execution of the query or the updategram much faster. The size of the cache is stored in the registry under the following registry key:

```
HKLM\SOFTWARE\Microsoft\MSSQLSERVER\Client\SQLXML3
```

You will find several values to control the sizes of different SQLXML caches. The default size of the schema cache is 31 schema files. You can increase the size if your application references more than 31 schemas. Otherwise SQLXML will purge a schema from the cache when it is full. Next time you reference the purged schema SQLXML will parse it again and purge another one. You get the idea right? Make sure your schema cache can hold all your schemas if you want maximum performance.

Mapping schemas are not the only item SQLXML caches to improve overall performance. Templates and XSLT stylesheets are also kept in dedicated caches. The cache sizes are also stored under registry key above.

Before we move on, I thought I would show some more concrete numbers to help you weighing the trade offs between flexibility and maintenance cost on one side against performance on the other.

I ran two series of tests, one for inserting data from an XML document into the database and one for querying results in an XML. I used SQL Server’s Northwind database, because you should all have it when you installed SQL Server. I modified the database only for the test where I needed to test the timing for client-side identity generation. I did not modify the database for any other test run. I ran each test 2000 times and averaged the results to get a reliable number. Then I normed the results to the fastest one of the tests, because I think you care more about the relative performance of the tests than how fast my particular server setup can run these benchmarks.

## **SQLXML Bencharks**

Now for the details. First let’s look at the query benchmark. I queried a hierarchy of nested elements, not just a simple element to table 1:1 scenario. I chose a hierarchy of Order and OrderDetail elements with the data from the tables Orders and Order Details. The five contestants in this benchmark were

- A SQL FOR XML AUTO query executed with the SqlCommand
- A SQL FOR XML AUTO query executed with the SqlXmlCommand

- an XPath query with SQLXML and an annotated schema,
- querying data into a DataSet with the SqlCommand and the SqlDataAdapter, setting up the DataRelations and writing the data out in the DataSet's default format
- querying the data into a DataSet with the SqlCommand and the SqlDataAdapter and then manually looping over the tables and create the XML with an XmlTextWriter.

I did not attempt the query the data into the faster DataReader because the code to create nested XML from several DataReaders can get quite convoluted.

And the winner, ladies and gentlemen, the winner is ... actually we have two winners. XPath and manually looping over the DataSet were equally fast as we can see in table 15.12. If we also consider that querying with XPath leads to much more maintainable code, I favor XPath queries over converting by-hand.

**1.12 Relative performance of different techniques to query data from SQL Server in XML format. Performance is listed relative to the fastest technique.**

Query Technique	Relative Performance
FOR XML AUTO (SQLXML)	100%
FOR XML AUTO (SQLClient)	102%
Xpath	150%
DataSet (WriteXml)	157%
DataSet (Explicit)	145%

The second test series examines the insert techniques from section 15.3.2. This time each test case had to insert an Order with two nested Order Detail elements. Specifically the lineup for these tests was:

- Reading the XML document into an XmlDocument, then composing a SQL Statement
- Reading the XML document into a DataSet, then composing a SQL statement
- Writing the XML to an Updategram, then transforming the updategram with XSLT to add the attributes required to handle the identity
- Reading the Xml with the XmlSerializer, generate a key value client-side, set the key value and serialize the data into an updategram.
- Parsing the XML document on the database server with an OPENXML statement.

The clear winner was the XmlDocument/SQL combination, followed by Updategram/XSLT the DataSet and team Updategram/XmlSerializer finishing last. Note that inserts with updategrams are even faster if we don't have to modify the XML document we insert. Without the extra parsing step involved Updategrams would lead the field of this benchmark.

**1.13 Relative performance of different techniques to insert data from an XML document into a SQL Server database. Performance is listed relative to the fastest technique.**

Insert Technique	Relative Performance
SqlDocumentByName	100%
DataSet	108%
UpdateGram w/ XSLT	107%
ClientSide w/ Serialization	154%
OPENXML	156%

So much for our little performance benchmark. Remember: actual mileage will vary with the complexity and the size of the XML documents you are processing. Do not take these results cast in stone! I strongly recommend that you download the benchmark code from the web site, adapt the code to your own schemas and see what results you are getting.

## SQL XML Pitfalls

Even though features like XPath queries and Updategrams make SQL Server appear like an XML database we need to realize that it's still the same old relational database engine under the hood. And the only interface to that relational engine is SQL. SQLXML does a good job to hide the SQL interface, but in some situations SQL's limitations just come right back in your face to remind you that certain things are not possible. The opposite is true, too. Certain things possible in SQL work differently in XPath. In the following section we will examine some of the limitations to help you stay clear of some of the pitfalls.

### *Watch Your Case*

XPath queries against an annotated schema are case sensitive. While SQL queries are case-insensitive the check of the XPath expression against the schema – like everything else in XML – is case sensitive. If SQLXML cannot validate the XPath expression against the schema it will return an error and not execute the query.

### *Typing*

You cannot include elements mapping to an image, text or ntext type column in the <before> section of an updategram. Elements in a <before> section wind up in the WHERE clause of an UPDATE or a DELETE statement, but SQL Server 2000 cannot compare against values in columns of these data types. Unfortunately you may not realize which elements map to a text type column until you start unit testing.

### *DateTime values*

The W3C standard for XML schemas defines that dateTime values in XML documents be formatted based on the ISO 8601 standard for dates and times. An XML value representing midnight on Christmas Day in the year 2002 in the US central time zone would look like:

```
<RequiredDate>2002-12-25T00:00:00.000-06:00</RequiredDate>
```

This format orders the fields in a way that you can apply alphanumerical a sorting algorithms to sort dateTime values, which is very helpful considering the poor support for date types in XSLT. However, SQL Server does not understand this particular flavor of ISO 8601 formatted dates. If you try to execute an Updategram containing a properly formatted dateTime value you will receive an error telling you that SQL Server does not like your date value – unless you annotate the element in the schema with the datatype annotation from the urn:schemas-microsoft-com:mapping-schema namespace:

```
<xs:element name="RequiredDate" type="xs:dateTime"  
  sql:datatype="dateTime" />
```

With the annotation in place SQLXML will convert the dateTime value to a format SQL Server recognizes prior to inserting or updating it.

## ***Before and After Updategram sections***

SQLXML needs to match up corresponding elements in the <before> and <after> sections of an updategram to determine what type of SQL to generate. It looks for the nodes identified as key-fields in the annotated schema and also for the updategram-specific id attribute in the updategram. If SQLXML fails to match up corresponding elements it generates two unrelated INSERT and DELETE statements to insert the element from the after section and delete the element from the before section. If you fail to catch a bug like this during development you could wind up deleting data from your production database that should not be deleted.

Make sure you always add the key-fields annotation to identify the key columns of a table. Also, if you know your application is going to change values in a key column to add the id attribute to the corresponding elements in your updategram.

## **Parent-Child Relationships**

One of the data guys I work with is a big advocate of guaranteeing data integrity through identity columns and foreign keys. Granted, identity columns work great if you like numeric key values, you only work with one database and you don't mind an extra round-trip to the database to retrieve the identity before you can insert it as a foreign key somewhere else. If you oppose to any of these items or if you prefer generating key values yourself instead of letting SQL Server do it, then you add the identity="useValue" annotation to your mapping schema and skip ahead. On the other hand, if you work with a data guy like I do and you have to deal with SQL Server generated identities you need to read this section.

Imagine you need to insert the data from the following XML document into a database where the OrderID column in the OrderHeader table is an IDENTITY column. The rows in the OrderLine table references their parent in the OrderHeader table by that OrderID column. Which of the insert techniques we discussed in section 15.3.2 would you choose?

```
<OrderHeader>
  <CustomerID>987</CustomerID>
  <RequiredDate>2002-12-25</RequiredDate>
  <OrderLine>
    <ProductID>12</ProductID>
    <UnitPrice>10</UnitPrice>
  </OrderLine>
</OrderHeader>
```

If you chose to implement your inserts by hand after you read sections 15.3.2.1 and 15.3.2.2 then you can skip right on to the Summary section because you do not have a problem – at least not with respect to inserting the foreign key values into the OrderLine table. You have to hand code all your SQL statements to insert your data, i.e. you can retrieve the IDENTITY after you insert the parent and propagate it to the children. If you chose to implement your inserts with DataSets and the SqlXmlAdapter then you did not pay attention when you read section 15.3.2.4. You

cannot insert nested XML elements if you need to propagate a SQL Server generated IDENTITY to children if you are doing the insert via a DiffGram, which is what the SqlXmlAdapter/SqlXmlCommand combination does.

If you chose to do your inserts via Updategrams despite the warning, then I better give you a few pointers how you can accomplish this. Updategrams allow you to specify a placeholder to reference the automatically generated identity from the child elements. We define the placeholder with the at-identity attribute at the root of an XML type.

```
<ROOT xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
  <updg:sync>
    <updg:after>
      <OrderHeader updg:at-identity="x">
        <OrderID>x</OrderID>
        <CustomerID>987</CustomerID>
        <RequiredDate>2002-12-25</RequiredDate>
        <OrderItem>
          <ProductID>1</ProductID>
          <Quantity>10</Quantity>
        </OrderItem>
      </OrderHeader>
    </updg:after>
  </updg:sync>
</ROOT>
```

With this restriction, writing a generic method to generate an Updategram as we did in 15.3.2.4 becomes harder. We can no longer insert the XML document as it is into an updategram. The at-identity attribute and the placeholders need to go into the XML document. To add them we can either read the XML document into an XmlDocument object to add the attribute and the identity references, or we can run the document through an XSLT. Either way we lose the major benefit we gained from using updategrams: easy to maintain code that was independent of the XML format our applications are dealing with. Furthermore the extra step parsing or transforming the document to add the identity placeholders requires extra processing cycles and demands more memory to build the complete updategram.

You have to decide for yourself if these issues around inserts are show stoppers adopting updategrams in your application. If you want to use them for updates and deletes you may as well stick with them for inserts. Or maybe your primary use case involves running XPath queries and you are already have to maintain a mapping schema, inserts happen rarely and you do not care so much about the performance hit. The best thing you can to do is to look at your XML schema and see how hard it is in your case to add the placeholders. Maybe you run a few tests to see how badly the performance is impacted in your scenario. The book's web site has one example on how to insert the at-identity attribute if you can assume that the relationship is realized with elements of the same name in parent and children. Maybe this code gets you going into finding a solution for your scenario.

## Summary

We have seen that we can implement XML-based data access without moving to a native XML database. We can stick with SQL Server 2000 and keep all our existing applications from before the XML era. You can also keep all our third party reporting tools and whatever else you have. SQL Server 2000 and SQLXML offer some very interesting features that simplify developing XML-based data access without major impact to an application's scalability. Especially when it comes to retrieving XML from SQL Server I cannot think of a good reason to not let SQLXML handle the XML formatting of the results.

Updategrams and Diffgrams to insert data and performing updates are not quite as much a hands-down winner as XPath queries. They are still "held hostage" by the gritty details of the relational model and SQL, but bear in mind that with the next release of SQL Server, XML, as well as the .NET Framework, become completely integrated into the database.

Overall, XML based data access already simplifies the development of XML-enabled applications. Annotated schemas keep all XML to database mapping rules in one place and XPath queries are a concise and powerful alternative to complex statements JOINing data from multiple tables.

After all the hierarchical structure of XML maps more natural to the hierarchical object models we use in our applications. Combine data binding with the XmlSerializer and SQLXML for example and you have a simple Object/Relational mapper. You can load and persist objects to the database without developing custom code. The mappings are defined in an annotated XML schema, which we can use to generate C# or Visual Basic.NET classes. The XmlSerializer handles the object-to-XML conversion while SQLXML does the XML-database mapping. There are many, many more use cases for XML and XML-based database access. You know have the knowledge to pick the best solution for your particular scenario.

---

**TopXML**

Great XML tools

For more great .NET and XML content go to <http://www.topxml.com>

---