XML and Internet Support Overview

Microsoft® SQL Server[™] 2000 introduces new features that support XML functionality. The combination of these features makes SQL Server 2000 an XML-enabled database server. These new features include:

- The ability to access SQL Server using HTTP.
- Support for XDR (XML-Data Reduced) schemas and the ability to specify XPath queries against these schemas.
- The ability to retrieve and write XML data:
 - Retrieve XML data using the SELECT statement and the FOR XML clause.
 - Write XML data using OPENXML rowset provider.
 - Retrieve XML data using the XPath query language.
- Enhancements to the Microsoft SQL Server 2000 OLE DB provider (SQLOLEDB) that allow XML documents to be set as command text and to return result sets as a stream.

For the latest updates relating to SQL Server support for XML, see the XML Developer Center on MSDN® at <u>Microsoft Web site</u>.

Note The Msxml2.dll is installed with SQL Server 2000, but additional tools are not installed. For example, Xmlinst.exe, the tool used to configure Microsoft Internet Explorer to use MSXML2, is not installed. The full MSXML2 package must be installed to obtain this functionality. MSXML2 can be downloaded from the XML Developer Center on MSDN at <u>Microsoft Web site</u>.

Getting Started with XML

To use the XML functionality that Microsoft SQL Server 2000 provides, you

must have a working knowledge of XML, URL syntax, and HTTP methods. You should also be familiar with these terms:

XML document

Is a document that contains XML elements and attributes.

Document Type Definition (DTD)

Defines the elements and attributes that can be used in an XML document.

Style sheet

Describes the way data is to be formatted or displayed. The Extensible Stylesheet Language (XSL) is the language that is commonly used with XML documents.

Form

Is a structured document used to collect and submit data for processing.

Template

A concept introduced in SQL Server 2000, a template is a valid XML document containing one or more SQL statements. The template files are used to specify queries (SQL and XPath queries). Instead of specifying queries in the URL, template files containing the queries are specified in the URL.

Virtual root

A concept introduced by Microsoft Internet Information Services (IIS), the virtual root is usually administered as part of IIS.

See Also

Using IIS Virtual Directory Management for SQL Server Utility

Accessing SQL Server Using HTTP

Creating XML Views Using Annotated XDR Schemas

Using XPath Queries

Retrieving and Writing XML Data

IIS Virtual Directory Management for SQL Server

The IIS Virtual Directory Management for SQL Server utility is provided to create a virtual root specific to Microsoft® SQL Server[™] 2000. You can interact with the IIS Virtual Directory Management for SQL Server application in two ways:

- Graphically, using the IIS Virtual Directory Management for SQL Server utility.
- Programmatically, using the IIS Virtual Directory Management for SQL Server object model.

See Also

Accessing SQL Server Using HTTP

System Requirements for IIS Virtual Directory Management

The IIS Virtual Directory Management for SQL Server utility can run on a computer running any edition of Microsoft® Windows NT® 4.0 or Microsoft Windows® 2000. Computers running Windows NT 4.0 require:

- Microsoft Internet Information Server 4.0 or higher (or Peer Web Services 4.0 or higher on Windows NT Workstation 4.0).
- Microsoft Management Console 1.2 (installed by the Windows NT Option Pack and by SQL Server 2000 Setup).

For computers running Microsoft Windows 2000 Professional, the Administrative Tools pack (Adminpak.msi) must be installed. This file is located in the %windir%\System32 folder of the Windows 2000 Server editions.

Using IIS Virtual Directory Management for SQL Server Utility

Before accessing a Microsoft® SQL Server[™] 2000 database using HTTP, you must set up an appropriate virtual directory. Use the IIS Virtual Directory Management for SQL Server utility (click **Configure SQL XML Support in IIS** in the **SQL Server Tools** program group) to define and register a new virtual directory, also known as the virtual root, on the computer running Microsoft Internet Information Services (IIS). This utility instructs IIS to create an association between the new virtual directory and an instance of Microsoft SQL Server. For information about the user interface for this utility, see <u>IIS Virtual Directory Management Utility</u>.

The name of the IIS server and the virtual directory must be specified as part of the URL. The information in the virtual directory (including login, password, and access permissions) is used to establish a connection to a specific database and execute the query.

The URL can be specified to:

• Directly access the database objects, such as tables.

In this case, the URL would include a virtual name of **dbobject** type.

• Execute template files.

A template is a valid XML document consisting of one or more SQL statements. When a template file is specified at the URL, the SQL commands stored in the template file are executed. SQL queries can be directly specified at the URL, but this is not recommended for security reasons.

• Execute XPath queries.

The XPath queries are executed against an annotated mapping schema file specified as part of the URL.

Virtual Names

To allow a template file, mapping schema file, or a database object (such as a table or view) as part of the URL, virtual names of type **template**, **schema**, and **dbobject** must be created. The virtual name is specified as part of the URL to execute a template file, an XPath query against a mapping schema file, or to access a database object directly.

The type (**template**, **schema**, **dbobject**) of the virtual name specified at the URL is also used to determine the file type specified at the URL (template file or a mapping schema file). For example, this URL accesses a SQL Server database using a template:

http://IISServer/nwind/TemplateVirtualName/Template.xml

TemplateVirtualName is a virtual name of **template** type, which identifies that the specified file (Template.xml) is a template file.

See Also

Accessing SQL Server Using HTTP

Creating the nwind Virtual Directory

This example creates the **nwind** virtual directory. The **nwind** virtual directory is used in most of the examples that are used to illustrate URL access to Microsoft® SQL Server[™] 2000.

Before you create the **nwind** virtual directory, you need a physical directory associated with the virtual directory that you are creating (for example, C:\Inetpub\Wwwroot\nwind where **nwind** is the physical directory associated with the **nwind** virtual directory that is created in the following procedure).

You also need to create two subdirectories in the physical directory associated with the virtual directory (for example, C:\Inetpub\Wwwroot\nwind\template, and C:\Inetpub\Wwwroot\nwind\schema). These are the directories associated with the virtual names of **template** and **schema** types that are created as part of creating **nwind** virtual directory.

To create the nwind virtual directory

- 1. In the **Microsoft SQL Server** program group, click **Configure SQL XML Support in IIS**.
- 2. Expand a server, and then click the Web site you want.
- 3. On the **Action** menu, point to **New**, and then click **Virtual Directory**. The property page for the new virtual directory is displayed on the screen.
- 4. On the General tab of the New Virtual Directory Properties dialog box, enter the name of the virtual directory. For this example, type nwind and the physical directory path (for example, C:\Inetpub\Wwwroot\nwind, assuming you have a subdirectory nwind created in the C:\Inetpub\Wwwroot directory). You can optionally use the Browse button to select the directory.

- 5. On the **Security** tab, select **SQL Server** and enter the valid SQL Server login information. When you go to the next tab, you will be asked to confirm the password you just entered.
- 6. On the **Data Source** tab, in the **SQL Server** box, enter the name of a server, for example (**local**), and optionally, the name of an instance of SQL Server 2000 if more than one instance is installed on the specified computer. In the **Database** box, enter **Northwind** as the name of the default database.
- 7. On the **Settings** tab, select the **Allow URL queries**, **Allow template queries**, **Allow XPath**, and **Allow POST** options.
- 8. On the **Virtual Names** tab, click **New** to create the virtual name for the template type.

In the **Virtual Name Configuration** dialog box:

- Enter **template** in the **Virtual name** box (it can be any user specified name). In the **Type** list, select **template**. Enter the path (for example, C:\Inetpub\Wwwroot\nwind\template, assuming there is a subdirectory **template** in the physical directory associated with the virtual directory, however the existence of the path is not checked). Click **Save** to save the virtual name.
- 9. On the **Virtual Names** tab, click **New** to create the virtual names for the schema type.
 - Enter **schema** in the **Virtual name** box (it can be any user specified name). In the **Type** list, select **schema**. Enter the path (for example, C:\Inetpub\Wwwroot\nwind\schema, assuming there is a subdirectory **schema** in the physical directory associated with the virtual directory). Click the **Save** button to save the virtual name.
- 10. On the **Virtual Names** tab, click **New** to create the virtual names for the template and schema types.

- Enter **dbobject** in the **Virtual name** box (it can be any user specified name). In the **Type** list, select **dbobject**. Click the **Save** button to save the virtual name.
- 11. Click **OK** to save the settings.

This creates a virtual directory **nwind**. The queries specified using this virtual directory are, by default, executed against the **Northwind** database.

To test the virtual directory, in the browser type: **http:**//<**IISServer**>/**nwind**? **sql=SELECT * FROM Employees FOR XML AUTO&root=root** and press ENTER.

See Also

Using IIS Virtual Directory Management for SQL Server Utility

IIS Virtual Directory Management for SQL Server Object Model

The IIS Virtual Directory Management for SQL Server object model consists of these objects:

- <u>SQLVDirControl object</u>
- <u>SQLVDirs collection object</u>
- <u>SQLVDir object</u>
- <u>VirtualNames collection object</u>
- <u>VirtualName object</u>

In an object model, the content and functionality of an application are provided by objects. The objects are units of related content and functionality. A collection object, on the other hand, is an object that contains a set of related objects. You can use a collection object to get to an individual object, usually with an **Item** method. For example, you can use the **Item** method of the **SQLVDirs** collection object to access one of the virtual directories.

In this object model, **SQLVDirControl** is the top-level object and is the only object that can be created directly. All other objects must be obtained from the **SQLVDirControl** object or its derivatives.

The object hierarchy (object model) is the way the objects in the application are arranged relative to each other. This illustration shows the object hierarchy in the object model to create a virtual root.

See Also

Accessing SQL Server Using HTTP

SQLVDirControl Object

SQLVDirControl is the only object in the object hierarchy that can be accessed directly using Automation. All other objects are accessed through this object or its derivatives.

SQLVDirControl supports these methods:

Connect

Connects to a specific Microsoft® Internet Information Services (IIS) server. The two parameters to this method are: *IIS Server name* and *Web Site number* (the number of the Web site in the metabase tree). If none of the parameters is supplied, the local server is the default value for the IIS server, and the first Web site (default Web site) on that IIS server is selected as the default.

Disconnect

Disconnects from the last connected IIS server and the Web site. There are no parameters to this method. You must call **Disconnect** to close the connection when you are finished or before you connect to another server or Web site.

Note You cannot issue multiple **Connect** calls to establish connections to many IIS servers. However, if you are trying to connect to the same IIS server but possibly different Web sites, you can call **Connect** multiple times without calling **Disconnect**.

SQLVDirs

The **SQLVDirs** method retrieves the virtual directory collection of the Web site to which you are connected. After you are connected to a specific IIS server and Web site, call the **SQLVDirs** to obtain the **SQLVDirs** collection object, which provides access to the virtual directory objects.

Examples

This example shows how to connect to a specific IIS server and Web site on the IIS server. The first Web site is selected on the IIS server.

Set ObjXML = CreateObject("SQLVDir.SQLVDirControl") ObjXML.Connect "IISServer", "1" Set ObjDirs = ObjXML.SQLVDirs

•••

ObjXML.Disconnect

SQLVDirs Collection Object

The **SQLVDirs** collection object is returned by the

SQLVDirControl.SQLVDirs method. With the **SQLVDirs** collection object, you can access a specific virtual directory (using the **Item** method), create a new virtual directory (using the **AddVirtualDirectory** method), or remove an existing virtual directory (using the **RemoveVirtualDirectory** method).

The **SQLVDirs** collection object supports these standard methods:

Next method

Retrieves the next virtual directory (or directories). An integer specified for **Next** determines the number of virtual directories to retrieve.

Skip method

Skips the virtual directory (or directories). A number specified for **Skip** determines how many virtual directories to skip.

Reset method

Resets the collection index to the first virtual directory.

Clone method

Returns a copy of the **SQLVDirs** collection object.

Count method

Returns the number of virtual directories.

Item method

Retrieves one virtual directory. You can specify an integer (starting with **0** for the first virtual directory) or the name of the virtual directory.

The **SQLVDirs** collection object also supports these methods:

AddVirtualDirectory method

Takes the name of the virtual directory to create. This method creates a new virtual directory in the metabase with all the defaults. However, some properties, like the default database are not set.

The **AddVirtualDirectory** method or **Item** method returns a **SQLVDir** object that represents the virtual directory.

RemoveVirtualDirectory method

Removes the virtual directory from the Microsoft® Internet Information Sevices (IIS) metabase.

Examples

This example establishes a connection to the first Web site on an IIS server. The first virtual directory object (index value **0**) on the connected Web site is accessed.

```
Set ObjXML = CreateObject("SQLVDir.SQLVDirControl")
ObjXML.Connect "IISServer", "1"
Set ObjVDirs = ObjXML.SQLVDirs
Set ObjVDir = ObjVDirs.Item(0)
' or ObjVDirs(0) since Item() is the default
```

... 01 ·

ObjXML.Disconnect

SQLVDir Object

The **SQLVDir** object is a virtual directory object obtained by calling the **Item** method (or by calling the **AddVirtualDirectory** method if you are creating a new virtual directory).

The **SQLVDir** object supports the following properties. You can get and set values for all of these properties except the **Password** property (for which you can set, but cannot get, the value).

Name property

Is the name of the virtual directory.

PhysicalPath property

Is the full physical path to the directory associated with the virtual directory.

ServerName property

Is the name of the server running Microsoft® SQL Server[™] 2000, which is the data source for the virtual directory.

DatabaseName property

Is the default database used in queries against this virtual directory.

UserName property

Is the user login that is used to connect to the data source.

Password property

Is the user password that is used to connect to the data source.

SecurityMode property

Is the login authentication method that is used with the virtual directory, such as SQL Authentication or Windows Integrated Authentication. You can specify one of these values.

Value	Description
1	SQL Server login

2	Microsoft Windows® anonymous login
4	Basic authentication
8	Windows Integrated Authentication

CAUTION If you are changing the connection settings (changing server name, database name, user name, password or the security mode), it is recommended that the virtual directory access be disallowed. The virtual directory can be disabled by setting the **AllowFlags** property to **0**.

AllowFlags property

Provides the type of access allowed through this virtual directory. You can specify one (or a combination) of these values.

Value	Description
1	URL queries
8	Template access
64	XPath queries

EnablePasswordSync property

Specifies whether Microsoft® Internet Information Services (IIS) is allowed to handle the anonymous password synchronization.

DLLPath property

Provides the full path to the Sqlisapi.dll.

AdditionalSettings property

Are user-defined settings appended to the OLE DB connection string.

The **SQLVDir** object also supports this method:

VirtualNames method

Is the collection of virtual name mappings for the virtual directory.

Examples

This example establishes a connection to the first Web site on an IIS server. The

first virtual directory object (index **0**) on the connected Web site is accessed. The **PhysicalPath** property of the object is set to **C:\inetpub.**

```
Set ObjXML = CreateObject("SQLVDir.SQLVDirControl")
ObjXML.Connect "IISServer", "1"
Set ObjVDirs = ObjXML.SQLVDirs
Set ObjVDir = ObjVDirs.Item(0)
ObjVDir.PhysicalPath = "C:\"
ObjVDir.PhysicalPath = ObjVDir.PhysicalPath & "inetpub"
...
ObjXML.Disconnect
```

VirtualNames Collection Object

The **VirtualNames** collection object is a collection of virtual names in the virtual directory object (**SQLVDir** object). The **VirtualNames** collection object is similar to the **SQLVDirs** object (which is a collection of virtual root objects). The **VirtualNames** collection object supports these standard methods:

Next method

Retrieves the next virtual name (or names). An integer specified for **Next** determines the number of virtual names to retrieve.

Skip method

Skips the virtual name (or names). A number specified for **Skip** determines how many virtual names to skip.

Reset method

Resets the collection index to the first virtual name.

Clone method

Returns a copy of the VirtualNames collection object.

Count method

Returns the number of virtual names.

Item method

Retrieves one virtual name. You can specify an integer (starting with **0** for the first virtual directory) or the name of the virtual name.

The VirtualNames collection object also supports these methods:

AddVirtualName method

Passes the name of the virtual name, type of the virtual name, and the directory path associated with the virtual name to create. The **AddVirtualName** method or **Item** method returns an interface to a **VirtualName** object that represents the virtual name.

RemoveVirtualName method

Removes the virtual name that is specified.

Examples

This example shows the steps for accessing a **VirtualNames** collection object.

```
Set ObjXML = CreateObject("SQLVDir.SQLVDirControl")
ObjXML.Connect "IISServer", "1"
Set ObjVDirs = ObjXML.SQLVDirs
Set ObjVDir = ObjVDirs.Item(0)
Set ObjNames = ObjVdir.VirtualNames
...
```

ObjXML.Disconnect

VirtualName Object

The **VirtualName** object is obtained by calling the **Item** method (or by calling the **AddVirtualName** method if you are creating a new virtual name).

The **VirtualName** object supports these properties:

Name property

Is the name of the virtual name that is being created.

Type property

Is the virtual name type. You can specify one of these values.

Value	Description
1	Virtual name of type dbobject .
2	Virtual name of type schema .
4	Virtual name of type template .

Path property

Is the directory path (absolute or relative) associated with the virtual name.

Examples

This example shows the steps for accessing an existing **VirtualName** object and for setting some of its attributes.

```
Set ObjXML = CreateObject("SQLVDir.SQLVDirControl")
ObjXML.Connect "IISServer", "1"
Set ObjVDirs = ObjXML.SQLVDirs
Set ObjVDir = ObjVDirs.Item(0)
Set ObjNames = ObjVdir.VirtualNames
Set ObjVName1 = ObjNames.Item(0)
ObjVName1.Type = 2
ObjVName1.Name = "MySchema"
ObjVName1.Path = "C:\inetpub\schema"
```

... ObjXML.Disconnect

This statement creates a new virtual name:

Set NewVName = ObjNames.AddVirtualName "MyNewSchema", 2, "

Creating the nwind Virtual Directory Using the Object Model

This Microsoft® Visual Basic® Scripting Edition (VBScript) sample creates the same **nwind** virtual directory that is described in <u>Creating the nwind Virtual</u> <u>Directory</u>.

```
Set ObjXML = CreateObject("SQLVDir.SQLVDirControl")
ObjXML.Connect 'Connect to the local computer and Web site "1"
```

```
Set ObjVDirs = ObjXML.SQLVDirs
Set ObjVDir = ObjVDirs.AddVirtualDirectory("nwind")
```

```
'General tab in UI
ObjVDir.PhysicalPath = "C:\Inetpub\wwwroot\nwind"
```

```
'Security tab in UI
ObjVDir.UserName = "sa" 'SQL Server login
ObjVDir.Password = "" 'SQL Server Password (is "" per default)
```

```
'Data source tab in UI
'(local) is default for the SQL Server
ObjVDir.DatabaseName = "Northwind"
```

```
'Settings tab in UI
objVDir.AllowFlags = 73 'afURL_QUERIES OR afTEMPLATES OR
```

```
'Virtual Name Configuration tab in the UI
Set objVNames = objVDir.VirtualNames
objVNames.AddVirtualName "dbobject", 1, ""
objVNames.AddVirtualName "schema", 2, "C:\Inetpub\wwwroot\nwir
objVNames.AddVirtualName "template", 4, "C:\Inetpub\wwwroot\nw
```

'Disconnect from the server. objXML.Disconnect

msgbox "Done."
Accessing SQL Server Using HTTP

You can access Microsoft® SQL Server[™] 2000 using HTTP. For more information about the URL syntax that is support by the SQL ISAPI extension, see <u>URL Access</u>. Before queries can be specified using HTTP, a virtual root must be created using the IIS Virtual Directory Management for SQL Server utility. For more information, see <u>Creating the nwind Virtual Directory</u>.

The HTTP access to SQL Server allows you to:

• Specify SQL queries directly in the URL, for example: http://IISServer/nwind?sql=SELECT+*+FROM+Customers+F

The FOR XML clause returns the result as an XML document instead of a standard rowset. The root parameter identifies the single top-level element.

• Specify templates directly in the URL.

Templates are valid XML documents containing one or more SQL statements. The templates allow you to put together data to form a valid XML document, which is not necessarily the case when queries are specified directly in the URL. For example:

http://IISServer/nwind?template=<ROOT+xmlns:sql="urn:sch@initial.com/initial.c

• Specify template files in the URL.

Writing long SQL queries at the URL can be cumbersome. In addition, browsers may have limitations on the amount of text that can be entered in the URL. To avoid these problems, templates can be written and stored in a file. A template is a valid XML document containing one or more SQL statements and XPath queries. You can specify a template file directly in a URL, for example:

http://IISServer/nwind/TemplateVirtualName/templatefile.xml

In the URL, **TemplateVirtualName** is the virtual name of **template** type that is created using the IIS Virtual Directory Management for SQL

Server utility.

Template files also enhance security by removing the details of database queries from the user. By storing the template file in the virtual root directory (or its subdirectories) where the database is registered, security can be enforced by removing the URL query-processing service on the virtual root, and leaving only the SQL Server XML ISAPI to process the files and return the result set.

• Write XPath queries against the annotated XML-Data Reduced (XDR) schemas (also referred to as mapping schemas).

Writing XPath queries against the mapping schemas is conceptually similar to creating views using the CREATE VIEW statement and writing SQL queries against them, for example:

http://IISServer/nwind/SchemaVirtualName/schemafile.xml/Cu

In the URL:

- **SchemaVirtualName** is the virtual name of **schema** type that is created using the IIS Virtual Directory Management for SQL Server utility.
- **Customer[@CustomerID="ALFKI"]** is the XPath query executed against the **schemafile.xml** specified in the URL.
- Specify database objects directly in the URL.

The database objects, such as tables and views, can be specified as part of the URL, and an XPath can be specified against the database object, for example:

http://IISServer/nwind/dbobjectVirtualName/XpathQuery

In the URL, **dbobjectVirtualName** is the virtual name of **dbobject** type that is created using IIS Virtual Directory Management for SQL Server utility.

Note When an operation that requires resources such as memory

(creating temporary tables and temporary stored procedures, declaring cursors, executing **sp_xml_preparedocument**, and so on) is executed at the URL, the resources must be freed by executing appropriate corresponding commands (such as, DROP TABLE, DROP PROCEDURE, DEALLOCATE the cursor, or EXECUTE **sp_xml_removedocument**).

XML Documents and Document Fragments

When you execute a template or a query with the **root** parameter, the result is a full XML document with a single top-level element. For example, this URL executes a template:

http://IISServer/VirtualRoot/TemplateVirutalName/MyTemplate.xml

This is a sample template file (MyTemplate.xml):

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:query>
SELECT *
FROM Customers
FOR XML AUTO
</sql:query>
</ROOT>
```

The <ROOT> tag in the template provides the top-level single element for the resulting XML document.

The queries can be specified directly in the URL. In this case, the **root** parameter specifies the top-level element of the document returned:

http://IISServer/VirtualRoot?sql=SELECT * FROM Customers FOR X

If you write the same query without the **root** parameter, an XML document fragment (an XML document without the single top-level element) is returned. This fragment has no header information. For example, this URL returns a document fragment:

http://IISServer/VirtualRoot?sql=SELECT * FROM Customers FOR X

The byte-order mark identifying the document encoding is returned when you request an XML document. A byte-order mark is a standard sequence of bytes identifying encoding type of the XML document. The XML parsers use this byte-order mark to determine the document encoding (such as Unicode). For example the byte-order mark, oxff, 0xfe identifies the document as Unicode. By default, the parser assumes the UTF-8 as the document encoding.

The byte-order mark is not returned when you request a XML fragment, because the byte-order mark belongs to the XML document header, which is missing in the XML fragment.

See Also

Retrieving XML Documents Using FOR XML

Using IIS Virtual Directory Management for SQL Server Utility

XML and Internet Support

Three-Tier System Architecture

The illustration shows the three-tier system architecture and describes the way HTTP requests from the client are handled.



The middle tier is the Microsoft® Internet Information Services (IIS) server on which you must first create a virtual root using the IIS Virtual Directory Management for SQL Server utility. The IIS server name specified in the URL identifies the IIS server. The IIS server examines the virtual root specified in the URL and determines whether an ISAPI DLL extension (Sqlisapi.dll) has been registered for the virtual root that is specified in the URL. The IIS server loads the DLL and passes on the URL request to the DLL. The Sqlisapi.dll extension communicates with the OLE DB Provider for SQL Server (SQLOLEDB) and establishes connection with the instance of Microsoft SQL Server[™] identified in the virtual root.

The entire XML functionality is implemented in Sqlxmlx.dll. When SQLOLEDB determines that the command is an XML command, the provider passes that command to Sqlxmlx.dll, which executes the command and returns the result to SQLOLEDB.

The template files, XML-Data Reduced (XDR) schema files, and Extensible Stylesheet Language (XSL) files reside on the IIS server. The XPath queries and the XDR schemas are handled on the IIS server. The XPath queries are translated into SQL commands by Sqlxmlx.dll.

The FOR XML clause and OPENXML are implemented on the server running SQL Server.

XML and Internet Support

Special Characters

Some characters have special meanings when they are used in a URL or in an XML document, and must be encoded properly for these meanings to take effect.

Special Characters in a URL

In queries executed at the URL, special characters are specified as *%xx*, where *xx* is the hexadecimal value of the character. The following table lists these special characters and describes their meanings. For more information, see the RFC1738 specification at <u>http://www.faqs.org/rfcs/rfc1738.html</u>.

Special character	Special meaning	Hexadecimal value
+	Indicates a space (spaces cannot be used in a URL).	%20
/	Separates directories and subdirectories.	%2F
?	Separates the actual URL and the parameters.	%3F
%	Specifies special characters.	%25
#	Indicates bookmarks.	%23
&	Separator between parameters specified in the URL.	%26

For example, consider this query:

SELECT * FROM Employees WHERE EmployeeID=?

Because the **?** character has a special meaning in the URL (separates the URL and the parameters being passed), it is encoded as **%3F** when this query is specified in the URL.

The following URL executes the query. In the URL, the parameter value is passed. For more information about executing SQL statement using HTTP, see

Executing SQL Statements Using HTTP.

http://IISServer/nwind?sql=SELECT * FROM Employees WHERE Er

Any special character (such as a + character) to the right of a **?** character is escaped by the browser (that is, a + character to the right of a **?** is converted to **%20**).

Special Characters in XML

Characters such as the > and < characters are XML markup characters and have special meaning in XML. When these characters are specified in SQL queries (or an XPath queries), they must be properly encoded (also referred to as entity encoding). The following table lists these special characters and describes their meanings. For more information, see the XML 1.0 specification at <u>XML 1.0</u> <u>Specifications</u>.

Special character	Special meaning	Entity encoding
>	Begins a tag.	>
<	Ends a tag.	<
"	Quotation mark.	"
1	Apostrophe.	'
&	Ampersand.	&

For example, consider this SQL query:

SELECT TOP 2 * FROM [Order Details] WHERE UnitPrice > 10 FOR XML AUTO

Because the > character has a special meaning in XML, it is encoded as **>** when this query is specified in a template (an XML document) This is the template with the query:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
```

```
<sql:query>
SELECT top 2 *
FROM [Order Details]
WHERE UnitPrice > 10
FOR XML AUTO
</sql:query>
</ROOT>
```

For more information about templates, see <u>Executing SQL Queries Using</u> <u>Templates</u> and <u>Executing XPath Queries Using Templates</u>.

Entity Encoding Within URL Encoding

At times you may have to specify both the URL encoding and entity encoding. For example, this template can be specified directly in the URL (instead of specifying the file name):

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:query>
SELECT top 2 *
FROM [Order Details]
WHERE UnitPrice > 10
FOR XML AUTO
</sql:query>
</ROOT>
```

In this case, the **&** character in the entity encoding **>** (specified for > markup character) has a special meaning in the URL and requires further encoding. The **&** character must be encoded as **%26**; otherwise it is treated as a parameter separator in the URL. The URL is then specified as:

http://IISServer/nwind?template=<ROOT xmlns:sql="urn:schemas-mi

See Also

Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML

Using IIS Virtual Directory Management for SQL Server Utility

XML and Internet Support

Executing SQL Statements Using HTTP

Microsoft® SQL Server[™] 2000 can be accessed directly by queries executed at the URL (if the **allow URL queries** option was selected when the virtual root was registered). Clients can make requests using HTTP methods GET and POST. For more information about the URL syntax that is support by the SQL ISAPI extension, see <u>URL Access</u>.

Examples

In the following examples, **nwind** is a virtual directory used to access the **Northwind** database. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In the example queries, if a query returns more than one element at the root of the document, the root element can be added by including either of these:

- A SELECT <ROOT> in the query.
- Passing a **root** keyword as a parameter to the query with a value **ROOT** (this value can be anything).

A. Specify a simple query

The following statement returns all of the customer data in the **Customers** table in the **Northwind** database. In this query, the XML mode is set to RAW.

http://IISServer/Nwind?sql=SELECT+top+2+CustomerID,+ContactNa

Here is the result set:

<ROOT>

```
<Customers CustomerID="ALFKI" ContactName="Maria Andears"
<Customers CustomerID="ANATR" ContactName="Ana Trujillo" />
</ROOT>
```

B. Specify a query on multiple tables

In this example, the SELECT statement returns information from the **Customers** and **Orders** tables in the **Northwind** database. The XML mode is set to AUTO.

http://IISServer/nwind?sql=SELECT+top+2+Customers.CustomerID,(

This is the partial result:

```
<ROOT>

<ROOT>

<Customers CustomerID="ALFKI">

<Orders OrderID="10643" OrderDate="1997-08-25T00:00:00" />

<Orders OrderID="10692" OrderDate="1997-10-03T00:00:00" />
</Customers>

<Customers CustomerID="ANATR">

<Orders OrderID="10308" OrderDate="1996-09-18T00:00:00" />
</customers>
```

C. Specify special characters in the query

The following query returns all distinct contact titles starting with **Sa** from the **Customers** table in the **Northwind** database. The example uses the LIKE clause and the special character % to search for the contact titles. In the LIKE clause, the special character % is specified as **%25**.

http://IISServer/nwind?sql=SELECT+DISTINCT+ContactTitle+FROM

Here is the result set:

```
<ROOT>
```

```
<Customers ContactTitle="Sales Agent" />
```

```
<Customers ContactTitle="Sales Associate" />
```

```
<Customers ContactTitle="Sales Manager" />
```

```
<Customers ContactTitle="Sales Representative" />
```

```
</ROOT>
```

In the following example, order and order detail information is retrieved from

the **Orders** and **Order Details** tables.

http://IISServer/nwind?sql=SELECT+'<ROOT>'+SELECT+Orders.Or

This is the partial result:

```
<ROOT>
<Orders OrderID="10248">
<Order_x0020_Details OrderID="10248" ProductID="11" UnitPrice=
<Order_x0020_Details OrderID="10248" ProductID="42" UnitPrice=
</Orders>
<Orders OrderID="10249">
<Orders OrderID="10249">
<Order_x0020_Details OrderID="10249" ProductID="14" UnitPrice=
</Orders>
</ROOT>
```

D. Specify a query without the FOR XML clause

You can specify SQL queries without the FOR XML clause. The result is returned as a stream. In the query, you can specify only one column because streaming is not supported over multiple column results. In this example, the query returns the first name of employees from the **Employees** table in the **Northwind** database. The result is returned as a concatenated string of first names.

```
http://IISServer/nwind?sql=SELECT+FirstName+FROM+Employees
```

E. Specify the contenttype keyword

The **contenttype** keyword specifies the content-type of the document returned. **text/XML** is the default content-type of the document except when **xsl** is specified in the URL. When **xsl** is specified in the URL and **contenttype** is not specified, then **contenttype** defaults to **text/html**.

In this example, the query returns a picture of an employee from the **Employees** table in the **Northwind** database. FOR XML mode is not specified because the returned data is compatible with the receiving application (that is, the browser can handle the returned data).

http://IISServer/nwind?sql=SELECT+Photo+FROM+Employees+WH

In retrieving images, **contenttype** is generally specified. If **contenttype** is specified, the ISAPI extension does not search for and remove any Access header information. Therefore, to retrieve any images that have the Access header information, **contenttype** should not be specified as shown in the previous example. In all other cases, **contenttype** should be specified as shown in this example:

```
http://iisserver/virtualroot?sql=SELECT+Picture+FROM+TableName-
```

The images can also be brought into an HTML document. In the following example, an .htm file (File1.htm) is created with these contents:

<img src="http://IISServer/nwind?sql=select photo from Employees w

When this file is opened in the browser, an employee photo is displayed.

F. Specify the xsl keyword

In this example, the query returns the first and last name of all employees in the **Employee** table in the **Northwind** database. **employee.xsl** processes the result set.

When **xsl** is specified in the URL but **contenttype** is not specified in the URL and there is no content-type defined in the XSL style sheet, **contenttype** defaults to **text/html**. Therefore, the result is displayed in the form of a table with two columns (**firstname**, **lastname**).

http://IISServer/nwind?sql=SELECT+FirstName,LastName+FROM+H

The .xsl file is provided here. This file must exist in the virtual root directory or one of its subdirectories (in which case the file path specified is relative to the virtual root directory). In this example, the .xsl file is stored in the virtual root directory.

```
<?xml version='1.0' encoding='UTF-8'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
<xsl:template match = '*'>
```

```
<xsl:apply-templates />
  </xsl:template>
  <xsl:template match = 'Employees'>
    < TR >
     <TD><xsl:value-of select = '@FirstName' /></TD>
     <TD><B><xsl:value-of select = '@LastName' /></B></TD>
    </TR>
  </xsl:template>
  <xsl:template match = '/'>
   <HTML>
    <HEAD>
      <STYLE>th { background-color: #CCCCCC }</STYLE>
    \langle HEAD \rangle
    <BODY>
     <TABLE border='1' style='width:300;'>
      <TR><TH colspan='2'>Employees</TH></TR>
      <TR><TH >First name</TH><TH>Last name</TH></TR>
      <xsl:apply-templates select = 'root' />
     </TABLE>
    \langle BODY \rangle
   </HTML>
  </xsl:template>
</xsl:stylesheet>
```

Instead of specifying the **contenttype** in the URL, **contenttype** can also be specified as the value of the **media-type** attribute of the <xsl:output> element. For example, <**xsl:output media-type="text/html"** /> can be added after the namespace declaration in the preceding XSL file.

G. Pass parameters to SQL statements

Parameters can be passed to SQL queries. In this example, employee information for a given employee ID is returned from the **Employees** table in the **Northwind** database. The value of **EmployeeID** is provided as input to the query. Note that

the ? character, used for a parameter marker in the URL, is a special character and is encoded as %3F. For more information about special characters, see <u>Special Characters</u>.

http://IISServer/nwind?sql=SELECT+FirstName,LastName+FROM+H

Here is the result set:

```
<ROOT>
<Employees FirstName="Nancy" LastName="Davolio" />
</ROOT>
```

In this query, two parameter values are passed to the query:

```
http://IISServer/nwind?sql=SELECT+'<ROOT>';SELECT+Employee
```

Here is the result set:

<ROOT>

```
<Employees EmployeeID="1" Title="Sales Representative" /> </ROOT>
```

See Also

Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML

Using IIS Virtual Directory Management for SQL Server Utility

XML and Internet Support

Executing Stored Procedures Using HTTP

Stored procedures can be executed at the URL, using either the Transact-SQL execute syntax: EXECUTE MySP; or the ODBC call syntax: {call+MySP}.

Parameters can be passed to stored procedures. Parameters are selected by taking any unused *name=value* pairs and supplying them as parameters to the query in the order supplied. For more information about the URL syntax that is support by the SQL ISAPI extension, see <u>URL Access</u>.

Examples

In the following examples, **nwind** is a virtual directory used to access the **Northwind** database. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

A. Execute a simple stored procedure

This example creates and executes a stored procedure that returns a category name from the **Categories** table in the **Northwind** database. The stored procedure takes no parameters.

```
IF EXISTS (SELECT name FROM sysobjects
WHERE name = 'CategoryInfo' AND type = 'P')
DROP PROCEDURE CategoryInfo
GO
CREATE PROCEDURE CategoryInfo
AS
SELECT CategoryName
FROM Categories
FOR XML AUTO
GO
```

This stored procedure can be executed using a URL:

http://IISServer/nwind?sql=EXECUTE+CategoryInfo&root=ROOT

B. Execute a stored procedure with a parameter

In this example, a stored procedure with parameters is executed using a URL. The stored procedure retrieves, for a given category ID, the category name from the **Category** table in the **Northwind** database.

```
IF EXISTS (SELECT name FROM sysobjects
WHERE name = 'CategoryInfoWithInputParam' AND type = 'P')
DROP PROCEDURE CategoryInfoWithInputParam
GO
```

```
CREATE PROCEDURE CategoryInfoWithInputParam
@CategoryID int
```

AS

```
SELECT '<ROOT>'
SELECT CategoryName
FROM Categories
WHERE Categories.CategoryID = @CategoryID
FOR XML AUTO
SELECT '</ROOT>'
GO
```

This stored procedure can be executed using a URL:

```
http://IISServer/nwind?sql=execute+CategoryInfoWithInputParam+1
```

Or

```
http://IISServer/nwind?sql=execute+CategoryInfoWithInputParam+@
```

The first example specifies the parameter value (1) by position (that is, without a parameter name). The second example specifies the parameter name with the value.

C. Execute a stored procedure using the Transact-SQL EXECUTE and ODBC Call syntax

This stored procedure returns employee information for a given employee from the **Employees** table in the **Northwind** database. The stored procedure takes the employee first name and last name as input and returns the employee ID, employee title, and the birth date.

```
IF EXISTS (SELECT name FROM sysobjects
WHERE name = 'FindEmp' AND type = 'P')
DROP PROCEDURE FindEmp
GO
```

```
CREATE PROCEDURE FindEmp @FName varchar(20), @LName va
SELECT EmployeeID, Title, BirthDate
FROM Employees
WHERE FirstName = @FName
AND LastName = @LName
FOR XML AUTO
GO
```

The Transact-SQL EXECUTE statement can be specified to execute the stored procedure:

http://IISServer/nwind?sql=SELECT+'<ROOT>';EXECUTE+FindEm

Or

```
http://IISServer/nwind?sql=SELECT+'<ROOT>';EXECUTE+FindEm
```

ODBC call syntax can also be specified to execute the stored procedure:

```
http://IISServer/nwind?sql=SELECT+'<ROOT>';{CALL+FindEmp}+
```

Or

```
http://IISServer/nwind?sql=SELECT+'<ROOT>';{CALL+FindEmp}+
```

See Also

Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML

Using IIS Virtual Directory Management for SQL Server Utility

XML and Internet Support

Executing Template Files Using HTTP

Writing long SQL queries at the URL can get cumbersome. Instead, templates can be used to specify queries (SQL or XPath). The template file name is specified in the URL. A template is a well-formed XML document containing one or more SQL statements and XPath queries.

Using a template you can:

- Specify SQL queries or XPath queries. When XPath queries are specified in the template, the mapping XML-Data Reduced (XDR) schema file against which the query is to be executed is also identified in the template.
- Specify a top-level element for the XML fragment that is returned by executing SQL or XPath queries; thereby, making the result of executing the template in the URL a valid XML document.
- Define parameters that can be passed to SQL statements or XPath queries.
- Declare namespaces.
- Specify an Extensible Stylesheet Language (XSL) style sheet to apply to the resulting document.

Template files also enhance security. Because the URLs (and thus the queries in the URL) can be edited, by having the queries stored in a file (template file), you can prevent users from modifying the queries and obtaining information you do not want them to see.

The security is enforced by removing the URL query-processing service on the virtual root and leaving only the Microsoft® SQL Server[™] XML ISAPI to process the files and return the result set. The virtual root is registered using IIS Virtual Directory Management for SQL Server utility.

Before templates can be specified in the URL, the virtual name of **template** type must be created using the IIS Virtual Directory Management for SQL Server utility. For more information, see <u>Using IIS Virtual Directory Management for SQL Server Utility</u>.

See Also

Using IIS Virtual Directory Management for SQL Server Utility Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML

Using XPath Queries

XML and Internet Support

Using XML Templates

This general form for a template shows the way SQL queries and XPath queries are specified:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql"
sql:xsl='XSL FileName' >
<sql:header>
<sql:param>..</sql:param>
...n
</sql:header>
<sql:query>
sql statement(s)
</sql:query>
<sql:xpath-query mapping-schema="SchemaFileName.xml">
XPath query
</sql:xpath-query>
```

Everything in a template is optional. The elements <header>, <param>, <query>, <XPath-query>, and the attribute mapping-schema are defined in the **sql** namespace. Therefore, the namespace declaration **xmlns:sql="urn:schemas-microsoft-com:xml-sql"** is required. The namespace can be named anything; **sql** is just an alias.

<ROOT>

This tag is specified to provide a single top-level element (also referred as root tag) for the resulting XML document. The <ROOT> tag can have any name.

```
<sql:header>
```

This tag is used to hold any header values. In the current implementation, only the <sql:param> element can be specified in this tag. The <sql:header> tag acts as containing tag, allowing you to define multiple parameters. With

all the parameter definitions in one place, processing the parameter definitions is more efficient.

<sql:param>

This element is used to define a parameter that is passed to the query inside the template. Each <param> element defines one parameter. Multiple <param> elements can be specified in the <sql:header> tag.

<sql:query>

This element is used to specify SQL queries. You can specify multiple <sql:query> elements in a template.

<sql:xpath-query>

This element is used to specify an XPath query. Because the XPath query is executed against the annotated XML-Data Reduced (XDR) schema, the schema file name must be specified using the **mapping-schema** attribute.

sql:xsl

This attribute is used to specify an Extensible Stylesheet Language (XSL) style sheet that will be applied to the resulting XML document. In specifying the XSL file, a relative or an absolute path can be specified. This relative path specified is relative to the directory associated with the virtual name of **template** type. For example, if the directory associated with the virtual name of **template** type is C:\Template, then the relative path, Xyz/MyXSL.xml specified for **sql:xsl** maps to C:\Template\Xyz\MyXSL.xml.

mapping-schema

This attribute is used to identify the annotated XDR schema. This attribute is specified only if you are executing an XPath query in the template. The XPath query is executed against the annotated XDR schema. In specifying the mapping schema file, a relative or an absolute path can be specified. This relative path specified is relative to the directory associated with the virtual name of **template** type. For example, if the directory associated with the virtual name of **template** type is C:\Template, then the relative path, Schema/MSchema.xml specified for **mapping-schema** maps to C:\Template\Schema\MSchema.xml.

Note Each <sql:query> or <sql:XPath-query> represents a separate transaction. Therefore, if you have multiple <sql:query> or <sql:XPath-query> tags in the template, and if one fails, the others will proceed.

If **contenttype** is set, Sqlisapi.dll returns that header information to the browser. If the **contenttype** is not set, the first character in the template file is used by the urlmon to determine the content-type. If the first character in the template is the < character or a Unicode byte order mark (0xFFFE), **text/xml** is returned to the browser as the content-type, and the browser displays the result. Otherwise, Sqlisapi.dll does not send the content-type header information that instructs the browser on how to display the result; therefore, you do not see the result in the browser.

Before templates can be specified in the URL, the virtual name of **template** type must be created using the IIS Virtual Directory Management for SQL Server utility. For more information, see <u>Using IIS Virtual Directory Management for SQL Server Utility</u>.

Storing the Templates

The template is stored in the directory associated with the virtual name of **template** type or one its subdirectories:

- If the template is stored in the directory associated with virtual name of **template** type, the URL query looks like: http://IISServer/nwind/TemplateVirtualName/TemplateFile.xm
- If the template is stored in the subdirectory associated with virtual name of **template** type (**xyz**), the URL query looks like: http://IISServer/nwind/TemplateVirtualName/xyz/TemplateFile

See Also

Using IIS Virtual Directory Management for SQL Server Utility

Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML

Using XPath Queries

XML and Internet Support

Executing SQL Queries Using Templates

In the following examples, **nwind** is a virtual directory created using the IIS Virtual Directory Management for SQL Server utility, and **template** is the virtual name of **template** type defined when the virtual directory is created (any name can be given to a virtual name when it is created). For more information, see <u>Using IIS Virtual Directory Management for SQL Server Utility</u>.

The <sql:query> tag is used to specify SQL statements.

Examples

A. Create a template file with a simple SELECT statement

This template specifies a simple SELECT statement.

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:query>
SELECT top 2 CustomerID, CompanyName
FROM Customers
FOR XML AUTO
</sql:query>
</ROOT>
```

This template is stored in a file (File1.xml) and executed using a URL:

```
http://IISServer/nwind/template/File1.xml
```

The query specified in the template is replaced by its result. Therefore, the XML document returned has the same structure as the template itself, including the <ROOT> tag that is added.

This is the result:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futter}
<Customers CustomerID="ANATR" CompanyName="Ana Trujillo F
```

</ROOT>

B. Execute a stored procedure in a template file

A stored procedure can also be executed in a template. The stored procedure is also specified in the <sql:query> tag.

Consider this stored procedure:

```
IF EXISTS (SELECT name FROM sysobjects
WHERE name = 'CategoryInfo' AND type = 'P')
DROP PROCEDURE CategoryInfo
GO
CREATE PROCEDURE CategoryInfo
AS
SELECT CategoryName
FROM Categories
WHERE Categories.CategoryID = 2
FOR XML AUTO
```

The stored procedure can be executed in a template:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:query>
exec CategoryInfo
</sql:query>
</ROOT>
```

This template is stored in a file (File1.xml) and executed using a URL:

http://IISServer/nwind/template/File1.xml

This is the result:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Categories CategoryName="Condiments" />
</ROOT>
```

C. Use entity references in a template

Because a template is an XML document, entity references must be used for special characters. This example uses the entity reference (**>**) for the special markup character (>).

Consider this template:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:query>
SELECT top 2 *
FROM [Order Details]
WHERE UnitPrice > 10
FOR XML AUTO
</sql:query>
</ROOT>
```

This template is stored in a file (File1.xml) and executed using a URL:

```
http://IISServer/nwind/template/File1.xml
```

This is the result:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Order_x0020_Details OrderID="10248" ProductID="11" UnitPrice=
Quantity="12" Discount="0" />
<Order_x0020_Details OrderID="10248" ProductID="72" UnitPrice=
Quantity="5" Discount="0" />
</ROOT>
```

If this template is specified directly in the URL, additional encoding is needed. Because the > character is a special character in XML and because it is specified.

Because the > character is a special character in XML and because it is specified in a template (an XML document), it is encoded as **>**. And because the **&** character is a special character in this URL, **&** must be encoded as **%26** when this template is specified in the URL.

The template is then specified in the URL as:

http://IISServer/nwind?template=<ROOT%20xmlns:sql="urn:schemas

D. Specify templates directly in the URL

Templates can be specified directly in the URL. In this example, a template containing a simple SELECT statement is specified in the URL:

http://IISServer/nwind?template=<ROOT+xmlns:sql="urn:schemas-m

CAUTION Specifying templates directly in the URL is not recommended for security reasons.

See Also

Using IIS Virtual Directory Management for SQL Server Utility

Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML
Passing Parameters to Templates

SQL queries requiring parameter values can be specified in templates. The <sql:header> tag is specified to define parameters. The parameters can be assigned default values. The default parameter values are used when a template is executed without specifying parameter values.

Templates can also be used to specify XPath queries against annotated XDR (XML-Data Reduced) schemas. The <sql:xpath-query> tag is used to specify the XPath query. For more information and an example, see <u>Using XPath Queries</u>.

Examples

In the following examples, **nwind** is a virtual directory created using the IIS Virtual Directory Management for SQL Server utility, and **template** is the virtual name of **template** type defined when the virtual directory is created (any name can be given to a virtual name when it is created). For more information, see <u>Using IIS Virtual Directory Management for SQL Server Utility</u>.

A. Specify default parameters in a template

Parameter values can be assigned default values in a template. The stored procedure in this example requires one input parameter (**@CategoryName**).

IF EXISTS (SELECT name FROM sysobjects

```
WHERE name = 'CategoryInfoWithInputParam' AND type = 'P')
DROP PROCEDURE CategoryInfoWithInputParam
```

```
GO
```

CREATE PROCEDURE CategoryInfoWithInputParam @CategoryName varchar(35)

AS

SELECT CategoryName, Description

FROM Categories

WHERE Categories.CategoryName = @CategoryName for xml a

A template with a call to execute the stored procedure can be created as shown in

the following example. The template specifies a default value (**Condiments**) for the parameter **@CategoryName**.

```
<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql'>
<sql:header>
<sql:param name='CategoryName'>Condiments</sql:param>
</sql:header>
<sql:query >
exec CategoryInfoWithInputParam @CategoryName
</sql:query>
</ROOT>
```

This template is saved in a file (File1.xml) and executed using a URL:

```
http://IISServer/nwind/template/File1.xml
```

Because no parameters are passed to the file, the default value (**Condiments**) is used.

This is the result:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Categories CategoryName="Condiments" Description="Sweet and sauces, relishes, spreads, and seasonings" />
</ROOT>
```

The template can be also be executed directly at the URL:

```
http://IISServer/nwind?template=<ROOT xmlns:sql='urn:schemas-mic
```

CAUTION Specifying queries or templates directly in the URL is not recommended for security reasons.

B. Pass a parameter value to a template

Parameters can be passed to template files. In this example, the stored procedure requires one input parameter (**@CategoryName**).

```
IF EXISTS (SELECT name FROM sysobjects
WHERE name = 'CategoryInfoWithInputParam' AND type = 'P')
DROP PROCEDURE CategoryInfoWithInputParam
GO
CREATE PROCEDURE CategoryInfoWithInputParam
@CategoryName varchar(35)
AS
SELECT CategoryName, Description
FROM Categories
```

WHERE Categories.CategoryName = @CategoryName for xml at

The stored procedure is called in the template as shown in the example that follows. The template specifies a default value for the parameter **@CategoryName**.

```
<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql'>
<sql:header>
<sql:param name='CategoryName'>Condiments</sql:param>
</sql:header>
<sql:query >
exec CategoryInfoWithInputParam @CategoryName
</sql:query>
</ROOT>
```

This template is stored in a file (File1.xml) and executed using a URL:

```
http://IISServer/nwind/template/File1.xml?CategoryName=Beverages
```

If a parameter value is passed to the file at run time, the specified value is used instead of the default value. In the following call to execute a template file, the value **Beverages** is passed to the file.

This is the result:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Categories CategoryName="Beverages" Description="Soft drinks,
```

```
coffees, teas, beers, and ales" /> </ROOT>
```

The template can be executed directly using a URL:

```
http://IISServer/nwind?template=<ROOT xmlns:sql='urn:schemas-mic
```

CAUTION Specifying queries or templates directly in the URL is not recommended for security reasons.

C. Pass multiple parameters to a template

Multiple parameters can be passed to a template. In this example, two parameters with default values are specified in <sql:header>. The template also specifies two queries that require parameter values.

This template consists of two SQL queries, each of which takes one parameter:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
 <sql:header>
   <sql:param name='CustomerID'>ALFKI</sql:param>
  <sql:param name='EmployeeID'>1</sql:param>
 </sql:header>
 <sql:query>
  SELECT CustomerID, CompanyName
  FROM Customers
  WHERE CustomerID=@CustomerID
  FOR XML AUTO
 </sql:query>
 <sql:query>
  SELECT EmployeeID,LastName,FirstName
  FROM Employees
  WHERE EmployeeID=@EmployeeID
  FOR XML AUTO
 </sql:query>
</ROOT>
```

This template is stored in a file (File1.xml) and executed using a URL: http://IISServer/nwind/template/template5.xml

This is the result:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterk
<Employees EmployeeID="1" LastName="Davolio" FirstName="Na
</ROOT>
```

In the following example, only the *CustomerID* parameter value is provided. Therefore, the default customer ID value **ALFKI** is ignored. Because no value is provided for *EmployeeID* parameter, the default value is used.

http://IISServer/nwind/template/template5.xml?CustomerID=BERGS

This is the result:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Customers CustomerID="BERGS" CompanyName="Berglunds sna
<Employees EmployeeID="1" LastName="Davolio" FirstName="Na
</ROOT>
```

The template is executed by passing both parameter values (default values are ignored).

http://IISServer/nwind/template/template5.xml?CustomerID=BERGS8

See Also

Using IIS Virtual Directory Management for SQL Server Utility

Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML

Specifying an XSL Style Sheet in a Template

An Extensible Stylesheet Language (XSL) style sheet can be applied to the query results. When you execute a template using HTTP, you can specify an XSL file in these ways:

- Use the **sql:xsl** attribute in the template.
- Use the **xsl** keyword as part of the URL to specify the XSL file that will be used to process the resulting XML data.

If the XSL file is specified both in the template using **sql:xsl** and in the URL using the keyword **xsl**, the XSL style sheet specified in the template is applied to the results first, and then the XSL file specified in the URL is applied.

Examples

In the following example, **nwind** is a virtual directory created using the IIS Virtual Directory Management for SQL Server utility, and **template** is the virtual name of **template** type defined when the virtual directory is created (any name can be given to a virtual name when it is created). For more information, see <u>Using IIS Virtual Directory Management for SQL Server Utility</u>.

A. Specify sql:xsl in a template to process the result

In this example, a template includes a simple SELECT statement. The query result is processed according to the instructions in the XSL file specified using **sql:xsl**.

```
<?xml version ='1.0' encoding='UTF-8'?>
<root xmlns:sql='urn:schemas-microsoft-com:xml-sql'
sql:xsl='MyXSL.xsl'>
<sql:query>
SELECT FirstName, LastName FROM Employees FOR XML AU
</sql:query>
</root>
```

For illustration purposes, the template (TemplateWithXSL.xml) is stored in the directory associated with the virtual name (**template**), of **template** type. The XSL file (MyXSL.xsl) is also stored in the same directory.

This is the XSL file:

```
<?xml version='1.0' encoding='UTF-8'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
<xsl:template match = '*'>
    <xsl:apply-templates />
  </xsl:template>
  <xsl:template match = 'Employees'>
    \langle TR \rangle
     <TD><xsl:value-of select = '@FirstName' /></TD>
     <TD><B><xsl:value-of select = '@LastName' /></B></TD>
    </TR>
  </xsl:template>
  <xsl:template match = '/'>
   <HTML>
    <HEAD>
      <STYLE>th { background-color: #CCCCCC }</STYLE>
    \langle HEAD \rangle
     \langle BODY \rangle
     <TABLE border='1' style='width:300;'>
      <TR><TH colspan='2'>Employees</TH></TR>
      <TR><TH >First name</TH><TH>Last name</TH></TR>
      <xsl:apply-templates select = 'root' />
     </TABLE>
     </BODY>
   </HTML>
  </xsl:template>
</xsl:stylesheet>
```

This URL executes the template:

http://IISServer/nwind/template/TemplateWithXSL.xml&contenttype=

Because the XSL file is applied to the result, the **contenttype** is set to **text/html**. Therefore, specifying the **contenttype** parameter in the URL is optional.

The result is displayed in a two-column table format (**FirstName** and **LastName**).

You can also specify the XSL file in the URL instead of in a template (using **sql:xsl**), . In this case, the XSL file must be stored in the directory associated with the virtual root (**nwind**) or one of its subdirectories, in which case the relative path must be specified in the URL. Assuming the XSL file is stored in the directory associated with the **nwind** virtual directory, this URL executes the template:

http://IISServer/nwind/template/templateFile.xml?xsl=MyXSL.xsl

If the XSL file is stored in a subdirectory (x) of the virtual root directory, the URL with a relative path is specified as:

http://IISServer/nwind/template/templateFile.xml?xsl=/x/MyXSL.xsl

If the XSL file is specified in the template using **sql:xsl** and in the URL using the keyword **xsl**, the XSL style sheet specified in the template is applied to the results first, and then the XSL file specified in the URL is applied.

Executing XPath Queries Using Templates

The <sql:xpath-query> tag is used to specify the XPath query in the template. The XPath query is executed against the annotated XML-Data Reduced (XDR) mapping schema specified using he **mapping-schema** attribute of the <xpathquery> element. For more information about XDR schemas, see <u>Creating XML</u> <u>Views Using Annotated XDR Schemas</u>. For more information about XPath queries, see <u>Using XPath Queries</u>.

The mapping XDR schema specified in the template must be stored in the directory associated with the **template** virtual name (created using IIS Virtual Directory Management for SQL Server utility) or one of its subdirectories, in which case you must specify the relative path in the **mapping-schema** attribute.

See Also

Using IIS Virtual Directory Management for SQL Server Utility Accessing SQL Server Using HTTP Creating XML Views Using Annotated XDR Schemas

Executing XPath Queries Using HTTP

The XPath queries against annotated XML-Data Reduced (XDR) schema can be specified directly in the URL. For more information about the URL syntax that is supported by the SQL ISAPI extension, see <u>URL Access</u>.

The annotated XDR schemas provide an XML view of the relational data. To execute an XPath query against an annotated XDR schema, the schema file is specified as part of the URL.

To specify an XPath query against an annotated XDR schema, you must create a virtual name of **schema** type using the IIS Virtual Directory Management for SQL Server utility. The XDR schema specified in the URL must be stored in the directory associated with virtual name of **schema** type or one of its subdirectories:

- If the annotated XDR schema is stored in the directory associated with the virtual name of **schema** type, the URL query looks like: http://IISServer/nwind/SchemaVirtualName/XDRSchema.xml/
- If the annotated XDR schema is stored in a subdirectory (**xyz**) associated with virtual name of **schema** type, the path relative to the directory associated with virtual name of **schema** type is included in the URL. In this case, the URL query looks like: http://IISServer/nwind/SchemaVirtualName/xyz/XDRSchema.:

Examples

In this example, **nwind** is a virtual directory created using the IIS Virtual Directory Management for SQL Server utility, and **schema** is the virtual name of **schema** type defined when the virtual directory is created (any name can be given to a virtual name when it is created).

A. Specify an XPath query in the URL

For example, consider this annotated XDR schema:

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

```
<ElementType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
<AttributeType name="ContactName" />
<AttributeType name="Phone" />
```

```
<attribute type="CustomerID" />
<attribute type="ContactName" />
<attribute type="Phone" />
</ElementType>
</Schema>
```

For illustration purposes, this XDR schema is stored as MySchema.xml in the directory associated with the virtual name of **schema** type.

This URL executes an XPath query against the XDR schema (MySchema.xml) specified in the URL. The XPath query requests all the customers with CustomerID of **ALFKI**.

http://IISServer/nwind/schema/Schema2.xml/Customer[@CustomerID

This is the result:

```
<Customer CustomerID="ALFKI" ContactName="Maria Anders" Phc
```

If the query returns more than one customer, you must specify the **root** keyword to return a well-formed XML document. The following XPath query returns all the customers. In the URL, the **root** keyword is specified:

```
http://IISServer/nwind/schema/Schema2.xml/Customer?root=root
```

This is the partial result:

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
<Customer CustomerID="ALFKI" ContactName="Maria Anders"
Phone="030-0074321" />
<Customer CustomerID="ANATR" ContactName="Ana Trujillo"
Phone="(5) 555-4729" />
...
</root>
```

See Also

Using Annotated XDR Schemas in Queries

Using XPath Queries

Creating XML Views Using Annotated XDR Schemas

Accessing Database Objects Using HTTP

The database objects, such as tables and views, can be accessed directly using a URL. In this case, the XPath query is specified directly against the database object to obtain the result (one row/one column value). For more information about the URL syntax that is supported by the SQL ISAPI extension, see <u>URL Access</u>.

In the URL, the virtual name of **dbobject** type is specified when accessing database objects directly.

The FOR XML queries can return references to binary data. You can retrieve the binary data associated with the reference by sending another URL request with the **dbobject** reference in it. This is the primary purpose for the **dbobject** virtual name type. For more information about queries that use FOR XML, see <u>Retrieving XML Documents Using FOR XML</u>.

Examples

In the following examples, **nwind** is a virtual directory created using the IIS Virtual Directory Management for SQL Server utility. The **dbobject** is the virtual name of **dbobject** type and **template** is a virtual name of **template** type defined when the virtual directory is created (any name can be given to a virtual name when it is created). For more information, see <u>Using IIS Virtual Directory</u> <u>Management for SQL Server Utility</u>.

A. Retrieve an employee's photo using the virtual name of dbobject type in the URL

The XPath query specified retrieves **Photo** column from **Employees** table. In the URL, **Employees[@EmployeeID='1']/@Photo** is the XPath query. In the query, **Employees** is the table name and **@EmployeeID='1'** is the predicate that finds an employee with an ID value of 1. **@Photo** is the column from which to retrieve the value.

http://IISServer/nwind/dbobject/Employees[@EmployeeID='1']/@Pho

The query is translated into the following SELECT statement:

SELECT	Photo
FROM	Employees
WHERE	EmployeeID='1'

Note The XPath query must identify a single row and a single column.

B. Execute a query to obtain references to image data and to apply an XSL style sheet to process the references

In this example, a SELECT statement is specified to retrieve the employee ID and photo. The query returns references to the image data. These references are used in the Extensible Stylesheet Language (XSL) file to retrieve the employee photos and to display them in the browser.

The query is specified in a template. For illustration purposes, the template file is saved as TemplateWithAnXSL.xml file in the **template** subdirectory of the virtual root (assuming this is the directory specified when the virtual name of **template** type is created).

```
<?xml version ='1.0' encoding='UTF-8'?>
<root xmlns:sql='urn:schemas-microsoft-com:xml-sql' sql:xsl='photo.x
<sql:query >
SELECT employeeID, photo FROM employees FOR XML AUTO
</sql:query>
```

</root>

This is the XSL file (Photo.xsl) to process the result set. For illustration purposes, this file is stored in the virtual root directory.

```
<?xml version='1.0' encoding='UTF-8'?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/TR/WD-xsl' >
<xsl:template match = '*'>
<xsl:apply-templates />
</xsl:template>
<xsl:template match = 'employees'>
```

```
\langle TR \rangle
     <TD><xsl:value-of select = '@employeeID' /></TD>
     <TD><B> <IMG><xsl:attribute name='src'>
            <xsl:value-of select = '@photo'/>
            </xsl:attribute>
          </IMG>
     </B></TD>
    </TR>
  </xsl:template>
  <xsl:template match = '/'>
   <HTML>
    <HEAD>
      <STYLE>th { background-color: #CCCCCC }</STYLE>
    <!-- <BASE href='http://IISServer/nwind/'></BASE> -->
    \langle HEAD \rangle
    \langle BODY \rangle
     <TABLE border='1' style='width:300;'>
      <TR><TH colspan='2'>Employees</TH></TR>
      <TR><TH >EmployeeID</TH><TH>Photo</TH></TR>
      <xsl:apply-templates select = 'root' />
     </TABLE>
    \langle BODY \rangle
   </HTML>
  </xsl:template>
</xsl:stylesheet>
```

This URL executes the template:

http://IISServer/nwind/template/TemplatewithAnXSL.xml?contenttype

After applying the XSL file, the query result is displayed as a two-column table (**EmployeeID** and **Photo**).

C. Specify special characters in the query

In a URL, the question mark (?) separates the URL and the parameters being passed to the URL. Any special characters, such as the plus sign (+) to the right of the ?, are escaped by the browser (that is, a + to the right of a ? is converted to **%20**).

The URL in this example produces an error because the + in the predicate expression is interpreted as an addition operator (because there is no ? in the URL). You must specify **%20** in place of + in the URL.

http://IISServer/nwind/dbobject/Orders[@OrderID=10248%20and%20

See Also

Using IIS Virtual Directory Management for SQL Server Utility

Using XPath Queries

Sample Applications to Post Templates

The topics in this section present simple applications that show how to post templates.

Using HTML Forms to Post Templates

HTML forms can be used to post templates. The input mechanism of HTML forms can be used to obtain user input for the values of the parameters that can be passed to an SQL statement. In the **TEXTAREA** element of the HTML form, *template* is used as the variable name for the **NAME** attribute. The body of the **TEXTAREA** is then sent as a value for *template*.

Examples

In the following examples, **nwind** is a virtual directory created using the IIS Virtual Directory Management for SQL Server utility (any name can be given to a virtual name when it is created). For more information, see <u>Using IIS Virtual</u> <u>Directory Management for SQL Server Utility</u>.

A. Post a simple template in a form

The HTML form in this example prompts the user to enter an employee ID. The ID value is used as an input parameter to the SELECT statement in the template. The query returns the first and last name of employees from the **Employees** table in the **Northwind** database. This form can be saved in an .htm file and opened in the browser.

```
<head>
<TITLE>Sample Form </TITLE>
</head>
<body>
For a given employee ID, employee first and last name is retrieved.
<form action="http://IISServer/nwind" method="POST">
<B>Employee ID Number</B>
<input type=text name=EmployeeID value='1'>
<input type=hidden name=contenttype value=text/xml>
<input type=hidden name=template value='
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql" >
<sql:header>
```

```
<sql:param name="EmployeeID">1</sql:param>
</sql:header>
<sql:query>
SELECT FirstName, LastName
FROM Employees
WHERE EmployeeID=@EmployeeID
FOR XML AUTO
</sql:query>
</ROOT>
'>
<input type="submit">
</form>
</body>
```

B. Post a template in an HTML form and provide an XSL file to process the output

In this example, a simple HTML form is used to post a template. The template contains a SELECT statement that returns first and last names from the **Employees** table in the **Northwind** database.

When this HTML document is opened in the browser, the user can specify the content-type and the Extensible Stylesheet Language (XSL) file to process the result set. If the content-type is specified as **text/html**, the XSL file processes the result set and produces a two-column table as output.

If the content-type is specified as **text/xml**, the result is displayed in form of an XML document.

Note The XSL file must reside in the physical directory (or one of its subdirectories) associated with the virtual directory. If the file is stored in the physical directory, only the file name has to be specified. If the file is stored in one of the subdirectories of the physical directory, the directory path relative to the physical directory is specified.

<body>

Hi there

```
<form action="http://IISServer/nwind" method="POST">
```

contenttype

```
<input name=contenttype value="text/html">
```


xsl

```
<input name=xsl value="emp.xsl"><br>
```

```
<input type=hidden name=template value='
```

<ROOT>

```
<sql:query xmlns:sql="urn:schemas-microsoft-com:xml-sql">
```

```
Select FirstName, LastName from Employees for xml auto</sql:query>
```

</ROOT>

'>

```
<input type="submit">
```

</form>

</body>

The XSL file is given below. The XSL transformation is applied to the result set.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match = "*">
<xsl:template match = "*">
</xsl:templates />
</xsl:template>
```

```
\langle TR \rangle
```

```
<TD><xsl:value-of select = "@FirstName" /></TD>
 <TD><B><xsl:value-of select = "@LastName" /></B></TD>
 </TR>
</xsl:template>
<xsl:template match = "/">
  <HTML>
  <HEAD>
  <STYLE>th { background-color: #CCCCCC }</STYLE>
  </\text{HEAD}>
  <BODY>
 <TABLE border="1" style="width:300;">
 <TR><TH colspan="2">Employees</TH></TR>
 <TR><TH >FirstName</TH><TH>LastName</TH>
 <xsl:apply-templates select = "ROOT" />
 </TABLE>
  </BODY>
  </HTML>
</xsl:template>
</xsl:stylesheet>
```

See Also

Using IIS Virtual Directory Management for SQL Server Utility

Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML

Using XPath Queries

Posting Templates Directly to the Virtual Directory

This Microsoft® Visual Basic® example shows how templates can be sent directly to the virtual directory without using a Web browser. In this example, a template consisting of a SELECT statement is sent directly to the **nwind** virtual directory. For information about how to create the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

Dim xmlHttp As New MSXML2.xmlHttp Dim doc As New MSXML2.DOMDocument Dim strQuery As String Dim strURL As String Dim strPostBody As String

```
'Set the post body - this is the query/request.
strPostBody = "<?xml version='1.0' encoding='UTF-16'?>
    "<root>    "&_
    "<sql:query xmlns:sql='urn:schemas-microsoft-com:xml-sql'
    "select OrderID, shipName from Orders for xml raw
    "</sql:query>    "&_
    "</root>    "
```

'Validate the document using the MSXML parser. doc.loadXML strPostBody

If doc.parseError.errorCode Then ' Do something with the error. End If

'Post the template. xmlHttp.Open "POST", "http://localhost/nwind", False xmlHttp.setRequestHeader "Content-type", "application/xml" xmlHttp.send doc ' Retrieve the results. Debug.Print xmlHttp.responseText

See Also

Using IIS Virtual Directory Management for SQL Server Utility

Accessing SQL Server Using HTTP

Retrieving XML Documents Using FOR XML

Using XPath Queries

Creating XML Views Using Annotated XDR Schemas

You can create XML views of relational data using XDR (XML-Data Reduced) schemas. These views can then be queried using XPath queries. This is similar to creating views using CREATE VIEW statements and specifying SQL queries against the view.

An XML schema describes the structure of an XML document and also various constraints on the data in the document. When you specify XPath queries against the schema, the structure of the XML document returned is determined by the schema against which the XPath query is executed.

In Microsoft® SQL Server[™] 2000, the XML-Data Reduced (XDR) language is used to create the schemas. The XDR is flexible and overcomes some of the limitations of the Document Type Definition (DTD), which also describes the document structure. Unlike DTDs, XDR schemas describe the structure of the document using the same syntax as the XML document. Additionally, in a DTD, all the data contents are character data. XDR language schemas allow you to specify the data type of an element or an attribute.

In an XDR schema, the <Schema> element encloses the entire schema. As properties of the <Schema> element, you can describe attributes that define the schema name and the namespaces in which the schema reside. In the XDR language, all element declarations must be contained within the <Schema> element.

The minimum XDR schema is:

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

</Schema>

The <Schema> element is derived from the **xml-data** namespace (**urn:schemas-microsoft-com:xml-data**).

Note This documentation assumes that you are familiar with XML-Data language.

Annotations to the XDR Schema

You can use an XDR schema with annotations that describe the mapping to the database to query the database and return the results in the form of an XML document. SQL Server 2000 introduces a number of annotations that you can use to map the XDR schema to the database tables and columns. XPath queries can be specified against the XML view created by the XDR schema to query the database and obtain results as an XML.

This is an alternative to the more complex process of writing a SQL query that uses the FOR XML EXPLICIT mode for describing the XML document structure as part of the query For more information about SELECT queries with the FOR XML EXPLICIT mode, see <u>Using EXPLICIT Mode</u>. However, for overcoming most of the limitations of XPath queries against mapping schemas, use SQL queries with the FOR XML EXPLICIT mode to return results in form of an XML document.

If you have public XDR schemas (such as a Microsoft BizTalk[™] schemas), you can perform either of these:

- Write the FOR XML EXPLICIT mode query so the data that is generated is valid against the public XDR schema; however, writing FOR XML EXPLICIT queries can be cumbersome.
- Make a private copy of the public XDR schema. Then add annotations to this private copy, thus generating a mapping schema. You can specify XPath queries against the mapping schema. As a result, what the query generates is the data in the namespace of the public schema. Creating annotated schemas and specifying XPath queries against them is a much simpler process than writing the complex FOR XML EXPLICIT queries. The illustration shows the process.

Note The Microsoft BizTalk[™] Framework is an effort to define a standard XML format to common business objects, such as Contacts, Orders, and Appointments. You can find copies of these business schemas at http://biztalk.org.

Mapping Schema

In the context of the relational database, it is useful to map the arbitrary XDR schema to a relational store. One way to achieve this is to annotate the XDR schema. An XDR schema with the annotations is referred to as a **mapping schema**, which provides information pertaining to how XML data is to be mapped to relational store. A mapping schema is, in effect, an XML view of the relational data. These mappings can be used to retrieve relational data as an XML document.

Microsoft SQL Server 2000 introduces a number of annotations that can be used in the XDR schema to map the elements and attributes to the database tables and columns. You can specify queries against the mapping schemas (XML views) using XPath (XML Path). The mapping schema describes the resulting document structure.

Namespace for Annotations

In an XDR schema, the annotations are specified using this namespace: **urn:schemas-microsoft-com:xml-sql**.

The example show that the easiest way to specify the namespace is to specify it in the <Schema> tag . The annotations must be namespace-qualified to the **urn:schemas-microsoft-com:xml-sql** namespace.

The namespace prefix that is used is arbitrary. In this documentation, the **sql** prefix is used to denote the annotation namespace and to distinguish annotations in this namespace from those in other namespaces.

Namespace for Data Types

XDR schemas allow you to specify the data type of an element or an attribute.

The data types are specified using this namespace: **urn:schemas-microsoft-com:datatypes**.

This is the minimum XDR schema with the namespace declarations:

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

</Schema>

The namespace prefix that is used is arbitrary. In this documentation, the **dt** prefix is used to denote the data type namespace and to distinguish annotations in this namespace from those in other namespaces.

The <Schema> element is derived from the **xml-data** namespace: **urn:schemas-microsoft-com:xml-data**.

Example of an XDR Schema

This example shows how annotations are added to the XDR schema. This XDR schema consists of an <Employee> element and the **EmpID**, **Fname**, and **Lname** attributes.

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

```
<ElementType name="Employee" >
<AttributeType name="EmpID" />
<AttributeType name="FName" />
<AttributeType name="LName" />
```

```
<attribute type="EmpID" />
<attribute type="FName" />
```
```
<attribute type="LName" />
</ElementType>
</Schema>
```

Now, annotations are added to this XDR schema to map its elements and attributes to the database tables and columns. This is the annotated XDR schema:

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

```
<ElementType name="Employee" sql:relation="Employees" >
<AttributeType name="EmpID" />
<AttributeType name="FName" />
<AttributeType name="LName" />
```

```
<attribute type="EmpID" sql:field="EmployeeID" />
<attribute type="FName" sql:field="FirstName" />
<attribute type="LName" sql:field="LastName" />
</ElementType>
</Schema>
```

In the mapping schema, the <Employee> element is mapped to the **Employees** table using **sql:relation** annotation. The attributes **EmpID**, **Fname**, and **Lname** are mapped to the **EmployeeID**, **FirstName**, and **LastName** columns in the **Employees** table using the **sql:field** annotations.

This annotated XDR schema provides the XML view of the relational data. This XML view can be queried using the XPath (XML Path) language. The query returns an XML document as a result, instead of the rowset returned by the SQL queries.

Note In the mapping schema, the specified relational values (such as table name and column name) are case-sensitive.

Annotations to the XDR Schema

Microsoft® SQL Server[™] 2000 introduces a number of annotations to the XDR schema language. These annotations can be used within the XDR schema to specify XML-to-relational mapping. This includes mapping between elements and attributes in the XDR schema to tables/views and columns in the databases. By default, an element name in an annotated schema maps to a table (view) name in the specified database and the attribute name maps to the column name. These annotations can also be used to specify the hierarchical relationships in XML (thus, representing the relationships in the database).

Annotation	Description	Topic link
sql:relation	Maps an XML item to the database table.	<u>Using sql:relation</u>
sql:field	Maps an XML item and the database column.	<u>Using sql:field</u>
sql:is-constant	Creates an XML element that does not map to any table. The element appears in the query output.	<u>Creating Constant</u> <u>Elements Using</u> <u>sql:is-constant</u>
sql:map-field	Allows schema items to be excluded from the result.	Excluding Schema Elements from the Resulting XML Document Using sql:map-field
<sql:relationship></sql:relationship>	Specifies relationships between XML elements. The key , key - relation , foreign-key and foreign-relation attributes are used to establish the relationship.	<u>Specifying</u> <u>Relationships Using</u> <sql:relationship></sql:relationship>
sql:limit-field sql:limit-value	Allows limiting the values returned based on a limiting value.	<u>Filtering Values</u> <u>Using sql:limit-field</u> and sql:limit-value

The table shows the list of annotations.

sql:key-fields	Allows specification of	Identifying Key
	column(s) that uniquely identify	Columns Using
	the rows in a table.	<u>sql:key-fields</u>
sql:target-	Allows placing the elements and	Specifying a Target
namespace	attributes from the default	Namespace Using
	namespace into a different	sql:target-namespace
	namespace for query results.	
sql:id-prefix	Creates valid XML ID , IDREF ,	Creating Valid ID,
	and IDREFS . Prepends the	IDREF, and IDREFS
	values of ID , IDREF , and	Type Attributes
	IDREFS with a string.	Using sql:id-prefix
sql:use-cdata	Allows specifying CDATA	Creating CDATA
	sections to be used for certain	Sections Using
	elements in the XML document.	<u>sql:use-cdata</u>
sql:url-encode	When XML element/attribute is	Requesting URL
	mapped to a SQL Server BLOB	References to BLOB
	column, allows requesting a	Data Using sql:url-
	reference (URI) to be returned	<u>encode</u>
	that can be used later for BLOB	
	data.	
sql:overflow-field	Identifies the database column	Retrieving
	that contains the overflow data.	Unconsumed Data
		Using sql:overflow-
		field

Note All of the examples presented in the topics in this section specify simple XPath queries against the annotated XDR schema described in each example. Prior familiarity with XPath language is assumed. For more information, see <u>Using XPath Queries</u>.

See Also

IIS Virtual Directory Management for SQL Server

Using XPath Queries

Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Default Mapping of XDR Elements and Attributes to Tables and Columns

In an annotated XDR schema, an <element>, by default, maps to the same name table/view, and an attribute maps to the same name column.

You can map the noncomplex subelements in the schema to the database columns. To map an <element> to a column in the database, the **content** attribute is specified for that element with the **textOnly** value. If **content=textOnly** is not specified in mapping an <element> to a database column, the **sql:field** annotation must be explicitly specified to map the <element> to a database column. For more information, see <u>Using sql:field</u>.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas</u> in <u>Queries</u>.

A. Specify default mapping

In this example, the <Employees> element maps to the **Employees** table in the **Northwind** database, and all the attributes map to same name columns in the **Employees** table. In this XDR schema, no annotations are specified.

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

```
<ElementType name="Employees" >
<AttributeType name="EmployeeID" />
<AttributeType name="FirstName" />
<AttributeType name="LastName" />
<attribute type="EmployeeID" />
<attribute type="FirstName" />
```

```
<attribute type="LastName" />
```

</ElementType>

</Schema>

Testing a sample XPath query against the schema

B. Map an XML <element> to a database column

By default, an XML <element> maps to a database table, and an <attribute> maps to database column. To map an <element> to a database column, **content** attribute is specified with **textOnly** value.

This XDR schema consists of <Employees> element with <FirstName> and <LastName> subelements and an **EmployeeID** attribute.

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

```
<ElementType name="FirstName" content="textOnly" />
<ElementType name="LastName" content="textOnly" />
```

```
<ElementType name="Employees" >
<AttributeType name="EmployeeID" />
```

```
<attribute type="EmployeeID" /> <element type="FirstName" />
```

```
<element type="LastName" />
</ElementType>
</Schema>
```

By default the <Employees> element in the XDR schema maps to the **Employees** table in the database. The **content** attribute is specified on <FirstName> and <LastName> subelements. Therefore, these subelements will map to the same name columns in the **Employees** table.

Note Mixed content (elements with both text and subelements) is not supported.

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Explicit Mapping of XDR Elements and Attributes to Tables and Columns

In providing an XML view of the relational database through an XDR schema, the elements and attributes in the schema must be mapped to database tables and columns. The rows in the database table/view will map to elements in XML. The column values in the database map to attributes or elements. To obtain a single value from the database, the mapping specified in the mapping XDR schema must have both relation and field specification.

By default, an element name in an annotated schema maps to the table (view) name in the specified database, and the attribute name (and noncomplex subelements with text-only content) maps to the column name. If the element/attribute name is the same as a table (view)/column name in the database, there is no need to explicitly specify any mappings.

However, if the element/attribute name is not the same as the table (view)/column name in the database, the following annotations are used to specify the mapping between an element/attribute in an XML document and the table/column in a database:

sql:relation

Maps an XML element to a database table.

sql:field

Maps an attribute or a noncomplex subelement to a database column.

When XPath queries are specified against the annotated XDR schema, the data for the elements and attributes in the schema is retrieved from the tables and columns to which they map.

See Also

Default Mapping of XDR Elements and Attributes to Tables and Columns

Using sql:relation

The **sql:relation** annotation is added to map an XML node in the XDR schema to a database table. A table/view name is specified as the value of **sql:relation** annotation.

The **sql:relation** annotation can be added to an <ElementType>, <element>, or <attribute> node in the XDR schema. **sql:relation** specifies the mapping between <ElementType>, <element>, or <attribute> in the schema to a table/view in a database.

When **sql:relation** is specified on <ElementType>, the scope of this annotation applies to all the attribute and subelement specifications in that <ElementType>. Therefore, it provides a shortcut in writing annotations. When **sql:relation** is specified directly on the <element>, there is also scoping introduced to attributes specified within an <ElementType>. **sql:relation** is ignored on <AttributeType>.

sql:relation is useful in cases in which identifiers that are valid in Microsoft® SQL Server[™] are invalid in XML. For example, **Order Details** is a valid table name in SQL Server but invalid in XML. In such cases, **sql:relation** annotation can be used to specify the mapping, for example:

<ElementType name="OD" sql:relation="[Order Details]">

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas</u> in <u>Queries</u>.

A. Specify sql:relation on <ElementType> containing attributes

In this example, the XDR schema consists of <Customer> element with **CustomerID**, **ContactName**, and **Phone** attributes. The **sql:relation** annotation is specified on the <ElementType>, mapping **Customer** element to the **Customers** table. The scope of this mapping applies to all the attributes in the <ElementType>. Therefore, all the attributes map to columns in the **Customers** table.

The default mapping takes places for the attributes, for example, the attributes map to same name columns in the **Customers** table.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Ustomer" sql:relation="Customers" >
<AttributeType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
<AttributeType name="ContactName" />
<AttributeType name="Phone" />
```

```
<attribute type="CustomerID" />
<attribute type="ContactName" />
<attribute type="Phone" />
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

B. Specify sql:relation on <ElementType> containing subelements and attributes

In this example, the XDR schema consists of <Customer> element with **CustomerID**, **ContactName** attributes and <Address> subelement. The **sql:relation** annotation is specified on the <ElementType>, mapping **Customer** element to the **Customers** table. The scope of this mapping applies to all the

attributes in the <ElementType>. Therefore, all the attributes map to columns in the **Customers** table.

The default mapping takes places for the attributes. The attributes map to columns with the same name in the **Customers** table.

In this example, the **content** attribute is specified on the <Address> subelement. Without the **content=textOnly** attribute, the <Address> element does not map to the **address** column in the **Customers** table because, by default, elements map to a table and not to a field.

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

```
<ElementType name="Address" content="textOnly" />
<ElementType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
<AttributeType name="ContactName" />
```

```
<attribute type="CustomerID" />
<attribute type="ContactName" />
<element type="Address" />
</ElementType>
</Schema>
```

As an alternative, instead of specifying **content=textOnly** attribute, you can specify **sql:field** annotation to map the <Address> subelement to the **Address** column:

```
<element type="Address" sql:field="Address" >
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Using sql:field

The **sql:field** annotation specifies the mapping between element or attribute in an annotated schema to a column in a database. **sql:field** can be added to an element or attribute. **sql:field** is ignored on <AttributeType> elements of the annotated schema. The **sql:field** attribute specifies the name of the mapped column in a table or view.

For example, **sql:field** can be used to specify the name of column when that name does not match with the field in schema specified in XDR. The value of **sql:field** must be a column name. Four-part column names such as *database.owner.table.columnname* are not allowed. This is true for all annotations that take a column name as its value.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas in Queries</u>.

A. Specify sql:field for an <attribute> of the XDR schema

In this annotated schema, the **sql:field** annotation is specified on the <attribute> element of the schema. The **sql:field** attribute maps the **Company** attribute in the schema to the **CompanyName** column in the **Customers** table.

Because the attribute name **CustomerID** in the XDR schema is the same as the **CustomerID** column in the **Customers** table, **sql:field** is not specified. The mapping is by default.

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

```
<ElementType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
<AttributeType name="Company" />
```

```
<attribute type="CustomerID" />
<attribute type="Company" sql:field="CompanyName" />
</ElementType>
</Schema>
```

In a mapping schema, attributes can be globally declared (for example, <AttributeType...>, declared outside the scope of the <ElementType>), and then referenced in <attribute type=...>, as shown in this schema.

In this schema, the **Contact** attribute is declared globally and referenced in the scope of the Customer <ElementType>.

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

```
<AttributeType name="Contact" />
```

```
<ElementType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
<AttributeType name="Company" />
```

```
<attribute type="CustomerID" />
```

```
<attribute type="Company" sql:field="CompanyName" />
<attribute type="Contact" sql:field="ContactName" />
</ElementType>
</Schema>
```

B. Specify sql:field for an <element> in the XDR schema

In this annotated schema, the **sql:field** annotation is specified on <element> in the schema. The **sql:field** annotation maps the <CompanyName> subelement in the schema to the **CompanyName** column in the **Customers** table.

Without the explicit annotation, the <CompanyName> subelement of the <Customer> element in the schema will not map to the CompanyName column of the **Customers** table because the default mapping of elements is to a relation, not to a field (the exception to this occurs when the <ElementType> contains a **textOnly** attribute).

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="CompanyName" />
<ElementType name="CompanyName" />
<ElementType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
```

```
<attribute type="CustomerID" />
<element type="CompanyName" sql:field="CompanyName" />
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

If **content="textOnly"** is specified on CompanyName <ElementType>, the **sql:field** annotation is not required on the <CompanyName> subelement. The CompanyName subelement will map to the **CompanyName** column in the **Customer** table.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="CompanyName" content="textOnly" />
<ElementType name="CompanyName" content="textOnly" />
<ElementType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
```

```
<attribute type="CustomerID" />
<element type="CompanyName" />
</ElementType>
</Schema>
```

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP

Executing Template Files Using HTTP

Specifying Relationships Using <sql:relationship>

The elements in an XML document can be related. The elements can be nested hierarchically, and **ID**, **IDREF**, or **IDREFS** relationships can be specified between the elements.

For example, in an XDR schema, a <customer> element contains <order> subelements. The <customer> element maps to **Customers** table and <order> element maps to **Orders** table in the database. These underlying tables, **Customers** and **Orders** are related because customers place orders. The **CustomerID** in **Orders** table is a foreign key referring to **CustomerID** primary key in **Customers** table. You can establish these relationships among mapping schema elements using the <sql:relationship> annotation.

In the annotated XDR schema, the <sql:relationship> annotation is used to nest the schema elements hierarchically based on the primary key and foreign key relationships among the underlying tables to which the elements map. In specifying the <sql:relationship> annotation, you must identify:

- The primary table (Customers) and the foreign table (Orders) and
- The necessary join condition (**CustomerID** in **Orders** is a foreign key referring to **CustomerID** primary key in **Customers** table).

This information is used in generating the proper hierarchy (for each <customer> element, the related <order> elements appear as subelements).

To provide the table names and the necessary join information, the following attributes are specified with the <sql:relationship> annotation. These attributes are valid only with the <sql:relationship> element:

• key-relation

Specifies the primary relation (table).

• key

Specifies the primary key of the **key-relation**. If the primary key is composed of multiple columns, values are specified with a space

between them. There is positional mapping between the values specified for the multicolumn key and the corresponding foreign key.

• foreign-relation

Specifies the foreign relation (table).

• foreign-key

Specifies the foreign key in the **foreign-relation** referring to **key** in **key-relation**. If the foreign key is composed of multiple attributes (columns), the foreign key values are specified with a space between them. There is positional mapping between the values specified for the multicolumn key and the corresponding foreign key.

Note You must ensure that the Microsoft® SQL Server[™] data types of the **key** and **foreign-key** are such that they can be implicitly converted if necessary.

The <sql:relationship> tag can be added only to <element> or <attribute> elements in an annotated schema. When <sql:relationship> is specified on an <attribute>, there should be a **sql:relation** and **sql:field** specified for the attribute to ensure that a single value is retrieved (multiple attributes of the same name are invalid in XML). When <sql:relationship> is specified on an <element>, the relationship may result in a single value or a set of values.

The <sql:relationship> tag is used to specify a single logical relationship between two entities. The attributes define the relations and fields used to define the logical relationship. Multiple instances of <sql:relationship> may be specified within an <element> or <attribute> in the annotated schema, which indicates a complex relationship between the <element> or <attribute> and its contained element. All instances of <sql:relationship> are used together to define the complex relationship.

When multiple instances of <sql:relationship> tag are specified within an <element> or <attribute>, the order in which they appear is significant.

sql:key-fields must be specified in an <element> containing a child element and a <sql:relationship>, defined between the element and the child, that does not provide the primary key of the table specified in the parent element. For more information, see <u>Identifying Key Columns Using sql:key-fields</u>. To produce proper nesting in the result, it is recommended that **sql:key-fields** be specified in

all schemas.

Note In the mapping schema, relational values such as table name and column name are case-sensitive.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas</u> in <u>Queries</u>.

A. Specify <sql:relationship> on an <element>

This annotated XDR schema includes <Customer> and <Order> elements. The <Order> element is a subelement of <Customer> element.

In the schema, the <sql:relationship> annotation is specified on the <Order> subelement. The annotation identifies **CustomerID** in the **Orders** table as a foreign key referring to the **CustomerID** primary key in the **Customers** table. Therefore, orders belonging to a customer appear as a subelement of that <Customer> element.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Urne=""
<Attributeschemas-microsoft-com:xml-sql">
<ElementType name=""
<Attributeschemas-microsoft-com:xml-sql">
<ElementType name=""
<Attributeschemas-microsoft-com:xml-sql">
<ElementType name=""
<Attributeschemas-microsoft-com:xml-sql">
<ElementType name=""
<Attributeschemas-microsoft-com:xml-sql">
<
<Attributeschemas-microsoft-com:xml-sql">
<
<Attributeschemas-microsoft-com:xml-sql">
<
<
<Attributeschemas-microsoft-com:xml-sql">
<
<
<
<
<
AttributeType name=""
Order" sql:relation=""
```

```
<attribute type="CustomerID" />
  <attribute type="OrderID" />
  <attribute type="OrderDate" />
</ElementType>
<ElementType name="Customer" sql:relation="Customers" >
  <AttributeType name="CustomerID" />
  <attribute type="CustomerID" />
  <element type="Order" >
       <sql:relationship
            key-relation="Customers"
           key="CustomerID"
           foreign-key="CustomerID"
           foreign-relation="Orders" />
  </element>
</ElementType>
</Schema>
```

Note In the mapping schema, the relational values such as the table name and column name are case-sensitive. In the previous example, **Customers** is the value of **sql:relation** attribute. The corresponding **key-relation** attribute value must also be **Customers**.

Testing a sample XPath query against the schema

B. Specify <sql:relationship> on an <attribute> and create document references using ID and IDREFS.

In this example, local document references are specified using **ID** and **IDREFS**. The sample XDR schema consists of <Customer> element that maps to the **Customers** table. This element consists of an <Order> subelement that maps to the **Orders** table.

In the example, <sql:relationship> is specified twice:

• <sql:relationship> is specified on the <Order> subelement. Therefore, orders belonging to a customer will appear as subelement of that

<Customer> element.

 <sql:relationship> is also specified on the OrderIDList attribute of the <Customer> element. This attribute is defined as IDREFS type referring to the OrderID attribute (an ID type attribute) of the <Order> element. Therefore, <sql:relationship> is required. In this case, the <sql:relationship> annotation allows a list of orders belonging to a customer to appear with that <Customer> element.

Attributes specified as **IDREFS** can be used to refer to **ID** type attributes, thus enabling intradocument links.

Because numbers are not valid ID values (must be name tokens), **sql:id-prefix** has been used to make the Order ID a string value. For more information, see <u>Using sql:id-prefix</u>.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Order" sql:relation="Orders" >
<ElementType name="Order" sql:relation="Orders" >
<AttributeType name="OrderID" dt:type="id" sql:id-prefix="Ord-",
<AttributeType name="OrderDate" />
```

```
<attribute type="OrderID" />
<attribute type="OrderDate" />
</ElementType>
```

```
<ElementType name="Customer" sql:relation="Customers">
<AttributeType name="CustomerID" />
<AttributeType name="ContactName" />
```

```
<attribute type="CustomerID" />
<attribute type="ContactName" />
<AttributeType name="OrderIDList" dt:type="idrefs"
```

```
sql:id-prefix="Ord-"/>
  <attribute type="OrderIDList" sql:relation="Orders"
                    sql:field="OrderID">
          <sql:relationship
            key-relation="Customers"
            key="CustomerID"
             foreign-relation="Orders"
             foreign-key="CustomerID" />
  </attribute>
  <element type="Order">
          <sql:relationship key-relation="Customers"
            key="CustomerID"
             foreign-relation="Orders"
             foreign-key="CustomerID" />
  </element>
 </ElementType>
</Schema>
```

C. Specify <sql:relationship> on multiple <element>s

In this example, the annotated XDR schema consists of the <Customer>, <Order>, and <OD> elements.

The <Order> element is a subelement of <Customer> element. <sql:relationship> is specified on the <Order> subelement so that orders belonging to a customer appear as subelements of <Customer>.

The <Order> element includes <OD> subelement. <sql:relationship> is specified on <OD> subelement so that the order details belonging to an order appear as subelements of that <Order> element.

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

```
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
```

```
<ElementType name="OD" sql:relation="[Order Details]" >
<AttributeType name="OrderID" />
<AttributeType name="ProductID" />
```

```
<attribute type="OrderID" />
<attribute type="ProductID" />
</ElementType>
```

```
<ElementType name="Order" sql:relation="Orders" >
<AttributeType name="CustomerID" />
<AttributeType name="OrderID" />
<AttributeType name="OrderDate" />
```

```
<attribute type="CustomerID" />
<attribute type="OrderID" />
<attribute type="OrderDate" />
<element type="OD" >
<sql:relationship
key-relation="Orders"
key="OrderID"
foreign-key="OrderID"
foreign-relation="[Order Details]" />
</element>
```

```
</ElementType>
```

```
<ElementType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
```

```
<attribute type="CustomerID" />
<element type="Order" >
<sql:relationship
```

```
key-relation="Customers"
key="CustomerID"
foreign-key="CustomerID"
foreign-relation="Orders" />
</element>
</ElementType>
</Schema>
```

D. Specify indirect relationships

In this example, the annotated XDR schema consists of the <Customer>, <OD> elements. The relationship between these elements is indirect (**Customers** table is related to **Order Details** table through the **Orders** table). To relate a customer to the order details, first the relationship between the **Customer** table and the **Orders** table is specified. Then, the relationship between the **Orders** and **Order Details** tables is specified.

This is the schema:

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Unitropy of the split of t
```

```
<attribute type="OrderID" />
<attribute type="ProductID" />
<attribute type="UnitPrice" />
</ElementType>
<ElementType name="Customer" sql:relation="Customers" >
```

```
<AttributeType name="CustomerID" />
  <attribute type="CustomerID" />
  <element type="OD" >
       <sql:relationship
           key-relation="Customers"
           key="CustomerID"
           foreign-relation="Orders"
           foreign-key="CustomerID"/>
       <sql:relationship
           key-relation="Orders"
           key="OrderID"
           foreign-relation="[Order Details]"
           foreign-key="OrderID" />
  </element>
</ElementType>
</Schema>
```

E. Specify multikey join relationships

In specifying a join using <sql:relationship>, you can specify a join involving two or more columns. In this case, the column names for **key** and **foreign-key** are listed using a space.

This example assumes these two tables exist:

- Cust(fname, lname)
- Ord(OrderID, fname, lname)

The **fname** and **lname** columns form the primary key of the **Cust** table. The **OrderID** is the primary key of the **Ord** table. The **fname** and **lname** in **Ord** table are foreign keys referring to **fname** and **lname** primary key of the **Cust** table.

This schema consists of <Cust> and <Ord> elements. <sql:relationship> is used to join them.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
    xmlns:dt="urn:schemas-microsoft-com:datatypes"
    xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Ord" sql:relation="Ord" >
  <AttributeType name="OrderID" />
  <attribute type="OrderID" />
</ElementType>
<ElementType name="Cust" sql:relation="Cust" >
  <AttributeType name="fname" />
  <AttributeType name="lname" />
  <attribute type="fname" />
  <attribute type="lname" />
  <element type="Ord" >
       <sql:relationship
           key-relation="Cust"
           key="fname lname"
           foreign-relation="Ord"
           foreign-key="fname lname"/>
  </element>
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Creating Constant Elements Using sql:is-constant

Because of the default mapping, every element and attribute in the XDR schema maps to a database table and column. At times, you may want create an element in the XDR schema that does not map to any database table or column but still appears in the XML document. These are called constant elements. To create a constant element, specify the **sql:is-constant** annotation. **sql:is-constant** takes a Boolean value (0 = FALSE, 1 = TRUE).

This annotation is specified on <ElementType>, which does not map to any database table, thereby making it a constant element. The **sql:is-constant** annotation can be used for:

- Adding a top-level element to the XML document. XML requires a single top-level element (<root> element) for the document.
- Creating container elements, for example, an <Orders> element that wraps all Orders.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas in Queries</u>.

A. Specify sql:is-constant to add a container element

In this annotated XDR schema, <OrderList> is defined as a constant element containing all the <Orders> subelements. The **sql:is-constant** annotation is specified on the **OrderList** <ElementType>, making it a constant, and therefore

not mapping to any database table. Although <OrderList> element does not map to any database table/column, it still appears in the resulting XML as a container element containing <Orders> subelements.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Orders" >
 <AttributeType name="OrderID" />
 <attribute type="OrderID" />
</ElementType>
<ElementType name="OrderList" sql:is-constant="1">
 <element type="Orders">
   <sql:relationship
          key-relation="Customers"
          foreign-relation="Orders"
          key="CustomerID"
          foreign-key="CustomerID" />
 </element>
</ElementType>
<ElementType name="Customers" >
 <AttributeType name="CustomerID" />
 <attribute type="CustomerID" />
 <element type="OrderList" />
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server

Using XPath Queries

Accessing SQL Server Using HTTP Executing Template Files Using HTTP
Excluding Schema Elements from the Resulting XML Document Using sql:map-field

Because of the default mapping, every element and attribute in the XDR schema maps to a database table and column. At times, you may want create an element in the XDR schema that does not map to any database table or column and does not appear in the XML. This is done by specifying the **sql:map-field** annotation.

The **sql:map-field** annotation differs from **sql:is-constant** in that the unmapped elements and attributes do not appear in the XML document. **sql:map-field** is especially useful if the schema cannot be modified or is used to validate XML from other sources yet contains data that is not stored in your database.

sql:map-field takes a Boolean value (0 = FALSE, 1 = TRUE). The **sql:map-field** annotation is valid only on an <attribute>, <element> or <ElementTypes> with text-only content (**content=textOnly**). The annotation is not valid on an <element> or <ElementTypes> that maps to tables.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas in Queries</u>.

A. Specify the sql:map-field annotation

Assume you have an XDR schema from some other source. This XDR schema consists of <Employees> element with **EmployeeID**, **FirstName**, **LastName**, and **HomeAddress** attributes.

In mapping this XDR schema to the **Employees** table in the database, **sql:map**-

field is specified on the **HomeAddress** attribute because the **Employees** table does not store home addresses of employees. As a result, this attribute is not returned in the resulting XML document when an XPath query is specified againt the mapping schema.

Default mapping takes place for the rest of the schema. The <Employees> element maps to the **Employees** table, and all the attributes map to the columns with the same name in the **Employees** table. For more information about default mapping, see <u>Default Mapping of XDR Elements and Attributes to Tables and Columns</u>.

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

```
<ElementType name="Employees" >
<AttributeType name="EmployeeID" />
<AttributeType name="FirstName" />
<AttributeType name="LastName" />
<AttributeType name="HomeAddress" />
```

```
<attribute type="EmployeeID" />
<attribute type="FirstName" />
<attribute type="LastName" />
<attribute type="HomeAddress" sql:map-field="0" />
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server

Using XPath Queries

Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Filtering Values Using sql:limit-field and sql:limitvalue

You can limit rows returned from a database query based on some limiting value. These annotations are used to identify the database column that contains the limiting values and to specify a specific limiting value to be used to filter the data returned.

The **sql:limit-field** annotation is used to identify a column that contains a limiting value. **sql:limit-field** is used to qualify the join relationship specified using <sql:relationship>. **sql:limit-field** must be used on an element or attribute that has <sql:relationship> specified.

The **sql:limit-value** annotation is used to specify the limited value in the column specified in a **sql:limit-field** annotation. This annotation is optional. If **sql:limit-value** is not specified, a null value is assumed.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas in Queries</u>.

A. Limit the customer addresses returned to a specific address type

In this example, a database contains two tables:

• Customer (CustomerID, CompanyName)

• Addresses (CustomerID, AddressType, StreetAddress)

A customer can have a shipping and/or a billing address (the **AddressType** column values are **Shipping** and **Billing**).

This is the mapping schema in which the **ShipTo** schema attribute maps to **StreetAddress** column in the **Addresses** relation. The values returned for this attribute are limited to only **Shipping** addresses by specifying the **sql:limit-field** and **sql:limit-value** annotations. Similarly, the **BillTo** schema attribute returns only the **Billing** address of a customer.

This is the schema:

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

```
<ElementType name="Customer" sql:relation="Customer" >
<AttributeType name="CustomerID" />
<AttributeType name="CompanyName" />
<AttributeType name="BillTo" />
<AttributeType name="ShipTo" />
```

```
<attribute type="CustomerID" />
<attribute type="CompanyName" />
<attribute type="BillTo"
sql:limit-field="AddressType"
sql:limit-value="billing"
sql:field="StreetAddress"
sql:relation="Addresses" >
<sql:relationship
key="CustomerID"
key-relation="Customer"
foreign-relation="Addresses"
foreign-key="CustomerID" />
```

```
</attribute>
<attribute type="ShipTo"
sql:limit-field="AddressType"
sql:limit-value="shipping"
sql:field="StreetAddress"
sql:relation="Addresses" >
<sql:relationship
key="CustomerID"
key-relation="Customer"
foreign-relation="Addresses"
foreign-key="CustomerID" />
</attribute>
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Identifying Key Columns Using sql:key-fields

When an XPath query is specified against the XDR schema, key information is required in most cases to obtain proper nesting in the result. Specifying the **sql:key-fields** annotation is a way to ensure that the appropriate hierarchy is generated.

Note To produce proper nesting in the result, it is recommended that **sql:key-fields** be specified in all schemas.

In many instances, it is necessary to understand how to uniquely identify the rows in a table to generate the appropriate XML hierarchy. The **sql:key-fields** annotation can be added to the <element> and <ElementType> to identify column(s) that uniquely identify rows in the table.

The value of **sql:key-fields** identifies the column(s) that uniquely identify the rows in the relation specified in the <ElementType>. If more than one column is required to uniquely identify a row, the column values are listed separated with a space.

sql:key-fields must be specified in an element containing a child element and a <sql:relationship>, defined between the element and the child, that does not provide the primary key of the table specified in the parent element.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas</u> in <u>Queries</u>.

A. Produce the appropriate nesting when <sql:relationship> does

not provide sufficient information

This example shows where **sql:key-fields** must be specified.

Consider the following schema. The schema specifies hierarchy between <Order> and <Customer> elements in which <Order> element is the parent and the <Customer> element is a child.

The <sql:relationship> tag is used to specify the parent-child relationship. <sql:relationship> identifies **CustomerID** as **foreign-key** in the **Orders** table referring to **CustomerID key** in the **Customers** table. This information provided in <sql:relationship> is not sufficient to uniquely identify rows the parent table (**Orders**). Therefore, without **sql:key-fields**, the hierarchy generated is inaccurate.

With **sql:key-fields** specified on <Order>, the annotation uniquely identifies the rows in the parent (**Orders** table) and its child elements appear below its parent.

This is the schema:

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Customer" sql:relation="Customers">
<ElementType name="Customer" sql:relation="Customers">
<AttributeType name="CustomerID" />
<AttributeType name="ContactName" />
```

```
<attribute type="CustomerID" />
<attribute type="ContactName" />
</ElementType>
```

```
<ElementType name="Order" sql:relation="Orders"
sql:key-fields="OrderID" >
<AttributeType name="OrderID" />
<AttributeType name="CustomerID" />
```

```
<attribute type="OrderID" />
```

```
<attribute type="CustomerID" />
<element type="Customer" >
<sql:relationship
key-relation="Orders"
key="CustomerID"
foreign-relation="Customers"
foreign-key="CustomerID" />
</element>
</ElementType>
</Schema>
```

Creating a working sample of this schema

B. Specify sql:key-fields to produce proper nesting in the result

In this schema, there is no hierarchy specified using <sql:relationship>. The schema still requires the **sql:key-fields** annotation specified to uniquely identify employees in the **Employees** table.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Region" content="textOnly" >
<ElementType name="Region" content="textOnly" >
<AttributeType name="EmployeeID" />
<attribute type="EmployeeID" />
</ElementType>
```

```
<ElementType name="Employees" sql:key-fields="EmployeeID" >
<element type="Region" />
</ElementType>
</Schema>
```

Creating a working sample of this schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Specifying a Target Namespace Using sql:targetnamespace

The **sql:target-namespace** annotation can be used to place elements and attributes from the default namespace into a different namespace. The **sql:target-namespace** attribute can be added only to the <Schema> tag in the XDR schema.

The value of **sql:target-namespace** is the namespace URI (Uniform Resource Identifier) to be used for generating elements and attributes specified in the mapping schema. This URI is applied to all elements and attributes in the default namespace. The XML document returned from queries against this schema contain **xmlns:prefix="uri"** declarations and prefix the element and attribute names accordingly. The URI that is used comes from the value of the **sql:target-namespace** annotation. However, the prefix is generated arbitrarily and does not correspond to any values in the schema (even if the prefixes are used in the schema).

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas in Queries</u>.

A. Specify a target namespace

In this example, **sql:target-namespace** annotation is used to specify the target namespace. As a result, all the elements and attributes that would have gone to the default namespace are redirected to the target namespace (**MyNamespace**).

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
    xmlns:dt="urn:schemas-microsoft-com:datatypes"
    xmlns:sql="urn:schemas-microsoft-com:xml-sql"
    sql:target-namespace="urn:MyNamespace">
<ElementType name="Orders" >
 <AttributeType name="OrderID" />
 <attribute type="OrderID"/>
</ElementType>
<ElementType name="Customers" >
 <AttributeType name="CustomerID" />
 <attribute type="CustomerID" />
 <AttributeType name="Contact" />
 <attribute type="Contact" sql:field="ContactName" />
 <element type="Orders" >
   <sql:relationship
      key="CustomerID"
      foreign-key="CustomerID"
      key-relation="Customers"
      foreign-relation="Orders" />
 </element>
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

```
IIS Virtual Directory Management for SQL ServerUsing XPath QueriesAccessing SQL Server Using HTTPExecuting Template Files Using HTTP
```

Creating Valid ID, IDREF, and IDREFS Type Attributes Using sql:id-prefix

An attribute can be specified to be an **ID** type attribute. Attributes specified as **IDREF** or **IDREFS** can then be used to refer to the **ID** type attributes, thus enabling intradocument links.

ID, **IDREF**, and **IDREFS** correspond to PK/FK (primary key/foreign key) relationships in the database, with few differences. In the XML document, the values of **ID** type attributes must be distinct. If you have **CustomerID** and **OrderID** attributes in an XML document, these values must be distinct. However, in a database, **CustomerID** and **OrderID** columns can have the same values (for example, **CustomerID** = **1** and **OrderID** = **1** are valid in the database).

For the **ID**, **IDREF**, and **IDREFS** attributes to be valid:

- The value of **ID** must be unique within the XML document.
- For every **IDREF** and **IDREFS**, the referenced **ID** values must be in the XML document.
- The value of an **ID**, **IDREF**, and **IDREFS** must be named token (for example, integer value 101 cannot be an **ID** value).
- The attributes of **ID**, **IDREF**, and **IDREFS** type cannot be mapped to columns of type **text**, **ntext**, **image**, or any other binary data type (for example, **timestamp**).

If an XML document contains multiple IDs, to ensure the values are unique, **sql:id-prefix** annotation is used. For more information about **sql:id-prefix**, see <u>Using sql:id-prefix</u>.

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Using sql:id-prefix

The **sql:id-prefix** annotation is used to create a valid XML **ID**, **IDREF**, or **IDREFS** attribute.

In an XML document, the values of **ID** type attributes must be distinct. If there are multiple **ID** type attributes in an XML document, to ensure that the values of these attributes are distinct, specify the **sql:id-prefix** attribute for the **ID** type attributes. **sql:id-prefix** is also used to create named tokens from numbers. The value specified for **sql:id-prefix** must be a valid name character.

The **sql:id-prefix** attribute is used to prepend the values of **ID**, **IDREF**, and **IDREFS** with a string, thereby, making it unique. No checks are made to ensure the validity of the prefixes and the uniqueness of the values of **ID**, **IDREF**, or **IDREFS**.

sql:id-prefix is ignored on attributes that are not of type **ID**, **IDREF**, or **IDREFS**.

Note Each value of the **ID**, **IDREF**, and **IDREFS** attributes is limited to 4,000 characters, including the prefix (if specified).

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas in Queries</u>.

A. Specify sql:id-prefix for an ID type attribute

In this XDR schema, **OrderID** and **EmployeeID** attributes are declared as **ID**

type. To ensure that the IDs are unique and valid, **sql:id-prefix** annotation is specified for these attributes:

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
    xmlns:dt="urn:schemas-microsoft-com:datatypes"
    xmlns:sql="urn:schemas-microsoft-com:xml-sql">
 <ElementType name="Order" sql:relation="Orders" sql:key-fields="(
  <AttributeType name="OrderID" dt:type="id" sql:id-prefix="Ord-".
  <AttributeType name="OrderDate" />
  <attribute type="OrderID" />
  <attribute type="OrderDate" />
 </ElementType>
 <ElementType name="Employee" sql:relation="Employees">
  <AttributeType name="EmployeeID" dt:type="id" />
  <AttributeType name="LastName" />
  <attribute type="EmployeeID" />
  <attribute type="LastName" />
  <a>AttributeType name="OrderList" dt:type="idrefs"</a>
                   sql:id-prefix="Ord-" />
  <attribute type="OrderList" sql:relation="Orders" sql:field="OrderI]
    <sql:relationship
         key-relation="Employees"
         key="EmployeeID"
         foreign-relation="Orders"
         foreign-key="EmployeeID" />
  </attribute>
  <element type="Order">
    <sql:relationship key-relation="Employees"
               key="EmployeeID"
               foreign-relation="Orders"
```

foreign-key="EmployeeID" />

```
</element>
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Data Type Coercions

The data type of an element or an attribute can be specified in an XDR schema. When an XDR schema is used to extract data from the database, the appropriate data format is output as a result of a query. The **dt:type** and **sql:datatype** annotations are used to control the mapping between XDR data types and Microsoft® SQL Server[™] 2000 data types.

dt:type

You can use the **dt:type** attribute to specify the XML data type of an attribute or element that maps to a column. The **dt:type** attribute can be specified on <AttributeType> or <ElementType>. The **dt:type** affects the document returned from the server and also the XPath query executed. When an XPath query is executed against a mapping schema containing **dt:type**, XPath uses the data type indicated when processing the query. For more information about how XPath uses **dt:type**, see <u>XPath Data Types</u>.

In a document returned, all SQL Server data types are converted into string representations. Some data types require additional conversions. The following table lists the conversions that are used for various **dt:type** values.

XML data type	SQL Server conversion
bit	CONVERT(bit, COLUMN)
date	LEFT(CONVERT(nvarchar(4000), COLUMN,
	126), 10)
fixed.14.4	CONVERT(money, COLUMN)
id/idref/idrefs	id-prefix + CONVERT(nvarchar(4000),
	COLUMN, 126)
nmtoken/nmtokens	id-prefix + CONVERT(nvarchar(4000),
	COLUMN, 126)
time/time.tz	SUBSTRING(CONVERT(nvarchar(4000),
	COLUMN, 126), 1+CHARINDEX(N'T',
	CONVERT(nvarchar(4000), COLUMN, 126)),
	24)
All others	No additional conversion

Note that some SQL Server values cannot be converted to some XML data types, either because the conversion is not possible (for example, "XYZ" to a number data type) or because the value exceeds the range of that data type (for example, -100000 converted to **ui2**). Incompatible type conversions may result in invalid XML documents or SQL Server errors.

Mapping from SQL Server Data Types to XML Data Types

The table shows a natural mapping from SQL Server data types to XML data types.

SQL Server data type	XML data type
bigint	i8
binary	bin.base64
bit	boolean
char	char
datetime	datetime
decimal	r8
float	r8
image	bin.base64
int	int
money	r8
nchar	string
ntext	string
nvarchar	string
numeric	r8
real	r4
smalldatetime	datetime
smallint	i2
smallmoney	fixed.14.4
sysname	string
text	string
timestamp	ui8
tinyint	ui1

varbinary	bin.base64
varchar	string
uniqueidentifier	uuid

sql:datatype

The XML data type **bin.base64** maps to various Microsoft® SQL Server[™] data types (**binary**, **image**, **varbinary**). To clearly map the XML data type **bin.base64** to a specific SQL Server data, the **sql:datatype** annotation is used. **sql:datatype** specifies the SQL Server data type of the column to which the attribute maps.

This is useful when data is being stored in the database. By specifying the **sql:datatype** annotation, you can identify the explicit SQL Server data type. The data item is then stored as the type specified in **sql:datatype**.

The valid values **for sql:datatype** are **text**, **ntext**, **image**, and **binary**).

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas</u> in <u>Queries</u>.

A. Specify dt:type on an attribute

In this XDR schema, **dt:type** is specified on the **OrdDate** and **ShipDate** attributes.

For the **ReqDate** attribute, no XPath data type is specified. Therefore, XPath returns the SQL Server **datetime** values retrieved from the **RequiredDate**

column in the database.

The **date** XPath data type is specified on **OrdDate** attribute. XPath returns only the date part of the values (and no time) retrieved from **OrderDate** column.

The **time** XPath data type is specified on **ShipDate** attribute. XPath returns only the time part of the values (and no date) retrieved from **ShippedDate** column.

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

```
<ElementType name="Order" sql:relation="Orders">
<AttributeType name="OID" />
<AttributeType name="CustID" />
<AttributeType name="OrdDate" dt:type="date" />
<AttributeType name="ReqDate" />
<AttributeType name="ShipDate" dt:type="time" />
```

```
<attribute type="OID" sql:field="OrderID" />
<attribute type="CustID" sql:field="CustomerID" />
<attribute type="OrdDate" sql:field="OrderDate" />
<attribute type="ReqDate" sql:field="RequiredDate" />
<attribute type="ShipDate" sql:field="ShippedDate" />
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

B. Specify sql:datatype on an attribute

In this example, **sql:datatype** is used to identify the SQL Server data type of the **Photo** column.

```
<?xml version="1.0" ?>
```

```
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
```

```
<ElementType name="Employee" sql:relation="Employees">
<AttributeType name="EID" />
<AttributeType name="fname" />
<AttributeType name="lname" />
<AttributeType name="photo" />
```

```
<attribute type="EID" sql:field="EmployeeID" />
<attribute type="fname" sql:field="FirstName" />
<attribute type="lname" sql:field="LastName" />
<attribute type="photo" sql:field="Photo" sql:datatype="image" />
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server

Using XPath Queries

Accessing SQL Server Using HTTP

Executing Template Files Using HTTP

Creating CDATA Sections Using sql:use-cdata

In XML, CDATA sections are used to escape blocks of text containing characters that would otherwise be recognized as markup.

Microsoft® SQL Server[™] data may contain characters that are considered special by the XML parser, for example, characters such as <, >, <=, & are treated as markup characters. If you want to avoid SQL Server data containing special characters being treated as markup, you can wrap them in a CDATA section. The text placed in the CDATA section is treated as plain text.

The **sql:use-cdata** annotation is used specify if the data returned by SQL Server be wrapped in a CDATA section. Use **sql:use-cdata** annotation to indicate if the value from the column specified by **sql:field** should be enclosed in a CDATA section. The **sql:use-cdata** annotation can be specified on <ElementType> or <element>, and takes a Boolean value (0 = FALSE, 1 = TRUE). **sql:use-cdata** cannot be used with **sql:url-encode** or on any of the attribute types **ID**, **IDREF**, **IDREFS**, **NMTOKEN**, or **NMTOKENS**.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas</u> in <u>Queries</u>.

A. Specify sql:use-cdata on an element

In this schema, **sql:use-cdata** is set to 1 (TRUE) for the <ProductName> element. As a result, the data for <ProductName> is returned in the CDATA section.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="ProductName" content="textOnly" />
<ElementType name="ProductName" content="textOnly" />
<ElementType name="Products" >
<element type="ProductName" sql:use-cdata="1" />
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Requesting URL References to BLOB Data Using sql:url-encode

In the annotated XDR schema, when an attribute (or element) is mapped to a Microsoft® SQL Server[™] BLOB column, the data is returned in Base 64-encoded format within XML. For a description of the SQL Server data types and their corresponding XML data types, see <u>Data Type Coercions</u>.

If you want a reference to the data (URI) to be returned that can be used later to retrieve the BLOB data in a binary format, specify the **sql:url-encode** annotation.

Specify **sql:url-encode** annotation to indicate that a URL to the field should be returned instead of the value of the field. **sql:url-encode** depends on the primary key to generate a singleton select in the URL. The primary key can be specified using **sql:key-fields** annotation. For more information, see <u>Identifying Key</u> <u>Columns Using sql:key-fields</u>.

The **sql:url-encode** annotation takes a Boolean type value (0 = FALSE, 1 = TRUE). **sql:url-encode** cannot be used with **sql:use-cdata** or on any of the attribute types **ID**, **IDREF**, **IDREFS**, **NMTOKEN**, or **NMTOKENS**.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas</u> in <u>Queries</u>.

A. Specify sql:url-encode to obtain a URL reference to BLOB data

In this example, the mapping schema specifies **sql:url-encode** on the **Photo**

attribute to retrieve the URI reference to the employee photo (instead of retrieving the binary data in Base 64-encoded format).

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Employee" sql:relation="Employees"
sql:key-fields="EmployeeID" >
<AttributeType name="EmployeeID" />
<AttributeType name="Photo" />
```

```
<attribute type="EmployeeID" />
<attribute type="Photo" sql:url-encode="1" />
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server

Using XPath Queries

Accessing SQL Server Using HTTP

Executing Template Files Using HTTP

Retrieving Unconsumed Data Using sql:overflow-field

When records are inserted in the database from an XML document using OPENXML, all the unconsumed data from the source XML document can be stored in a column. In retrieving data from the database using annotated schemas, the **sql:overflow-field** attribute can be specified to identify the column in the table in which the overflow data is stored.

This data is then retrieved in these ways:

- Attributes stored in the overflow column are added to the element containing the **sql:overflow-field** annotation.
- The subelements, and their descendents, stored in the overflow column in the database are added as subelements, following the content that is explicitly specified in the schema (no order is preserved).

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas in Queries</u>.

A. Specify sql:overflow-field for an <ElementType> in the XDR schema

The example assumes this table exists:

CREATE TABLE Customers2 (CustomerID VARCHAR(10),

```
ContactName VARCHAR(30),
OverflowData NVARCHAR(200))
GO
INSERT INTO Customers2 VALUES (
'ALFKI',
'Joe',
N'<xyz><address>111 Maple, Seattle</address></xyz>')
GO
```

In this example, the mapping schema retrieves the unconsumed data stored in the **OverflowData** column of the **Customers2** table. The **sql:overflow-field** attribute is specified on the <ElementType>.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes"
xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Customers2" sql:overflow-field="OverflowDa
<AttributeType name="CustomerID" />
<AttributeType name="ContactName" />
```

```
<attribute type="CustomerID" />
<attribute type="ContactName"/>
</ElementType>
</Schema>
```

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP
Executing Template Files Using HTTP

Specifying Default Values for Attributes in the XDR Schema

In a database columns can be assigned default values. Similarly, in an XDR schema, default values can be set for attributes (elements cannot be assigned default values in the XDR schema). The XDR schema allows the **default** attribute specification on <AttributeType>.

If a column value associated with an attribute is NULL, that attribute is not returned for that instance of the element. But if the **default** attribute is specified on the <AttributeType>, then the attribute is returned with the default value specified.

For example, in extracting data from the database into an XML document, if one of the attribute values is missing, a default value of that attribute in the XDR schema is used.

Note The default values may not appear in the document that is returned, rather this value is used by the validating parser whenever the attribute is not present.

The default value is used if the parser used is schema-aware. That is, for the MSXML parser, you must ensure that the **resolveExternals** flag is set to TRUE (the default), and the parser then fetches the schemas. Once parsed, the individual instances have the attributes (for which the default is specified), regardless of whether the attribute was included in the XML document. The DOM supplies the default value.

Examples

To create working samples using these examples, you must create the **nwind** virtual directory (to access the **Northwind** database) and a virtual name of **template** type. For more information about creating the **nwind** virtual directory, see <u>Creating the nwind Virtual Directory</u>.

In creating working samples in each example, templates are used to specify XPath queries against the mapping XDR schema. There are different ways of using annotated XDR schemas in queries, for example, inline schemas and schemas in the URL. For more information, see <u>Using Annotated XDR Schemas</u>

in Queries.

A. Specify the default value for an attribute in the XDR schema

In this example, attribute **Title** is given a default value of **XYZ**. When employee records are retrieved, a default value is assigned for the employees who do not have a title.

```
<?xml version="1.0" ?>

<Schema xmlns="urn:schemas-microsoft-com:xml-data"

xmlns:dt="urn:schemas-microsoft-com:datatypes"

xmlns:sql="urn:schemas-microsoft-com:xml-sql">

<ElementType name="Employees" >

<AttributeType name="EID" sql:field="EmployeeID"/>

<AttributeType name="FirstName" />

<AttributeType name="LastName" />

<AttributeType name="Title" default="XYZ"/>

<attribute type="EID" sql:field="EmployeeID" />

<attribute type="EID" sql:field="EmployeeID" />

<attribute type="EID" sql:field="EmployeeID" />

<attribute type="EID" sql:field="EmployeeID" />

<attribute type="LastName" />

<attribute type="LastName" />

<attribute type="LastName" />

<attribute type="Title" />

<attribute t
```

</Schema>

Testing a sample XPath query against the schema

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Using Annotated XDR Schemas in Queries

These are the ways queries can be specified against annotated schema to retrieve data from the database:

• Specify XPath queries in a template against the XDR schema

The <sql:xpath-query> element allows you to specify an XPath query against the XML view defined by the annotated schema. The annotated schema against which the XPath query is to be executed is identified by using **mapping-schema** attribute of the <sql:xpath-query> element.

Templates are valid XML documents that contain one or more queries. The FOR XML and XPath queries return a document fragment. Templates act as containers for the resulting document fragments (templates provide a way to specify a single, top-level element).

The examples in this topic use templates to specify an XPath query against an annotated schema to retrieve data from the database.

For more information about templates, see <u>Executing Template Files</u> <u>Using a URL</u>.

• Inline Mapping Schemas

An annotated schema can be included directly in a template. The **sql:is-mapping-schema** annotation is used to specify an inline annotated schema. **sql:is-mapping-schema** takes a Boolean type value (0 = FALSE, 1 = TRUE). **sql:is-mapping-schema** is specified on the <Schema> element in the template.

The **sql:id** attribute uniquely identifies the element in which it is contained. **sql:id** is of the **ID** type attribute and is specified on the <Schema> element. The value assigned to **sql:id** is then used to reference the inline annotated schema using the **mapping-schema** attribute in **<sql:xpath-query>**.

For example, this is a template with an inline annotated schema is specified:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
```

```
sql:id="MyMappingSchema"
sql:is-mapping-schema="1">
```

<ElementType name="Employees" > <AttributeType name="EmployeeID" /> <AttributeType name="FirstName" /> <AttributeType name="LastName" />

```
<attribute type="EmployeeID" />
<attribute type="FirstName" />
<attribute type="LastName" />
</ElementType>
</Schema>
```

```
<sql:xpath-query mapping-schema="#MyMappingSchema">
Employees
</sql:xpath-query>
</ROOT>
```

For illustration purposes, this template is stored in the template subdirectory of the virtual root directory, and the file name is InlineSchemaTemplate.xml.

This URL executes the template:

http://IISServer/VirtualRoot/template/InlineSchemaTemplate.x

In the URL, **template** is a virtual name (created by using the IIS Virtual Directory Management for SQL Server utility) of the **template** type, followed by the template file name.

• Mapping Schema in the URL

An XPath query can be specified against the annotated schema directly in a URL. This is performed by creating a virtual name of **schema** type and by specifying the annotated schema and the XPath query at the URL.

For example, consider this annotated schema:

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

```
<ElementType name="Customer" sql:relation="Customers" >
<AttributeType name="CustomerID" />
<AttributeType name="ContactName" />
<AttributeType name="Phone" />
```

```
<attribute type="CustomerID" />
<attribute type="ContactName" />
<attribute type="Phone" />
</ElementType>
</Schema>
```

For illustration purposes, this XDR schema is stored in the schema subdirectory of the virtual root directory, and the file name is Schema2.xml.

An XPath query against the annotated schema can be specified directly in the URL:

http://IISServer/VirtualRoot/schema/Schema2.xml/Customer[@

In the URL, **schema** is the virtual name of **schema** type (created by using the IIS Virtual Directory Management for SQL Server utility). Schema2.xml is the annotated schema file followed by an XPath query requesting all the customers with a **CustomerID** of **ALFKI**.

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP Executing Template Files Using HTTP

Schema Caching

Schema caching significantly improves the performance of an XPath query. When an XPath query is executed against an annotated XDR schema, the schema is stored in memory, and the necessary data structures are built in memory. If schema caching is set, the schema remains in memory, thereby improving performance for subsequent XPath queries.

You can set the schema cache size by adding the following key in the registry:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSSQLServer\Client\SQL

The schema size is set based on the available memory and the number of schemas you are using. The default SchemaCacheSize size is 31. If you set SchemaCacheSize higher, more memory is used. Therefore, you can increase the cache size if schema access seems slow, or decrease the cache size if memory is low.

For performance reasons, it is recommended that you set SchemaCacheSize higher than the number of mapping schemas you usually use. As the number of schemas increase, if SchemaCacheSize is less than the number of schemas you have, the performance degrades.

Note During development, it is recommended that you do not cache the schemas, because the changes to the schemas are not reflected in the cache for about two minutes.

See Also

IIS Virtual Directory Management for SQL Server Using XPath Queries Accessing SQL Server Using HTTP XML Error Messages

Using XPath Queries

The Microsoft® SQL Server[™] 2000 support for annotated XDR schemas allows you to create XML views of the relational data stored in the database. You can use a subset of the XPath language to query the XML views created by an annotated XDR schema.

The XPath query can be specified as part of a URL or within a template. The mapping schema determines the structure of this resulting fragment, and the values are retrieved from the database. This process is conceptually similar to creating views using the CREATE VIEW statement and writing SQL queries against them.

Note To understand XPath queries, you must be familiar with the concepts of templates (for more information, see <u>Using XML Templates</u>), HTTP access to SQL Server (for more information, see <u>Accessing SQL Server Using HTTP</u>), mapping schema (for more information, see <u>Creating XML Views Using Annotated XDR Schemas</u>), and the XPath standard defined by the World Wide Web Consortium (W3C).

An XML document consists of nodes such as an element node, attribute node, text node, and so on. For example, consider this XML document:

```
<root>
```

```
<Customer cid= "C1" name="Janine" city="Issaquah">
<Order oid="O1" date="1/20/1996" amount="3.5" />
<Order oid="O2" date="4/30/1997" amount="13.4">Customer was
very satisfied</Order>
</Customer>
<Customer cid="C2" name="Ursula" city="Oelde" >
<Order oid="O3" date="7/14/1999" amount="100" note="Wrap it |
<Urgency>Important</Urgency>
</Order>
</Order oid="O4" date="1/20/1996" amount="10000"/>
</Customer>
</root>
```

In this document, Customer is an element node, cid is an attribute node, and Important is a text node.

XPath (XML Path Language) is a graph navigation language. XPath is used to select a set of nodes from an XML document. Each XPath operator selects a node-set based on a node-set selected by a previous XPath operator. For example, given a set of <Customer> nodes, XPath can select all <Order> nodes with the **date** attribute value **7/14/1999**. The resulting node-set contains all the orders with order date 7/14/1999.

Note XPath language is defined by the W3C as a standard navigation language. The XPath language specification, XML Path Language (XPath) version 1.0 W3C Proposed Recommendation 8 October 1999, can be found at the W3C Web site at <u>http://www.w3.org/TR/1999/PR-xpath-19991008.html</u>. A subset of this specification is implemented in SQL Server 2000. For more information, see <u>XPath Guidelines and Limitations</u>.

Supported Functionality

The table shows the features of the XPath language that are implemented in SQL Server 2000.

Feature	Item	Link to sample queries
Axes	attribute, child,	Specifying Axes in
	parent, and self axes	XPath Queries
Boolean-valued		Specifying Arithmetic
predicates including		Operators in XPath
successive and nested		<u>Queries</u>
predicates		
All relational operators	=, !=, <, <=, >, >=	Specifying Relational
		Operators in XPath
		<u>Queries</u>
Arithmetic operators	+, -, *, div	Specifying Arithmatic
		Operators in XPath
Explicit conversion	number(), string(),	Specifying Explicit
functions	Boolean()	Conversion Functions in
		XPath Queries
Boolean operators	AND, OR	Specifying Boolean

		Operators in XPath
		<u>Queries</u>
Boolean functions	true(), false(), not()	Specifying Boolean
		Functions in XPath
		<u>Queries</u>
XPath variables		Specifying XPath
		Variables in XPath
		<u>Queries</u>

Unsupported Functionality

The table shows the features of the XPath language that are not implemented in SQL Server 2000.

Feature	Item
Axes	ancestor, ancestor-or-self, descendant, descendant-or-self (//), following, following-sibling, namespace, preceding, preceding-sibling
Numeric-valued predicates	
Arithmetic operators	mod
Node functions	ancestor, ancestor-or-self, descendant, descendant-or-self (//), following, following-sibling, namespace, preceding, preceding-sibling
String functions	<pre>string(), concat(), starts-with(), contains(), substring-before(), substring- after(), substring(), string-length(), normalize(), translate()</pre>
Boolean functions	lang()
Numeric functions	<pre>sum(), floor(), ceiling(), round()</pre>
Union operator	

Specifying an XPath Query

XPath queries can be specified directly in the URL or in a template that is specified in the URL. Parameters can be passed to the XPath queries specified directly in the URL or in the template using XPath variables.

XPath Queries in a URL

XPath queries can be directly specified in the URL, for example:

http://IISServer/VirtualRoot/SchemaVirtualName/SchemaFile/XPathQ

The **root** parameter is specified to provide a single top-level element. Any value can be specified for this parameter. If the query returns only one element (or if you want to receive a collection of top-level nodes), you do not have to specify this parameter.

The **SchemaVirtualName** in the URL is a virtual name of **schema** type created using the IIS Virtual Directory Management for SQL Server utility. For more information, see <u>IIS Virtual Directory Management for SQL Server</u>.

When you specify XPath queries in the URL, note the following URL-specific behavior:

- XPath may contain characters such as # or + that have special meanings in the URLs. Escape these characters using the URL percent encoding, or specify the XPath in a template. For example, the URL http://IISServer/VirtualRoot/VirtualName/SchemaFile/Customers[@ is truncated at the # symbol, resulting in an invalid XPath.
- XPath expressions such as .. or // that resemble special file paths are interpreted by some browsers and modified before passing the URL to the server. Consequently, XPaths containing these expressions may not work as expected from the URL. For example:
 - The URL

http://IISServer/VirtualRoot/VirtualName/SchemaFile/Cust may be transformed by the browser to http://IIServer/VirtualRoot/VirtualName/SchemaFile/, which is invalid XPath. • The URL

http://IISServer/VirtualRoot/VirtualName/SchemaFile//Cus may be transformed by the browser to http://IISServer/VirtualRoot/VirtualName/SchemaFile/Cust which is different XPath.

XPath Queries in a Template

You can write the XPath queries in a template and specify the template in the URL. For example, this is a template with an XPath query:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:xpath-query mapping-schema="FilePath/AnnotatedSchemaFile.
Specify the XPath query
</sql:xpath-query>
</ROOT>
```

This template file is stored in the directory specified at the time a virtual name of type **template** is created. For more information about creating virtual names, see <u>Using IIS Virtual Directory Management for SQL Server Utility</u>.

This URL executes the template:

```
http://IISServer/VirtualRoot/VirtualName/TemplateFile.xml
```

The **VirtualName** specified in the URL is of **template** type.

Note There is no namespace support for XPath queries specified directly in the URL. If you want to use a namespace in an XPath query, **template** should be used. For more information about templates, see <u>Executing Template Files Using a URL</u>.

When you specify XPath queries in a template, note the following behavior:

• XPath may contain characters such as < or & that have special meanings in XML (and template is an XML document). You must escape these characters using XML &-encoding, or specify the XPath in the URL.

See Also

Retrieving XML Documents Using FOR XML Accessing SQL Server Using HTTP IIS Virtual Directory Management for SQL Server XML Error Messages

Guidelines for Using XPath Queries

Microsoft® SQL Server[™] 2000 implements a subset of the World Wide Web Consortium (W3C) XPath specification, which is located at <u>http://www.w3.org/TR/1999/PR-xpath-19991008.html</u>" <u>target=_blank>http://www.w3.org/TR/1999/PR-xpath-19991008.html</u>. The implementation of XPath queries in SQL Server 2000 differs from the W3C specification in these areas:

• Root queries

SQL Server 2000 does not support the root query (/). Every XPath query must begin at a top-level <ElementType> in the schema.

• Reporting errors

The W3C XPath specification defines no error conditions. XPath queries that fail to select any nodes return an empty node-set. In SQL Server 2000, a query may return many types of error messages. For more information, see <u>Errors in XPath Queries</u>.

• Document order

In SQL Server 2000, document order is not always determined. Consequently, numeric predicates and axes that use document order (such as **following**) are not implemented.

The lack of document order also means that the string value of a node can be evaluated only when that node maps to a single column in a single row. An element with subelements or an **IDREFS** or **NMTOKENS** node cannot be converted to string.

Note In some cases, the **key-fields** annotation or keys from the **relationship** annotation can result in a deterministic document order. However, this is not the primary use of these annotations For more information, see <u>Identifying Key Columns Using sql:key-fields</u> and <u>Specifying Relationships Using <sql:relationship></u>.

• Data types

SQL Server 2000 has limitations in implementing the XPath **string**, **number**, and **boolean** data types. For more information, see <u>XPath</u> <u>Data Types</u>.

• Cross-product queries

SQL Server 2000 does not support cross-product XPath queries, such as **Customer[Order/@OrderDate=Order/@ShippedDate]**. This query selects all Customers with any **Order** for which the **OrderDate** equals the **ShippedDate** of any Order.

However, SQL Server 2000 does support queries such as **Customer[Order[@OrderDate=@ShippedDate]]**, which selects **Customers** with any **Order** for which the **OrderDate** equals its **ShippedDate**.

See Also

Using XPath Queries

Specifying a Location Path

XPath queries are specified in the form of an expression. There are various kinds of expressions. A location path is an expression that selects a set of nodes relative to the context node. The result of evaluating an expression that is a location path is a node-set.

Types of Location Paths

A location path can take either of these forms:

• Absolute location path

An absolute location path starts at the root node of the document. It consists of a slash mark (/) optionally followed by a relative location path. The slash mark (/) selects the root node of the document.

• Relative location path

A relative location path starts at the context node in the document. A location path consists of a sequence of one or more location steps separated by a slash mark (/). Each step selects a set of nodes relative to the context node. The initial sequence of steps selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are joined. For example, **child::Order/child::OrderDetail** selects the <OrderDetail> element children of the <Order> element children of the context node.

Note In this implementation of XPath, every XPath query begins at the root context, even if the XPath is not explicitly absolute. For example, an XPath query beginning with **Customer** is treated as **/Customer**. In the XPath query, **Customer[Order]**, Customer begins at the root context but the **Order** begins at the **Customer** context. For more information, see <u>XPath Guidelines and Limitations</u>.

Location Steps

A location path (absolute or relative) is composed of location steps that contain three parts:

• Axis

The axis specifies the tree relationship between the nodes selected by the location step and the context node. The **parent**, **child**, **attribute**, and **self** axes are supported. If a **child** axis is specified in the location path, all the nodes selected by the query are the children of the context node. If a **parent** axis is specified, the node selected is the parent node of the context node. If an **attribute** axis is specified, the nodes selected are the attributes of the context node.

• Node test

A node test specifies the node type selected by the location step. Every axis (**child**, **parent**, **attribute**, and **self**) has a principal node type. For the **attribute** axis, the principal node type is <attribute>. For the **parent**, **child**, and **self** axes, the principal node type is <element>.

For example, if the location path specifies **child::Customer**, the <Customer> element children of the context node are selected. Because the **child** axis has <element> as its principal node type, the node test, **Customer**, is TRUE if **Customer** is an <element> node.

• Selection predicates (zero or more)

A predicate filters a node-set with respect to an axis. Specifying selection predicates in an XPath expression is similar to specifying a WHERE clause in a SELECT statement. The predicate is specified between brackets. Applying the test specified in the selection predicates filters the nodes returned by the node test. For each node in the node-set to be filtered, the predicate expression is evaluated with that node as the context node, with the number of nodes in the node-set as context size. If the predicate expression evaluates to TRUE for that node, the node is included in the resulting node-set.

The syntax for a location step is the axis name and node test separated by two colons (::), followed by zero or more expressions, each in square brackets. For example, in the XPath expression (location path) **child::Customer[@CustomerID='ALFKI']**, selects all the <Customer> element children of the context node. Then the test in the predicate is applied to the node-set, which returns only the <Customer> element nodes with attribute value 'ALFKI' for its CustomerID attribute.

Specifying an Axis

The axis specifies the tree relationship between the nodes selected by the location step and the context node. These axes are supported:

• child

Contains the child of the context node.

This XPath expression (location path) selects from the current context node all the <Customer> children:

child::Customer

In this XPath query, **child** is the axis. **Customer** is the node test.

• parent

Contains the parent of the context node.

This XPath expression selects all the <Customer> parents of the <Order> children:

child::Customer/child::Order[parent::Customer/@customerID=

This is same as specifying **Child::Customer**. In this XPath query, **child** and **parent** are the axes. **Customer** and **Order** are the node tests.

• attribute

Contains the attribute of the context node.

This XPath expression selects **CustomerID** attribute of the context node:

attribute::CustomerID

• self

Contains the context node itself.

This XPath expression selects the current node if it is the <Order> node:

self::Order

In this XPath query, **self** is the axis, and **Order** is the node test.

Specifying a Node Test in the Location Path

A node test specifies the node type selected by the location step. Every axis (**child**, **parent**, **attribute**, and **self**) has a principal node type. For the **attribute** axis, the principal node type is <attribute>. For the **parent**, **child**, and **self** axes, the principal node type is <element>.

Note The wildcard node test * (for example, **child::***) is not supported.

Node Test: Example 1

The location path **child::Customer** selects <Customer> element children of the context node.

In the example, **child** is the axis and **Customer** is the node test. The principal node type for the **child** axis is <element>. Therefore, the node test is TRUE if the <Customer> node is an <element> node. If the context node has no <Customer> children, an empty set of nodes is returned.

Node Test: Example 2

The location path **attribute::CustomerID** selects **CustomerID** attribute of the context node.

In the example, **attribute** is the axis and **CustomerID** is the node test. The principal node type of the **attribute** axis is <attribute>. Therefore, the node test is TRUE if **CustomerID** is an <attribute> node. If the context node has no **CustomerID**, an empty set of nodes is returned.

Note In this implementation of XPath, if a location step refers to an <element> or an <attribute> type that is not declared in the schema, an error is generated. This is different from the implementation of XPath in MSXML, which returns an empty node set.

Abbreviated Syntax for the Axes

The following abbreviated syntax for the location path is supported:

• **attribute::** can be abbreviated to **@**.

The location path **Customer[@CustomerID="ALFKI"]** is the same as **child::Customer[attribute::CustomerID="ALFKI"]**.

• **child::** can be omitted from a location step.

Thus, child is the default axis. The location path **Customer/Order** is the same as **child::Customer/child::Order**.

• **self::node()** can be abbreviated to one period (.), and **parent::node()** can be abbreviated to two periods (..).

Specifying Selection Predicates in the Location Path

A predicate filters a node-set with respect to an axis (similar to a WHERE clause in a SELECT statement). The predicate is specified between brackets. For each node in the node-set to be filtered, the predicate expression is evaluated with that node as the context node, with the number of nodes in the node-set as context size. If the predicate expression evaluates to TRUE for that node, the node is included in the resulting node-set.

XPath also allows position-based filtering. A predicate expression evaluating to a number selects that ordinal node. For example, the location path **Customer[3]** returns the third customer. Such numeric predicates are not supported. Only predicate expressions that return a Boolean result are supported.

Note For information about the limitations of this XPath implementation of XPath and the differences between it and the W3C specification, see <u>XPath</u> <u>Guidelines and Limitations</u>.

Selection Predicate: Example 1

This XPath expression (location path) selects from the current context node all the <Customer> element children that have the **CustomerID** attribute with value of **ALFKI**:

```
/child::Customer[attribute::CustomerID="ALFKI"]
```

In this XPath query, **child**, and **attribute** are the axis name. **Customer** is the node test (TRUE if **Customer** is an <element node>, because <element> is the principal node type for the **child** axis). **attribute::CustomerID="ALFKI"** is the predicate. In the predicate, **attribute** is the axis and **CustomerID** is the node test (TRUE if **CustomerID** is an attribute of the context node, because <attribute> is the principal node type of **attribute** axis).

Using the abbreviated syntax, the XPath query can also be specified as:

/Customer[@CustomerID="ALFKI"]

Selection Predicate: Example 2

This XPath expression (location path) selects from the current context node all the <Order> grandchildren that have the **OrderID** attribute with the value **1**:

```
/child::Customer/child::Order[attribute::OrderID="1"]
```

In this XPath expression, **child** and **attribute** are the axis names. **Customer**, **Order**, and **OrderID** are the node tests. **attribute::OrderID="1**" is the predicate.

Using the abbreviated syntax, the XPath query can also be specified as:

```
/Customer/Order[@OrderID="1"]
```

Selection Predicate: Example 3

This XPath expression (location path) selects from the current context node all the <Customer> children that have one or more <ContactName> children:

child::Customer[child::ContactName]

The example assumes that the <ContactName> is a <child> element of the <Customer> element in the XML document, which is referred to as **element-centric** mapping in an annotated XDR schema. For more information, see <u>Creating XML Views Using Annotated XDR Schemas</u>.

In this XPath expression, **child** is the axis name. **Customer** is the node test (TRUE if **Customer** is an <element> node, because <element> is the principal node type for **child** axis). **child::ContactName** is the predicate. In the predicate, **child** is the axis and **ContactName** is the node test (TRUE if **ContactName** is an <element> node).

This expression returns only the <Customer> element children of the context node that have <ContactName> element children.

Using the abbreviated syntax, the XPath query can also be specified as:

Customer[ContacName]

Selection Predicate: Example 4

This XPath expression selects <Customer> element children of the context node

that do not have <ContactName> element children:

child::Customer[not(child::ContactName)]

The example assumes that the <ContactName> is a subelement of <Customer> element in the XML document and the **ContactName** is a field that is not required in the database.

In this example, **child** is the axis. **Customer** is the node test (TRUE if **Customer** is an <element> node). **not(child::ContactName)** is the predicate. In the predicate **child** is the axis and **ContactName** is the node test (TRUE if **ContactName** is an <element> node).

Using the abbreviate syntax, the XPath query can also be specified as:

```
Customer[not(ContactName)]
```

Selection Predicate: Example 5

This XPath expression selects from the current context node all the <Customer> children that have the **CustomerID** attribute:

```
child::Customer[attribute::CustomerID]
```

In this example, **child** is the axis and **Customer** is node test (TRUE if **Customer** is an <element> node). **attribute::CustomerID** is the predicate. In the predicate, **attribute** is the axis and **CustomerID** is the predicate (TRUE if **CustomerID** is an <attribute> node).

Using the abbreviated syntax, the XPath query can also be specified as:

```
Customer[@CustomerID]
```

See Also

Creating XML Views Using Annotated XDR Schemas

Retrieving XML Documents Using FOR XML

Accessing SQL Server Using HTTP
Sample XPath Queries

The sample XPath queries refer to the following mapping schema. The mapping schema is an annotated XML-Data Reduced (XDR) schema. For more information about mapping schemas, see <u>Creating XML Views Using Annotated XDR Schemas</u>.

Note Before you can execute the sample XPath queries using a URL, you must create a virtual root to access the **Northwind** database and the virtual names of **template** and **schema** types. For information about creating the sample **nwind** virtual directory and the virtual names, see <u>Creating the nwind Virtual Directory</u>. For more information about accessing Microsoft® SQL Server[™] using HTTP, see <u>Accessing SQL Server Using HTTP</u>.

There are two ways to execute XPath queries against the annotated XDR schemas:

- Create a template with an XPath query in it. This template is then executed in the URL (for example, http://IISServer/VirtualRoot/TemplateVirtualName/TemplateFile.xml). In the template, you specify the mapping schema against which the XPath query is to be executed. In this case, the mapping schema must be stored in the directory (or one of its subdirectories, in which case a relative path is specified as the value of the mapping-schema attribute in the template) associated with virtual name of template type.
- The XPath query can be directly specified in the URL (for example, http://IISServer/VirtualRoot/SchemaVirtualName/SchemaFile.xml/XPatl In this case, the schema file must be stored in the directory associated with the virtual name of **schema** type.

Sample Annotated XDR Schema

In all the examples in this section, for illustration purposes, the XPath queries are specified in a template and the template is executed using HTTP. Therefore, you must use this mapping schema file, (SampleSchema1.xml), which is saved

in the directory associated with virtual name of **template** type:

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"</pre>
    xmlns:dt="urn:schemas-microsoft-com:datatypes"
    xmlns:sql="urn:schemas-microsoft-com:xml-sql">
 <ElementType name="Customer" sql:relation="Customers">
  <AttributeType name="CustomerID" dt:type="id" />
  <AttributeType name="CompanyName" />
  <AttributeType name="ContactName" />
  <AttributeType name="City" />
  <AttributeType name="Fax" />
  <AttributeType name="Orders" dt:type="idrefs" sql:id-prefix="Ord</pre>
  <attribute type="CustomerID" />
  <attribute type="CompanyName" />
  <attribute type="ContactName" />
  <attribute type="City" />
  <attribute type="Fax" />
  <attribute type="Orders" sql:relation="Orders" sql:field="OrderID";
   <sql:relationship
         key-relation="Customers"
         kev="CustomerID"
         foreign-relation="Orders"
         foreign-key="CustomerID" />
  </attribute>
  <element type="Order">
   <sql:relationship
        key-relation="Customers"
        key="CustomerID"
        foreign-relation="Orders"
```

```
foreign-key="CustomerID" />
```

</element> </ElementType>

```
<ElementType name="Order" sql:relation="Orders">
<AttributeType name="OrderID" dt:type="id" sql:id-prefix="Ord-",
<AttributeType name="EmployeeID" />
<AttributeType name="OrderDate" />
<AttributeType name="RequiredDate" />
<AttributeType name="ShippedDate" />
```

```
<attribute type="OrderID" />
<attribute type="EmployeeID" />
<attribute type="OrderDate" />
<attribute type="RequiredDate" />
<attribute type="ShippedDate" />
```

```
<element type="OrderDetail">
<sql:relationship
key-relation="Orders"
key="OrderID"
foreign-relation="[Order Details]"
foreign-key="OrderID" />
</element>
```

</ElementType>

```
<ElementType name="OrderDetail" sql:relation="[Order Details]"
sql:key-fields="OrderID ProductID">
<AttributeType name="ProductID" dt:type="idref"
sql:id-prefix="Prod-" />
<AttributeType name="UnitPrice"/>
<AttributeType name="Quantity" />
```

```
<attribute type="ProductID" />
```

```
<attribute type="UnitPrice" sql:field="UnitPrice" />
<attribute type="Quantity" />
```

```
<element type="Discount" sql:field="Discount"/>
</ElementType>
```

```
<ElementType name="Discount" dt:type="string"
sql:relation="[Order Details]"/>
```

```
<ElementType name="Employee" sql:relation="Employees">
<AttributeType name="EmployeeID" />
<AttributeType name="LastName" />
<AttributeType name="FirstName" />
<AttributeType name="Title" />
```

```
<attribute type="EmployeeID" />
<attribute type="LastName" />
<attribute type="FirstName" />
<attribute type="Title" />
</ElementType>
</Schema>
```

Note

The sample queries are grouped by the type of XPath operation that is performed by the query.

Specifying Axes in XPath Queries

The following examples show how axes are specified in XPath queries. The XPath queries in these examples are specified against the mapping schema contained in SampleSchema1.xml. For information about this sample schema, see <u>Sample XPath Queries</u>.

Examples

A. Retrieve child elements of the context node

This XPath query selects all the <Customer> child elements of the context node:

/child::Employee

In the query, **child** is the axis and **Customer** is the node test (TRUE if **Customer** is an <element> node, because <element> is the primary node type associated with the **child** axis).

The **child** axis is the default. Therefore, the query can be written as:

/Employee

To test the XPath query against the mapping schema

B. Retrieve grandchildren of the context node

This XPath query selects all the <Order> element children of the <Customer> element children of the context node:

/child::Customer/child::Order

In the query, **child** is the axis and **Customer** and **Order** are the node tests (these node tests are TRUE if **Customer** and **Order** are <element> nodes, because the <element> node is the primary node for the **child** axis). For each node matching <Customer>, the nodes matching <Orders> are added to the result. Only <Order> is returned in the result set.

The **child** axis is the default. Therefore, the query can be specified as:

/Customer/Order

To test the XPath query against the mapping schema

C. Use .. to specify the parent axis

This query retrieves all the <Order> elements whose parent is <Customer> element with a **CustomerID** attribute value of **ALFKI**. The query uses **parent** axis in the predicate to find parent of <Order> element.

/child::Customer/child::Order[../@CustomerID="ALFKI"]

The **child** axis is the default axis. Therefore, the query can be specified as:

```
/Customer/Order[../@CustomerID="ALFKI"]
```

The XPath query is equivalent to:

/Customer[@CustomerID="ALFKI"]/Order.

Note The XPath query **/Order[../@CustomerID="ALFKI"]** will return an error because there is no parent of **Order**. Although there may be elements in the mapping schema that contain **Order**, the XPath did not begin at any of them; consequently, **Order** is considered to be the top-level element type in the document.

To test the XPath query against the mapping schema

D. Specify the attribute axis

This XPath query selects all the <Customer> child elements of the context node with a **CustomerID** attribute value of **ALFKI**:

```
/child::Customer[attribute::CustomerID="ALFKI"]
```

In the predicate **attribute::CustomerID**, **attribute** is the axis and **CustomerID** is the node test (if **CustomerID** is an attribute the node test is TRUE, because the <attribute> node is the primary node for the **attribute** axis).

A shortcut to the **attribute** axis (@) can be specified, and because **child** is the default axis, it can be omitted from the query:

/Customer[@CustomerID="ALFKI"]

To test the XPath query against the mapping schema

Specifying Boolean-Valued Predicates in XPath Queries

The following examples show how Boolean-valued predicates are specified in XPath queries. The XPath queries in these examples are specified against the mapping schema contained in SampleSchema1.xml. For information about this sample schema, see <u>Sample XPath Queries</u>.

Examples

A. Specify multiple predicates

This XPath query uses multiple predicates to find order information for a given order ID and a customer ID:

/child::Customer[attribute::CustomerID="ALFKI"]/child::Order[attribu

A shortcut to the **attribute** axis (@) can be specified, and because the **child** axis is the default, it can be omitted from the query:

/Customer[@CustomerID="ALFKI"]/Order[@OrderID="Ord-10643"]

To test the XPath query against the mapping schema

B. Specify successive and nested predicates

This query shows using successive predicates. The query returns all the <Customer> child elements of the context node that have both a **City** attribute with a value of **London** and a **Fax** attribute:

/child::Customer[attribute::City="London"][attribute::Fax]

The query returns the <Customer> elements that satisfy both the conditions specified in the predicates.

A shortcut to the **attribute** axis (@) can be specified, and because the **child** axis is the default, it can be omitted from the query:

```
/Customer[@City="London"][@Fax]
```

The following XPath query illustrates the use of nested predicates. The query returns all the <Customer> child elements of the context node that include <Order> subelements with at least one of <Order> element that has an **EmployeeID** attribute value of **2**.

/Customer[Order[@EmployeeID=2]]

To test the XPath query against the mapping schema

C. Specify a top-level predicate

This query returns the <Customer> child element nodes of the context node that have <Order> element children. The query tests the location path as the top-level predicate:

```
/child::Customer[child::Order]
```

The **child** axis is the default. Therefore, the query can be specified as:

/Customer[Order]

To test the XPath query against the mapping schema

Specifying Relational Operators in XPath Queries

The following examples show how relational operators are specified in XPath queries. The XPath queries in these examples are specified against the mapping schema contained in SampleSchema1.xml. For information about this sample schema, see <u>Sample XPath Queries</u>.

Examples

A. Specify relational operator

This XPath query returns the <Customer> elements with at least one child <Order> containing an <OrderDetail> child with a **Quantity** attribute with a value greater than **5**:

/child::Customer[Order/OrderDetail[@Quantity>5]]

The predicate specified in the brackets filters the <Customer> elements. Only the <Customer> elements that have at least one <OrderDetail> grandchild with a **Quantity** attribute value greater than **5** are returned.

The **child** axis is the default. Therefore, the query can be specified as:

/Customer[Order/OrderDetail[@Quantity>5]]

To test the XPath query against the mapping schema

B. Specify relational operator in the XPath query and use Boolean function to compare the result

This query returns all the <Order> element children of the context node that have an **EmployeeID** attribute value that is less than **4**:

/child::Customer/child::Order[(attribute::EmployeeID < 4)=true()]</pre>

A shortcut to the **attribute** axis (@) can be specified, and because the **child** axis is the default, it can be omitted from the query:

/Customer/Order[(@EmployeeID < 4)=true()]</pre>

Note When this query is specified in a template, the < character must be entity encoded because the < character has special meaning in an XML document. In a template, use < to specify the < character.

To test the XPath query against the mapping schema

Specifying Arithmetic Operators in XPath Queries

The following example shows how arithmetic operators are specified in XPath queries. The XPath queries in these example is specified against the mapping schema contained in SampleSchema1.xml. For information about this sample schema, see <u>Sample XPath Queries</u>.

Examples

A. Specify the * arithmetic operator

This XPath query returns <OrderDetail> elements that satisfy the predicate specified:

/child::OrderDetail[@UnitPrice * @Quantity = 98]

In the query, **child** is the axis and **OrderDetail** is the node test (TRUE if **OrderDetail** is an <element node>, because <element> node is the primary node for the **child** axis). For all the <OrderDetail> element nodes, the test in the predicate is applied, and only those nodes that satisfy the condition are returned.

Note The numbers in XPath are double-precision floating-point numbers, and comparing floating-point numbers as in the example causes rounding.

To test the XPath query against the mapping schema

Specifying Explicit Conversion Functions in XPath Queries

The following examples show how explicit conversion functions are specified in XPath queries. The XPath queries in these examples are specified against the mapping schema contained in SampleSchema1.xml. For information about this sample schema, see <u>Sample XPath Queries</u>.

Examples

A. Use the number() explicit conversion function

The **number()** function converts an argument to a number.

Assume the value of **EmployeeID** is nonnumeric, the following query converts **EmployeeID** to a number and compares it with the value **4**. The query returns all <Employee> element children of the context node with the **EmployeeID** attribute that has a numeric value of **4**:

/child::Employee[number(attribute::EmployeeID)=4]

A shortcut to the **attribute** axis (@) can be specified, and because the **child** axis is the default, it can be omitted from the query:

/Employee[number(@EmployeeID)=4]

In relational terms, the query returns an employee with an EmployeeID of 4.

To test the XPath query against the mapping schema

B. Use the string() explicit conversion function

The **string()** function converts an argument to a string.

The following query converts **EmployeeID** to a string and compares it with the value **4**. The query returns all <Employee> element children of the context node with the **EmployeeID** attribute that has a string value of **4**:

/child::Employee[string(attribute::EmployeeID)="4"]

A shortcut to the **attribute** axis (@) can be specified, and because the **child** axis is the default, it can be omitted from the query:

```
/Employee[string(@EmployeeID)="4"]
```

In relational terms, the query returns an employee who has an EmployeeID of 4.

The following query returns <Customer> elements with a **ContactName** attribute that is a nonempty string:

```
Customer[string(@ContactName)=true()]
```

To test the XPath query against the mapping schema

Specifying Boolean Operators in XPath Queries

The following example shows how Boolean operators are specified in XPath queries. The XPath queries in this examples is specified against the mapping schema contained in SampleSchema1.xml. For information about this sample schema, see <u>Sample XPath Queries</u>.

Examples

A. Specify the OR Boolean operator

This XPath query returns the <Customer> element children of the context node with the **CustomerID** attribute value of **ALFKI** or **ANATR**:

/child::Customer[attribute::CustomerID="ALFKI" or attribute::Custom

A shortcut to the **attribute** axis (@) can be specified, and because the **child** axis is the default, it can be omitted:

/Customer[@CustomerID="ALFKI" or @CustomerID="ANATR"]

In the predicate, **attribute** is the axis and **CustomerID** is the node test (TRUE if **CustomerID** is an <attribute> node, because the <attribute> node is the primary node for the **attribute** axis). The predicate filters the <Customer> elements and returns only those that satisfy the condition specified in the predicate.

To test the XPath queries against the mapping schema

Specifying Boolean Functions in XPath Queries

The following examples show how Boolean functions are specified in XPath queries. The XPath queries in these examples are specified against the mapping schema contained in SampleSchema1.xml. For information about this sample schema, see <u>Sample XPath Queries</u>.

Examples

A. Specify the not() Boolean function

This query returns all the <Customer> child elements of the context node that do not have <Order> subelements:

/child::Customer[not(child::Order)]

The **child** axis is the default. Therefore, the query can be specified as:

/Customer[not(Order)]

To test the XPath query against the mapping schema

B. Specify the true() and false() Boolean functions

This query returns all <Customer> element children of the context node that do not have <Order> subelements. In relational terms, this query returns all customers who have not placed any orders.

```
/child::Customer[child::Order=false()]
```

The **child** axis is the default. Therefore, the query can be specified as:

```
/Customer[Order=false()]
```

This query is equivalent to:

```
/Customer[not(Order)]
```

The following query returns all the customers who have placed at least one order:

/Customer[Order=true()]

This query is equivalent to:

/Customer[Order]

To test the XPath query against the mapping schema

Specifying XPath Variables in XPath Queries

The following examples show how XPath variables are passed in XPath queries. The XPath queries in these examples are specified against the mapping schema contained in SampleSchema1.xml. For information about this sample schema, see <u>Sample XPath Queries</u>.

Examples

A. Use the XPath variables

A sample template consists of two XPath queries. Each of the XPath queries takes one parameter. The template also specifies default values for these parameters. The default values are used if parameter values are not specified. Two parameters with default values are specified in <sql:header>.

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:header>
<sql:param name='CustomerID'>ALFKI</sql:param>
<sql:param name='EmployeeID'>1</sql:param>
</sql:header>
<sql:xpath-query mapping-schema="SampleSchema1.xml">
Customer[@CustomerID=$CustomerID]
</sql:xpath-query >
<sql:xpath-query mapping-schema="SampleSchema1.xml">
Employee[@CustomerID=$CustomerID]
</sql:xpath-query >
<sql:xpath-query mapping-schema="SampleSchema1.xml">
Employee[@EmployeeID=$CustomerID]
</sql:xpath-query >
</ROOT>
```

This template is stored in a file (MyTemplate.xml) and executed using a URL:

```
http://IISServer/VirtualRoot/template/MyTemplate.xml
```

In the URL, no parameters are passed. Therefore, the default parameter values are used.

In the following URL, the *CustomerID* parameter value is provided. Therefore, the default customer ID value **ALFKI** is ignored. Because no value is provided for the *EmployeeID* parameter, the default value is used.

http://IISServer/VirtualRoot/template/MyTemplate.xml?CustomerID=I

In the following URL, both parameter values are passed (default values are ignored).

http://IISServer/VirtualRoot/template/MyTemplate.xml?CustomerID=1

Note The XPath query can be specified directly in the URL: http://IISServer/nwind/schema/SampleSchema1.xml/Customer[@Customer] CustomerID=ANATR&root=root.

The virtual name **schema** is of **schema** type. The schema file is stored in the directory associated with virtual name of **schema** type. The **root** parameter is used to specify a top-level element for the resulting XML document (root can be any value).

XPath Data Types

Microsoft® SQL Server[™] 2000, XPath, and XDR (XML-Data Reduced) have very different data types. For example, XPath does not have integer or date data types, but SQL Server and XDR have many. XDR uses nanosecond precision for time values, and SQL Server uses at most 1/300-second precision. Consequently, mapping one data type to another is not always possible. For more information about mapping SQL Server data types to XDR data types, see <u>Data Type</u> <u>Coercions</u>.

XPath has three data types: **string**, **number**, and **boolean**. The **number** data type is always an IEEE 754 double-precision floating-point. The SQL Server **float(53)** data type is the closest to XPath **number**. However, **float(53)** is not exactly IEEE 754. For example, neither NaN (Not-a-Number) nor infinity is used. Attempting to convert a nonnumeric string to **number** and trying to divide by zero results in an error.

XPath Conversions

When you use an XPath query such as **OrderDetail[@UnitPrice** > "**10.0**"], implicit and explicit data type conversions can change the meaning of the query in subtle ways. Therefore, it is important to understand how XPath data types are implemented. The XPath language specification, XML Path Language (XPath) version 1.0 W3C Proposed Recommendation 8 October 1999, can be found at the W3C Web site at <u>http://www.w3.org/TR/1999/PR-xpath-19991008.html</u>.

XPath operators are divided into four categories:

- Boolean operators (and, or)
- Relational operators (<, >, <=, >=)
- Equality operators (=, !=)
- Arithmetic operators (+, -, *, div, mod)

Each category of operator converts its operands differently. XPath operators implicitly convert their operands if necessary. Arithmetic operators convert their operands to **number**, and result in a number value. Boolean operators convert their operands to **boolean**, and result in a Boolean value. Relational operators and equality operators result in a Boolean value. However, they have different conversion rules depending on the original data types of their operands, as shown in this table.

Operand	Relational operator	Equality operator
Both operands are node-sets	TRUE if and only if there is a node in one set and a node in the second set such that the comparison of their string values is TRUE.	Same
One is a node-set, the other a string	TRUE if and only if there is a node in the node-set such that when converted to number , the comparison of it with the string converted to number is TRUE.	TRUE if and only if there is a node in the node-set such that when converted to string , the comparison of it with the string is TRUE.
One is a node-set, the other a number	TRUE if and only if there is a node in the node-set such that when converted to number , the comparison of it with the number is TRUE.	Same
One is a node-set, the other a boolean	TRUE if and only if there is a node in the node-set such that when converted to boolean and then to number , the comparison of it with the boolean converted to number is TRUE.	TRUE if and only if there is a node in the node-set such that when converted to boolean , the comparison of it with the boolean is TRUE.

Neither is a node-	Convert both operands to	Convert both operands to a	
set	number and then	common type and then	
	compare.	compare. Convert to	
		boolean if either is	
		boolean, number if either	
		is number ; otherwise,	
		convert to string .	

Note Because XPath relational operators always convert their operands to **number**, **string** comparisons are not possible. To include date comparisons, SQL Server 2000 offers this variation to the XPath specification: When a relational operator compares a **string** to a **string**, a node-set to a **string**, or a string-valued node-set to a string-valued node-set, a **string** comparison (not a **number** comparison) is performed.

Node-Set Conversions

Node-set conversions are sometimes nonintuitive. A node-set is converted to a **string** by taking the string value of only the first node in the set. A node-set is converted to **number** by converting it to **string**, and then converting **string** to **number**. A node-set is converted to **boolean** by testing for its existence.

Note Because SQL Server 2000 does not perform positional selection (for example, the XPath query **Customer[3]** means the third customer. This type of positional selection is not supported in SQL Server 2000.) on node-sets, the node-set-to-**string** or node-set-to-**number** conversions as described by the XPath specification are not implemented. SQL Server 2000 uses "any" semantics wherever the XPath specification specifies "first" semantics. For example, based on the W3C XPath specification, this XPath query

Order[OrderDetail/@UnitPrice > 10.0] selects those orders with the first **OrderDetail** that has a **UnitPrice** greater than **10.0**. In SQL Server 2000, this XPath query selects those orders with any **OrderDetail** that has a **UnitPrice** that is greater than **10.0**.

Conversion to **boolean** generates an existence test; therefore, the XPath query **Products[@Discontinued=true()]** is equivalent to the SQL expression "Products.Discontinued is not null", not the SQL expression

"Products.Discontinued = 1". To get the latter meaning, first convert the node-set to a non-**boolean** type, such as **number**. For example, **Products[number(@Discontinued) = true()]**.

Because most operators are defined to be TRUE if they are TRUE for any or one of the nodes in the node-set, these operations always evaluate to FALSE if the node-set is empty. Thus, if A is empty, both A = B and A != B are FALSE, and **not(A=B)** and **not(A!=B)** are TRUE.

Usually, an attribute or element that maps to a column exists if the value of that column in the database is not null. Elements that map to rows exist if any of their children exist. For more information see, <u>Using sql:relation</u> and <u>Using sql:field</u>.

Note Elements annotated with **is-constant** always exist. Consequently, XPath predicates cannot be used on **is-constant** elements. For more information, see <u>Creating Constant Elements Using sql:is-constant</u>.

When a node-set is converted to **string** or **number**, its XDR type (if any) is inspected in the annotated schema and that type is used to determine the conversion that is required.

Mapping XDR Data Types to XPath Data Types

The XPath data type of a node is derived from the XDR data type in the schema, as shown in this table (the node **EmployeeID** is used for illustrative purpose).

VDD Jota toma	Equivalent XPath data	
XDR data type	туре	SQL Server conversion used
None	N/A	None
bin.base64		EmployeeID
bin.hex		
boolean	boolean	CONVERT(bit, EmployeeID)
number, int,	number	CONVERT(float(53), EmployeeID)
float,		
i1, i2, i4, i8,		
r4, r8		
ui1, ui2, ui4, ui8		

id, idref, idrefs entity, entities enumeration notation nmtoken, nmtokens char dateTime dateTime.tz string uri uuid	string	CONVERT(nvarchar(4000) , EmployeeID , 126)
fixed14.4	N/A (there is no data type in XPath that is equivalent to the fixed14.4 XDR data type)	CONVERT(money , EmployeeID)
date	string	LEFT(CONVERT(nvarchar(4000) , EmployeeID , 126), 10)
time	string	SUBSTRING(CONVERT(nvarchar(4000) ,
time.tz		EmployeeID, 126), 1 + CHARINDEX(N'T', CONVERT(nvarchar(4000), EmployeeID , 126)), 24)

The date and time conversions are designed to work whether the value is stored in the database using the SQL Server **datetime** data type or a **string**. Note that the SQL Server **datetime** data type does not use **timezone** and has a smaller precision than the XML **time** data type. To include the **timezone** data type or additional precision, store the data in SQL Server 2000 using a **string** type.

When a node is converted from its XDR data type to the XPath data type, additional conversion is sometimes necessary (from one XPath data type to

another XPath data type). For example, consider this XPath query:

(@m + 3) = 4

If @m is of the **fixed14.4** XDR data type, the conversion from XDR data type to XPath data type is accomplished using:

CONVERT(money, m)

In this conversion, the node **m** is converted from **fixed14.4** to **money**. However, adding the value of 3, requires additional conversion:

CONVERT(float(CONVERT(money, m))

The XPath expression is evaluated as:

CONVERT(float(CONVERT(money, m)) + CONVERT(float(53), 3) =

As shown in the following table, this is the same conversion that is applied for other XPath expressions (such as literals or compound expressions).

	X is unknown	X is string	X is number	X is boolean
string(X)	CONVERT	-	CONVERT	CASE WHEN
	(nvarchar		(nvarchar	X THEN
	(4000) , X,		(4000) , X,	N'true' ELSE
	126)		126)	N'false' END
number(X)	CONVERT	CONVERT	-	CASE WHEN
	(float(53) , X)	(float(53),		X THEN 1
		X)		ELSE 0 END
boolean(X)	-	LEN(X) > 0	X != 0	-

Examples

A. Convert a data type in an XPath query

In the following XPath query specified against an annotated XDR schema, the query selects all the **Employee** nodes with the **EmployeeID** attribute value of **E**-

1, where "**E-**" is the prefix specified using the **sql:id-prefix** annotation.

```
Employee[@EmployeeID="E-1"]
```

The predicate in the query is equivalent to the SQL expression:

N'E-' + CONVERT(nvarchar(4000), Employees.EmployeeID, 126) = N'E-1'

Because **EmployeeID** is one of the **id** (**idref, idrefs, nmtoken, nmtokens**, and so on) data type values in the XDR schema, **EmployeeID** is converted to the **string** XPath data type using the conversion rules described previously.

CONVERT(nvarchar(4000), Employees.EmployeeID, 126)

The **"E-"** prefix is added to the string, and the result is then compared with **N'E-1'**.

B. Perform several data type conversions in an XPath query

Consider this XPath query specified against an annotated XDR schema: OrderDetail[@UnitPrice * @Quantity > 98]

This XPath query returns all the <OrderDetail> elements satisfying the predicate **@UnitPrice * @Quantity > 98**. If the **UnitPrice** is annotated with a **fixed14.4** data type in the annotated schema, this predicate is equivalent to the SQL expression:

CONVERT(float(53), CONVERT(money, [Order Details].UnitPrice)) * CONVERT(float(53), [Order Details].Quantity) > CONVERT(float(53), 98)

In converting the values in the XPath query, the first conversion converts the XDR data type to the XPath data type. Because the XDR data type of **UnitPrice** is **fixed14.4**, as described in the previous table, this is the first conversion that is used:

CONVERT(money, [Order Details].UnitPrice))

Because the arithmetic operators convert their operands to the **number** XPath data type, the second conversion (from one XPath data type to another XPath

data type) is applied in which the value is converted to **float(53)** (**float(53)** is close to the XPath **number** data type):

CONVERT(float(53), CONVERT(money, [Order Details].UnitPrice))

Assuming the **Quantity** attribute has no XDR data type, **Quantity** is converted to a **number** XPath data type in a single conversion:

CONVERT(float(53), [Order Details].Quantity)

Similarly, the value 98 is converted to the **number** XPath data type:

CONVERT(float(53), 98)

Note If the XDR data type used in the schema is incompatible with the underlying SQL Server data type in the database, or if an impossible XPath data type conversion is performed, SQL Server may return an error. For example, if **EmployeeID** attribute is annotated with **id-prefix** annotation, the XPath **Employee[@EmployeeID=1]** generates an error because **EmployeeID** has the **id-prefix** annotation and cannot be converted to **number**.
XML and Internet Support

Retrieving and Writing XML Data

You can execute SQL queries to return results as XML rather than standard rowsets. These queries can be executed directly or from within stored procedures. To retrieve results directly, you use the FOR XML clause of the SELECT statement, and within the FOR XML clause you specify an XML mode: RAW, AUTO, or EXPLICIT.

For example, this SELECT statement retrieves information from **Customers** and **Orders** table in the **Northwind** database. This query specifies the AUTO mode in the FOR XML clause:

SELECT Customers.CustomerID, ContactName, CompanyName, Orders.CustomerID, OrderDate FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID AND (Customers.CustomerID = N'ALFKI' OR Customers.CustomerID = N'XYZAA') ORDER BY Customers.CustomerID FOR XML AUTO

Whereas you can use the FOR XML clause to retrieve data as an XML document, you can use the Transact-SQL OPENXML function to insert data represented as an XML document. OPENXML is a rowset provider similar to a table or a view, providing a rowset over in-memory XML documents. OPENXML allows access to XML data as if it is a relational rowset by providing a rowset view of the internal representation of an XML document. The records in the rowset can be stored in database tables. OPENXML can be used in SELECT, and SELECT INTO statements where a source table or view can be specified.

The following example shows the use of OPENXML in an INSERT statement and a SELECT statement. The sample XML document consists of <Customers> and <Orders> elements. First, the **sp_xml_preparedocument** stored procedure parses the XML document. The parsed document is a tree representation of the nodes (elements, attributes, text, comments, and so on) in the XML document. OPENXML then refers to this parsed XML document and provides a rowset view of all or parts of this XML document. An INSERT statement using OPENXML can insert data from such a rowset into a database table. Several OPENXML calls can be used to provide rowset view of various parts of the XML document and process them, for example, inserting them into different tables (this process is also referred to as "Shredding XML into tables"). In the following example, an XML document is shredded in a way that <Customers> elements are stored in the **Customers** table and <Orders> elements are stored in the **Orders** table using two INSERT statements.

The example also shows a SELECT statement with OPENXML that retrieves **CustomerID** and **OrderDate** from the XML document.

DECLARE @hDoc int EXEC sp_xml_preparedocument @hDoc OUTPUT, N'<ROOT> <Customers CustomerID="XYZAA" ContactName="Joe" CompanyName="Company1"> <Orders CustomerID="XYZAA" OrderDate="2000-08-25T00:00:00"/> <Orders CustomerID="XYZAA" OrderDate="2000-10-03T00:00:00"/> </Customers> <Customers CustomerID="XYZBB" ContactName="Steve" CompanyName="Company2">No Orders yet! </Customers> </ROOT>' -- Use OPENXML to provide rowset consisting of customer data. **INSERT** Customers **SELECT** * FROM OPENXML(@hDoc, N'/ROOT/Customers') WITH Customers -- Use OPENXML to provide rowset consisting of order data. **INSERT** Orders **SELECT ***

```
FROM OPENXML(@hDoc, N'//Orders')
WITH Orders
-- Using OPENXML in a SELECT statement.
SELECT * FROM OPENXML(@hDoc, N'/ROOT/Customers/Orders')
-- Remove the internal representation of the XML document.
EXEC sp_xml_removedocument @hDoc
```

This illustration shows the parsed XML tree of the preceding XML document that was created by **sp_xml_pareparedocument**.



See Also

OPENXML

Writing XML Using OPENXML

Retrieving XML Documents Using FOR XML

XML and Internet Support

Retrieving XML Documents Using FOR XML

You can execute SQL queries against existing relational databases to return results as XML documents rather than as standard rowsets. To retrieve results directly, use the FOR XML clause of the SELECT statement, and within the FOR XML clause, specify one of these XML modes:

- RAW
- AUTO
- EXPLICIT

These modes are in effect only for the execution of the query for which they are set. They do not affect the results of any subsequent queries. In addition to specifying the XML mode, you can also request the XML-Data schema.

See Also

Executing Template Files Using a URL

<u>SELECT</u>

XML and Internet Support

Basic Syntax of the FOR XML Clause

The basic syntax for specifying the XML mode in the FOR clause is:

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

Arguments

XML mode

Specifies the XML mode. XML mode determines the shape of the resulting XML.

mode can be RAW, AUTO, or EXPLICIT.

XMLDATA

Specifies that an XML-Data schema should be returned. The schema is prepended to the document as an inline schema.

ELEMENTS

If the ELEMENTS option is specified, the columns are returned as subelements. Otherwise, they are mapped to XML attributes. This option is supported in AUTO mode only.

BINARY BASE64

If the BINARY Base64 option is specified, any binary data returned by the query is represented in base64-encoded format. To retrieve binary data using RAW and EXPLICIT mode, this option must be specified. In AUTO mode, binary data is returned as a reference by default.

See Also

<u>SELECT</u>

Guidelines for Using the FOR XML Clause

The FOR XML clause is valid only in the SELECT statement and is subject to these limitations:

• FOR XML is not valid in subselections, whether it is in UPDATE, INSERT, or DELETE statements, a nested SELECT statement, or other statements (SELECT INTO, assignment). For example, subselects as shown in these examples are not supported:

Example A

```
SELECT *
FROM Table1
WHERE ......(SELECT * FROM Table2 FOR XML RAW)
```

Example B

DECLARE @doc nchar(3000) SET @doc = (SELECT * FROM Customers WHERE Custome

- FOR XML is not valid for any selection that is used with a COMPUTE BY or FOR BROWSE clause, for example: SELECT OrderID, UnitPrice FROM [Order Details] ORDER BY OrderID COMPUTE SUM(UnitPrice) BY OrderI]
- GROUP BY and aggregate functions are currently not supported with FOR XML AUTO. For example: SELECT max(price), min(price), avg(price)
 FROM titles
 FOR XML AUTO
- FOR XML is not valid in a SELECT statement used in a view definition or in a user-defined function that returns a rowset. For example, this

statement is not allowed: CREATE VIEW AllOrders AS SELECT * FROM Orders FOR

However, a statement such as the following is allowed:

SELECT * FROM ViewName FOR XML AUTO are allowed.

- FOR XML cannot be used in a selection that requires further processing in a stored procedure.
- FOR XML cannot be used with cursors.
- Generally, FOR XML cannot be used for any selections that do not produce direct output to the Microsoft® SQL Server[™] 2000 client.
- FOR XML cannot be used in a stored procedure when called in an INSERT statement.
- When a SELECT statement with a FOR XML clause specifies a fourpart name in the query, the server name is not returned in the resulting XML document when the query is executed on the local computer. However, the server name is returned as the four-part name when the query is executed on a network server.

For example, consider this query:

SELECT TOP 1 LastName FROM ServerName.Northwind.dbo.Employees FOR XML AUTO

When **ServerName** is a local server, the query returns:

<Northwind.dbo.Employees LastName="Buchanan"/>

When **ServerName** is a network server, the query returns:

<ServerName.Northwind.dbo.Employees LastName="Buchana"

This can be avoided by specifying this alias:

SELECT TOP 1 LastName FROM ServerName.Northwind.dbo.Employees x FOR XML AUTO

This query returns:

<x ="Buchanan"/>

• Using derived tables in a SELECT statement with FOR XML AUTO may not produce the nesting you want.

The FOR BROWSE mode is implemented when a query with the FOR XML AUTO mode is specified. The FOR XML AUTO mode uses the information provided by the FOR BROWSE mode in determining the hierarchy in the result set.

For example, consider the following query. A derived table **P** is created in the query.

```
SELECT c.CompanyName,
o.OrderID,
o.OrderDate,
p.ProductName,
p.Quantity,
p.UnitPrice,
p.Total
FROM Customers AS c
JOIN
Orders AS o
ON
c.CustomerID = o.CustomerID
JOIN
(
```

```
SELECT od.OrderID,

pr.ProductName,

od.Quantity,

od.UnitPrice,

od.Quantity * od.UnitPrice AS total

FROM Products AS pr

JOIN

[Order Details] AS od

ON

pr.ProductID = od.ProductID

) AS p

ON

o.OrderID = p.OrderID

FOR XML AUTO
```

```
This is the partial result:
```

```
<c CompanyName="Vins et alcools Chevalier">
<o OrderID="10248" OrderDate="1996-07-04T00:00:00">
<pr ProductName="Queso Cabrales">
<od Quantity="12" UnitPrice="14.0000" total="168.0000"
</pr>
<pr ProductName="Singaporean Hokkien Fried Mee">
<od Quantity="10" UnitPrice="9.8000" total="98.0000"/>
</pr>
```

In the resulting XML document, the element is missing, and the <pr> and <od> elements are returned. This occurs because the query optimizer eliminates the **P** table in the result and returns a result set consisting of the **od** and **pr** tables.

This can be avoided by rewriting the query. For example, you can rewrite the query is to create a view and use it in the SELECT statement:

```
CREATE VIEW p AS
     SELECT od.OrderID,
         pr.ProductName,
         od.Quantity,
         od.UnitPrice,
         od.Quantity * od.UnitPrice AS total
     FROM Products AS pr
         JOIN
         [Order Details] AS od
         ON
         pr.ProductID = od.ProductID
And then write the SELECT statement:
SELECT c.CompanyName,
   o.OrderID,
   o.OrderDate,
   p.ProductName,
   p.Quantity,
   p.UnitPrice,
   p.total
FROM Customers AS c
   JOIN
   Orders AS o
   ON
   c.CustomerID = o.CustomerID
   JOIN
    р
   ON
   o.OrderID = p.OrderID
```

```
FOR XML AUTO
```

This is the partial result:

```
<c CompanyName="Vins et alcools Chevalier">
<o OrderID="10248" OrderDate="1996-07-04T00:00:00">
Quantity="12"
UnitPrice="14.0000"
total="168.0000"/>
</o>
```

In addition, SQL Server names containing characters that are invalid in XML names (such as spaces) are translated into XML names in a way in which the invalid characters are translated into escaped numeric entity encoding.

There are only two nonalphabetic characters that can begin an XML name: the colon (:) and the underscore (_). Because the colon (:) is already reserved for namespaces, the underscore (_) is chosen as the escape character. The escape rules used for encoding are:

- Any UCS-2 character that is not a valid XML name character (according to the XML 1.0 specification) is escaped as _xHHHH_, where HHHH stands for the four-digit hexadecimal UCS-2 code for the character in the most significant bit-first order. For example, the table name **Order Details** is encoded as Order_x0020_Details.
- Characters that do not fit into the UCS-2 realm (the UCS-4 additions of the range U+00010000 to U+0010FFFF) are encoded as __xHHHHHHHH_, where HHHHHHHH stands for the eight-digit hexadecimal UCS-4 encoding of the character.
- The underscore character does not need to be escaped unless it is followed by the character x. For example, the table name **Order_Details** is not encoded.
- The colon (:) in identifiers is not escaped so that the namespace element and attribute names can be generated by the FOR XML query. For

example, the following query generates a namespace attribute with a colon in the name: SELECT 'namespace-urn' as 'xmlns:namespace', 1 as 'namespace:a' FOR XML RAW

The query produces this result:

```
<row xmlns:namespace="namespace-urn" namespace:a="1"/>
```

• In a SELECT query, casting of any column to a binary large object (BLOB) makes it a temporary entity (losing its associated table name and column name). This causes AUTO mode queries to generate an error because it does not know where to place this value in the XML hierarchy, for example:

CREATE TABLE MyTable (Col1 int PRIMARY KEY, Col2 bi INSERT INTO MyTable VALUES (1, 0x7)

This query produces an error because of the casting to a BLOB:

SELECT Col1, CAST(Col2 as image) as Col2 FROM MyTable FOR XML AUTO

If you remove the casting, the query produces results as expected:

SELECT Col1, Col2 FROM MyTable FOR XML AUTO

This is the result:

<Computed Col1="1" Col2="dbobject/Computed[@Col1='1']/(

See Also

Executing SQL Statements Using HTTP Executing Template Files Using HTTP SELECT XML and Internet Support

Using RAW Mode

RAW mode transforms each row in the query result set into an XML element with the generic identifier *row*. Each column value that is not NULL is mapped to an attribute of the XML element in which the attribute name is the same as the column name.

The BINARY BASE64 option must be specified in the query to return the binary data in base64-encoded format. In RAW mode, retrieving binary data without specifying the BINARY BASE64 option results in an error.

When an XML-Data schema is requested, the schema, declared as a namespace, appears at the beginning of the data. In the result, the schema namespace reference is repeated for every top-level element.

Examples

The queries in these examples can be executed using SQL Query Analyzer. To execute these queries using HTTP, see <u>Accessing SQL Server Using HTTP</u>.

A. Retrieve customer and order information using the RAW mode

This query returns customer and order information. RAW mode is specified in the FOR XML clause.

SELECT Customers.CustomerID, Orders.OrderID, Orders.OrderDate FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID ORDER BY Customers.CustomerID FOR XML RAW

This is the partial result:

<row CustomerID="ALFKI" OrderID="10643" OrderDate="1997-08-<row CustomerID="ANATR" OrderID="10308" OrderDate="1996-09 <row CustomerID="ANATR" OrderID="10625" OrderDate="1997-08 <row CustomerID="AROUT" OrderID="10355" OrderDate="1996-11

The same query can be specified using an outer join to return all customers in the result set, regardless of whether there are any orders for those customers.

SELECT C.CustomerID, O.OrderID, O.OrderDate FROM Customers C LEFT OUTER JOIN Orders O ON C.CustomerII ORDER BY C.CustomerID FOR XML RAW

This is the partial result:

<row CustomerID="BONAP" OrderID="11076" OrderDate="1998-05 <row CustomerID="FISSA"/> <row CustomerID="PARIS"/> <row CustomerID="RICSU" OrderID="11075" OrderDate="1998-05-

B. Specify the XMLDATA option to request XML-Data schema

This query returns the XML-DATA schema that describes the document structure:

SELECT TOP 2 Customers.CustomerID, Orders.OrderID, Orders.Orde FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID ORDER BY Customers.CustomerID FOR XML RAW , XMLDATA

This is the partial result:

<Schema name="Schema3" xmlns="urn:schemas-microsoft-com:xmlxmlns:dt="urn:schemas-microsoft-com:datatypes"> <ElementType name="row" content="empty" model="closed"> <AttributeType name="row" content="empty" model="closed"> <AttributeType name="closed"> <AttributeType name="CustomerID" dt:type="string"/> <AttributeType name="OrderID" dt:type="i4"/> <AttributeType name="OrderID" dt:type="i4"/> <AttributeType name="OrderDate" dt:type="dateTime"/> <attribute type="CustomerID"/><attribute type="OrderID"/>

```
<attribute type="OrderDate"/>
</ElementType>
</Schema>
<row xmlns="x-schema:#Schema3" CustomerID="ALFKI" OrderID='
OrderDate="1997-08-25T00:00:00"/>
<row xmlns="x-schema:#Schema3" CustomerID="ALFKI" OrderID='
OrderDate="1997-10-03T00:00:00"/>
```

Note The <Schema> is declared as a namespace. To avoid namespace collisions when multiple XML-Data schemas are requested in different FOR XML queries, the namespace identifier (**Schema3** in this example) changes with every query execution. The namespace identifier is made up of **Schema** followed by an integer.

C. Retrieve binary data

This query returns an employee photo from **Employees** table. Photo is an **image** column in the **Employees** table. The BINARY BASE64 option is specified in the query to return the binary data in base64-encoded format.

SELECT TOP 1 Photo FROM Employees WHERE EmployeeID=1 FOR XML RAW, BINARY BASE64

This is the result:

<row Photo="Binary data in base64 format"/>

D. Directly specify a URL to retrieve binary data

Because the RAW mode does not support addressing the binary data as URLs, this example creates a URL directly, using the DBOBJECT/TABLE[@PK1="v1"]/@COLUMN syntax. This returns a reference to an image data that can be used in subsequent operations.

SELECT TOP 1 EmployeeID,

'dbobject/Employees[@EmployeeID='+CAST(EmployeeID as nvarchar(4000))+']/@Photo' Photo FROM Employees FOR XML RAW

This is the result:

<row EmployeeID="3" Photo="dbobject/Employees[@EmployeeID3]/@Photo"/>

See Also

<u>SELECT</u>

XML and Internet Support

Using AUTO Mode

AUTO mode returns query results as nested XML elements. Each table in the FROM clause, from which at least one column is listed in the SELECT clause, is represented as an XML element. The columns listed in the SELECT clause are mapped to the appropriate attribute of the element. When the ELEMENTS option is specified, the table columns are mapped to subelements instead of attributes. By default, AUTO mode maps the table columns to XML attributes.

A table name (or the alias if provided) maps to the XML element name. A column name (or the alias if provided) maps to an attribute name or noncomplex subelement name when the ELEMENTS option is specified in the query.

The hierarchy (nesting of the elements) in the result set is based on the order of tables identified by the columns specified in the SELECT clause; therefore, the order in which column names are specified in the SELECT clause is significant.

The tables are identified and nested in the order in which the column names are listed in the SELECT clause. The first, leftmost table identified forms the top element in the resulting XML document. The second leftmost table (identified by columns in the SELECT statement) forms a subelement within the top element, and so on.

If a column name listed in the SELECT clause is from a table that is already identified by a previously specified column in the SELECT clause, the column is added as an attribute (or as a subelement if ELEMENTS option is specified) of the element already created, instead of opening a new level of hierarchy (adding a new subelement for that table).

For example, execute this query:

SELECT Customers.CustomerID, Orders.OrderID, Customers.Contact FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID FOR XML AUTO

This is the partial result:

```
<Customers CustomerID="ALFKI" ContactName="Maria Anders">
<Orders OrderID="10643"/>
<Orders OrderID="10692"/>
<Orders OrderID="10702"/>
<Orders OrderID="10835"/>
<Orders OrderID="10952"/>
<Orders OrderID="1091"/>
</Customers>
```

Note that in the SELECT clause, **CustomerID** identifies the **Customers** table. Therefore, a <Customers> element is created and **CustomerID** is added as its attribute. Next, the **OrderID** column name identifies the **Orders** table. An <Orders> element is added as a subelement of <Customers>, and the **OrderID** attribute is added to the <Orders> element. Now, the **ContactName** column identifies the **Customers** table, which was already identified by the **CustomerID** column. Therefore, no new element is created. Instead, **ContactName** attribute is added to the <Customers> element that is already created.

This query specifies the ELEMENT option. Therefore, an **element-centric** document is returned.

SELECT Customers.CustomerID, Orders.OrderID, Customers.Contact FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID FOR XML AUTO, ELEMENTS

This is the partial result:

```
<Customers>
<CustomerID>ALFKI</CustomerID>
<ContactName>Maria Anders</ContactName>
<Orders><OrderID>10643</OrderID></Orders>
<Orders><OrderID>10692</OrderID></Orders>
<Orders><OrderID>10702</OrderID></Orders>
<Orders><OrderID>10835</OrderID></Orders>
```

```
<Orders><OrderID>10952</OrderID></Orders>
<Orders><OrderID>11011</OrderID></Orders>
</Customers>
```

This query returns employee and order information. Again, the AUTO mode is specified in the FOR XML clause.

```
SELECT Employees.EmployeeID, LastName, FirstName,
OrderID, OrderDate, Orders.EmployeeID
FROM Orders, Employees
WHERE Orders.EmployeeID = Employees.EmployeeID
ORDER BY Employees.EmployeeID
FOR XML AUTO
```

The partial result is shown below. The table name appears as a tag for the XML element in the output. There is one <Employee> element for each value of **EmployeeID**.

<Employees EmployeeID="1" LastName="Davolio" FirstName="Nan <Orders OrderID="10258" OrderDate="1996-07-17T00:00:00" Empl <Orders OrderID="10270" OrderDate="1996-08-01T00:00:00" Empl </Employees>

<Employees EmployeeID="2" LastName="Fuller" FirstName="Andre <Orders OrderID="10248" OrderDate="1996-07-04T00:00:00" Empl <Orders OrderID="10249" OrderDate="1996-07-05T00:00:00" Empl </Employees>

Because the **Employees** table is identified before the **Orders** table in the SELECT clause, the <Employees> element appears as the outmost element in the resulting hierarchy that contains the <Orders> subelements.

In this example, comparing the **EmployeeID** values from one row to the next creates the <Employees> elements in the resulting XML document. This is done because **EmployeeID** is the primary key of the table. If **EmployeeID** is not identified as the primary key of the **Employees** table, all the column values from the **Employees** table specified in the SELECT statement (**EmployeeID**,

LastName, and **FirstName**) are compared from one row to the next. If any of the values differ from one row to the next, then a new <Employees> element is added in the result.

In comparing these column values, if any of the columns to be compared are of type **text**, **ntext**, or **image**, FOR XML assumes that values are different (although they may be the same because Microsoft® SQL Server[™] 2000 does not support comparing large objects); and elements are added to the result for each row selected.

When a column in the SELECT clause cannot be associated with any of the tables identified in the FROM clause (in case of an aggregate column or computed column), the column is added in the XML document in the deepest nesting level in place when it is encountered in the list. If such a column appears as the first column in the SELECT clause, the column is added to the top element.

If the * wildcard character is specified in the SELECT clause, the nesting is determined in the same way as described above (based on the order the rows are returned by the query engine).

The GROUP BY and aggregate functions are not supported in the AUTO mode. However, for a work around in which a nested SELECT is used to retrieve the information, see Example C that follows.

If BINARY BASE64 option is specified in the query, the binary data is returned in base64 encoding format. By default (if BINARY BASE64 option is not specified), the AUTO mode supports URL encoding of binary data. That is, instead of returning the binary data, a reference (a relative URL to the virtual root of the database where query is executed) is returned that can be used to access the actual binary data in subsequent operations. The query must provide enough information such as primary key columns to identify the image.

In a query specified against a table or view, if an alias is specified for the binary column of the view, the alias is returned in the URL encoding of the binary data. In subsequent operations, the alias is meaningless, and the URL encoding cannot be used to retrieve the image. Therefore, do not use aliases when querying a view using FOR XML AUTO mode.

When a view is created using a SELECT statement with TOP *n* option or DISTINCT option, the primary key information is lost. Therefore, if a query is

specified against this view to retrieve a binary column using OPEN XML AUTO mode, an error is returned. For example, if you create the following view:

CREATE VIEW MyView as SELECT TOP 2 * FROM Employees

This query generates an error because **MyView** does not have **EmployeeID** as its primary key:

SELECT EmployeeID, Photo FROM MyView WHERE EmployeeID = 1 FOR XML AUTO

If the same query is specified against the **Employees** table, you get these results:

SELECT EmployeeID, Photo FROM Employees WHERE EmployeeID = 1 FOR XML AUTO

Examples

The queries in these examples can be executed using SQL Query Analyzer. To execute these queries using HTTP, see <u>Accessing SQL Server Using HTTP</u>.

A. Retrieve employee and order information using the AUTO mode

This query returns employee and order information. AUTO mode is specified in the FOR XML clause:

SELECT Employees.EmployeeID, LastName, FirstName, OrderID, OrderDate, Orders.EmployeeID FROM Orders, Employees WHERE Orders.EmployeeID = Employees.EmployeeID ORDER BY Employees.EmployeeID FOR XML AUTO The partial result is shown below. The table name appears as a tag for the XML element in the output. There is one <Employee> element for each value of **EmployeeID**.

- <Employees EmployeeID="1" LastName="Davolio" FirstName="Nan <Orders OrderID="10258" OrderDate="1996-07-17T00:00:00" Empl <Orders OrderID="10270" OrderDate="1996-08-01T00:00:00" Empl </Employees>
- <Employees EmployeeID="2" LastName="Fuller" FirstName="Andre <Orders OrderID="10248" OrderDate="1996-07-04T00:00:00" Empl <Orders OrderID="10249" OrderDate="1996-07-05T00:00:00" Empl </Employees>

If the same query is specified in such a way that in the SELECT clause the column from the **Orders** table is specified before the columns in the **Employees** table, the hierarchy produced has the <Orders> element as top element and the <Employees> elements at the next level in the hierarchy.

SELECT OrderID, OrderDate, Orders.EmployeeID, Employees.EmployeeID, LastName, FirstName FROM Orders, Employees WHERE Orders.EmployeeID = Employees.EmployeeID ORDER BY Employees.EmployeeID FOR XML AUTO

This is the partial result:

<Orders OrderID="10258" OrderDate="1996-07-17T00:00:00" Emplo <Employees EmployeeID="1" LastName="Davolio" FirstName="Na </Orders>

<Orders OrderID="10270" OrderDate="1996-08-01T00:00:00" Emplo <Employees EmployeeID="1" LastName="Davolio" FirstName="Na </Orders>

In the following query, the **Orders** table is the leftmost table based on the columns specified in the SELECT statement. As a result the <Orders> elements

are created as top elements. The columns in the **Employees** table are specified next in the SELECT statement. The <Employees> element appears nested inside the <Orders> element. Finally, a column in the **Orders** table is specified in the SELECT statement. However, because the **Orders** table is already at the top level in the hierarchy, this column is added to that element, and no further elements are created.

SELECT OrderID, Orders.EmployeeID,

Employees.EmployeeID, LastName, FirstName, OrderDate FROM Orders, Employees WHERE Orders.EmployeeID = Employees.EmployeeID AND Employees.EmployeeID=1 or Employees.EmployeeID=2 ORDER BY Employees.EmployeeID FOR XML AUTO

This is the partial result:

<Orders OrderID="10258" EmployeeID="1" OrderDate="1996-07-17" <Employees EmployeeID="1" LastName="Davolio" FirstName="Na </Orders>

<Orders OrderID="10270" EmployeeID="1" OrderDate="1996-08-01" <Employees EmployeeID="1" LastName="Davolio" FirstName="Na </Orders>

B. Specify aliases for table names

This query returns customer and order information. Aliases are used for table names.

```
SELECT C.CustomerID, O.OrderID, O.OrderDate
FROM Customers C LEFT OUTER JOIN Orders O ON C.CustomerII
ORDER BY C.CustomerID
FOR XML AUTO
```

The partial result set is shown below. The element names are the same as the aliases specified for the tables used in the query.

```
<C CustomerID="ALFKI">
<O OrderID="10643" OrderDate="1997-08-25T00:00:00"/>
<O OrderID="10692" OrderDate="1997-10-03T00:00:00"/>
</C>
<C CustomerID="ANATR">
<O OrderID="10308" OrderDate="1996-09-18T00:00:00"/>
```

</C>

In a nested query, if an alias is specified in the inner query, depending on how the optimizer handles the query, the alias is not preserved. For example:

```
SELECT TOP 2 *
FROM (SELECT FirstName+' '+LastName as FullName FROM Empl-
FOR XML AUTO
```

The query produces this result:

```
<Employees FullName="Nancy Davolio"/>
<Employees FullName="Andrew Fuller"/>
```

In the result, the element name is <Employees> instead of <EMP>.

C. Specify GROUP BY and aggregate functions

The GROUP BY and aggregate functions are not currently supported with FOR XML AUTO mode. The following query uses a nested query approach to find the number of orders a customer has placed. This query returns customer information including the number of orders the customer has placed, the order information and the order detail information.

The inner SELECT statement produces a table with customer information along with the number of orders the customer has placed (GROUP BY and COUNT() function are used). This inner table is then joined with tables in FROM clause of the outer query where the FOR XML mode is specified.

SELECT Cust.CustomerID, ContactName, NoOfOrders, O.OrderID, O.CustomerID,

OD.ProductID, OD.Quantity

```
FROM (SELECT C.CustomerID, C.ContactName, count(*) as NoOfO
FROM Customers C left outer join
Orders O ON C.CustomerID = O.CustomerID
GROUP BY C.CustomerID, C.ContactName) Cust
left outer join Orders O on Cust.CustomerID = O.CustomerID
left outer join [Order Details] OD on O.OrderID = OD.OrderID
FOR XML AUTO
```

This is the partial result:

```
<Cust CustomerID="ALFKI" ContactName="Maria Anders" NoOfOr

<O OrderID="10643" CustomerID="ALFKI">

<OD ProductID="28" Quantity="15"/>

<OD ProductID="39" Quantity="21"/>

<OD ProductID="46" Quantity="2"/>

</O>

<O OrderID="10692" CustomerID="ALFKI">

<OD ProductID="63" Quantity="20"/>

</O>

</Cust>
```

D. Specify computed columns in the AUTO mode

This query returns concatenated employee names and the order ID of the orders the employee has taken. The computed column is assigned to the innermost level encountered at that point. The concatenated employee names are added as attributes of <Order> element in the result.

```
SELECT FirstName+' '+LastName as Name,
Orders.OrderID
FROM Employees left outer join Orders on
Employees.EmployeeID=Orders.EmployeeID
ORDER BY Name
FOR XML AUTO
```

This is the partial result:

```
<Orders Name="Andrew Fuller" OrderID="10265"/>
<Orders Name="Andrew Fuller" OrderID="10277"/>
```

To get the <Emp> elements with **Name** attribute containing the order subelements, the query is rewritten using a subselect. The innerselect creates a temporary **Emp** table with the computed column containing the names of the employees. This table is then joined with the **Orders** table to get the result.

```
SELECT Emp.name, Orders.OrderID
FROM (SELECT FirstName+' '+LastName as Name,
EmployeeID
FROM Employees) Emp
left outer join Orders on Emp.EmployeeID = Orders.EmployeeIE
ORDER BY Emp.Name
FOR XML AUTO
```

This is the partial result:

```
<Emp name="Andrew Fuller">
<Orders OrderID="10265"/>
<Orders OrderID="10277"/>
<Orders OrderID="10280"/>
</Emp>
```

E. Return binary data

This query returns an employee photo from the **Employees** table. **Photo** is an **image** column in the **Employees** table. The AUTO mode, by default, returns a reference (relative URL to the virtual root of the database where the query is executed) to the binary data. The **EmployeeID** key attribute must be specified to identify the image. In retrieving an image reference as in this example, the primary key of the table must also be specified in the SELECT clause to uniquely identify a row.

SELECT EmployeeID, Photo

FROM Employees WHERE EmployeeID=1 FOR XML AUTO

This is the result:

<Employees EmployeeID="1" Photo="dbobject/Employees[@Employ</pre>

The same query is executed with the BINARY BASE64 option. The query returns the binary data in base64-encoded format.

SELECT Photo FROM Employees WHERE EmployeeID=1 FOR XML AUTO, BINARY Base64

This is the result:

<Employees Photo="Here you see the Picture in base64 format"/>

In retrieving binary data using AUTO mode, a reference (a relative URL to the virtual root of the database where the query is executed), instead of the binary data, is returned by default (for example, BINARY BASE64 option is not specified). In case-insensitive databases, if the table or column name specified in the query does not match the table or column name in the database, the query executes; however, the case returned in the reference will not be consistent. For example:

SELECT TOP 2 PHOTO, EMPLOYEEID FROM EMPLOYEES FOR

This is the result:

<EMPLOYEES PHOTO="dbobject/EMPLOYEES[@EmployeeID='1' EMPLOYEEID="1"/> <EMPLOYEES PHOTO="dbobject/EMPLOYEES[@EmployeeID='2' EMPLOYEEID="2"/>

This could be a problem, especially if two templates request data from the same

table in a case-insensitive database but use queries with different cases. To avoid such a problem, it is recommended that the case of the table or column name specified in the queries match the case of table or column name in the database.

F. Understand the encoding

This example shows various encoding that takes place in the result.

- 1. Create this table: CREATE TABLE [Special Chars] (Col1 char(1) primary key,
- Add following data to the table: INSERT INTO [Special Chars] values ('&', 0x20) INSERT INTO [Special Chars] values ('#', 0x20)
- This query returns the data from the table. The FOR XML AUTO mode is specified. Binary data is returned as a reference.
 SELECT * FROM [Special Chars] FOR XML AUTO

This is the result:

```
<Special_x0020_Chars
Col1="#"
Col_x0023__x0026_2="dbobject/Special_x0020_Chars[@Col1='#']/@
/>
<Special_x0020_Chars
Col1="&"
Col_x0023__x0026_2="dbobject/Special_x0020_Chars[@Col1='&am
/>
```

This is the process for encoding special characters in the result:

• In the query result, the special XML and URL characters in the element and attribute names returned are encoded using the hexadecimal value of the corresponding Unicode character encoded. In the preceding result, the element name <Special Chars> is returned as <Special_x0020_Chars>. The attribute name <Col#&2> is returned as <Col_x0023__x0026_2> (both XML and URL special characters are encoded).

- If the values of the elements or attribute contain any of the five standard XML character entities (', '''', <, >, and &), these special XML characters are always encoded using XML character encoding. In the above result, the value & in the value of attribute <Col1> is encoded as &. However, the # character remains # because it is a valid XML character (not a special XML character).
- If the values of the elements or attributes contain any special URL characters that have special meaning in the URL, they are encoded only in the DBOBJECT URL value and encoded only when the special character is part of a table or column name. In the result, the character # that is part of table name **Col#&2** is encoded as _x0023_ in the DBOJBECT URL.

G. Specify the ELEMENTS option

This query returns customer and order information. The query specifies the ELEMENTS option. As a result, the table columns are mapped to subelements.

SELECT Customers.CustomerID, ContactName, OrderID, OrderDate FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID ORDER BY Customers.CustomerID FOR XML AUTO, ELEMENTS

This is the result:

```
<Customers>
<CustomerID>ALFKI</CustomerID>
<ContactName>Maria Anders</ContactName>
<Orders>
<OrderID>10835</OrderID>
```

```
<OrderDate>1998-01-15T00:00:00</OrderDate>
</Orders>
<Orders>
<OrderID>10952</OrderID>
<OrderDate>1998-03-16T00:00:00</OrderDate>
</Orders>
</Customers>
```

In the **element-centric** mapping, you can specify the same alias for different columns in the query. This results in multiple subelements with the same name (this is not allowed in attribute-centric mapping), for example:

SELECT FirstName name, LastName name FROM Employees FOR XML AUTO, EXPLICIT

This is the partial result:

```
<Employees>
<name>Nancy</name>
<name>Davolio</name>
</Employees>
```

•••

See Also

<u>SELECT</u>

Using EXPLICIT Mode

In an EXPLICIT mode, the query writer controls shape of the XML document returned by the execution of the query. The query must be written in a specific way so that the additional information about the expected nesting is explicitly specified as part of the query. You can also specify additional configurations at the column level using the **directives**. When you specify EXPLICIT mode, you must assume the responsibility for ensuring that the generated XML is well-formed and valid (in case of an XML-DATA schema).

Processing EXPLICIT Mode Queries and the Universal Table

The EXPLICIT mode transforms the rowset resulting from the query execution into an XML document. For the EXPLICIT mode to produce the XML document, the rowset must have certain format. This requires the SELECT query to be written in a certain way to produce the rowset with a specific format (called the universal table), which can then be processed to produce the requested XML document.

First the EXPLICIT mode requires the query to produce two meta data columns:

- The first column specified in the SELECT clause must be a named (**Tag**) tag number. The **Tag** column stores the tag number of the current element. **Tag** is an integer data type.
- The second column specified must be a named (**Parent**) tag number of the parent element. The **Parent** column stores the tag number of the parent element. **Tag** is an integer data type.

These columns are used to determine the parent-child hierarchy in the XML tree. This information is then used to produce the desired XML tree. If the parent tag value stored in **Parent** column is 0 or NULL, the row is placed on the top level of the XML hierarchy.

The remainder of the universal table fully describes the resulting XML document. An example of a universal table showing the nesting for the

Tag	Parent	Customer!1!cid	Customer!1!name	Order!2! id	Order!2!date	OrderDetail!3!id!id	OrderDetail!3!pid!idref
1	NULL	C1	"Janine"	NULL	NULL	NULL	NULL
2	1	C1	NULL	01	1/20/1996	NULL	NULL
3	2	C1	NULL	01	NULL	OD1	P1
3	2	C1	NULL	01	NULL	OD2	P2
2	1	C1	NULL	02	3/29/1997	NULL	NULL

<Customer>, <Order>, and <OrderDetail> elements is shown in this illustration.

The data in the rowset (universal table) is partitioned vertically into groups. Each group becomes an XML element in the result.

A query that generates this sample universal table will produce the following XML document in the EXPLICIT mode (one of the examples below describes the query). Only the partial output is shown:

```
<Customer cid="C1" name="Janine">
<Order id="O1" date="1/20/1996">
<OrderDetail id="OD1" pid="P1"/>
<OrderDetail id="OD2" pid="P2"/>
</Order>
<Order id="O2" date="3/29/1997">
....
</Customer>
```

The FOR XML EXPLICIT mode requires that the SELECT query specify the column names in the universal table in a certain way. It requires that the SELECT query associate the element names with the tag numbers and provide the property names (attribute names by default) in the column names of the universal table. In addition, to get the correct children instances associated with their parent, the rowset needs to be ordered such that the parent is followed immediately by its children.

To summarize, the information provided in the column names of the universal table, the values in the **Tag** and **Parent** meta columns, and the data in the universal table format are used to generate the desired XML document in the EXPLICIT mode.

Specifying Column Names in a Universal Table

The SELECT query must specify the column names in a universal table. The column names in the universal table are encoded using XML generic identifiers and attribute names. The encoding of the element name, the attribute names, and other transformation information in the column name in the universal table are specified as:

ElementName!TagNumber!AttributeName!Directive

Arguments

ElementName

Is the resulting generic identifier of the element (for example, if **Customers** is specified as *ElementName*, then <Customers> is the element tag).

TagNumber

Is the tag number of the element. *TagNumber*, with the help of the two meta data columns (**Tag** and **Parent**) in the universal table, is used to express the nesting of XML elements in the XML tree. Every *TagNumber* correspond to exactly one *ElementName*.

AttributeName

Is either the name of the XML attribute (if *Directive* is not specified) or the name of the contained element (if *Directive* is either **xml**, **cdata**, or **element**). If *Directive* is specified, *AttributeName* can be empty. In this case, the value contained in the column is directly contained by the element with the specified *ElementName*.

Directive

Is an optional directive. If *Directive* is not specified, *AttributeName* must be specified. If *AttributeName* is not specified and *Directive* is not specified (for example, **Customer!1**), an element directive is implied (for example, **Customer!1!!element**), and data is contained.

Directive has two purposes. This option is used to encode ID, IDREF, and IDREFS by using the keywords **ID**, **IDREF**, and **IDREFS**. It is also used to indicate how to map the string data to XML using the keywords **hide**,
element, **xml**, **xmltext**, and **cdata**. Combining directives between these two groups is allowed in most of the cases, but not combining among themselves.

ID

An element attribute can be specified to be an **ID** type attribute. **IDREF** and **IDREFS** attributes can then be used to refer to them, enabling intradocument links. If XMLDATA is not requested, this keyword has no effect.

IDREF

Attributes specified as **IDREF** can be used to refer to **ID** type attributes, enabling intradocument links. If XMLDATA is not requested, this keyword has no effect.

IDREFS

Attributes specified as **IDREFS** can be used to refer to **ID** type attributes, enabling intradocument links. If XMLDATA is not requested, this keyword has no effect.

hide

The attribute is not displayed. This may be useful for ordering the result by an attribute that will not appear in the result.

element

This does not generate an attribute. Instead it generates a contained element with the specified name (or generate contained element directly if no attribute name is specified). The contained data is encoded as an entity (for example, the < character becomes **<**). This keyword can be combined with **ID**, **IDREF**, or **IDREFS**.

xml

This is the same as an element directive except that no entity encoding takes place (for example, the < character remains <). This directive is not allowed with any other directive except **hide**.

xmltext

The column content should be wrapped in a single tag that will be integrated with the rest of the document. This directive is useful in

fetching overflow (unconsumed) XML data stored in a column by OPENXML. For more information, see <u>Writing XML Using OPENXML</u>.

If *AttributeName* is specified, the tag name is replaced by the specified name; otherwise, the attribute is appended to the current list of attributes of the enclosing elements and by putting the content at the beginning of the containment without entity encoding. The column with this directive must be a text type (**varchar, nvarchar, char, nchar, text, ntext**). This directive can be used only with **hide**. This directive is useful in fetching overflow data stored in a column.

If the content is not a well-formed XML, the behavior is undefined.

cdata

Contains the data by wrapping it with a CDATA section. The content is not entity encoded. The original data type must be a text type (**varchar**, **nvarchar**, **text**, **ntext**). This directive can be used only with **hide**. When this directive is used, *AttributeName* must not be specified.

Examples

The queries in these examples can be executed using SQL Query Analyzer. To execute these queries using HTTP, see <u>Accessing SQL Server Using HTTP</u>.

The process for writing queries using EXPLICIT mode is explained in detail in Examples A and B. This process applies to the other examples that follow.

A. Retrieve customer and order information

This example retrieves customer and order information. Assume you want the following hierarchy generated:

```
<Customer CustomerID="ALFKI">
```

```
<Order OrderID=10643>
<Order OrderID=10692>
```

```
•••
```

```
</Customer>
<Customer CustomerID="ANATR" >
<Order OrderID=10308 >
<Order OrderID=10625 >
...
```

</Customer>

The universal table produced by the query from which the resulting XML tree is produced contains two meta data columns: **Tag** and **Parent**. Therefore, in specifying the query the SELECT clause must specify these columns. The values in these columns are used in generating the XML hierarchy.

The <Customer> element is at the top level. In this example, this element is assigned a **Tag** value of 1 (this can be any number, but there is unique number associated with each element name). Because <Customer> is a top-level element, its **Parent** tag value is NULL.

The <Order> element is a child of the <Customer> element. Therefore, the **Parent** tag value for <Order> element is 1 (identifying <Customer> as its parent element). The <Order> element is assigned a **Tag** value of 2.

You can write a query with two SELECT statements and use UNION ALL to combine the results of the statements:

• In the first SELECT statement in the query, all the <Customer> elements and their attribute values are obtained. In a query with multiple SELECT statements, only the column names (universal table column names) that are specified in the first query are used. The column names specified in the subsequent SELECT statements are ignored. Therefore, the column names for the universal table that specify the XML element and attribute names are included in this query:

SELECT 1	as Tag,
NULL	as Parent,
Customers.Cus	tomerID as [Customer!1!CustomerID],
NULL	as [Order!2!OrderID]
FROM Customers	

• In the second query, all <Order> elements and their attribute values are retrieved:

SELECT 2,

1, Customers.CustomerID, Orders.OrderID FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID

- The two SELECT statements in the query are combined with a UNION ALL.
- The universal table rowset (containing all data and meta data) is scanned one row at a time, in a forward-only manner, producing the resulting XML tree. To yield the proper XML document hierarchy, it is also important to specify the order of rows in the universal table. This is achieved by using the ORDER BY clause in the query.
- This is the final query:

SELECT 1 as Tag, NULL as Parent, Customers.CustomerID as [Customer!1!CustomerID], NULL as [Order!2!OrderID] FROM Customers

```
UNION ALL
SELECT 2,
1,
Customers.CustomerID,
Orders.OrderID
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
ORDER BY [Customer!1!CustomerID], [Order!2!OrderID]
```

FOR XML EXPLICIT

The resulting universal table is a four-column table. For illustration purposes, only a few rows are shown.

Tag	Parent	Customer!1!CustomerID	Order!2!OrderID
1	NULL	ALFKI	NULL
2	1	ALFKI	10643
2	1	ALFKI	10692
2	1	ALFKI	10702
2	1	ALFKI	11011
2	1	ALFKI	•••
1	NULL	ANATR	NULL
2	1	ANATR	10308
2	1	ANATR	10625
2	1	ANATR	•••

The processing of the rows in the universal table to produce the resulting XML tree is described here:

- The first row identifies Tag value 1. All columns with the Tag value 1 are identified. In this case, there is only one column:
 Customer!1!CustomerID. This column name is composed of element name (Customer), tag number (1), and attribute name (CustomerID). Therefore, a <Customer> element is created, and an attribute
 CustomerID is added to it. The column value is then assigned as the attribute value.
- 2. The second row has the Tag value 2. Therefore, all columns with the Tag value 2 are identified. There is only one column with the Tag value 2: Order!2!OrderID. The column name is composed of element name (Order), tag number (2) and attribute name (OrderID). This row also identifies <Customer> as its parent (Parent value is 1). As a result, an <Order> element is created as a child of the <Customer> element and an attribute OrderID is added to it. The column value is

then assigned as the attribute value.

- 3. All the subsequent rows with **Tag** value 2 are processed in the same manner.
- 4. A row with Tag value 1 is identified. It identifies the Customer!1!CustomerID column with the Tag value 1. This column identifies a <Customer> element with no parent (Parent is NULL). Thus, both the previous <Order> tag and the previous <Customer> tag are closed. A new <Customer> tag is opened, and the process is repeated.

Because *Directive* is not specified in the query, the attribute name is the name of the XML attribute. This is the partial result set:

```
<Customer CustomerID="ALFKI">
<Order OrderID="10643" />
<Order OrderID="10692" />
<Order OrderID="10702" />
<Order OrderID="11011" />
</Customer>
<Customer CustomerID="ANATR">
<Order OrderID="10308" />
<Order OrderID="10625" />
</Customer>
```

B. Specify the element directive

This example retrieves the customer and order information. Assume you want the following hierarchy generated: (note that <OrderID> is a subelement of <Order> and not an attribute):

```
<Customer CustomerID="ALFKI">
<Order OrderDate="1997-08-25T00:00:00">
<OrderID>10643</OrderID>
</Order>
```

```
<Order OrderDate="1997-10-03T00:00:00">
<OrderID>10692</OrderID>
</Order>
...
</Customer>
```

~/Customer/

The <Customer> element is at the top level. In this example, it is assigned a **Tag** value of 1. Because <Customer> is a top-level element, its **Parent** tag value is NULL.

The <Order> element is a child of <Customer> element. Therefore, the **Parent** tag value for <Order> element is 1 (identifying <Customer> as its parent element) and it is assigned a **Tag** value of 2.

The <Order> element has <OrderID> as a contained element (not an attribute). Therefore, in retrieving this value, the **element** directive must be specified.

You can write a query with two SELECT statements and use a UNION ALL to combine the results of the statements:

• In the first SELECT statement in the query, all the <Customer> elements and their attribute values are obtained. In a query with multiple SELECT statements, only the column names (universal table column names) that are specified in the first query are used. The column names specified in the subsequent SELECT statements are ignored. Therefore, the column names for the universal table that specify the XML element and attribute names are included in this query:

SELECT 1	as Tag,	
NULL	as Parent,	
Customers.C	CustomerID as [Customer!1!CustomerID],
NULL	as [Order!2!OrderID!element],	
NULL	as [Order!2!OrderDate]	

FROM Customers

• In the second query, all <Order> elements and their attribute values are retrieved. This query selects **Customers.CustomerID** because of the required grouping of parent with children using ORDER BY clause. SELECT 2, 1,

Customers.CustomerID, Orders.OrderID, Orders.OrderDate FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID

- The two SELECT statements in the query are combined with a UNION ALL.
- The ORDER BY clause is used to specify the order of the rows in the universal table rowset that is generated.

• This is the final query: **SELECT 1** as Tag, NULL as Parent, Customers.CustomerID as [Customer!1!CustomerID], as [Order!2!OrderID!element], NULL NULL as [Order!2!OrderDate] FROM Customers UNION ALL SELECT 2, 1, Customers.CustomerID, Orders.OrderID, Orders.OrderDate FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID ORDER BY [Customer!1!CustomerID], [Order!2!OrderID!ele] FOR XML EXPLICIT

The resulting universal table is a five-column table. For illustration purposes, only a few rows are shown.

Tag	Parent	Customer!1!CustomerID	Order!2!OrderID!element	Order!2!Ord
1	NULL	ALFKI	NULL	NULL
2	1	ALFKI	10692	1997-10- 03T00:00:00
2	1	ALFKI	10702	1997-10- 13T00:00:00
2	1	ALFKI	10835	1998-01- 15T00:00:00
•••		•••	•••	•••
1	NULL	ANATR	10308	1996-09- 18T00:00:00
1	NULL	ANATR		•••

The processing of the rows in the rowset to produce the resulting XML tree is described here:

- The first row identifies Tag value 1. Therefore, all the columns with Tag value 1 are identified. In this case there is only one column: Customer!1!CustomerID column. This column name is composed of element name (Customer), tag number (1) and attribute name (CustomerID). Therefore, a <Customer> element is created and an attribute CustomerID is added to it. The column value is then assigned as the attribute value.
- The second row has Tag value 2. All the columns with Tag value 2 are identified. There are two columns (Order!2!OrderID!element and Order!2!OrderDate) with the tag number 2.
 - Column Order!2!OrderDate is composed of element name (Order), tag number (2) and the attribute name (OrderDate). This row identifies <Customer> as its parent (Parent value is 1). Therefore, an <Order> element is created as a child of the <Customer> element, and an attribute OrderID is added to it. The column value is assigned as the attribute value.

- The column name, **Order!2!OrderID!element** consists of the directive (**element**). Therefore, a contained element (<OrderID>) is generated. The column value is assigned as the element value.
- 3. All the subsequent rows with **Tag** value 2 are processed in the same manner.
- 4. A row with Tag value 1 is identified. It identifies Customer!1!CustomerID column with Tag value 1. This column identifies a <Customer> element with no parent (Parent is NULL). Therefore, both the previous <Order> tag and the previous <Customer> tag are closed. A new <Customer> tag is opened, and the process is repeated.

Note In the query, if the column name (**Order!2!OrderID!element**) is changed so that the attribute name is not specified (**Order!2!!element**), the query generates the contained element directly.

C. Specify the element directive and the entity encoding

If the directive is set to **element**, the contained data is entity encoded. For example, if one of the customer contact names in the **Customers** table is **Mar<ia Anders**, the following query encodes the contained data:

```
--Update customer record.
UPDATE Customers
SET ContactName='Mar<ia Anders'
WHERE ContactName='Maria Anders'
GO
```

The following query returns the customer ID and contact name information.

The process of writing the query to produce the universal table and the processing of the universal table rowset to produce the resulting XML document

is similar to the process described in Example A and Example B.

SELECT 1 as Tag, NULL as Parent, Customers.CustomerID as [Customer!1!CustomerID], Customers.ContactName as [Customer!1!ContactName!element] FROM Customers ORDER BY [Customer!1!CustomerID] FOR XML EXPLICIT GO -- set the value back to original UPDATE Customers SET ContactName='Maria Anders' WHERE ContactName='Mar<ia Anders' GO

The partial result is shown below. Because the **element** directive is specified in the query, the attribute name specified is the name of the contained element. Also the **ContactName** is entity encoded (the < character in the **ContactName** is returned as **<**)

```
<Customer CustomerID="ALFKI">
<ContactName>Mar&lt;ia Anders</ContactName>
</Customer>
<Customer CustomerID="ANATR">
<ContactName>Ana Trujillo</ContactName>
</Customer>
```

D. Specify the xml directive

The **xml** directive is similar to **element** directive except that the contained data is not entity encoded (the < character remains <). For example, if one of the customer contact names in the **Customers** table is **Mar**<**ia Andears**, the following query does not entity encode the contained data and generates an XML document that is not well-formed.

-- Update a customer record.

```
UPDATE Customers
SET ContactName='Mar<ia Anders'
WHERE ContactName='Maria Anders'
GO
```

The following query returns the customer ID and contact name information.

The process of writing the query to produce the universal table and the processing of the universal table rowset to produce the resulting XML document is similar to the process described in Example A and Example B.

```
SELECT 1 as Tag, NULL as Parent,
Customers.CustomerID as [Customer!1!CustomerID],
Customers.ContactName as [Customer!1!ContactName!xml]
FROM Customers
ORDER BY [Customer!1!CustomerID]
FOR XML EXPLICIT
GO
-- Set customer record back to the original.
UPDATE Customers
SET ContactName='Maria Anders'
WHERE ContactName='Mar<ia Anders'
GO
```

The partial result is shown below. Because the directive is specified in the query, the attribute name specified is the name of the contained element.

```
<Customer CustomerID="ALFKI">
<ContactName>Mar<ia Anders</ContactName>
</Customer>
<Customer CustomerID="ANATR">
<ContactName>Ana Trujillo</ContactName>
</Customer>
```

E. Specify the hide directive

This example shows the use of the **hide** directive. This directive is useful when you want the query to return an attribute for ordering the rows in the universal table returned by the query but you do not want that attribute in the final resulting XML document.

Assume you want the following hierarchy generated where the <Customer> elements are ordered by CustomerID, and within each <Customer> element, the <Order> elements are sorted by OrderID. Note that the **OrderID** attribute is not in the resulting XML document:

<Customer CustomerID="ALFKI"> <Order OrderDate="1997-08-25T00:00:00" /> <Order OrderDate="1997-10-03T00:00:00" /> <Order OrderDate="1997-10-13T00:00:00" /> </Customer>

In this case, the **OrderID** is retrieved for ordering purposes but in specifying the column name for this attribute, the **hide** directive is specified. As a result the **OrderID** attribute is not displayed as part of the resulting XML document.

The process of writing the query to produce the universal table and the processing of the universal table rowset to produce the resulting XML document is similar to the process described in Example A and Example B.

This is the query:

```
SELECT 1 as Tag,
NULL as Parent,
Customers.CustomerID as [Customer!1!CustomerID],
NULL as [Order!2!OrderID!hide],
NULL as [Order!2!OrderDate]
FROM Customers
UNION ALL
SELECT 2,
1,
Customers.CustomerID,
Orders.OrderID,
```

Orders.OrderDate FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID ORDER BY [Customer!1!CustomerID], [Order!2!OrderID!hide] FOR XML EXPLICIT

This is the partial result. The **OrderID** attribute is not in the resulting document.

```
<Customer CustomerID="ALFKI">
<Order OrderDate="1997-08-25T00:00:00" />
<Order OrderDate="1997-10-03T00:00:00" />
<Order OrderDate="1997-10-13T00:00:00" />
</Customer>
<Order CustomerID="ANATR">
<Order OrderDate="1996-09-18T00:00:00" />
<Order OrderDate="1997-08-08T00:00:00" />
</Customer>
```

F. Specify the cdata directive

If the directive is set to **cdata**, the contained data is not entity encoded but is put in the CDATA section. The **cdata** attributes must be nameless.

The following query wraps the contact names in the CDATA sections. The process of writing the query to produce the universal table and the processing of the universal table rowset to produce the resulting XML document is similar to the process described in Example A and Example B.

SELECT 1	as Ta	ig,
NULL	as Pare	ent,
Customers.	CustomerID	as [Customer!1!CustomerID],
Customers.	ContactName	e as [Customer!1!!cdata]
FROM Custom	ers	
ORDER BY [C	ustomer!1!Cu	lstomerID]
FOR XML EXI	PLICIT	

The partial result is shown below. The contained data is wrapped in the CDATA section, and the contained data is not entity encoded (the contact name remains **Mar<ia Ande!rs**).

```
<Customer CustomerID="ALFKI">
<![CDATA[Maria Anders]]>
</Customer>
<Customer CustomerID="ANATR">
<![CDATA[Ana Trujillo]]>
</Customer>
```

G. Specify the ID and IDREF directives

In an XML document, an element attribute can be specified as an **ID** type attribute and the **IDREF** attributes in the document can then be used to refer to them, thereby enabling intradocument links (similar to the primary key and foreign key relationship in relational databases).

The query in this example returns an XML document that consists of the **ID** and **IDREF** attributes. The example retrieves customer and order information. The query is to return this XML document:

```
<Customer CustomerID="ALFKI">
<Order CustomerID="ALFKI" OrderDate="1997-08-25T00:00:00">
<OrderID>10643</OrderID>
</Order>
<Order CustomerID="ALFKI" OrderDate="1997-10-03T00:00:00">
<Order CustomerID="ALFKI" OrderDate="1997-10-03T00:00:00">
<OrderID>10692</OrderID>
</Order>
</Customer>
```

•••

Assume also that the **CustomerID** attribute of the <Customer> element is to be of **ID** type and the **CustomerID** attribute of <Order> element is to be an **IDREF** type. Because an order can belong to only one customer, an **IDREF** is specified.

The process of writing the query to produce the universal table and the

processing of the universal table rowset to produce the resulting XML document is similar to the process described in Example A and Example B. The only addition to the query is that the directives (**ID** and **IDREF**) are specified as part of the columns.

This is the query:

SELECT 1	as Tag,	
NULL	as Parent,	
Customers.Custo	omerID as [Customer!1!CustomerID!id],	
NULL	as [Order!2!OrderID!element],	
NULL	as [Order!2!CustomerID!idref],	
NULL	as [Order!2!OrderDate]	
FROM Customers		
UNION ALL		
SELECT 2,		
1,		
Customers.CustomerID,		
Orders.OrderID, Orders.CustomerID,		
Orders.OrderDate		
FROM Customers, Orders		
WHERE Customers.CustomerID = Orders.CustomerID ORDER BY [Customer!1!CustomerID!id], [Order!2!OrderID!element FOR XML EXPLICIT_XML DATA		
$1 \cup 1 \setminus 1 \cup $		

The **ID** or **IDREF** directives specified in the query mark the elements/attributes in the XML-Data schema. In the query:

- **ID** directive is specified as part of the universal table column name (**Customer!1!CustomerID!id**). This directive makes the **CustomerID** attribute of the <Customer> element in the returned XML documents an **ID** type attribute. In the XML-Data schema the **dt:type** value is **ID** in the **AttributeType** declaration.
- **IDREF** directive is specified as part of the universal table column name (**Order!2!OrderID!idref**). This directive makes the **OrderID** attribute

of the <Order> element in the returned XML documents an **IDREF** type attribute. In the XML-Data schema the **dt:type** value is **IDREF** in the **AttributeType** declaration.

You can obtain the XML-Data schema by specifying the XMLDATA schema option in the query. Note that the **ID** and **IDREF** directives specified in the query overwrite the data types in the XML-Data schema.

This is the partial result. Because the XMLDATA schema option is specified in the query, the schema is prepended to the result.

```
<Schema name="Schema1" xmlns="urn:schemas-microsoft-com:xml-</pre>
<ElementType name="Customer" content="mixed" model="open">
 <AttributeType name="CustomerID" dt:type="id"/>
 <attribute type="CustomerID"/>
</ElementType>
<ElementType name="Order" content="mixed" model="open">
 <AttributeType name="CustomerID" dt:type="idref"/>
 <AttributeType name="OrderDate" dt:type="dateTime"/>
 <element type="OrderID"/>
 <attribute type="CustomerID"/>
 <attribute type="OrderDate"/>
</ElementType>
<ElementType name="OrderID" content="textOnly" model="closed"
       dt:type="i4"/>
</Schema>
<Customer xmlns="x-schema:#Schema1" CustomerID="ALFKI">
 <Order CustomerID="ALFKI" OrderDate="1997-08-25T00:00:00">
  <OrderID>10643</OrderID>
 </Order>
 <Order CustomerID="ALFKI" OrderDate="1997-10-03T00:00:00">
  <OrderID>10692</OrderID>
 </Order>
</Customer>
```

H. Specify the ID and IDREFS attributes

An element attribute can be specified as an **ID** type attribute, and the **IDREFS** attribute can then be used to refer to it, thereby enabling intradocument links (similar to the primary key, foreign key relationships in relational databases).

This example shows how the **ID** and **IDREFS** directives can be specified as part of the column names in a query to create XML attributes of **ID** and **IDREFS** types. Because IDs cannot be integer values, the ID values in this example are converted (type casted); prefixes are used for the ID values.

In the ORDER BY clause, the customer name is specified as the ordering attribute to show that attributes that are not **ID** can be used to sort the result.

Assume these tables exist in the database:

```
-- Create Customers2 table.
CREATE TABLE Customers2 (CustomerID int primary key,
       CustomerName varchar(50))
GO
-- Insert records in Customers2 table.
INSERT INTO Customers2 values (1, 'Joe')
INSERT INTO Customers2 values (2, 'Bob')
INSERT INTO Customers2 values (3, 'Mary')
Go
-- Create Orders2 table.
CREATE TABLE Orders2 (OrderID int primary key,
       CustomerID int references Customers2)
GO
-- Insert records in Orders2 table.
INSERT INTO Orders2 values (5, 3)
INSERT INTO Orders2 values (6, 1)
INSERT INTO Orders2 values (9, 1)
INSERT INTO Orders2 values (3, 1)
INSERT INTO Orders2 values (8, 2)
INSERT INTO Orders2 values (7, 2)
GO
```

Assume a query is to return an XML document with this hierarchy:

```
<Cust CustID="1" CustName="Joe" OrderIDList="O-3 O-6 O-9">
<Order Oid="O-3"/>
<Order Oid="O-6"/>
<Order Oid="O-9"/>
</Cust>
<Cust CustID="2" CustName="Bob" OrderIDList="O-7 O-8">
<Order Oid="O-7"/>
<Order Oid="O-7"/>
</Cust>
</Cust>
</Cust CustID="3" CustName="Mary" OrderIDList="O-5">
</Cust>
```

The **OrderIDList** attribute of the <Cust> element is a multivalued attribute referring to the **Oid** attribute of <Order> element. To establish this link, the **Oid** attribute must be declared of **ID** type, and the **OrderIDList** attribute of the <Cust> element must be declared of **IDREFS** type. Because a customer can place many orders, **IDREFS** type is used.

The process of writing the query to produce the universal table and the processing of the universal table rowset to produce the resulting XML document is similar to the process described in Example A and Example B. The only addition to the query is that the directives (**ID** and **IDREFS**) are specified as part of the columns.

This is the query:

-- Generate Customer element without IDREFS attribute.

```
SELECT 1 AS tag,

NULL AS parent,

CustomerID AS "Cust!1!CustID",

CustomerName AS "Cust!1!CustName",

NULL AS "Cust!1!OrderIDList!idrefs",

NULL AS "Order!2!Oid!id"

FROM Customers2
```

UNION ALL

-- Now add the IDREFS. Note that Customers2.CustomerName

-- is repeated because it is listed in the ORDER BY clause

-- (otherwise, NULL would suffice).

SELECT 1 AS tag,

NULL AS parent,

Customers2.CustomerID,

Customers2.CustomerName,

'O-'+CAST(Orders2.OrderID as varchar(5)),

NULL

FROM Customers2 join Orders2 on Customers2.CustomerID = Order

UNION ALL

- -- Now add the subelements (Orders2).
- -- Customers2.CustomerID is repeated because it is the parent key.
- -- Customers2.CustomerName is repeated because it is listed
- -- in the ORDER BY clause.

SELECT 2 AS tag,

1 AS parent,

Customers2.CustomerID,

Customers2.CustomerName,

NULL,

'O-'+CAST(Orders2.OrderID as varchar(5))

FROM Customers2 JOIN Orders2 ON Customers2.CustomerID = Ord

-- Now order by name and by key. No order on the last column -- is required because the key of Orders2 is not a parent. ORDER BY "Cust!1!CustID", "Order!2!Oid!id", "Cust!1!OrderIDList FOR XML EXPLICIT, XMLDATA

The **ID** or **IDREFS** directives specified in the query mark the elements/attributes in the XML-Data schema. In the query:

• **ID** directive is specified as part of the universal table column name

(**Order!2!Oid!id**). The directive makes the **Oid** attribute of the <Order> element in the returned XML documents an **ID** type attribute. In the XML-Data schema the **dt:type** value is **ID** in the **AttributeType** declaration.

• **IDREF** directive is specified as part of the universal table column name (**Cust!1!OrderIDList!idrefs**). The directive makes the **OrderIDList** attribute of the <Cust> element in the returned XML documents an **IDREF** type attribute. In the XML-Data schema the **dt:type** value is IDREFS in the **AttributeType** declaration.

You can obtain the XML-Data schema by specifying the XMLDATA option in the query. Note that the **ID** and **IDREFS** directives specified in the query overwrite the data types in the XML-Data schema.

This is the result:

```
<Schema name="Schema8" xmlns="urn:schemas-microsoft-com:xml-
xmlns:dt="urn:schemas-microsoft-com:datatypes">
<ElementType name="Cust" content="mixed" model="open">
 <AttributeType name="CustID" dt:type="i4"/>
 <AttributeType name="CustName" dt:type="string"/>
 <AttributeType name="OrderIDList" dt:type="idrefs"/>
 <attribute type="CustID"/>
 <attribute type="CustName"/>
 <attribute type="OrderIDList"/>
</ElementType>
<ElementType name="Order" content="mixed" model="open">
 <AttributeType name="Oid" dt:type="id"/>
 <attribute type="Oid"/>
</ElementType>
</Schema>
<Cust xmlns="x-schema:#Schema8" CustID="1" CustName="Joe"
   OrderIDList="O-3 O-6 O-9">
 <Order Oid="O-3"/>
```

```
<Order Oid="O-6"/>
<Order Oid="O-9"/>
</Cust>
<Cust xmlns="x-schema:#Schema8" CustID="2" CustName="Bob"
OrderIDList="O-7 O-8">
<Order Oid="O-7"/>
<Order Oid="O-7"/>
<Order Oid="O-8"/>
</Cust>
<Cust xmlns="x-schema:#Schema8" CustID="3" CustName="Mary"
OrderIDList="O-5">
<Order Oid="O-5"/>
</Cust>
```

I. Specify the xmltext directive

This example shows how data in the overflow column is addressed using the **xmltext** directive in a SELECT statement using EXPLICIT mode.

Consider the **Person** table. This table has the **Overflow** column that stores unconsumed part of XML document.

CREATE TABLE Person(PersonID varchar(5), PersonName varchar(2 INSERT INTO Person VALUES ('P1','Joe',N'<SomeTag attr1="data"> INSERT INTO Person VALUES ('P2','Joe',N'<SomeTag attr2="data"/> INSERT INTO Person VALUES ('P3','Joe',N'<SomeTag attr3="data" F

This query retrieves columns from the **Person** table. For the **Overflow** column, *AttributeName* is not specified, but *directive* is set to **xmltext** as part of providing universal table column name.

SELECT 1 as Tag, NULL as parent, PersonID as [Parent!1!PersonID], PersonName as [Parent!1!PersonName], overflow as [Parent!1!!xmltext] -- No AttributeName; xmltext direct FROM Person FOR XML EXPLICIT Because *AttributeName* is not specified for the **Overflow** column and the **xmltext** directive is specified, in the resulting XML document the attributes in the <overflow> element are appended to the attribute list of the enclosing <Parent> element, and because the **PersonID** attribute in the <xmltext> element conflicts with the **PersonID** attribute retrieved on the same element level, the attribute in the <xmltext> element is ignored (even if **PersonID** is NULL). Generally, an attribute overrides an attribute of the same name in the overflow.

This is the result:

```
<Parent PersonID="P1" PersonName="Joe" attr1="data">
content</Parent>
<Parent PersonID="P2" PersonName="Joe" attr2="data">
</Parent>
<Parent PersonID="P3" PersonName="Joe" attr3="data">
content</Parent>
```

If the overflow data had subelements and the same query is specified, the subelements in the **Overflow** column are added as the subelements of the enclosing <Parent> element.

For example, change the data in the **Person** table so that the **Overflow** column now has subelements:

```
TRUNCATE TABLE Person
INSERT INTO Person VALUES ('P1','Joe',N'<SomeTag attr1="data">
INSERT INTO Person VALUES ('P2','Joe',N'<SomeTag attr2="data"/>
INSERT INTO Person VALUES ('P3','Joe',N'<SomeTag attr3="data" F
```

If the same query is executed, the subelements in the <xmltext> element are added as subelements of the enclosing <Parent> element.

SELECT 1 as Tag, NULL as parent, PersonID as [Parent!1!PersonID], PersonName as [Parent!1!PersonName], overflow as [Parent!1!!xmltext] -- no AttributeName, xmltext directi FROM Person FOR XML EXPLICIT This is the result:

```
<Parent PersonID="P1" PersonName="Joe" attr1="data">
content</Parent>
<Parent PersonID="P2" PersonName="Joe" attr2="data">
</Parent>
<Parent PersonID="P3" PersonName="Joe" attr3="data">
<name>content</name></Parent>
```

If *AttributeName* is specified with the **xmltext** directive, the attributes of the <overflow> element are added as attributes of the subelements of the enclosing <Parent> element. The name specified for *AttributeName* becomes the name of the subelement

In this query, *AttributeName* (<overflow>) is specified along with the **xmltext** directive.

```
SELECT 1 as Tag, NULL as parent,
PersonID as [Parent!1!PersonID],
PersonName as [Parent!1!PersonName],
overflow as [Parent!1!overflow!xmltext] -- overflow is AttributeNan
-- xmltext is directive
```

FROM Person FOR XML EXPLICIT

This is the result:

```
<Parent PersonID="P1" PersonName="Joe">
<overflow attr1="data">
content</overflow>
</Parent>
<Parent PersonID="P2" PersonName="Joe">
<overflow attr2="data"/>
</Parent>
<Parent PersonID="P3" PersonName="Joe">
```

```
<overflow attr3="data" PersonID="P">
<name>content</name></overflow>
</Parent>
```

In this query element, *directive* is specified for **PersonName** attribute. This results in **PersonName** added as subelement of the enclosing <Parent> element. The attributes of the <xmltext> are still appended to the enclosing <Parent> element. The contents of <overflow> element (subelements and so on) are prepended to the other subelements of the enclosing <Parent> elements.

```
SELECT 1 as Tag, NULL as parent,
PersonID as [Parent!1!PersonID],
PersonName as [Parent!1!PersonName!element], -- element directive
overflow as [Parent!1!!xmltext]
FROM Person
FOR XML EXPLICIT
```

This is the result:

```
<Parent PersonID="P1" attr1="data">
content <PersonName>Joe</PersonName>
</Parent>
<Parent PersonID="P2" attr2="data">
<PersonName>Joe</PersonName>
</Parent>
<Parent PersonID="P3" attr3="data">
<name>content</name> <PersonName>Joe</PersonName>
</Parent>
```

If the **xmltext** column data contain attributes on the root element, these attributes are not shown in XML-Data schema and the MSXML parser does not validate the resulting XML document fragment, for example:

```
SELECT 1 as Tag,
0 as Parent,
N'<overflow a="1"/>' as 'overflow!1!!xmltext'
```

FOR XML EXPLICIT, xmldata

This is the result. Note that in the returned schema, the overflow attribute **a** is missing from the schema.

```
<Schema name="Schema12" xmlns="urn:schemas-microsoft-com:xml
xmlns:dt="urn:schemas-microsoft-com:datatypes">
<ElementType name="overflow" content="mixed" model="open">
</ElementType>
</Schema>
<overflow xmlns="x-schema:#Schema12" a="1"> </overflow>
```

J. Obtain an XML document consisting of customers, orders, and order details

The query in this example generates the universal table rowset described in the conceptual discussion earlier in the topic.

Assume this is the hierarchy to be generated:

```
<Customer cid="C1" name="Janine">
<Order id="O1" date="1/20/1996">
<OrderDetail id="OD1" pid="P1"/>
<OrderDetail id="OD2" pid="P2"/>
</Order>
<Order id="O2" date="3/29/1997">
...
</Customer>
```

The process of writing the query to produce the universal table and the processing of the universal table rowset to produce the resulting XML document is similar to the process described in Example A and Example B.

```
SELECT 1 as Tag,
NULL as Parent,
C.CustomerID as [Customer!1!cid],
C.ContactName as [Customer!1!name],
```

NULL as [Order!2!id], NULL as [Order!2!date], NULL as [OrderDetail!3!id!id], NULL as [OrderDetail!3!pid!idref] FROM Customers C **UNION ALL** SELECT 2 as Tag, 1 as Parent, C.CustomerID, NULL, O.OrderID, O.OrderDate, NULL, NULL FROM Customers C, Orders O WHERE C.CustomerID = O.CustomerID UNION ALL SELECT 3 as Tag, 2 as Parent, C.CustomerID, NULL, O.OrderID, NULL, OD.OrderID, **OD.ProductID** FROM Customers C, Orders O, [Order Details] OD WHERE C.CustomerID = O.CustomerID AND O.OrderID = OD.OrderID ORDER BY [Customer!1!cid], [Order!2!id] FOR XML EXPLICIT

This is the partial result:

<Customer cid="ALFKI" name="Maria Anders">

```
<Order id="10643" date="1997-08-25T00:00:00">
    </orderDetail id="10643" pid="28"></OrderDetail>
    </orderDetail id="10643" pid="39"></OrderDetail>
    </Order>
    </order id="10692" date="1997-10-03T00:00:00">
        </orderDetail id="10692" pid="63"></OrderDetail>
    </orderDetail id="10692" pid="63"></OrderDetail>
    </orderPetail id="10692" pid="63"></OrderDetail>
    </orderPetail id="10702" date="1997-10-13T00:00:00">
        </orderDetail id="10702" pid="3"></orderDetail>
    </orderPetail id="10702" pid="3"></orderDetail>
    </orderDetail id="10702" pid="3"></orderDetail>
    </orderDetail id="10702" pid="3"></orderDetail>
    </orderDetail</or>
```

See Also

<u>SELECT</u>

Specifying the XMLDATA Schema Option in a Query

The primary purpose for specifying XMLDATA in a query is to receive XML data type information that can be used where data types are necessary (for example, in handling numeric expressions). Otherwise, everything in an XML document is a textual string. Generating an XML-Data schema is an overhead on the server, is likely to affect performance, and should be used only when data types are needed.

If the database column from which values are retrieved is of type **sql_variant**, there is no data type information in the XML-Data schema. If a given query designates different XML elements with same name, XMLDATA may produce an invalid XML-Data schema. This is because element name collisions and data type names are not resolved (you might have two elements with same name but different data types).

Example

A. Specify the XMLDATA schema option

This query specifies the XMLDATA schema option. The query returns customer and order information.

```
SELECT Customers.CustomerID, ContactName,
Orders.OrderID, OrderDate, Orders.CustomerID,
ProductID, Quantity
FROM Customers, Orders, [Order Details]
WHERE Customers.CustomerID = Orders.CustomerID
AND Orders.OrderID = [Order Details].OrderID
ORDER BY Customers.CustomerID, Orders.OrderID
FOR XML AUTO, XMLDATA
```

This is the partial result. The XML-Data schema is generated and prepended to the result. The table name **[Order Details]** is an invalid XML name because of the space in the table name. This invalid character is converted into escaped

numeric encoding.

<Schema name="Schema1" xmlns="urn:schemas-microsoft-com:xml-<ElementType name="Customers" content="eltOnly" model="closed" <element type="Orders"/>

<AttributeType name="CustomerID" dt:type="string"/>

<a href="https://www.example.com/activation-commutatio-commutation-commutation-commutation-commutation-commutation

<attribute type="CustomerID"/>

<attribute type="ContactName"/>

</ElementType>

<ElementType name="Orders" content="eltOnly" model="closed">

<element type="Order_0020_Details"/>

<AttributeType name="OrderID" dt:type="i4"/>

<AttributeType name="OrderDate" dt:type="dateTime"/>

<AttributeType name="CustomerID" dt:type="string"/>

<attribute type="OrderID"/>

<attribute type="OrderDate"/>

<attribute type="CustomerID"/>

</ElementType>

<ElementType name="Order_0020_Details" content="empty" model= <AttributeType name="ProductID" dt:type="i4"/>

<AttributeType name="Quantity" dt:type="i2"/>

<attribute type="ProductID"/>

<attribute type="Quantity"/>

</ElementType>

</Schema>

<Customers xmlns="x-schema:#Schema1" CustomerID="ALFKI" Coi <Orders OrderID="10643" OrderDate="1997-08-25T00:00:00"

CustomerID="ALFKI">

<Order_0020_Details ProductID="28" Quantity="15"/>

<Order_0020_Details ProductID="39" Quantity="21"/>

</Orders>

<Orders OrderID="10692" OrderDate="1997-10-03T00:00:00" CustomerID="ALFKI">

```
<Order_0020_Details ProductID="63" Quantity="20"/>
</Orders>
</Customers>
```

The query in the following example assigns the same alias to the **EmployeeID** and **LastName** columns, and **Employees** table specified in the FROM clause:

SELECT EmployeeID emp, LastName emp FROM Employees emp FOR XML AUTO, ELEMENTS, XMLDATA

Only the resulting XML-Data schema is shown. In the schema there are three <emp> elements. Also note that two of the <emp> elements have different data types.

```
<Schema name="Schema2" xmlns="urn:schemas-microsoft-com:xml-
<ElementType name="emp" content="mixed" model="closed">
```

```
<element type="emp"/>
```

```
<element type="emp"/>
```

</ElementType>

```
<ElementType name="emp" content="textOnly" model="closed" dt:ty]
<ElementType name="emp" content="textOnly" model="closed" dt:ty]
</Schema>
```

XML and Internet Support

Writing XML Using OPENXML

OPENXML is a Transact-SQL keyword that provides a rowset over in-memory XML documents. OPENXML is a rowset provider similar to a table or a view. OPENXML allows access to XML data as if it is a relational rowset by providing a rowset view of the internal representation of an XML document. The records in the rowset can be stored in database tables (similar to the rowsets provided by tables and views).

OPENXML can be used in SELECT, and SELECT INTO statements wherever rowset providers, such as a table, a view or OPENROWSET can appear as the source. For information about the syntax of OPENXML, see <u>OPENXML</u>.

To write queries against an XML document using OPENXML, you must first call **sp_xml_preparedocument**, which parses the XML document and returns a handle to the parsed document that is ready for consumption. The parsed document is a tree representation of various nodes (elements, attributes, text, comment, and so on) in the XML document. The document handle is passed to OPENXML, which then provides a rowset view of the document based on the parameters passed to it.

The internal representation of an XML document must be removed from memory by calling **sp_xml_removedocument** system stored procedure to free the memory.

This illustration shows the process.



Note To understand OPENXML, familiarity with XPath queries and understanding of XML is required. For more information about XPath support in Microsoft® SQL Server[™] 2000, see <u>Using XPath Queries</u>.

OPENXML Parameters

The parameters to OPENXML include:

- An XML document handle (*idoc*)
- An XPath expression to identify the nodes to be mapped to rows (*rowpattern*)
- A description of the rowset to be generated
- Mapping between the rowset columns and the XML nodes

XML Document Handle (idoc)

The document handle is returned by the **sp_xml_preparedocument** stored procedure.

XPath Expression to Identify the Nodes to Be Processed (*rowpattern*)

The XPath expression specified as *rowpattern* identifies a set of nodes in the XML document. Each node identified by *rowpattern* corresponds to a single row in the rowset generated by OPENXML.

The nodes identified by the XPath expression can be any XML node (elements, attributes, processing instructions, and so on) in the XML document. If *rowpattern* identifies a set of elements in the XML document, there is one row in the rowset for each element node identified. For example, if *rowpattern* ends in an attribute, a row is created for each attribute node selected by *rowpattern*.

Description of the Rowset to Be Generated

A rowset schema must be provided to OPENXML to generate the rowset. You can specify the rowset schema by using the optional WITH clause. These options are available for specifying the rowset schema:

• Specify the complete schema in the WITH clause.

In specifying the rowset schema you specify the column names and their data types and their mapping to the XML document.

You can specify the column pattern (using the *ColPattern* parameter in the *SchemaDeclaration*). The column pattern specified is used to map a rowset column to the XML node identified by *rowpattern* and also to determine the type of mapping.

If *ColPattern* is not specified for a column, the rowset column maps to the XML node with same name based on the mapping specified by the *flags* parameter. However, if *ColPattern* is specified as part of schema specification in the WITH clause, it overwrites the mapping specified in the *flags* parameter.

• Specify the name of an existing table in the WITH clause.

You can simply specify an existing table name whose schema can be used by OPENXML to generate the rowset.

• Do not specify the WITH clause.

In this case, OPENXML returns a rowset in the edge table format. This is called an edge table because, in this table format, every edge in the parsed XML document tree maps to a row in the rowset.

Edge tables represent the fine-grained XML document structure (for example, element/attribute names, the document hierarchy, the namespaces, processing instructions, and so on) in a single table. The edge table format allows you to get additional information that is not exposed through the metaproperties. For more information about metaproperties, see <u>Specifying Metaproperties in OPENXML</u>.

The additional information provided by edge table allows you to store and query the data type of an element/attribute, the node type (element node, attribute node, or a value node), store and query information about the XML document structure, and to possibly build your own XML document management system.

Using an edge table, you can write stored procedures that take XML documents as a BLOB input, produce the edge table, and then extract and analyze the document on its finest level (find the document hierarchy, element/attribute names, namespaces, processing instructions, and so on).

The edge table also can serve as a storage format for XML documents when mapping to other relational formats does not make sense, and an **ntext** field is not providing enough structural information.

Whenever you would use an XML parser to examine the XML document, you can use edge table to get the same information.

Column	Data	
name	type	Description

This table describes the structure of the edge table.
id	Bigint	Is the unique ID of the document node.
		The root element has an ID value 0. The negative ID values are reserved.
parentid	Bigint	Identifies the parent of the node. The parent identified by this ID is not necessarily the parent element but it depends on the NodeType of the node whose parent is identified by this ID. For example, if the node is a text node, the parent of it may be an attribute node. If the node is at the top level in the XML document, its ParentID is NULL.
nodetype	Int	Identifies the node type. Is an integer that corresponds to the XML DOM node type numbering (see DOM for node information). Some of the node types are: 1 = Element node 2 = Attribute node 3 = Text node
localname	nvarchar	Gives the local name of the element or attribute. Is NULL if the DOM object does not have a name.
prefix	nvarchar	Is the namespace prefix of the node name.
namespaceuri	nvarchar	Is the namespace URI of the node. If the value is NULL, no namespace is present.
datatype	nvarchar	Is the actual data type of the element or attribute row and is NULL otherwise. The data type is inferred from the inline DTD or from the inline schema.
prev	Bigint	Is the XML ID of the previous sibling element. Is NULL if there is no direct

		previous sibling.
text	Ntext	Contains the attribute value or the
		element content in text form (or is NULL
		if the edge table entry does not need a
		value).

Mapping Between the Rowset Columns and the XML Nodes

In the OPENXML statement, you can optionally specify the type of mapping (attribute-centric, element-centric) between the rowset columns and the XML nodes identified by the *rowpattern*. This information is used in transformation between the XML nodes and the rowset columns.

There are two ways to specify the mapping (you can specify both):

• Use the *flags* parameter.

The mapping specified by the *flags* parameter assumes name correspondence where the XML nodes map to corresponding rowset columns with same name.

• Use the *ColPattern* parameter.

ColPattern, an XPath expression, is specified as part of *SchemaDeclaration* in the WITH clause. The mapping specified in *ColPattern* overwrites the mapping specified by *flags* parameter.

ColPattern can be used to specify the special nature of the mapping (in case of attribute-centric and element-centric mapping) that overwrites or enhances the default mapping indicated by the *flags*.

ColPattern is specified if:

- The column name in the rowset is different from the element/attribute name to which it is mapped. In this case *ColPattern* is used to identify the XML element/attribute name to which the rowset column maps.
- You want to map a metaproperty attribute to the column. In this

case, *ColPattern* is used to identify the metaproperty to which the rowset column maps. For more information about using metaproperties , see <u>Specifying Metaproperties in OPENXML</u>.

Both the *flags* and *ColPattern* parameters are optional. If no mapping is specified, attribute-centric mapping (default value of *flags* parameter) is assumed by default.

Attribute-centric Mapping

If the *flags* parameter in OPENXML is set to attributes map to the columns in the rowset based on the name correspondence. Name correspondence means that XML attributes of a given name are stored in a column in the rowset with the same name.

If the column name is different from the attribute name to which it maps, *ColPattern* must be specified.

If XML attribute has a namespace qualifier, the column name in the rowset must have the qualifier as well.

Element-centric Mapping

Setting the *flags* parameter in OPENXML to 2 (XML_ELEMENTS) specifies the **element-centric** mapping. It is similar to **attribute-centric** mapping except for these differences:

- The name correspondence of the mapping (for example, a column mapping to an XML element with the same name) chooses the noncomplex subelements, unless a column-level pattern is specified. In the retrieval case, if subelement is complex (contains further subelements), the column is set to NULL. Attribute values of the subelements are disregarded.
- Multiple subelements with the same name overwrite each other in the order retrieved. Fusion on the parent appends subelement in case of name equivalence.

See Also

<u>sp_xml_preparedocument</u>

sp_xml_removedocument

<u>OPENXML</u>

XML and Internet Support

Using OPENXML

The examples in this topic show how OPENXML is used in creating a rowset view of an XML document. For information about the syntax of OPENXML, see <u>OPENXML</u>. The examples show all aspects of OPENXML except specifying metaproperties in OPENXML. For more information about specifying metaproperties in OPENXML, see <u>Specifying Metaproperties in OPENXML</u>.

Examples

In retrieving the data, *rowpattern* is used to identify the nodes in the XML document that determine the rows. *rowpattern* is expressed in the XPath pattern language used in the MSXML XPath implementation. For example, if the pattern ends in an element or an attribute, a row is created for each element or attribute node selected by *rowpattern*.

The *flags* value provides default mapping. In the *SchemaDeclaration*, if no *ColPattern* is specified, the mapping specified in *flags* is assumed. The *flags* value is ignored if *ColPattern* is specified in *SchemaDeclaration*. The specified *ColPattern* determines the mapping (**attribute-centric** or **element-centric**) and also the behavior in dealing with overflow and unconsumed data.

A. Execute a simple SELECT statement with OPENXML

The XML document in this example consists of the <Customer>, <Order>, and <OrderDetail> elements. The OPENXML statement retrieves customer information in a two-column rowset (**CustomerID** and **ContactName**) from the XML document.

First, the **sp_xml_preparedocument** stored procedure is called to obtain a document handle. This document handle is passed to OPENXML.

In the OPENXML statement:

- *rowpattern* (/ROOT/Customer) identifies the <Customer> nodes to process.
- The *flags* parameter value is set to **1** indicating **attribute-centric**

mapping. As a result, the XML attributes map to the columns in the rowset defined in *SchemaDeclaration*.

• In *SchemaDeclaration* (in the WITH clause), the specified *ColName* values match the corresponding XML attribute names. Therefore, the *ColPattern* parameter is not specified in *SchemaDeclaration*.

And then, the SELECT statement retrieves all the columns in the rowset provided by OPENXML.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
 <Order OrderID="10248" CustomerID="VINET" EmployeeID="5"
     OrderDate="1996-07-04T00:00:00">
   <OrderDetail ProductID="11" Quantity="12"/>
   <OrderDetail ProductID="42" Quantity="10"/>
 </Order>
</Customer>
<Customer CustomerID="LILAS" ContactName="Carlos Gonzlez">
 <Order OrderID="10283" CustomerID="LILAS" EmployeeID="3"
     OrderDate="1996-08-16T00:00:00">
   <OrderDetail ProductID="72" Quantity="3"/>
 </Order>
</Customer>
</ROOT>'
-- Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
-- Execute a SELECT statement using OPENXML rowset provider.
SELECT *
```

```
FROM OPENXML (@idoc, '/ROOT/Customer',1)
```

WITH (CustomerID varchar(10),

ContactName varchar(20)) EXEC sp_xml_removedocument @idoc

This is the result:

CustomerID ContactName

VINETPaul HenriotLILASCarlos Gonzlez

If the same SELECT statement is executed with *flags* set to **2**, indicating **element-centric** mapping, because <Customer> elements do not have any subelements, the values of **CustomerID** and **ContactName** for both the customers are returned as NULL.

If in the XML document, the <CustomerID> and <ContactName> are subelements, the **element-centric** mapping retrieves the values.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customer>
 <CustomerID>VINET</CustomerID>
 <ContactName>Paul Henriot</ContactName>
 <Order OrderID="10248" CustomerID="VINET" EmployeeID="5"</pre>
   <OrderDetail ProductID="11" Quantity="12"/>
   <OrderDetail ProductID="42" Quantity="10"/>
 </Order>
</Customer>
<Customer>
 <CustomerID>LILAS</CustomerID>
 <ContactName>Carlos Gonzlez</ContactName>
 <Order OrderID="10283" CustomerID="LILAS" EmployeeID="3" (
   <OrderDetail ProductID="72" Quantity="3"/>
 </Order>
```

</Customer> </ROOT>' -- Create an internal representation of the XML document. EXEC sp_xml_preparedocument @idoc OUTPUT, @doc -- Execute a SELECT statement using OPENXML rowset provider. SELECT * FROM OPENXML (@idoc, '/ROOT/Customer',2) WITH (CustomerID varchar(10), ContactName varchar(20)) EXEC sp_xml_removedocument @idoc

This is the result:

CustomerID ContactName

VINET Paul Henriot

LILAS Carlos Gonzlez

B. Specify *ColPattern* for mapping between rowset columns and the XML attributes/elements

This example shows how the XPath pattern is specified in the optional *ColPattern* parameter to provide mapping between rowset columns and the XML attributes (and elements).

The XML document in this example consists of the <Customer>, <Order>, and <OrderDetail> elements. The OPENXML statement retrieves customer and order information as a rowset (**CustomerID**, **OrderDate**, **ProdID**, and **Qty**) from the XML document.

First, the **sp_xml_preparedocument** stored procedure is called to obtain a document handle. This document handle is passed to OPENXML.

In the OPENXML statement:

• *rowpattern* (/ROOT/Customer/Order/OrderDetail) identifies the <OrderDetail> nodes to process.

• For illustration purposes, the *flags* parameter value is set to **2** indicating **element-centric** mapping. However, the mapping specified in *ColPattern* overwrites this mapping (the XPath pattern specified in *ColPattern* maps the columns in the rowset to attributes thus resulting in an **attribute-centric** mapping).

In *SchemaDeclaration* (in the WITH clause), *ColPattern* is also specified with the *ColName* and *ColType* parameters. The optional *ColPattern* is the XPath pattern specified to indicate:

- The OrderID, CustomerID, and OrderDate columns in the rowset map to the attributes of the parent of the nodes identified by *rowpattern*. *rowpattern* identifies the <OrderDetail> nodes. Therefore, the CustomerID and OrderDate columns map to CustomerID and OrderDate attributes of the <Order> element.
- The **ProdID** and **Qty** columns in the rowset map to the **ProductID** and **Quantity** attributes of the nodes identified in *rowpattern*.

And then the SELECT statement retrieves all the columns in the rowset provided by OPENXML.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
<Order OrderID="10248" CustomerID="VINET" EmployeeID="5"
OrderDate="1996-07-04T00:00:00">
<OrderDate="1996-07-04T00:00:00">
<OrderDate="1996-07-04T00:00:00">
<OrderDetail ProductID="11" Quantity="12"/>
<OrderDetail ProductID="42" Quantity="10"/>
</Order>
</Customer>
<Customer CustomerID="LILAS" ContactName="Carlos Gonzlez">
<Order OrderID="10283" CustomerID="LILAS" EmployeeID="3"
OrderOrderID="1996-08-16T00:00:00">
```

```
<OrderDetail ProductID="72" Quantity="3"/>
 </Order>
</Customer>
</ROOT>'
-- Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
-- Execute a SELECT stmt using OPENXML rowset provider.
SELECT *
FROM OPENXML (@idoc, '/ROOT/Customer/Order/OrderDetail',2)
   WITH (OrderID
                            '../@OrderID',
                    int
      CustomerID varchar(10) '../@CustomerID',
      OrderDate datetime '../@OrderDate',
                        '@ProductID',
      ProdID
                int
                      '@Quantity')
      Qty
               int
```

This is the result:

OrderID	CustomerID	OrderDate	ProdII	D Q	ty
10248	VINET	1996-07-04 00:00:00	0.000	 11	12
10248	VINET	1996-07-04 00:00:00	0.000	42	10
10283	LILAS	1996-08-16 00:00:00	.000	72	3

The XPath pattern specified as *ColPattern* can also be specified to map the XML elements to the rowset columns (resulting in **element-centric** mapping). In the following example, the XML document <CustomerID> and <OrderDate> are subelements of <Orders> element. Because *ColPattern* overwrites the mapping specified in *flags* parameter, the *flags* parameter is not specified in OPENXML.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
<Order EmployeeID="5" >
```

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

<CustomerID>VINET</CustomerID>

<OrderDate>1996-07-04T00:00:00</OrderDate>

<OrderDetail ProductID="11" Quantity="12"/>

<OrderDetail ProductID="42" Quantity="10"/>

</Order>

</Customer>

```
<Customer CustomerID="LILAS" ContactName="Carlos Gonzlez">
<Order EmployeeID="3" >
```

```
<OrderID>10283</OrderID>
```

<CustomerID>LILAS</CustomerID>

<OrderDate>1996-08-16T00:00:00</OrderDate>

<OrderDetail ProductID="72" Quantity="3"/>

</Order>

</Customer>

</ROOT>'

```
-- Create an internal representation of the XML document.
```

```
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
```

```
-- Execute a SELECT stmt using OPENXML rowset provider. SELECT *
```

```
FROM OPENXML (@idoc, '/ROOT/Customer/Order/OrderDetail')
WITH (CustomerID varchar(10) '../CustomerID',
```

```
OrderDate datetime '../OrderDate',
```

ProdID int '@ProductID',

```
Qty int '@Quantity')
```

EXEC sp_xml_removedocument @idoc

C. Combining attribute-centric and element-centric mapping

In this example, the *flags* parameter is set to **3**, indicating that both **attribute-centric** and **element-centric** mapping is to be applied. In this case, the **attribute-centric** mapping is applied first, and then **element-centric** mapping is applied for all the columns not yet dealt with.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customer CustomerID="VINET" >
  <ContactName>Paul Henriot</ContactName>
 <Order OrderID="10248" CustomerID="VINET" EmployeeID="5"
     OrderDate="1996-07-04T00:00:00">
   <OrderDetail ProductID="11" Quantity="12"/>
   <OrderDetail ProductID="42" Quantity="10"/>
 </Order>
</Customer>
<Customer CustomerID="LILAS" >
  <ContactName>Carlos Gonzlez</ContactName>
 <Order OrderID="10283" CustomerID="LILAS" EmployeeID="3"
     OrderDate="1996-08-16T00:00:00">
   <OrderDetail ProductID="72" Quantity="3"/>
 </Order>
</Customer>
</ROOT>'
```

-- Create an internal representation of the XML document. EXEC sp_xml_preparedocument @idoc OUTPUT, @doc

```
-- Execute a SELECT statement using OPENXML rowset provider.
SELECT *
FROM OPENXML (@idoc, '/ROOT/Customer',3)
WITH (CustomerID varchar(10),
ContactName varchar(20))
```

EXEC sp_xml_removedocument @idoc

This is the result

CustomerID ContactName

VINET Paul Henriot LILAS Carlos Gonzlez

The **attribute-centric** mapping is applied for **CustomerID**. There is no **ContactName** attribute in the <Customers> element; therefore, **element-centr**ic mapping is applied.

D. Specify text() XPath function as *ColPattern*

The XML document in this example consists of the <Customer> and <Order> elements. The OPENXML statement retrieves a rowset consisting of the **oid** attribute from the <Order> element, the ID of the parent of the node (identified by *rowpattern*), and the leaf-value string of the element content.

First, the **sp_xml_preparedocument** stored procedure is called to obtain a document handle. This document handle is passed to OPENXML.

In the OPENXML statement:

- *rowpattern* (/root/Customer/Order) identifies the <Order> nodes to process.
- The *flags* parameter value is set to **1**, indicating **attribute-centric** mapping. As a result, the XML attributes map to the rowset columns defined in *SchemaDeclaration*.
- In *SchemaDeclaration* (in the WITH clause), the rowset column names, **oid** and **amount**, match the corresponding XML attribute names. Therefore, the *ColPattern* parameter is not specified. For the **comment** column in the rowset, the XPath function (**text()**) is specified as *ColPattern*. This overwrites the **attribute-centric** mapping specified in *flags*, and the column contains the leaf-value string of the element content.

And then, the SELECT statement retrieves all the columns in the rowset

provided by OPENXML.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
--sample XML document
SET @doc ='
<root>
 <Customer cid= "C1" name="Janine" city="Issaquah">
   <Order oid="01" date="1/20/1996" amount="3.5" />
   <Order oid="O2" date="4/30/1997" amount="13.4">Customer was
   </Order>
 </Customer>
 <Customer cid="C2" name="Ursula" city="Oelde" >
   <Order oid="O3" date="7/14/1999" amount="100" note="Wrap it ]
       white red">
      <Urgency>Important</Urgency>
      Happy Customer.
   </Order>
   <Order oid="O4" date="1/20/1996" amount="10000"/>
 </Customer>
</root>
```

```
-- Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
```

```
-- Execute a SELECT statement using OPENXML rowset provider.

SELECT *

FROM OPENXML (@idoc, '/root/Customer/Order', 1)

WITH (oid char(5),

amount float,

comment ntext 'text()')

EXEC sp_xml_removedocument @idoc
```

This is the result:

oid	amount	comment
O 1	3.5	NULL
O2	13.4	Customer was very satisfied
O3	100.0	Happy Customer.
O4	10000.0	NULL

E. Specify *TableName* in the WITH clause

This example specifies *TableName* in the WITH clause instead of *SchemaDeclaration* in the WITH clause. This is useful if you have a table with the structure you want and no column patterns (*ColPattern* parameter) are required.

The XML document in this example consists of the <Customer> and <Order> elements. The OPENXML statement retrieves order information in a three-column rowset (**oid**, **date**, and **amount**) from the XML document.

First, the **sp_xml_preparedocument** stored procedure is called to obtain a document handle. This document handle is passed to OPENXML.

In the OPENXML statement:

- *rowpattern* (/root/Customer/Order) identifies the <Order> nodes to process.
- There is no *SchemaDeclaration* in the WITH clause. Instead, a table name is specified. Therefore, the table schema is used as the rowset schema.
- The *flags* parameter value is set to **1**, indicating **attribute-centric** mapping. Therefore, attributes of the elements (identified by *rowpattern*) map to the rowset columns with the same name.

And then the SELECT statement retrieves all the columns in the rowset provided by OPENXML.

-- Create a test table. This table schema is used by OPENXML as the

```
-- rowset schema.
CREATE TABLE T1(oid char(5), date datetime, amount float)
DECLARE @idoc int
DECLARE @doc varchar(1000)
-- Sample XML document
SET @doc ='
<root>
 <Customer cid= "C1" name="Janine" city="Issaquah">
   <Order oid="01" date="1/20/1996" amount="3.5" />
   <Order oid="O2" date="4/30/1997" amount="13.4">Customer was
       satisfied</Order>
 </Customer>
 <Customer cid="C2" name="Ursula" city="Oelde" >
   <Order oid="O3" date="7/14/1999" amount="100" note="Wrap it ]
       white red">
     <Urgency>Important</Urgency>
   </Order>
   <Order oid="O4" date="1/20/1996" amount="10000"/>
 </Customer>
</root>
--Create an internal representation of the XML document.
```

```
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
```

```
    Execute a SELECT statement using OPENXML rowset provider.
    SELECT *
    FROM OPENXML (@idoc, '/root/Customer/Order', 1)
    WITH T1
    EXEC sp_xml_removedocument @idoc
```

This is the result:

oid date amount

----- ------- -------

- O1 1996-01-20 00:00:00.000 3.5
- O2 1997-04-30 00:00:00.000 13.4
- O3 1999-07-14 00:00:00.000 100.0
- O4 1996-01-20 00:00:00.000 10000.0

F. Obtain the result in an edge table format

In this example, the WITH clause is not specified in the OPENXML statement. As a result, the rowset generated by OPENXML has an edge table format. The SELECT statement returns all the columns in the edge table.

The sample XML document in the example consists of the <Customer>, <Order>, and <OrderDetail> elements.

First, the **sp_xml_preparedocument** stored procedure is called to obtain a document handle. This document handle is passed to OPENXML.

In the OPENXML statement:

- *rowpattern* (/ROOT/Customer) identifies the <Customer> nodes to process.
- The WITH clause is not provided; therefore, OPENXML returns the rowset in an edge table format.

And then the SELECT statement retrieves all the columns in the edge table.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<ROOT>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
<Order CustomerID="VINET" EmployeeID="5" OrderDate=
"1996-07-04T00:00:00">
<Order Detail OrderID="10248" ProductID="11" Quantity="12"/>
<OrderDetail OrderID="10248" ProductID="42" Quantity="10"/>
</Order>
```

</Customer>

```
<Customer CustomerID="LILAS" ContactName="Carlos Gonzlez">
<Order CustomerID="LILAS" EmployeeID="3" OrderDate=
"1996-08-16T00:00:00">
<OrderDetail OrderID="10283" ProductID="72" Quantity="3"/>
</Order>
</Customer>
</ROOT>'
--Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
-- Execute a SELECT statement using OPENXML rowset provider.
SELECT *
FROM OPENXML (@idoc, '/ROOT/Customer')
EXEC sp_xml removedocument @idoc
```

The result is returned as an edge table. You can write queries against the edge table to obtain information:

- The following query returns the number of Customer nodes in the document. Because the WITH clause is not specified, OPENXML returns an edge table. The SELECT statement queries the edge table. SELECT count(*)
 FROM OPENXML(@idoc, '/')
 WHERE localname = 'Customer'
- This query returns local names of XML nodes of element type. SELECT distinct localname FROM OPENXML(@idoc, '/') WHERE nodetype = 1 ORDER BY localname

G. Specify rowpattern ending with an attribute

The XML document in this example consists of the <Customer>, <Order>, and <OrderDetail> elements. The OPENXML statement retrieves order details

information in a three-column rowset (**ProductID**, **Quantity**, and **OrderID**) from the XML document.

First, the **sp_xml_preparedocument** is called to obtain a document handle. This document handle is passed to OPENXML.

In the OPENXML statement:

- *rowpattern* (/ROOT/Customer/Order/OrderDetail/@ProductID) ends with an XML attribute (**ProductID**). In the resulting rowset, a row is created for each attribute node selected in the XML document.
- In this example, the *flags* parameter is not specified. Instead, the mappings are specified by the *ColPattern* parameter.

In *SchemaDeclaration* (in the WITH clause), *ColPattern* is also specified with the *ColName* and *ColType* parameters. The optional *ColPattern* is the XPath pattern specified to indicate:

- The XPath pattern (.) specified as *ColPattern* for the **ProdID** column in the rowset identifies the context node (current node). As per the *rowpattern* specified, it is the **ProductID** attribute of the <OrderDetail> element.
- The *ColPattern*, **.**/@Quantity, specified for the Qty column in the rowset identifies the Quantity attribute of the parent (<OrderDetail>) node of the context node (<ProductID>).
- Similarly, the *ColPattern*, .././@OrderID, specified for the OID column in the rowset identifies the OrderID attribute of the parent (<Order>) of the parent (<OrderDetail>) node of the context node (<ProductID>).

And then, the SELECT statement retrieves all the columns in the rowset provided by OPENXML.

DECLARE @idoc int DECLARE @doc varchar(1000)

```
--Sample XML document
SET @doc ='
<ROOT>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
 <Order OrderID="10248" CustomerID="VINET" EmployeeID="5"
      "1996-07-04T00:00:00">
   <OrderDetail ProductID="11" Quantity="12"/>
   <OrderDetail ProductID="42" Quantity="10"/>
 </Order>
</Customer>
<Customer CustomerID="LILAS" ContactName="Carlos Gonzlez">
 <Order OrderID="10283" CustomerID="LILAS" EmployeeID="3" (
      "1996-08-16T00:00:00">
   <OrderDetail ProductID="72" Quantity="3"/>
 </Order>
</Customer>
</ROOT>'
-- Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
-- Execute a SELECT stmt using OPENXML rowset provider.
SELECT *
FROM OPENXML (@idoc, '/ROOT/Customer/Order/OrderDetail/@P
   WITH ( ProdID int '.',
             int '../@Quantity',
       Qty
              int '../../@OrderID')
       OID
```

EXEC sp_xml_removedocument @idoc

This is the result:

ProdID	Qty	OID
	10	10240
11	12	10248
42	10	10248
72	3	10283

H. Specify an XML document with multiple text nodes

If you have multiple text nodes in an XML document, a SELECT statement with a *ColPattern* (**text()**) returns only the first text node instead of all. For example:

```
DECLARE @h int
EXEC sp_xml_preparedocument @h OUTPUT,
N'
<root xmlns:a="urn:1">
<a:Elem abar="asdf">
T<a>a</a>U
</a:Elem>
</root>',
'<ns xmlns:b="urn:1" />'
```

```
SELECT * FROM openxml(@h, '/root/b:Elem')
WITH (Col1 varchar(20) 'text()')
```

The SELECT statement returns \mathbf{T} as the result (and not \mathbf{TaU})

I. Retrieve individual values from multivalued attributes

An XML document can have attributes that are multivalued. For example the **IDREFS** attribute can be multivalued. In an XML document, the multivalued attribute values are specified as a string with the values separated by a space. In the following XML document, the **attends** attribute of the <Student> element and the **attendedBy** attribute of <Class> are multivalued. Retrieving individual values from a multivalued XML attribute and storing each value in a separate row in the database requires additional work. This example shows the process.

This sample XML document consists of the following elements:

• <Student>

Consists of **id** (student ID), **name**, and **attends** attributes. The **attends** attribute is a multivalued attribute.

• <Class>

Consists of **id** (class ID), **name**, and **attendedBy** attributes. The **attendedBy** attribute is a multivalued attribute.

This **attends** attribute in <Student> and the **attendedBy** attribute in <Class> represent a **m:n** relationship between **Student** and **Class** tables. A student can take many classes and a class can have many students.

Assume you want to shred this document and save it in the database as follows:

- Save the <Student> data in the **Students** table.
- Save the <Class> data in the **Courses** table.
- Save he **m:n** relationship data (between Student and Class) in the **CourseAttendence** table. Additional work is required to extract the values. To retrieve this information and store it in the table, use these stored procedures:
 - Insert_Idrefs_Values

Inserts the values of course ID and student ID in the CourseAttendence table.

• Extract_idrefs_values

Extracts the individual student IDs from each <Course> element. An edge table is used to retrieve these values.

Here are the steps:

1. Create the following tables:

```
DROP TABLE CourseAttendance
DROP TABLE Students
DROP TABLE Courses
GO
CREATE TABLE Students(
id varchar(5) primary key,
name varchar(30)
)
```

```
GO
  CREATE TABLE Courses(
                varchar(5) primary key,
          id
                  varchar(30),
          name
          taughtBy varchar(5)
  )
  GO
  CREATE TABLE CourseAttendance(
                varchar(5) references Courses(id),
         id
         attendedBy varchar(5) references Students(id),
         constraint CourseAttendance_PK primary key (id, atte
  )
  go
2. Create these stored procedures:
  DROP PROCEDURE f idrefs
  GO
  CREATE PROCEDURE f_idrefs
          varchar(500),
    @t
    @idtab varchar(50),
    @id
           varchar(5)
  AS
  DECLARE @sp int
  DECLARE @att varchar(5)
  SET @sp = 0
  WHILE (LEN(@t) > 0)
  BEGIN
    SET @sp = CHARINDEX('', @t+'')
    SET @att = LEFT(@t, @sp-1)
    EXEC('INSERT INTO '+@idtab+' VALUES ("'+@id+"', "'+
    SET @t = SUBSTRING(@t+'', @sp+1, LEN(@t)+1-@sp)
  END
  Go
```

DROP PROCEDURE fill idrefs GO **CREATE PROCEDURE fill idrefs** @xmldoc int. @xpath varchar(100), @from varchar(50), varchar(50), @to @idtable varchar(100) AS DECLARE @t varchar(500) DECLARE @id varchar(5) /* Temporary Edge table */ **SELECT** * INTO #TempEdge FROM OPENXML(@xmldoc, @xpath) DECLARE fillidrefs cursor CURSOR FOR SELECT CAST(iv.text AS nvarchar(200)) AS id, CAST(av.text AS nvarchar(4000)) AS refs FROM #TempEdge c, #TempEdge i, #TempEdge iv, #TempEdge a, #TempEdge av WHERE c.id = i.parentid AND UPPER(i.localname) = UPPER(@from) AND i.id = iv.parentid AND c.id = a.parentid AND UPPER(a.localname) = UPPER(@to) a.id = av.parentid AND **OPEN** fillidrefs cursor

```
FETCH NEXT FROM fillidrefs cursor INTO @id, @t
WHILE (@@FETCH STATUS <> -1)
```

```
BEGIN

IF (@@FETCH_STATUS <> -2)

BEGIN

execute f_idrefs @t, @idtable, @id

END

FETCH NEXT FROM fillidrefs_cursor INTO @id, @t

END

CLOSE fillidrefs_cursor

DEALLOCATE fillidrefs_cursor

Go
```

3. This is the sample document that is shredded and the data is stored in the preceding tables. DECLARE @h int EXECUTE sp_xml_preparedocument @h OUTPUT, ' <Data> <Student id = "s1" name = "Student1" attends = "c1 c3 c6" <Student id = "s2" name = "Student2" attends = "c2 c4" /> <Student id = "s3" name = "Student3" attends = "c2 c4 c6" / <Student id = "s4" name = "Student4" attends = "c1 c3 c5" / <Student id = "s5" name = "Student5" attends = "c1 c3 c5 cf <Student id = "s6" name = "Student6" /> <Class id = "c1" name = "Intro to Programming" attendedBy = "s1 s4 s5" /> <Class id = "c2" name = "Databases" attendedBy = "s2 s3" /> <Class id = "c3" name = "Operating Systems" attendedBy = "s1 s4 s5" /> <Class id = "c4" name = "Networks" attendedBy = "s2 s3" />

```
<Class id = "c5" name = "Algorithms and Graphs"
attendedBy = "s4 s5"/>
```

```
<Class id = "c6" name = "Power and Pragmatism"
```

attendedBy = "s1 s3 s5" /> </Data>'

INSERT INTO Students SELECT * FROM OPENXML(@h,

INSERT INTO Courses SELECT * FROM OPENXML(@h, '//* Using the edge table */ EXECUTE fill_idrefs @h, '//Class', 'id', 'attendedby', 'Course/

SELECT * FROM Students SELECT * FROM Courses SELECT * FROM CourseAttendance

EXECUTE sp_xml_removedocument @h

See Also

sp_xml_preparedocument

sp_xml_removedocument

OPENXML

Writing XML Using OPENXML

XML and Internet Support

Specifying Metaproperties in OPENXML

Metaproperty attributes in an XML document are attributes that describe the properties of an XML item (element, attribute, or any other DOM node). These attributes do not physically exist in the XML document text; however, OPENXML provides these metaproperties for all the XML items. These metaproperties allow you to extract information (such as local positioning and namespace information) of XML nodes, which provide more details than is visible in the textual representation.

You can map these metaproperties to the rowset columns in an OPENXML statement using the *ColPattern* parameter. The columns will contain the values of the metaproperties to which they are mapped. For more information about the syntax of OPENXML, see <u>OPENXML</u>.

To access the metaproperty attributes, a namespace specific to Microsoft® SQL Server[™] 2000 (**urn:schemas-microsoft-com:xml-metaprop**) is provided that allows the user to access the metaproperty attributes. If the result of an OPENXML query is returned in an edge table format, the edge table contains one column for each metaproperty attribute (except for the **xmltext** metaproperty).

Some of the metaproperty attributes are used for processing purposes. For example, **xmltext** metaproperty attribute is used for overflow handling. Overflow handling refers to the unconsumed, unprocessed data in the document. One of the columns in the rowset generated by OPENXML can be identified as overflow column by mapping it to **xmltext** metaproperty using the *ColPattern* parameter. The column then receives the overflow data (the *flags* parameter determines whether the column contains only the unconsumed data or everything).

The following table lists the metaproperty attributes that each parsed XML element possesses. These metaproperty attributes can be accessed using the namespace **urn:schemas-microsoft-com:xml-metaprop**. Any value set by the user directly in the XML document using these metaproperties is disregarded.

Note You cannot reference these metaproperties in any XPath navigation.

Metaproperty		
attribute	Description	
@mp:id	Provides system-generated, document-wide identifier of the DOM node (element, attribute, and so on). This ID is guaranteed to refer to the same XML node as long as the document is not reparsed. An XML ID of 0 indicates that the element is a root	
	element. Its parent XML ID is NULL.	
@mp:localname	Stores the local part of the name of the node. It is used with prefix and namespace URI (Uniform Resource Identifier) to name element or attribute nodes.	
@mp:namespaceuri	Provides the namespace URI of the current element. If the value of this attribute is NULL, no namespace is present	
@mp:prefix	Stores the namespace prefix of the current element name. If no prefix is present (NULL) and a URI is given, indicates that the specified namespace is the default namespace. If no URI is given, no namespace is attached.	
@mp:prev	Stores the previous sibling relative to a node, thereby, providing information about the ordering of elements in the document.	
	@mp:prev contains the XML ID of the previous sibling that has the same parent element. If an element is at the beginning of the sibling list, @mp:prev is NULL.	
@mp:xmltext	This metaproperty is used for processing purposes. Is the textual serialization of the element and its attributes and subelements as used in the overflow handling of OPENXML.	

This table shows additional parent properties that are provided that allow you to

retrieve information about the hierarchy.

Parent metaproperty attribute	Description
@mp:parentid	Corresponds to/@mp:id
@mp:parentlocalname	Corresponds to/@mp:localname
@mp:parentnamespacerui	Corresponds to
	/@mp:namespaceuri
@mp:parentprefix	Corresponds to/@mp:prefix

Examples

A. Map the OPENXML rowset columns to the metaproperties

This example creates a rowset view of the sample XML document by using OPENXML. This example shows how the various metaproperty attributes can be mapped to rowset columns in an OPENXML statement using the *ColPattern* parameter.

In the OPENXML statement:

- The **id** column is mapped to the **@mp:id** metaproperty attribute indicating that the column contains the system-generated unique XML ID of the element.
- The **parent** column is mapped to *@mp:parentid*, indicating that the column contains the XML ID of the parent of the element.
- The **parentLocalName** column is mapped to **@mp:parentlocalname**, indicating that the column contains the local name of the parent.

And then, the SELECT statement returns the rowset provided by OPENXML:

DECLARE @idoc int DECLARE @doc varchar(1000) -- Sample XML document

```
SET @doc ='
<root>
  <Customer cid= "C1" name="Janine" city="Issaquah">
      <Order oid="O1" date="1/20/1996" amount="3.5" />
      <Order oid="O2" date="4/30/1997" amount="13.4">Customer was
      </Customer>
      <Customer cid="C2" name="Ursula" city="Oelde" >
      <Order oid="O3" date="7/14/1999" amount="100" note="Wrap it |
      <Urgency>Important</Urgency>
      </Order>
      <Order oid="O4" date="1/20/1996" amount="10000"/>
      </Customer>
</root>
```

```
-- Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
```

```
-- Execute a SELECT statement using OPENXML rowset provider.

SELECT *

FROM OPENXML (@idoc, '/root/Customer/Order', 9)

WITH (id int '@mp:id',

oid char(5),

date datetime,

amount real,

parentIDNo int '@mp:parentid',

parentLocalName varchar(40) '@mp:parentlocalname')

EXEC sp_xml_removedocument @idoc
```

This is the result:

id	oid	date	amount	parentIl	DNo	parentLocalName
6	01	1996-01-20 ()0:00:00.000	3.5	2	Customer
10	02	1997-04-30	00:00:00.000	13.4		Customer

19	O3	1999-07-14 00:00:00.000	100.0	15	Customer
25	04	1996-01-20 00:00:00.000	10000.0	15	Customer

B. Retrieve the entire XML document

In this example, OPENXML creates a one-column rowset view of the sample XML document. This column (**Col1**) is mapped to the **xmltext** metaproperty, making it an overflow column. Therefore, the column receives the unconsumed data, which is the entire document in this case.

And then the SELECT statement returns the entire rowset.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
SET @doc ='
<?xml version="1.0"?>
<root>
 <Customer cid= "C1" name="Janine" city="Issaguah">
   <Order oid="O1" date="1/20/1996" amount="3.5" />
   <Order oid="O2" date="4/30/1997" amount="13.4">Customer was
       satisfied</Order>
 </Customer>
 <Customer cid="C2" name="Ursula" city="Oelde" >
   <Order oid="O3" date="7/14/1999" amount="100" note="Wrap it ]
       white red">
  <MyTag>Testing to see if all the subelements are returned</MyTag
     <Urgency>Important</Urgency>
   </Order>
   <Order oid="O4" date="1/20/1996" amount="10000"/>
 </Customer>
</root>
```

-- Create an internal representation of the XML document. EXEC sp_xml_preparedocument @idoc OUTPUT, @doc -- Execute a SELECT statement using OPENXML rowset provider. SELECT * FROM OPENXML (@idoc, '/') WITH (Col1 ntext '@mp:xmltext')

To retrieve the entire document without the XML declaration, the query can be specified as:

```
SELECT *
FROM OPENXML (@idoc, '/root')
WITH (Col1 ntext '@mp:xmltext')
EXEC sp_xml_removedocument @idoc
```

The query returns the root element with the name root and the data contained by the root element

C. Specify the xmltext metaproperty to retrieve the unconsumed data in a column

This example creates a rowset view of the sample XML document by using OPENXML. The example shows how to retrieve unconsumed XML data by mapping the **xmltext** metaproperty attribute to a rowset column in OPENXML.

The **comment** column is identified as the overflow column by mapping it to the *@mp:xmltext* metaproperty. The *flags* parameter is set to **9** (XML_ATTRIBUTE and XML_NOCOPY), indicating **attribute-centric** mapping and that only the unconsumed data should be copied to the overflow column.

And then the SELECT statement returns the rowset provided by OPENXML.

In this example, **@mp:parentlocalname** metaproperty is set for a column (**ParentLocalName**) in the rowset generated by OPENXML. As a result, this column contains the local name of the parent element.

Two additional columns are specified in the rowset (**parent** and **comment**). The **parent** column is mapped to **@mp:parentid**, indicating that the column contains the XML ID of the parent element of the element. The comment column is identified as the overflow column by mapping it to **@mp:xmltext**

metaproperty.

```
DECLARE @idoc int
DECLARE @doc varchar(1000)
-- sample XML document
SET @doc ='
<root>
 <Customer cid= "C1" name="Janine" city="Issaquah">
   <Order oid="01" date="1/20/1996" amount="3.5" />
   <Order oid="O2" date="4/30/1997" amount="13.4">Customer was
 </Customer>
 <Customer cid="C2" name="Ursula" city="Oelde" >
   <Order oid="O3" date="7/14/1999" amount="100" note="Wrap it]
     <Urgency>Important</Urgency>
   </Order>
   <Order oid="O4" date="1/20/1996" amount="10000"/>
 </Customer>
</root>
```

```
-- Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
```

```
-- Execute a SELECT statement using OPENXML rowset provider.

SELECT *

FROM OPENXML (@idoc, '/root/Customer/Order', 9)

WITH (oid char(5),

date datetime,

comment ntext '@mp:xmltext')

EXEC sp_xml_removedocument @idoc
```

This is the result. Because the **oid** and **date** columns are already consumed, they do not appear in the overflow column.

oid date comment

01	1996-01-20 00:00:00.000	<order amount="3.5"></order>	
O2	1997-04-30 00:00:00.000	<order amount="13.4">Customer w</order>	
	satisfied 0</td <td>Order></td>	Order>	
O3	1999-07-14 00:00:00.000	<order amount="100" note="Wrap i</td></tr><tr><td></td><td>white red"></order>	> <urgency></urgency>
	Important<		
O4	1996-01-20 00:00:00.000	<order amount="10000"></order>	

See Also

Writing XML Using OPENXML

<u>OPENXML</u>
XML System Stored Procedures

Microsoft[®] SQL Server[™] 2000 provides these system stored procedures that are used in conjunction with OPENXML:

- <u>sp_xml_preparedocument</u>
- <u>sp_xml_removedocument</u>

To write queries using OPENXML, you must first create an internal representation of the XML document by calling **sp_xml_preparedocument**. The stored procedure returns a handle to the internal representation of the XML document. This handle is then passed to OPENXML, which provides rowset views of the document based on Xpaths; namely one row pattern and one or more column patterns.

The internal representation of an XML document can be removed from memory by calling **sp_xml_removedocument** system stored procedure.

See Also

OPENXML

Sample XML Applications

The topics in this section present simple applications that demonstrate how to pass an XML document from the client to the server. The XML document is then shredded using OPENXML and the necessary updates are applied to the database tables.

Sample HTML Form to Insert Records Using OPENXML

This sample HTML form prompts a user to enter an employee ID, first name, and last name. After the user has entered the data in the form, an XML document containing the employee element to be inserted in the database is created. The XML document is passed as a parameter to the template.

Before executing this example, you must create a virtual root. For more information, see <u>Creating the nwind Virtual Directory</u>.

This example shows:

- How to create a simple HTML form.
- How to create an XML document from the data entered in the HTML form.
- How to pass the XML document to the template.
- How to use the OPENXML clause in an INSERT statement to add the record in the database.

The template executes a stored procedure. The XML document is passed to the stored procedure as a text parameter.

The stored procedure:

- Calls **sp_xml_preparedocument** to create an internal representation of the XML document passed as a text parameter.
- Calls the INSERT statement to insert the employee record in the **Employee** table. The record to be inserted is provided by OPENXML, which creates a rowset view of the XML document.

These are the steps to create a working sample:

- 1. Create this table: CREATE TABLE Employee(eid int, fname varchar(20), lnam
- Create these stored procedure in the database: CREATE PROC sp_insert_employee @empdata ntext AS

DECLARE @hDoc int EXEC sp_xml_preparedocument @hDoc OUTPUT, @emp INSERT INTO Employee SELECT * FROM OPENXML(@hDoc, '/Employee') WITH Employee EXEC sp_xml_removedocument @hDoc

3. Create the following template. The template must be stored in the directory associated with the virtual name of template type (if you have created the sample **nwind** virtual directory, the template is stored in the template subdirectory of the virtual root directory). Save the template as MyTemplate.xml.

```
<root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:header>
<sql:param name="empdata"><Employee/></sql:param>
</sql:header>
<sql:query>exec sp_insert_employee @empdata
</sql:query>
</root>
```

This is the HTML form:

```
<html>
<body>
<form action="http://IISServer/nwind/template
/MyTemplate.xml" method="post">
```

```
<input type="hidden" id="e" name="empdata">
<input type="hidden" name="contenttype" value="text/xml">
EmployeeID: <input type=text id=eid value="1"><br>
First Name: <input type=text id=fname value="Harry"><br>
Last Name: <input type=text id=lname value="Smith"><br>
<input type=submit onclick="Insert_Employee(e, eid,
lname, fname)" value="Insert Employee"><br><br>
```

```
<script>
```

```
function Insert_Employee(e, eid, lname, fname)
{
    e.value = '<Employee eid=''' + eid.value +
    ''' lname=''' + lname.value + ''' fname=''' +
    fname.value + '''/>';
    }
</script>
</form>
</body>
</html>
```

Sample HTML Form to Update Records Using OPENXML

This sample application shows how data in an XML documents can be used to update records in a database table. The application shows the process of:

- Executing a template from an HTML form.
- Passing an XML document as a parameter to the template.
- Executing SQL statements (stored procedures) in a template.
- Using the OPENXML rowset provider with UPDATE to apply the updates.

The application assumes that the client has an XML document that is created using another application. The application uses OPENXML to shred the XML document and creates the rowset that is passed to UPDATE statement.

Before executing this example, you must create a virtual root. For more information, see <u>Creating the nwind Virtual Directory</u>.

The template executes a stored procedure (**sp_update_employee**). The XML document is passed to the stored procedure as a text parameter.

The stored procedure:

- Calls **sp_xml_preparedocument** to create an internal representation of the XML document passed as a text parameter.
- Calls the **UPDATE** statement to update the employee records in the **Employee** table. **OPENXML** provides the rowset view of the XML, which is used in the UPDATE statement.

These are the steps to create a working sample:

- 1. Create this table: CREATE TABLE Employee(eid int, fname varchar(20), lnam
- Add sample data: INSERT INTO Employee VALUES (1, 'Nancy', 'Davolio') INSERT INTO Employee VALUES (2, 'Andrew', 'Fuller')
- 3. Create this stored procedure in the database: CREATE PROC sp_update_employee @empdata ntext AS DECLARE @hDoc int exec sp_xml_preparedocument @hDoc OUTPUT,@empdata **UPDATE Employee** SET Employee.fname = XMLEmployee.fname, Employee.lname = XMLEmployee.lname FROM OPENXML(@hDoc, '/root/Employee') WITH Employee XMLEmployee WHERE Employee.eid = XMLEmployee.eid EXEC sp xml removedocument @hDoc SELECT * Employee from FOR XML AUTO
- 4. Create the following template. The template must be stored in the directory associated with the virtual name of template type (if you have created the sample **nwind** virtual directory, the template is stored in the template subdirectory of the virtual root directory). Save the template as UpdateEmployee.xml.

```
<root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<sql:header>
```

<sql:param name="empdata"><Employee/></sql:param> </sql:header> <sql:query>

```
exec sp_update_employee @empdata
</sql:query>
</root>
```

This is the HTML form:

<html>

<body>

<form name="Employee"

action="http://localhost/nwind/Template/UpdateEmployee.XML" method="POST">

This app assumes that client has this simple

XML document created using some other application and you want to update the tables based on the data in this document.

```
<input type=hidden name="contenttype" value="text/xml">
```

```
<textarea name="empdata" cols=50 rows=5>
```

<root>

<Employee eid="1" lname="Leverling" <Employee eid="2" lname="Peacock" fn <:/root>

</textarea>

```
<br><input type=Submit value="Submit">
```

</form>

</body>

</html>

See Also

IIS Virtual Directory Management for SQL Server

Executing Template Files Using HTTP

Using XPath Queries

Sample Visual Basic Application to Update Records Using OPENXML and ADO

The application is based on the assumption that the client has an XML document (created with some other application) that is to be used to apply updates to the database.

This example shows:

- Writing a simple Microsoft® Visual Basic® application to update the database using XML documents.
- Using ADO to execute XML templates.
- Creating and execute templates and pass parameters to the templates.
- Creating a rowset from an XML document using OPENXML.

These are the steps to create a working sample:

- 1. Create this table: CREATE TABLE Employee(eid int, fname varchar(20), lname
- 2. Add sample data: INSERT INTO Employee VALUES (1, 'Nancy', 'Davolio') INSERT INTO Employee VALUES (2, 'Andrew', 'Fuller')
- 3. Create the following stored procedure in the database: CREATE PROC update_employee @empdata nvarchar(4000) AS
 DECLARE @hDoc int exec sp_xml_preparedocument @hDoc OUTPUT,@empdata UPDATE Employee

```
SET
Employee.fname = XMLEmployee.fname,
Employee.lname = XMLEmployee.lname
FROM OPENXML(@hDoc, 'update/Employee')
WITH Employee XMLEmployee
WHERE Employee.eid = XMLEmployee.eid
EXEC sp_xml_removedocument @hDoc
```

- 4. Create a Visual Basic project (a standard EXE project is sufficient).
- 5. Add Microsoft ActiveX[®] Data Objects 2.6 Library to the project references.
- 6. Add the following code:

'The code uses ADO to establish a SQL Server connection and 'template to the server. The template executes a stored procedu '(update_employee) which accepts an XML document as input 'procedure uses OPENXML to shred the document and genera 'which is used to update the records in the Employee table. 'The template is then executed on the server and the resulting : 'is returned to the client. The stream contains the resulting XM 'document.

Dim cmd As New ADODB.Command Dim conn As New ADODB.Connection Dim strmIn As New ADODB.Stream Dim strmOut As New ADODB.Stream

' Open a connection to the SQL Server. conn.Provider = "SQLOLEDB" conn.Open "server=(local); database=Northwind; uid=sa; "

Set cmd.ActiveConnection = conn

' Build the command string in the form of an XML template. SQLxml = "<root xmlns:sql=""urn:schemas-microsoft-com:xi SQLxml = SQLxml & "exec update_employee N'<update><E SQLxml = SQLxml & "<Employee eid=""2"" lname=""Peaco SQLxml = SQLxml & "</sql:query></root>"

' Set the command dialect to XML. cmd.Dialect = "{5d531cb2-e6ed-11d2-b252-00c04f681b71}"

' Open the command stream and write our template to it. strmIn.Open strmIn.WriteText SQLxml strmIn.Position = 0

Set cmd.CommandStream = strmIn

' Execute the command, open the return stream, and read the r strmOut.Open strmOut.LineSeparator = adCRLF cmd.Properties("Output Stream").Value = strmOut cmd.Execute , , adExecuteStream strmOut.Position = 0 Debug.Print strmOut.ReadText

OLE DB Provider for SQL Server Extensions for XML

The Microsoft® OLE DB Provider for SQL Server (SQLOLEDB) supports a new dialect called DBGUID_MSSQLXML to execute XML templates with embedded queries (such as SQL FOR XML and XPath queries). Templates are valid XML documents containing one or more queries. The FOR XML and XPath queries return a document fragment. The templates act as a container for the resulting document fragment.

Setting an XML Command Using ICommandText

The **ICommandText::SetCommandText** and **ICommand::Execute** methods have been enhanced to allow XML documents to be set as command text, to execute the command and retrieve the results as a stream that can then be used in further processing, such as passing the XML document to DOM (Document Object Model).

The XML templates can be passed to the **ICommandText::SetCommandText** method. When XML templates are set as command text using **ICommandText::SetCommandText**, the consumer must pass DBGUID_MSSQLXML as the GUID of the command syntax. This new GUID indicates that the command text is an XML template.

The consumer must call **ICommand::Execute** to execute XML templates. To obtain an XML document as a result set, the **riid** is set to **IStream**, in which case the provider returns the result set as a stream.

Limitations of ICommandText

The template being passed to **ICommandText::SetCommandText** can be large. And if the template being executed is stored in a file, overhead is required to read the file, buffer its contents, and then set command text using **ICommandText::SetCommandText**.

In addition, the **ICommandText::SetCommandText** expects the command string to be a Unicode string. If the actual XML file is in some encoding,

additional overhead is required to convert the file to Unicode before passing it to **IcommandText::SetCommandText** as a command.

Setting an XML Command Using ICommandStream

The OLE DB (version 2.6) interface **ICommandStream**, although similar to **ICommandText**, passes a command as a stream object rather than as a string.

SQLOLEDB has implemented the optional **ICommandStream** interface on the command object. The **ICommandStream** interface allows you to pass a stream to the command object.

The **ICommandStream** interface allows a command to be in any encoding that the XML parser understands. Thus, when **ICommand::Execute** is called, the command text is read out of the stream directly and no conversion is required. Executing XML commands using **ICommandStream** interface is more efficient.

Both the **ICommandStream::GetCommandStream** and **ICommandStream::SetCommandStream** interfaces are implemented in SQLOLEDB.

For **ICommandStream**, the default dialect (DBGUID_DEFAULT) is DBGUID_MSSQLXML. The dialects supported by **ICommandStream::SetCommandStream** are provider-specific. SQLOLEDB supports DBGUID_MSSSQLXML only (DBGUID_SQL and DBGUID_XPATH are not supported.)

If you read from the stream returned by **GetCommandStream** before EXECUTE is called, **EXECUTE** may fail unless EXECUTE can read from the proper position in the stream.

Support for the OLE DB (Version 2.6) DBPROPSET_STREAM Property Set

SQLOLEDB has implemented DBPROPSET_STREAM property set (in the **Stream** property group), which includes these properties:

DBPROP_OUTPUTSTREAM

The value passed in this property is a Variant containing a pointer to either **IStream** or **ISequentialStream**. When this property is set,

ICommand::Execute returns results in the stream specified by this property. This avoids extra copies of the data because you can pass the stream to other users, such as the XML parser.

DBPROP_OUTPUTENCODING

This property specifies the requested encoding for the stream returned by the Execute method. Some of the commonly used encodings are UTF-8, ANSI, and Unicode. The UTF-8 is the default encoding if the value of this property is NULL.

Requesting ISequentialStream on ICommand::Execute

You can request **ISequentialStream** on **ICommand::Execute**.

While reading from a stream as long as there is data to read, **ISequentialStream::Read** will return S_OK. After the end of the stream is reached, a subsequent read will return S_FALSE, unless there were errors during the execution of the command. If there were any errors, DB_S_ERRORSOCCURED is returned only on the first read after the end of the

stream was reached. All the subsequent reads will return S_FALSE. In executing the command, if there are any errors, the errors are returned as

processing instructions (PIs) in the stream. All the errors are returned after the last read. Applications that do not have access to error objects can examine the stream contents for the PI containing the errors.

ISequentialStream is supported only when the selected result is a single-column rowset.

SQLOLEDB Provider-Specific Properties

To support XML-specific behavior, SQLOLEDB has implemented the following provider-specific properties in the DBPROPSET_SQLSERVERSTREAM property set (**Stream** property group). These properties allow you to specify the mapping schema against which an XPath query can be specified as a command, or to specify an XSL file to process the result. Some of these properties are useful for enhancing security and performance.

SSPROP_STREAM_BASEPATH

This property is used to specify the base path. This base path is used in resolving relative paths specified for the XSL file, mapping schema, or external schema references in a template.

SSPROP_STREAM_MAPPINGSCHEMA

This property is used for specifying a schema for the XPath queries. The path specified can be relative or absolute. If the path specified is relative, the base path specified in SSPROP_STREAM_BASEPATH is used to resolve the relative path. If the base path is not specified, the relative path is relative to the current directory

SSPROP_STREAM_XSL

This property is used for specifying an XSL file. The path specified can be relative or absolute. If the path specified is relative, the base path specified in SSPROP_STREAM_BASEPATH is used to resolve the relative path. If the base path is not specified, the relative path is relative to the current directory.

SSPROP_STREAM_CONTENTTYPE

If an XSL style sheet is applied to the result, the media-type property on <xsl:output> in the XSL file is returned as the value of this property.

SSPROP_STREAM_FLAGS

This property is used to specify certain security restrictions. For example, you may not want to allow URL references to files or absolute paths to files (such as external sites). You may not want to allow queries in the templates. The property can be assigned values STREAM_FLAGS_DISALLOW_URL, STREAM_FLAGS_DISALLOW_ABSOLUTE_PATH, or STREAM_FLAGS_DISALLOW_QUERY.

For more information about these properties, see <u>Initialization and Authorization</u> <u>Properties</u>.

Using ICommandStream to Set an XML Command

The OLE DB (version 2.6) interface **ICommandStream** passes a command as a stream object rather than as a string.

This interface allows command to be in any encoding that the XML parser understands. When **ICommand::Execute** is called, the command text is read out of the stream directly and no conversion is required. Therefore, executing XML commands using **ICommandStream** interface is more efficient.

To set XML as a command using ICommandStream and retrieving the result as an XML document