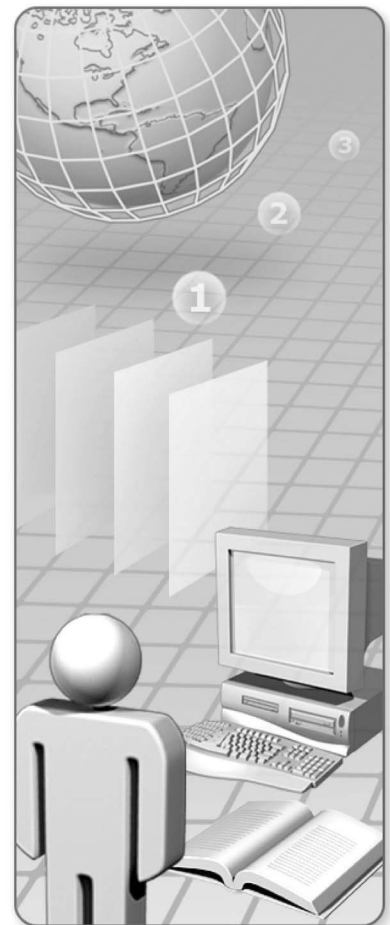


Module 3

Using XML

Contents:

Lesson 1: Retrieving XML by Using FOR XML	3-2
Lesson 2: Shredding XML by Using OPENXML	3-19
Lesson 3: Using the xml Data Type	3-30
Lab: Working with XML	3-45



Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

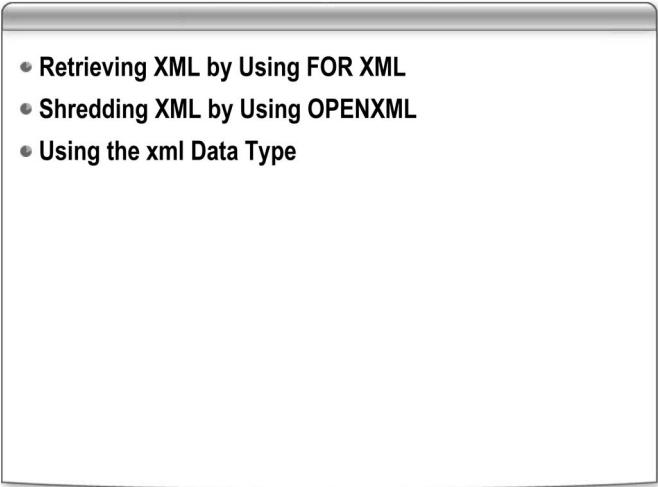
The names of manufacturers, products, or URLs are provided for informational purposes only and Microsoft makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of Microsoft of the manufacturer or product. Links are provided to third party sites. Such sites are not under the control of Microsoft and Microsoft is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. Microsoft is not responsible for webcasting or any other form of transmission received from any linked site. Microsoft is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Microsoft of the site or the products contained therein.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

©2006 Microsoft Corporation. All rights reserved.

Microsoft, JScript, MSDN, Outlook, PowerPoint, Visual Basic, Visual C#, Visual C++, Visual FoxPro, Windows, and Windows Server are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

- 
- Retrieving XML by Using FOR XML
 - Shredding XML by Using OPENXML
 - Using the xml Data Type

***** Illegal for non-trainer use *****

Module objectives

After completing this module, students will be able to:

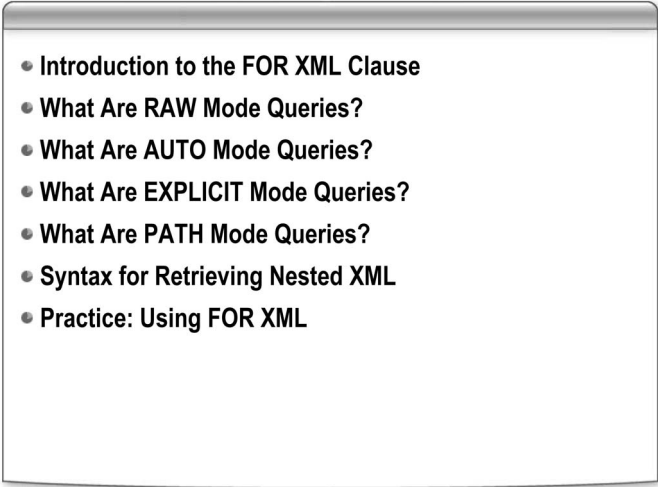
- Retrieve XML by using FOR XML.
- Shred XML by using OPENXML.
- Use the **xml** data type.

Introduction

Working with Extensible Markup Language (XML) is a common requirement of many of today's applications. There are many cases when an application developer must transform data between XML and relational formats, and even store XML and manipulate data natively in a relational database.

In this module, you will learn how to retrieve data from the database in XML format by using the FOR XML clause. You will also learn how to shred XML documents for storage in relational tables by using the OPENXML function. Finally, you will learn how to use the Microsoft® SQL Server™ 2005 **xml** data type to store XML data natively in the database and how to perform operations such as querying and modifying the **xml** data.

Lesson 1: Retrieving XML by Using FOR XML

- 
- Introduction to the FOR XML Clause
 - What Are RAW Mode Queries?
 - What Are AUTO Mode Queries?
 - What Are EXPLICIT Mode Queries?
 - What Are PATH Mode Queries?
 - Syntax for Retrieving Nested XML
 - Practice: Using FOR XML

***** Illegal for non-trainer use *****

Lesson objectives

After completing this lesson, students will be able to:

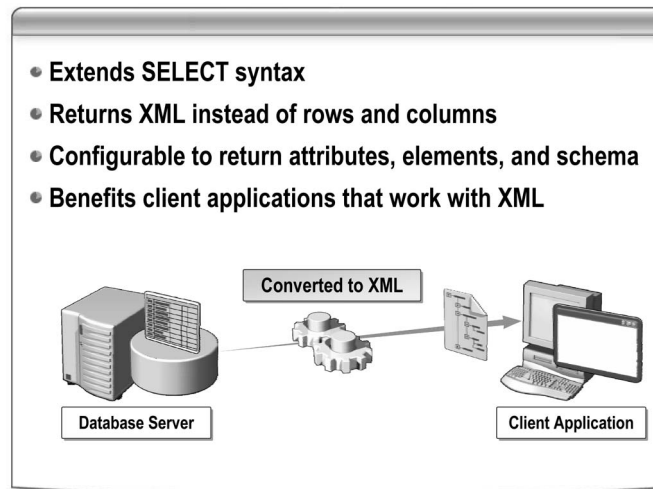
- Describe the purpose of the FOR XML clause.
- Describe RAW mode queries and explain their syntax.
- Describe AUTO mode queries and their syntax.
- Describe EXPLICIT mode queries and their syntax.
- Describe PATH mode queries and their syntax.
- Describe the syntax for retrieving nested XML.

Introduction

The FOR XML clause is central to XML data retrieval in SQL Server 2005. This clause instructs the SQL Server query engine to return the data as an XML stream rather than as a rowset. Application developers can build solutions that extract XML business documents such as orders, invoices, or catalogs directly from the database.

This lesson describes how to use the FOR XML clause and the various options that you can use to configure the format of the returned XML.

Introduction to the FOR XML Clause



***** Illegal for non-trainer use *****

Introduction

You can use the FOR XML clause in a Transact-SQL SELECT statement to retrieve data as XML instead of rows and columns. You control the format of the XML by specifying one of four modes: RAW, AUTO, EXPLICIT, or PATH. In addition, you can specify various options to control the output.

The FOR XML syntax

The FOR XML clause is appended to the SELECT statement, as shown in the following syntax.

```
FOR XML
{
    { RAW [ ( 'ElementName' ) ] | AUTO }
    [
        <CommonDirectives>
        [ , { XMLDATA | XMLSCHEMA [ ( 'TargetNamespaceURI' ) ] } ]
        [ , ELEMENTS [ XSINIL | ABSENT ] ]
    ]
| EXPLICIT
    [
        <CommonDirectives>
        [ , XMLDATA ]
    ]
| PATH [ ( 'ElementName' ) ]
    [
        <CommonDirectives>
        [ , ELEMENTS [ XSINIL | ABSENT ] ]
    ]
}

<CommonDirectives> ::=
[ , BINARY BASE64 ]
[ , TYPE ]
[ , ROOT [ ( 'RootName' ) ] ]
```

The most commonly used FOR XML modes and options are described in the following table.

Mode/option	Description
RAW mode	Transforms each row in the result set into an XML element that has a generic identifier row as the element tag. You can optionally specify a name for the row element when you use this directive.
AUTO mode	Returns query results in a simple, nested XML tree. Each table in the FROM clause for 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 element attributes.
EXPLICIT mode	A custom format specified in the query formats the resulting XML data.
PATH mode	Provides a simpler way to mix elements and attributes, and to introduce additional nesting for representing complex properties.
ELEMENTS option	Returns columns as subelements rather than as attributes for RAW, AUTO, and PATH modes.
BINARY BASE64 option	Returns binary data fields, such as images, as base-64-encoded binary.
ROOT option	Adds a top-level element to the resulting XML. When a query returns multiple rows as XML, by default the results are not enclosed in a single root element. XML data without a single root element is known as a <i>fragment</i> . However, if you need to return a fully well-formed XML document with a single root element, you should specify this option. You can optionally specify a name for this root element.
TYPE option	Returns the query results as the xml data type.
XMLDATA option	Returns an inline XML-Data Reduced (XDR) schema.
XMLSCHEMA option	Returns an inline World Wide Web Consortium (W3C) XML Schema (XSD).

Examples of using FOR XML

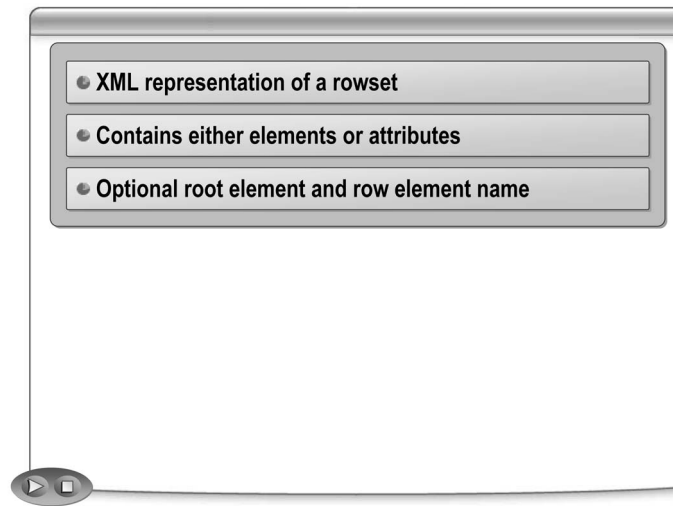
There are a number of situations in which you might want to retrieve data in XML format rather than as a rowset. For example, consider the following data access scenarios:

- **Retrieving data to be published in a Web site** By retrieving the data as XML, you can apply an Extensible Stylesheet Language Transformations (XSLT) style sheet to render the data as Hypertext Markup Language (HTML). You can also apply different style sheets to the same XML data to generate alternative presentation formats to support different client devices without rewriting any data access logic.
- **Retrieving data to exchange with a trading partner** XML is a natural format for data that you want to send to a trading partner. By retrieving business data in XML format, you can easily integrate your systems with external organizations, no matter what data technologies they use internally.

For More Information For more information about XML in general, see “Understanding XML” on the MSDN® Web site.

For more information about the FOR XML clause, see “Constructing XML Using FOR XML” in SQL Server Books Online.

What Are RAW Mode Queries?



***** Illegal for non-trainer use *****

Introduction

You use RAW mode queries to retrieve an XML representation of a rowset. Applications can process the XML in the generic format or apply an XSLT style sheet to transform the XML into the required business document format or user interface representation. Consider the following features of RAW mode:

- An element represents each row in the result set that the query returns.
- An attribute with the same name as the column name or alias used in the query represents each column in the result set, unless the ELEMENTS option is specified, in which case each column is mapped to a subelement of the row element.
- RAW mode queries can include aggregated columns and GROUP BY clauses.

Retrieving data in generic row elements

The following example shows how you can retrieve an XML fragment containing order data by using a FOR XML query in RAW mode.

```
SELECT      Cust.CustomerID CustID, CustomerType, SalesOrderID
FROM        Sales.Customer Cust JOIN Sales.SalesOrderHeader [Order]
           ON Cust.CustomerID = [Order].CustomerID
ORDER BY    Cust.CustomerID
FOR XML RAW
```

This query produces an XML fragment in the format that contains generic <row> elements, shown in the following example.

```
<row CustID="1" CustomerType="S" SalesOrderID="43860"/>
<row CustID="1" CustomerType="S" SalesOrderID="44501"/>
```

Notice that the **CustomerID** column uses the alias **CustID**, which determines the attribute name.

Retrieving data as elements

The following example shows how you can retrieve the same data as elements instead of attributes by specifying the ELEMENTS option.

```
SELECT      Cust.CustomerID CustID, CustomerType, SalesOrderID
FROM        Sales.Customer Cust JOIN Sales.SalesOrderHeader [Order]
           ON Cust.CustomerID = [Order].CustomerID
ORDER BY    Cust.CustomerID
FOR XML RAW, ELEMENTS
```

This query produces an XML fragment in the format shown in the following example.

```
<row>
  <CustID>1</CustID>
  <CustomerType>S</CustomerType>
  <SalesOrderID>43860</SalesOrderID>
</row>
<row>
  <CustID>1</CustID>
  <CustomerType>S</CustomerType>
  <SalesOrderID>44501</SalesOrderID>
</row>
```

Retrieving data by using a root element and a customized row element name

The following example shows how you can retrieve the same data by using a root element specified with the ROOT option and modify the row element name by using the RAW mode optional argument.

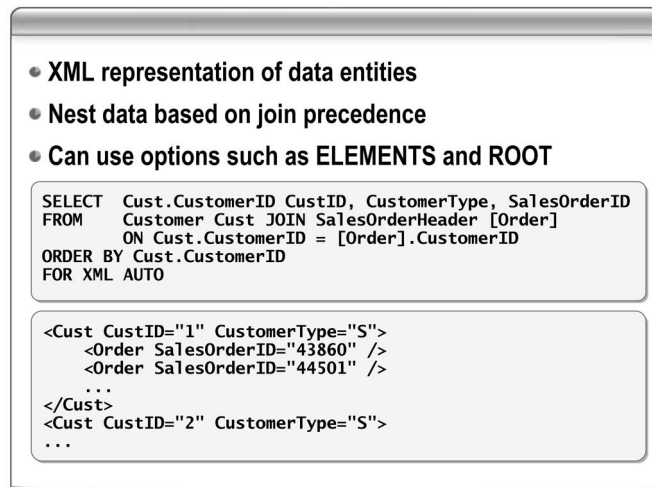
```
SELECT      Cust.CustomerID CustID, CustomerType, SalesOrderID
FROM        Sales.Customer Cust JOIN Sales.SalesOrderHeader [Order]
           ON Cust.CustomerID = [Order].CustomerID
ORDER BY    Cust.CustomerID
FOR XML RAW('Order'), ROOT('Orders')
```

This query produces a well-formed XML document in the format shown in the following example.

```
<Orders>
  <Order CustID="1" CustomerType="S" SalesOrderID="43860"/>
  <Order CustID="1" CustomerType="S" SalesOrderID="44501"/>
</Orders>
```

Note that if the ELEMENTS option was also specified, the resulting rows would contain elements instead of attributes.

What Are AUTO Mode Queries?



***** Illegal for non-trainer use *****

Introduction

AUTO mode queries produce XML representations of data entities. Consider the following features of AUTO mode:

- Each row returned by the query is represented by an XML element with the same name as the table from which it was retrieved (or the alias used in the query).
- Each JOIN in the query results in a nested XML element, reducing the duplication of data in the resulting XML fragment. The order of the JOIN statements affects the order of the nested elements.
- To ensure that the child elements are collated correctly with their parent, use an ORDER BY clause to return the data in the correct hierarchical order.
- Each column in the result set is represented by an attribute, unless the ELEMENTS option is specified, in which case, each column is represented by a child element.
- Aggregated columns and GROUP BY clauses are not supported in AUTO mode queries (although you can use an AUTO mode query to retrieve aggregated data from a view that uses a GROUP BY clause).

Retrieving nested data by using AUTO mode

The following example shows how you can use an AUTO mode query to return an XML fragment containing a list of orders.

```
SELECT      Cust.CustomerID CustID, CustomerType, SalesOrderID
FROM        Sales.Customer Cust JOIN Sales.SalesOrderHeader [Order]
            ON Cust.CustomerID = [Order].CustomerID
ORDER BY    Cust.CustomerID
FOR XML AUTO
```

This query produces an XML fragment in the format shown in the following example.

```
<Cust CustID="1" CustomerType="S">
  <Order SalesOrderID="43860"/>
  <Order SalesOrderID="44501"/>
</Cust>
```

Note that the **CustomerID** column and the **Customer** and **SalesOrderHeader** tables use aliasing to determine the attribute and element names.

Retrieving data as elements

The following example shows how you can retrieve the same data as elements instead of attributes by specifying the **ELEMENTS** option.

```
SELECT      Cust.CustomerID CustID, CustomerType, SalesOrderID
FROM        Sales.Customer Cust JOIN Sales.SalesOrderHeader [Order]
           ON Cust.CustomerID = [Order].CustomerID
ORDER BY    Cust.CustomerID
FOR XML AUTO, ELEMENTS
```

This query produces an XML fragment in the format shown in the following example.

```
<Cust>
  <CustID>1</CustID>
  <CustomerType>S</CustomerType>
  <Order>
    <SalesOrderID>43860</SalesOrderID>
  </Order>
  <Order>
    <SalesOrderID>44501</SalesOrderID>
  </Order>
</Cust>
...
```

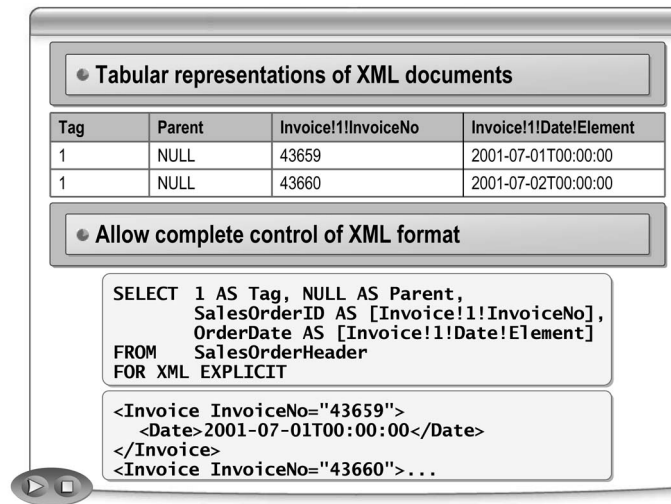
As for RAW mode, you can also use the **ROOT** option, as shown in the following example.

```
SELECT      Cust.CustomerID CustID, CustomerType, SalesOrderID
FROM        Sales.Customer Cust JOIN Sales.SalesOrderHeader [Order]
           ON Cust.CustomerID = [Order].CustomerID
ORDER BY    Cust.CustomerID
FOR XML AUTO, ELEMENTS, ROOT('Orders')
```

This query produces an XML document in the format shown in the following example.

```
<Orders>
  <Cust>
    <CustID>1</CustID>
    <CustomerType>S</CustomerType>
    <Order>
      <SalesOrderID>43860</SalesOrderID>
    </Order>
    <Order>
      <SalesOrderID>44501</SalesOrderID>
    </Order>
  </Cust>
  ...
</Orders>
```

What Are EXPLICIT Mode Queries?



***** Illegal for non-trainer use *****

Introduction

Sometimes the business documents you must exchange with trading partners require an XML format that cannot be retrieved by using RAW or AUTO mode queries. When data from a table is mapped to an XML element, the columns in the table can be represented as

- The value of the element.
- An attribute.
- A child element.

Universal tables

The key to understanding the retrieval of custom XML documents is the concept of a universal table. A universal table is a tabular representation of an XML document. Each row in a universal table represents data that will be represented as an element in the resulting XML document.

The first two columns in a universal table define the hierarchical position in the resulting XML document of the element that contains the data for the row. These columns are:

Tag. A numerical value that uniquely identifies the tag for the element that contains the data in this row

Parent. A numerical value that identifies the immediate parent tag for this element

Each different XML tag in the resulting document that maps to a table or view in the database must be represented by a different **Tag** value in the universal table. The **Parent** value determines the hierarchical position of the tag in the resulting document. Tags at the top level of the XML fragment (and therefore having no immediate parent element) have a parent value of NULL. For example, the XML document described earlier contains an **Invoice** element with no parent and a **LineItem** element, which is a child of the **Invoice** element. In the universal table, the tags for the two elements are assigned an arbitrary **Tag** number to identify them, and the **Parent** column is used to define how the elements are nested.

Defining column mappings in a universal table

The remaining columns in a universal table contain the data that will be represented in the document. The column name specifies whether the data will be represented as an element value, an attribute, or a child element. Columns for data in a universal table have a name with up to four parts, in the following format.

ElementName! TagNumber! AttributeName! Directive

The following table describes the parts of a column name.

Name part	Description
<i>ElementName</i>	The name of the element that contains the data in this row.
<i>TagNumber</i>	The unique number that identifies the tag (as specified in the Tag column). The same <i>ElementName</i> must be used consistently with a given <i>TagNumber</i> .
<i>AttributeName</i>	(Optional) The name of the attribute or child element that represents the data in this column. If this column is unspecified, the data is represented as an element value.
<i>Directive</i>	(Optional) Additional formatting instructions that represent the data as a child element or other XML-specific format.

For example, the following universal table defines an XML invoice document.

Tag	Parent	Invoice!1!InvoiceNo	Invoice!1!Date!Element
1	NULL	43659	2001-07-01T00:00:00
1	NULL	43660	2001-07-01T00:00:00

The XML represented by this table looks like the following example.

```
<Invoice InvoiceNo="43659">
  <Date>2001-07-01T00:00:00</Date>
</Invoice>
<Invoice InvoiceNo="43660">
  <Date>2001-07-01T00:00:00</Date>
</Invoice>
```

Creating a query to construct the universal table

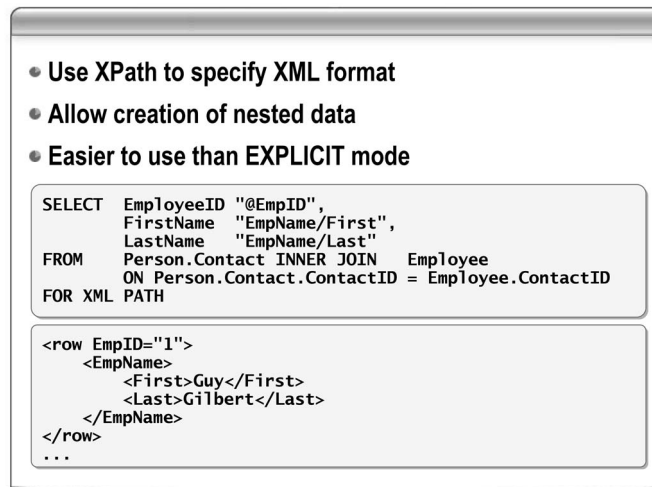
After you have determined the universal table you require to retrieve the desired XML document, you can construct the Transact-SQL query necessary to generate the table by using aliases to name the columns. You can assign the values for the **Tag** and **Parent** columns explicitly, as shown in the following example.

```
SELECT 1 AS Tag,
       NULL AS Parent,
       Sales.SalesOrderID AS [Invoice!1!InvoiceNo],
       OrderDate AS [Invoice!1!Date!Element]
FROM    SalesOrderHeader
FOR XML EXPLICIT
```

This example creates the XML shown earlier in this topic by combining attributes for the **SalesOrderID** and elements for the **OrderDate** columns.

For More Information For more information about EXPLICIT mode and universal tables, see “Using EXPLICIT Mode” in SQL Server Books Online.

What Are PATH Mode Queries?



***** Illegal for non-trainer use *****

Introduction

PATH mode produces customized XML by using a syntax named XPath to map values to attributes, elements, subelements, text nodes, and data values. This ability to map relational columns to XML nodes makes it possible to generate complex XML without the complexity of an EXPLICIT mode query.

Consider the following features of XPath syntax:

- XML nodes in a tree are expressed as paths, separated by slash marks (/).
- Attributes are indicated with an at sign (@) prefix.
- Relative paths can be indicated by using a single dot (.) to represent the current node and a double dot (..) to represent the current node's parent.

Retrieving data by using PATH mode

The following example shows how you can use a PATH mode query to return an XML fragment containing a list of employees.

```
SELECT EmployeeID "@EmpID",
       FirstName "EmpName/First",
       LastName "EmpName/Last"
FROM   Person.Contact INNER JOIN
       Employee ON Person.Contact.ContactID = Employee.ContactID
FOR XML PATH
```

This query produces an XML fragment in the format shown in the following example.

```
<row EmpID="1">
  <EmpName>
    <First>Guy</First>
    <Last>Gilbert</Last>
  </EmpName>
</row>
```

```
<row EmpID="2">
  <EmpName>
    <First>Kevin</First>
    <Last>Browne</Last>
  </EmpName>
</row>
```

Notice that the **EmployeeID** column is mapped to the **EmpID** attribute with an at sign (@). The **FirstName** and **LastName** columns are mapped as subelements of the **EmpName** element with the slash mark (/).

Modifying the row element name

The following example shows how you can use the optional *ElementName* argument to the PATH mode query to modify the default row element name.

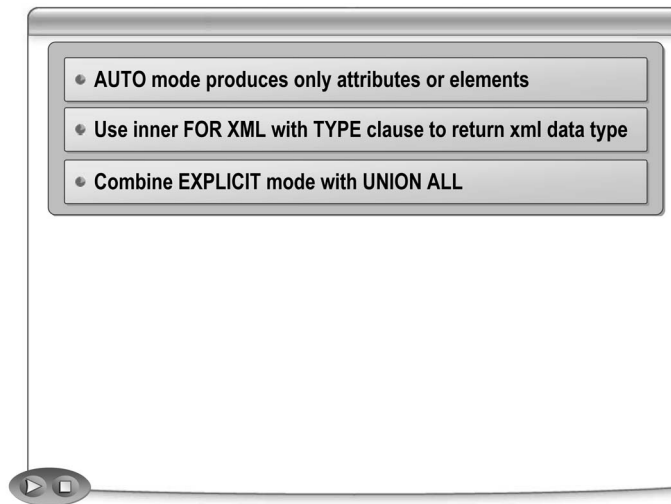
```
SELECT EmployeeID "@EmpID",
       FirstName  "EmpName/First",
       LastName   "EmpName/Last"
FROM   Person.Contact INNER JOIN
       Employee ON Person.Contact.ContactID = Employee.ContactID
FOR XML PATH('Employee')
```

This query produces an XML fragment in the format shown in the following example.

```
<Employee EmpID="1">
  <EmpName>
    <First>Guy</First>
    <Last>Gilbert</Last>
  </EmpName>
</Employee>
<Employee EmpID="2">
  <EmpName>
    <First>Kevin</First>
    <Last>Browne</Last>
  </EmpName>
</Employee>
```

Notice that the **EmployeeID** column is mapped to the **EmpID** attribute (@EmpID), and the **FirstName** and **LastName** columns are mapped as subelements of the **EmpName** element (**EmpName/First** and **EmpName/Last**).

Syntax for Retrieving Nested XML



***** Illegal for non-trainer use *****

Introduction

Nesting XML enables you to represent parent/child relationships as an XML hierarchy—for example, customers and their orders, or orders and their order items. There are several ways to nest XML elements by using the FOR XML clause, including:

- Defining joins in AUTO mode queries
- Specifying the TYPE option in subqueries to return **xml** values
- Combining multiple universal tables by using the UNION ALL statement in an EXPLICIT mode query

Using AUTO mode to return nested XML

When you define a JOIN in an AUTO mode query, SQL Server nests the resulting elements that map to the tables in the order in which they appear in the SELECT clause.

AUTO mode enables you to specify whether columns are mapped to elements or attributes within the XML fragment by using the ELEMENTS option. However, you can specify only one or the other option for the entire fragment, providing only limited control over the output format.

The following example outputs all columns as attributes because the ELEMENTS option is not specified.

```
SELECT      Cust.CustomerID CustID, CustomerType, SalesOrderID
FROM        Sales.Customer Cust JOIN Sales.SalesOrderHeader [Order]
           ON Cust.CustomerID = [Order].CustomerID
ORDER BY    Cust.CustomerID
FOR XML AUTO
```

This query produces an XML fragment in the format shown in the following example.

```
<Cust CustID="1" CustomerType="S">
  <Order SalesOrderID="43860"/>
  <Order SalesOrderID="44501"/>
</Cust>
```

Adding the ELEMENTS option to the preceding Transact-SQL statement would produce the following output.

```
<Cust>
  <CustID>1</CustID>
  <CustomerType>S</CustomerType>
  <Order>
    <SalesOrderID>43860</SalesOrderID>
  </Order>
  <Order>
    <SalesOrderID>44501</SalesOrderID>
  </Order>
</Cust>
...
```

Using TYPE to return the xml data type in a subquery

SQL Server 2005 includes the **xml** data type. Specifying the TYPE directive in a FOR XML query returns the results as an **xml** value instead of as a **varchar** string. The most significant impact of this is the ability to nest FOR XML queries to return multilevel XML results in AUTO and RAW mode queries. The following example shows how to use the TYPE directive to nest FOR XML queries.

```
SELECT Name CategoryName,
       (SELECT Name SubCategoryName
        FROM   Production.ProductSubCategory SubCategory
        WHERE  SubCategory.ProductCategoryID=Category.ProductCategoryID
        FOR XML AUTO, TYPE, ELEMENTS)
FROM   Production.ProductCategory Category
FOR XML AUTO
```

The resulting output from the preceding query is shown in this example.

```
<Category CategoryName="Accessories">
  <SubCategory>
    <SubCategoryName>Bike Racks</SubCategoryName>
  </SubCategory>
  <SubCategory>
    <SubCategoryName>Bike Stands</SubCategoryName>
  </SubCategory>
</Category>
...
```

Alternatively, if the outside query did not contain a FOR XML AUTO clause, the output would appear with the subcategories as a column containing **xml** data, as follows.

CategoryName	
Accessories	<SubCategory><SubCategoryName>Bike Racks</SubCategoryName></SubCategory><SubCategory><SubCategoryName>Bike Stands</SubCategoryName></SubCategory>

There is no column name for the **xml** data because the query does not specify an alias for the inner SELECT.

For More Information For more information about using the TYPE directive, see “TYPE Directive in FOR XML Queries” in SQL Server Books Online.

Nesting tables by using EXPLICIT mode

To use EXPLICIT mode to generate an XML document containing multiple tags, you must write individual queries for each tag and then merge them by using a UNION ALL operator.

Consider the following requirements when using the UNION ALL operator in EXPLICIT mode queries:

- Each query must return a consistent set of columns so that the results can be merged successfully. You can assign NULL entries to columns not used by the current query.
- You should use an ORDER BY clause to merge the results into the correct XML hierarchy.

For example, an XML invoice document might require the following format.

```
<Invoice InvoiceNo="43659">
  <Date>2001-07-01T00:00:00</Date>
  <LineItem ProductID="709">Bike Socks, M</LineItem>
  <LineItem ProductID="711">Helmet, Blue</LineItem>
</Invoice>
```

Because this format contains two XML elements (**Invoice** and **LineItem**) that are mapped to tables or views, you must use two queries to retrieve the data.

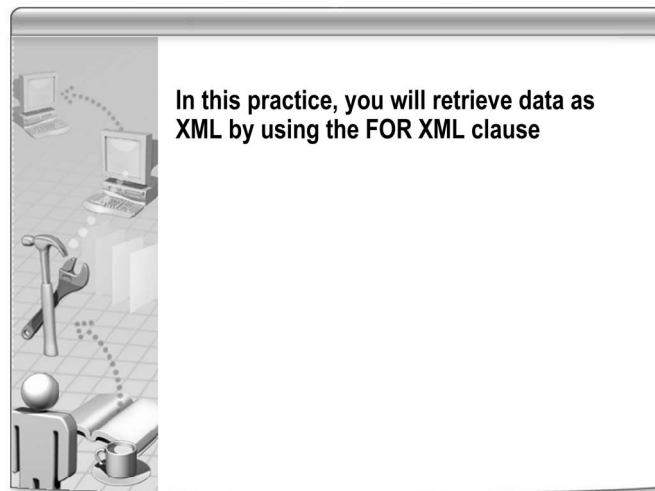
The first query must produce an **Invoice** element that contains an **InvoiceNo** attribute and a **Date** child element. Because the query will be combined with another query by using the UNION ALL operator, you must also specify columns as placeholders for the **ProductID** and **Name** fields that are returned in the other query.

The second query must retrieve **LineItem** elements under the **Invoice** element. These contain the **ProductID** and the **Name** assigned to the value of the **LineItem** element. You must also retrieve the **OrderID** order to join items to their orders.

The following example shows the two queries needed to retrieve the **Invoice** and **LineItem** elements combined by using the UNION operator to retrieve the preceding XML document.

```
SELECT 1 AS Tag,
       NULL AS Parent,
       SalesOrderID AS [Invoice!1!InvoiceNo],
       OrderDate AS [Invoice!1!Date!Element],
       NULL AS [LineItem!2!ProductID],
       NULL AS [LineItem!2]
FROM    Sales.SalesOrderHeader
UNION ALL
SELECT 2 AS Tag, 21
       1 AS Parent,
       OrderDetail.SalesOrderID,
       NULL,
       OrderDetail.ProductID,
       Product.Name
FROM    Sales.SalesOrderDetail OrderDetail JOIN
       Sales.SalesOrderHeader OrderHeader
       ON OrderDetail.SalesOrderID= OrderHeader.SalesOrderID
       JOIN Production.Product Product
       ON OrderDetail.ProductID = Product.ProductID
ORDER BY [Invoice!1!InvoiceNo], [LineItem!2!ProductID]
FOR XML EXPLICIT
```

Practice: Using FOR XML



***** Illegal for non-trainer use *****

Goals The goal of this practice is to enable you to use the FOR XML clause and to use the different modes and options.

Preparation Ensure that virtual machine 2779A-MIA-SQL-03 is running and that you are logged on as **Student**.

If a virtual machine has not been started, perform the following steps:

1. Close any other running virtual machines.
2. Start the virtual machine.
3. In the **Log On to Windows** dialog box, complete the logon procedure by using the user name **Student** and the password **Pa\$\$w0rd**.

To use RAW mode ► Perform the following steps to retrieve XML by using the FOR XML clause in RAW mode:

1. Use Microsoft Windows® Explorer to view the contents of the D:\Practices folder.
2. Double-click **FORXML.sql** to open it in SQL Server Management Studio.
3. In the **Connect to Database Engine** dialog box, specify the values in the following table, and then click **Connect**.

Property	Value
Server type	Database Engine
Server name	MIAMI
Authentication	Windows Authentication

4. Select the query under the comment **RAW mode**, and on the toolbar, click **Execute**.
5. Review the query results. Notice that the XML contains a **row** element for each row in the result set and an attribute for each column.

Tip You can display the results in an XML Editor window by clicking the data in the first cell of the results table. Close the XML Editor window when you have viewed the results.

6. Select the query under the comment **ELEMENTS with RAW mode**, and on the toolbar, click **Execute**.
7. Review the query results. Notice that the XML returned is element-centric (that is, it contains a child element for each column).
8. Select the query under the comment **Named element in RAW mode**, and on the toolbar, click **Execute**.
9. Review the query results. Notice that the resultant XML includes a **Product** element for each row instead of the default **row** element.
10. Select the query under the comment **root element**, and on the toolbar, click **Execute**.
11. Review the query results. Notice that the results are enclosed in a **Products** root element.
12. Select the query under the comment **Join in RAW mode**, and on the toolbar, click **Execute**.
13. Review the query results. Notice that the resultant XML includes a **row** element for each row in the result set; no attempt is made to nest the XML into a parent/child hierarchy.

To use AUTO mode

- You must perform the following steps to retrieve XML by using the FOR XML clause in AUTO mode:
1. Select the query under the comment **AUTO mode**, and on the toolbar, click **Execute**.
 2. Review the query results. Notice that AUTO mode automatically uses the table name or alias for the row element name.
 3. Select the query under the comment **Join in AUTO mode**, and on the toolbar, click **Execute**.
 4. Review the query results. Notice that AUTO mode automatically nests child elements under parent elements.
 5. Select the query under the comment **Nested XML with TYPE**, and on the toolbar, click **Execute**.
 6. Review the query results. Notice that the resultant XML consists of the results of one FOR XML query nested within the results of another. (You must scroll through the results because not all products have a review associated with them.)

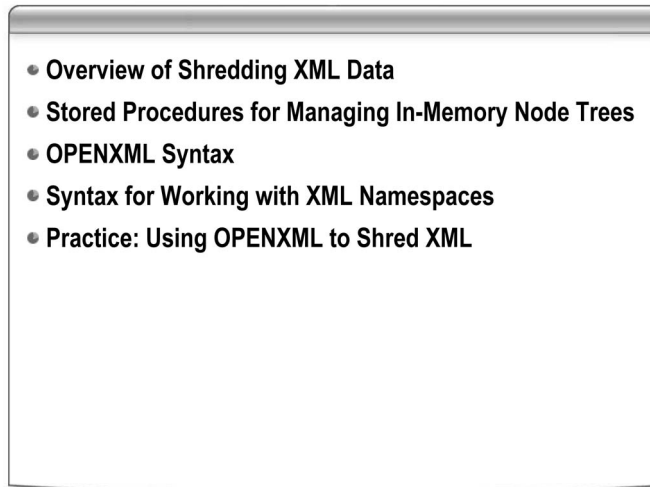
To use EXPLICIT mode

- You must perform the following steps to retrieve data by using PATH mode:
1. Select the query under the comment **EXPLICIT Mode**, and on the toolbar, click **Execute**.
 2. Review the query results. Notice that the resultant XML consists of elements, attributes, and values determined by the universal table defined in the query.
 3. Select the query under the comment **Nesting in EXPLICIT Mode**, and on the toolbar, click **Execute**.
 4. Review the query results. Notice that the resultant XML contains nested results with a mixture of elements, attributes, and values.

To use PATH mode

- You must perform the following steps to retrieve data by using PATH mode:
1. Select the query under the comment **PATH mode**, and on the toolbar, click **Execute**.
 2. Review the query results. Notice that the resultant XML consists of elements, attributes, and values defined by using XPath-like expressions in the query.
 3. Close SQL Server Management Studio. Click **No** if prompted to save files.

Lesson 2: Shredding XML by Using OPENXML



***** Illegal for non-trainer use *****

Lesson objectives

After completing this lesson, students will be able to:

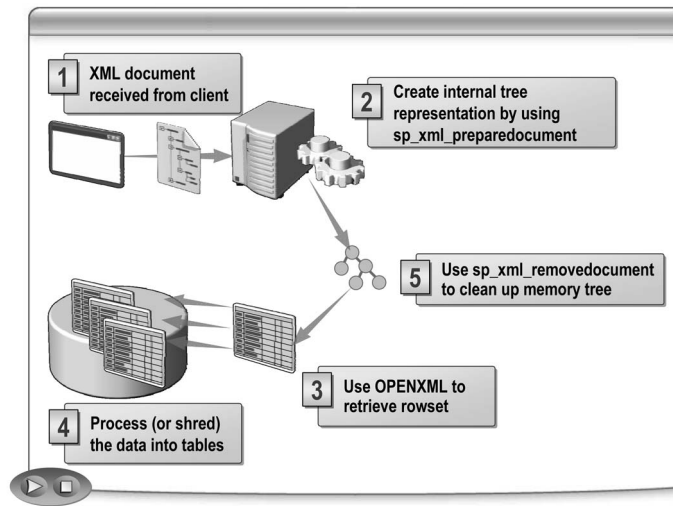
- Describe the purpose of OPENXML.
- Describe the stored procedures that are used to manage an in-memory node tree.
- Describe the OPENXML syntax.
- **Describe the syntax for shredding XML that contains namespaces.**

Introduction

A rowset contains the tabular result set for a query. In a trading partner integration scenario, you might need to generate a rowset from an XML document. For example, a retailer might send orders to a supplier as XML documents. The supplier must then generate rowsets from the XML to insert the data into one or more tables in the database. The process of transforming XML data to a rowset is known as “shredding” the XML data.

In this lesson, you will learn how to shred XML by using the OPENXML function and its related stored procedures.

Overview of Shredding XML Data



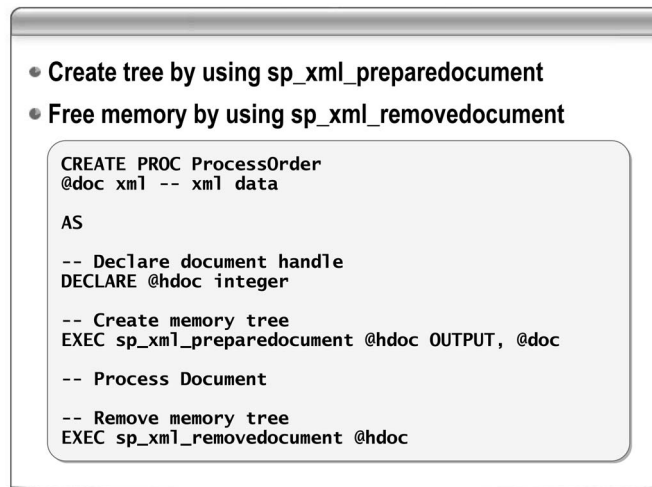
***** Illegal for non-trainer use *****

Process for shredding XML data

Processing XML data as a rowset involves the following five steps:

- 1. Receive an XML document.** When an application receives an XML document, it can process the document by using Transact-SQL code. For example, when a supplier receives an XML order from a retailer, the supplier logs the order in a SQL Server database. Usually, the Transact-SQL code to process the XML data is implemented in the form of a stored procedure, and the XML string is passed as a parameter.
- 2. Generate an internal tree representation.** Use the `sp_xml_preparedocument` stored procedure to parse the XML document and transform it into an in-memory tree structure before processing the document. The tree is conceptually similar to a Document Object Model (DOM) representation of an XML document. You can use only a valid, well-formed XML document to generate the internal tree.
- 3. Retrieve a rowset from the tree.** You use the OPENXML function to generate an in-memory rowset from the data in the tree. Use XPath query syntax to specify the nodes in the tree to be returned in the rowset.
- 4. Process the data from the rowset.** Use the rowset created by OPENXML to process the data, in the same way that you would use any other rowset. You can select, update, or delete the data by using Transact-SQL statements. The most common use of OPENXML is to insert rowset data into permanent tables in a database. For example, an XML order received by a supplier might contain data that must be inserted into **SalesOrderHeader** and **SalesOrderDetail** tables.
- 5. Destroy the internal tree when it is no longer required.** Because the tree structure is held in memory, use the `sp_xml_removedocument` stored procedure to free the memory when the tree is no longer required.

Stored Procedures for Managing In-Memory Node Trees



***** Illegal for non-trainer use *****

Introduction

Before you can process an XML document by using Transact-SQL statements, you must parse the document and transform it into an in-memory tree structure.

Creating the tree by using `sp_xml_preparedocument`

The `sp_xml_preparedocument` stored procedure parses an XML document and generates an internal tree representation of the document. The following table describes the parameters for the `sp_xml_preparedocument` system stored procedure.

Parameter	Description
<code>xmltext</code>	The original XML document to be processed. This parameter can accept any of the following text data types: nchar , varchar , nvarchar , text , ntext , or xml .
<code>hdoc</code>	A handle to the parsed XML tree.
<code>xpath_namespaces</code>	(Optional) XML namespace declarations that are used in row and column XPath expressions in OPENXML statements.

The following example shows how to use the `sp_xml_preparedocument` system stored procedure to parse an XML document that has been passed to a new, custom stored procedure.

```
CREATE PROC ProcessOrder @doc xml
AS
DECLARE @hdoc integer
EXEC sp_xml_preparedocument @hdoc OUTPUT, @doc
```

**Removing the tree by
using
sp_xml_removedocument**

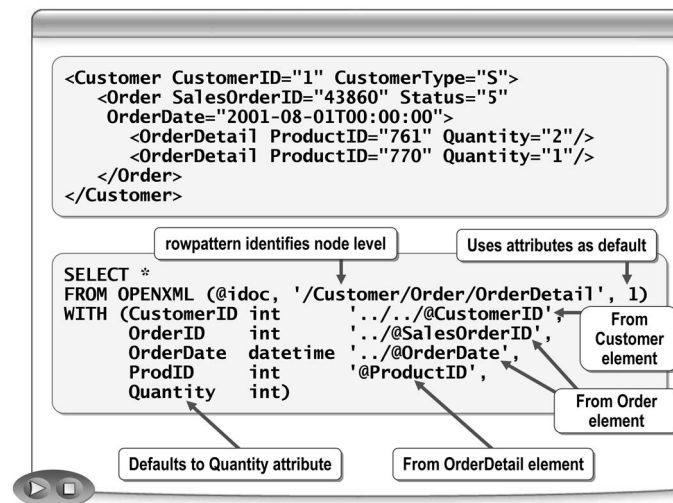
SQL Server stores parsed documents in the internal cache. To avoid running out of memory, use the **sp_xml_removedocument** system stored procedure to release the document handle and destroy the tree structure when it is no longer required.

You must call **sp_xml_removedocument** in the same query batch as the **sp_xml_preparedocument** used to generate the node tree. This is because the **hdoc** parameter used to reference the tree is a local variable, and if it goes out of scope, there is no way to remove the tree from memory.

The following example shows how to use the **sp_xml_removedocument** system stored procedure:

```
EXEC sp_xml_removedocument @hdoc
```


OPENXML Syntax



***** Illegal for non-trainer use *****

Introduction

After you have parsed an XML document by using the `sp_xml_preparedocument` stored procedure and a handle to the internal tree has been returned, you can generate a rowset from the parsed tree.

The OPENXML syntax

You use the OPENXML function to retrieve a rowset from the tree. You can then write Transact-SQL SELECT, UPDATE, or INSERT statements that modify a database.

The OPENXML function has the following syntax.

```
OpenXML(idoc, rowpattern [, flags])
[WITH (SchemaDeclaration | TableName)]

<SchemaDeclaration> ::=
  ColumnName ColumnType [colpattern],n
```

The following table describes parameters of the OPENXML function.

Parameter	Description
<i>rowpattern</i>	XPath query defining the nodes that should be returned.
<i>idoc</i>	Handle to the internal tree representation of the XML document.
<i>flags</i>	Optional bit mask that determines attribute or element centricity. <i>flags</i> accepts the following values: <ul style="list-style-type: none"> ■ 0—Use the default mapping (attributes). ■ 1—Retrieve attribute values. ■ 2—Retrieve element values. ■ 3—Retrieve both attribute and element values.
<i>SchemaDeclaration</i>	Rowset schema declaration for the columns to be returned by using a combination of column names, data types, and patterns.
<i>TableName</i>	Name of an existing table, the schema of which should be used to define the columns returned.

Using the OPENXML statement

You can use an OPENXML function anywhere you can use a rowset provider, such as a table, a view, or the OPENROWSET function. OPENXML is primarily used in SELECT statements, as shown in the following example.

```
SELECT *
FROM OPENXML (@idoc, '/Customer/Order/OrderDetail')
WITH (ProductID int,
      Quantity int)
```

The *rowpattern* argument sets the node level to the **OrderDetail** element and retrieves attributes because no flag value is specified.

In the preceding example, **@idoc** is a handle to the internal tree representation of the following XML order document.

```
<Customer CustomerID="1" CustomerType="S">
  <Order SalesOrderID="43860" Status="5"
    OrderDate="2001-08-01T00:00:00">
    <OrderDetail ProductID="761" Quantity="2"/>
    <OrderDetail ProductID="770" Quantity="1"/>
  </Order>
</Customer>
```

The following table shows the rowset returned by the preceding OPENXML statement.

ProductID	Quantity
761	2
770	1

Using a schema declaration

In situations when you must retrieve data from throughout the hierarchy, you can use a column pattern containing an XPath pattern. The XPath pattern in a *colpattern* parameter is relative to the XPath pattern specified in the *rowpattern* parameter. You can use column patterns to retrieve data from elements that use both attribute-centric and element-centric mappings.

The following example shows how a schema declaration can reference nodes throughout the XML hierarchy by using the same XML document as the preceding example.

```
SELECT *
FROM OPENXML (@idoc, '/Customer/Order/OrderDetail', 1)
WITH (CustomerID int      '../../@CustomerID',
      OrderID int         '../@SalesOrderID',
      OrderDate datetime  '../@OrderDate',
      ProdID int          '@ProductID',
      Quantity int)
```

The *flags* value is set to 1 in the preceding example so that attributes are retrieved by default unless specified otherwise by using a *colpattern* parameter. The **CustomerID** attribute must be retrieved from the **Customer** element, which is two levels above the current **OrderDetail** element. This is achieved by using the `../` combination twice to navigate up two levels. The **Quantity** attribute is automatically retrieved without requiring a column pattern because the attribute name exactly matches the column name, unlike the **ProdID** column.

The following table shows the rowset returned by the preceding OPENXML statement.

OrderID	CustomerID	OrderDate	ProdID	Quantity
43860	1	2001-08-01 00:00:00.000	761	2
43860	1	2001-08-01 00:00:00.000	770	1

For More Information For more information about using the OPENXML function directive, see “Querying XML Using OPENXML” in SQL Server Books Online.

Syntax for Working with XML Namespaces

- **sp_xml_preparedocument** accepts namespaces
- Use namespace prefix in all XPath expressions

```
<Customer xmlns="urn:AW_NS" xmlns:o="urn:AW_OrderNS"
CustomerID="1" CustomerType="S">
  <o:Order SalesOrderID="43860" Status="5"
    OrderDate="2001-08-01T00:00:00">
    <o:OrderDetail ProductID="761" Quantity="2"/>
    <o:OrderDetail ProductID="770" Quantity="1"/>
  </o:Order>
</Customer>

EXEC sp_xml_preparedocument @idoc OUTPUT, @doc,
<ROOT xmlns:rootNS="urn:AW_NS" xmlns:orderNS="urn:AW_OrderNS"/>'

SELECT * FROM OPENXML (@idoc,
'/rootNS:Customer/orderNS:Order/orderNS:OrderDetail')
WITH...
```

***** Illegal for non-trainer use *****

Introduction

Many XML documents contain namespaces to help applications recognize elements and avoid collisions between elements with the same name but different purposes. The **sp_xml_preparedocument** stored procedure supports the use of namespaces when creating the memory tree for shredding.

Passing namespaces to **sp_xml_preparedocument**

The **sp_xml_preparedocument** stored procedure accepts an optional parameter that specifies a placeholder XML element with the required namespace declarations, as shown in the following example, which includes two namespaces.

```
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc,
'<ROOT xmlns:rootNS="urn:AW_NS" xmlns:orderNS="urn:AW_OrderNS" />'
```

The following example shows an XML document that uses these namespaces.

```
<Customer xmlns="urn:AW_NS" xmlns:o="urn:AW_OrderNS"
CustomerID="1" CustomerType="S">
  <o:Order SalesOrderID="43860" Status="5"
    OrderDate="2001-08-01T00:00:00">
    <o:OrderDetail ProductID="761" Quantity="2"/>
    <o:OrderDetail ProductID="770" Quantity="1"/>
  </o:Order>
</Customer>
```

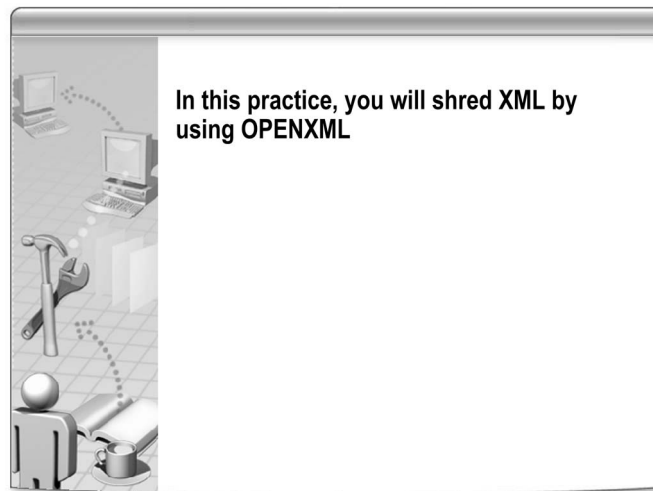
Using namespaces with XPath and OPENXML

By passing the namespace to the **sp_xml_preparedocument** stored procedure, you can use the namespace information within your XPath expression, as shown in the following example.

```
SELECT *
FROM      OPENXML (@idoc,
                '/rootNS:Customer/orderNS:Order/orderNS:OrderDetail')
WITH (OrderID      int           '@SalesOrderID',
      CustomerID   int           '@CustomerID',
      OrderDate    datetime      '@OrderDate',
      ProdID       int           '@ProductID',
      Quantity     int)
```

For More Information For more information about XML namespaces, see “Understanding XML Namespaces” on the MSDN Web site.

Practice: Using OPENXML to Shred XML



***** Illegal for non-trainer use *****

Goals The goal of this practice is to enable you to use the OPENXML function and its different arguments.

Preparation Ensure that virtual machine 2779A-MIA-SQL-03 is running and that you are logged on as **Student**.

If a virtual machine has not been started, perform the following steps:

1. Close any other running virtual machines.
2. Start the virtual machine.
3. In the **Log On to Windows** dialog box, complete the logon procedure by using the user name **Student** and the password **Pa\$\$w0rd**.

To use the OPENXML statement

- You must perform the following steps to shred XML by using the OPENXML statement.
1. Use Microsoft Windows Explorer to view the contents of the D:\Practices folder.
 2. Double-click **OPENXML.sql** to open it in SQL Server Management Studio.
 3. In the **Connect to Database Engine** dialog box, specify the values in the following table, and then click **Connect**.

Property	Value
Server type	Database Engine
Server name	MIAMI
Authentication	Windows Authentication

4. Review the Transact-SQL script, noting that an **xml** variable is passed to the **sp_xml_preparedocument** stored procedure. Much of the code is initially commented out except for the **OPENXML using attributes only** section.
5. On the toolbar, click **Execute**.

To shred XML by using elements only

6. Review the query results, noting that the **Item** attributes are returned correctly, but that NULL is returned for the **ProductName** subelement because the *flags* argument for the OPENXML function is set to **1**, and the query searches for attributes but not elements.

► Perform the following steps to shred an XML document based on elements only:

1. Comment out the SELECT query under the comment **OPENXML using attributes only**.

Tip You can comment out multiple lines of code by selecting them and clicking the **Comment out the selected lines** button on the toolbar.

2. Uncomment the SELECT query under the comment **OPENXML using elements only**.

Tip You can uncomment multiple lines of code by selecting them and clicking the **Uncomment the selected lines** button on the toolbar. Make sure that you deselect the code before executing the query.

3. On the toolbar, click **Execute**.
4. Review the query results. Notice that NULL values are returned for each attribute but that the **ProductName** subelement is returned because the *flags* argument for the OPENXML function is set to **2**, so the query searches for elements and not attributes.

To shred XML by using attributes or elements

► You must perform the following steps to shred an XML document based on attributes or elements:

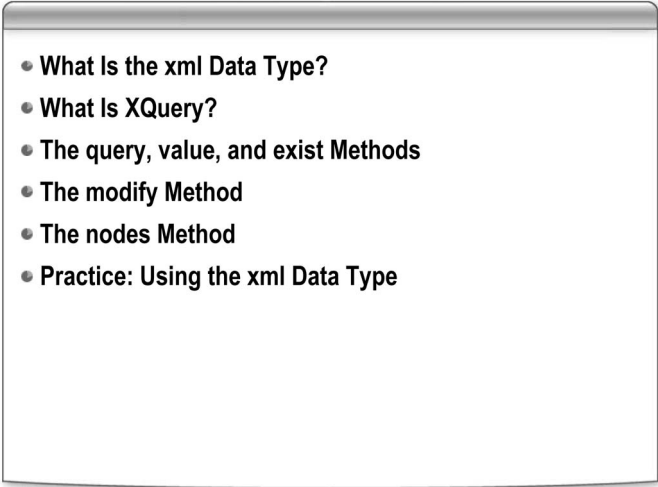
1. Comment out the Comment out the SELECT query under the comment **OPENXML using elements only**.
2. Uncomment the SELECT query under the comment **OPENXML using either attributes or elements**.
3. On the toolbar, click **Execute**.
4. Review the query results. Notice that the correct values are returned for each column because the *flags* argument for the OPENXML function is set to **3**, and the query searches both attributes and elements.

To shred XML by using a colpattern parameter

► You must perform the following steps to shred an XML document based on column patterns:

1. Comment out the SELECT query under the comment **OPENXML using either attributes or elements**.
2. Uncomment the SELECT query under the comment **OPENXML using colpattern**.
3. On the toolbar, click **Execute**.
4. Review the query results. Notice that the correct values are returned for each column because the *flags* argument for the OPENXML function is set to **1** and the *colpattern* paths navigate to other locations in the node hierarchy.
5. Close SQL Server Management Studio without saving the file.

Lesson 3: Using the xml Data Type

- 
- What Is the xml Data Type?
 - What Is XQuery?
 - The query, value, and exist Methods
 - The modify Method
 - The nodes Method
 - Practice: Using the xml Data Type

***** Illegal for non-trainer use *****

Lesson objectives

After completing this lesson, students will be able to:

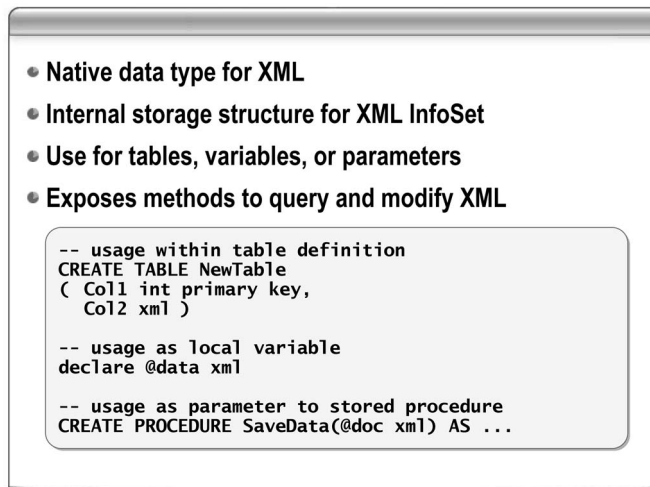
- Define the **xml** data type.
- Define XQuery.
- Describe the **query**, **value**, and **exist** methods that are used to query XML.
- Describe the **modify** method, which is used to modify XML.
- Describe the **nodes** method, which is used to shred XML.

Introduction

SQL Server 2005 provides the **xml** data type for storing XML documents and fragments in a table, variable, or parameter. The **xml** data type provides some important advantages over storing the data as a **text** or **varchar** field, including the ability to work with Extensible Query Language (XQuery) and modification or retrieval of specific elements or attributes within the XML data.

In this lesson, you will learn about the **xml** data type and the various methods relating to it. You will also learn how to use XQuery to query XML data.

What Is the xml Data Type?



***** Illegal for non-trainer use *****

Introduction

The **xml** data type is a native data type for storing XML documents or fragments as columns, local variables, or parameters of up to 2 gigabytes (GB) in size. The ability to store XML natively in a relational database brings many advantages for application developers.

Benefits of native XML storage

The benefits of native XML storage include the following:

- Both structured data and semi-structured data are stored in a single location, making it easier to manage.
- You can define variable content within a relational model.
- You can choose the most suitable data model for your application's specific requirements while still taking advantage of a highly optimized data storage and querying environment.

Native XML functionality

The SQL Server 2005 **xml** data type stores the InfoSet of an XML document in an efficient internal format. The data can be treated as if it were the original XML document with the exception that insignificant white space, order of attributes, namespace prefixes, and the XML declaration are not retained. SQL Server 2005 provides the following functionality for the **xml** data type:

XML indexing. Columns defined as **xml** can be indexed by using XML indexes and full-text indexes. This can significantly enhance the performance of queries that retrieve XML data.

XQuery-based data retrieval methods. The **xml** data type provides the **query**, **value**, and **exist** methods. These can be used to extract data from the XML data by using an XQuery expression.

XQuery-based data modification. The **xml** data type provides the **modify** method, which uses an extension of the XQuery specification to perform updates on the XML data.

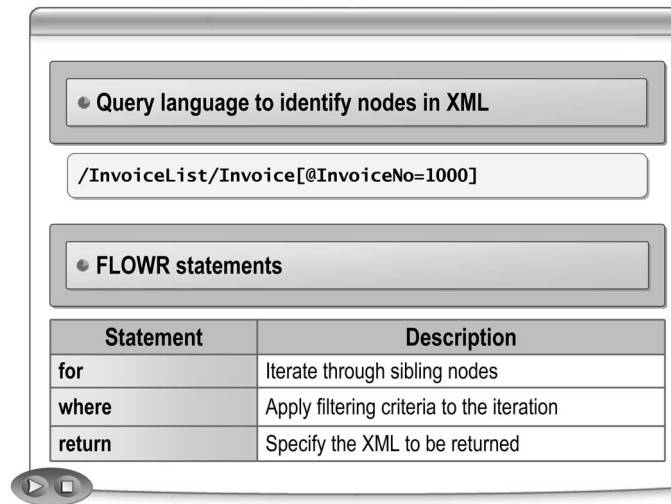
Typed XML. XML that is associated with an XML schema. The schema defines the elements and attributes that are valid in an XML document of this type and specifies a

namespace for them. When the **xml** data type is used to store typed XML, SQL Server validates the XML against the schema and optimizes the internal storage of the data by assigning appropriate SQL Server data types to the data based on the XML data types defined in the schema.

For More Information For more information about the XML InfoSet, see the “XML Information Set” W3C specification.

For more information about the **xml** data type, see “xml Data Type” in SQL Server Books Online.

What Is XQuery?



***** Illegal for non-trainer use *****

Introduction

XQuery is used to query XML data. XQuery syntax includes and extends XPath 2.0 expressions and makes it possible to perform complex queries against an XML data source. The **xml** data type in SQL Server provides methods through which data in an **xml** value can be retrieved or updated by specifying an XQuery expression.

The XQuery support in SQL Server 2005 is based on a working draft of the W3C XQuery 1.0 language specification (available on the W3C Web site), and therefore, there might be some minor incompatibilities with the specification when it is released.

XQuery syntax

An XQuery query consists of two main sections: an optional *prolog* section, in which namespaces can be declared and schemas can be imported; and a *body*, in which XQuery expressions are used to specify the data to be retrieved. The XQuery expression can be a simple path that describes the XML nodes to be retrieved or a complex expression that generates an XML result.

An XQuery path is based on the XPath language and describes the location of a node in an XML document. Paths can be absolute (describing the location of the node by traversing the XML tree from the root element) or relative (describing the location of a node relative to a previously identified node). The examples in the following table show some simple XQuery paths.

Example path	Description
<code>/InvoiceList/Invoice</code>	All Invoice elements immediately contained within the root InvoiceList element
<code>(/InvoiceList/Invoice) [2]</code>	The second Invoice element within the root InvoiceList element
<code>(InvoiceList/Invoice/@InvoiceNo) [1]</code>	The InvoiceNo attribute of the first Invoice element in the root InvoiceList element
<code>(InvoiceList/Invoice/Customer/text())[1]</code>	The text of the first Customer element in an Invoice element in the InvoiceList root element
<code>/InvoiceList/Invoice[@InvoiceNo=1000]</code>	All Invoice elements in the InvoiceList element that have an InvoiceNo attribute with the value 1000

FLOWR statements

The XQuery language specification includes **for**, **let**, **order by**, **where**, and **return** statements, commonly known as FLOWR (pronounced “flower”) statements. SQL Server 2005 supports the **for**, **where**, and **return** statements, which are described in the following table.

Statement	Description
for	Used to iterate through a group of nodes at the same level in an XML document.
where	Used to apply filtering criteria to the node iteration. XQuery includes functions such as count that can be used with the where statement.
return	Used to specify the XML returned from within an iteration.

The following example shows an XQuery expression that includes the **for**, **where**, and **return** keywords.

```
for $i in /InvoiceList/Invoice
where count($i/Items/Item) > 1
return $i
```

This example returns each **Invoice** element that includes more than one **Item** element in its **Items** child element.

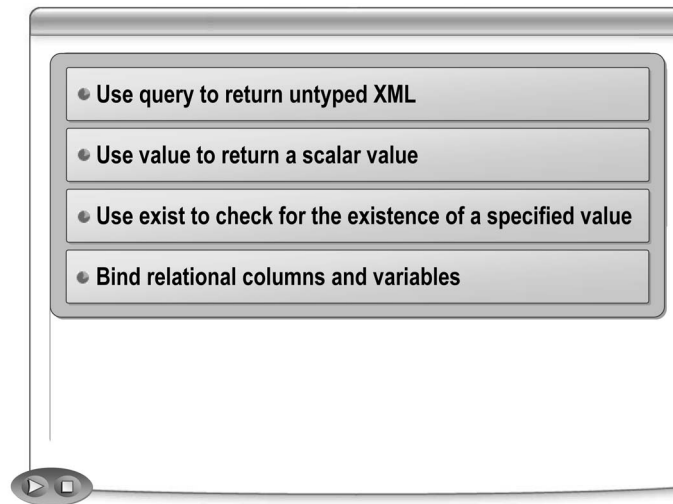
Working with namespaces

If the XML being queried contains a namespace, the XQuery can include a namespace declaration in the query prolog using the following syntax.

```
xml.method('declare default element namespace "http://namespace";
method body')
```

For More Information For more information about using namespaces with XQuery, see “XQuery Basics” in SQL Server Books Online.

The query, value, and exist Methods



***** Illegal for non-trainer use *****

Introduction

The SQL Server 2005 **xml** data type provides four methods that can be used to query or modify XML data. These methods are called by using the *data_type.method_name* syntax familiar to most developers. Understanding the purpose of each of the methods will help you build applications that process XML natively in the database.

The query method

The **query** method is used to extract XML from an **xml** data type. The XML retrieved by the query method is determined by an XQuery expression passed as a parameter. The following example shows how to use the **query** method.

```
SELECT xmlCol.query('declare default element namespace
                    "http://schemas.adventure-works.com/InvoiceList";
                    <InvoiceNumbers>
                    {
                    for $i in /InvoiceList/Invoice
                    return <InvoiceNo>
                        {number($i/@InvoiceNo)}
                    </InvoiceNo>
                    }
                    </InvoiceNumbers>')
```

The value method

The **value** method is used to return a single value from an XML document. To use the **value** method, you must specify an XQuery expression that identifies a single node in the XML being queried and the Transact-SQL data type of the value to be returned. The following example shows how to use the **value** method.

```
SELECT xmlCol.value('declare default element namespace
                    "http://schemas.adventure-works.com/InvoiceList";
                    /InvoiceList/Invoice/@InvoiceNo[1]', 'int')
```

The exist method

The **exist** method is used to determine whether a specified node exists in an XML document. The **exist** method returns 1 if one or more instances of the specified node

exist in the document and 0 if the node does not exist. The following example shows how to use the **exist** method.

```
SELECT xmlCol.exist('declare default element namespace
                    "http://schemas.adventure-works.com/InvoiceList";
                    /InvoiceList/Invoice[@InvoiceNo=1000]')
```

Binding relational columns and variables

SQL Server 2005 provides Microsoft-specific extensions that enhance XQuery to allow you to reference relational columns or variables. This is known as *binding* the relational column or variable. When a SELECT statement that retrieves data from a table includes an **xml** method to retrieve XML from an **xml** column, the **sql:column** function can be used to include non-**xml** column values in the XML, as shown in the following example.

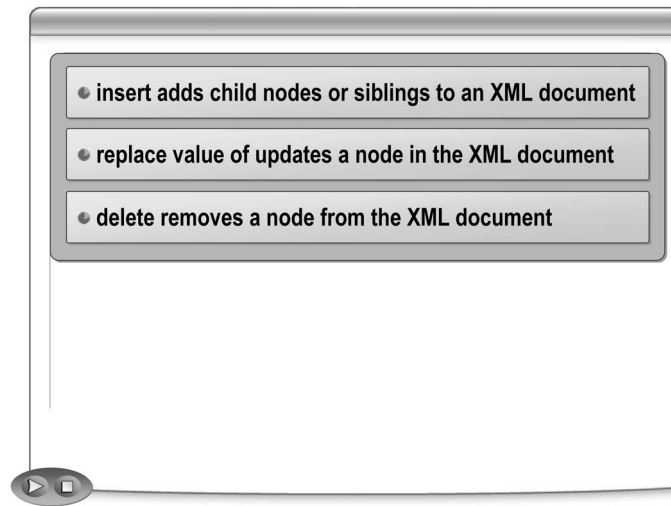
```
SELECT StoreName, Invoices.query('declare default element namespace
                                "http://schemas.adventure-works.com/InvoiceList";
                                <Invoices>
                                <Store>{sql:column("StoreName")}</Store>
                                {
                                  for $i in /InvoiceList/Invoice
                                  return $i
                                }
                                </Invoices>') InvoicesWithStoreName
FROM Stores
```

Similarly, the **sql:variable** extension can be used to reference a variable in a stored procedure, as shown in the following example.

```
CREATE PROCEDURE GetInvoice(@store int, @invoiceNo int)
AS
SELECT Invoices.query('declare default element namespace
                    "http://schemas.adventure-works.com/InvoiceList";
                    <Invoices>
                    {
                      for $i in /InvoiceList/Invoice
                      where $i/@InvoiceNo = sql:variable("@invoiceNo")
                      return $i
                    }
                    </Invoices>')
FROM #Stores
WHERE StoreID=@store
```

For More Information For more information about the **xml** data type methods, see “xml Data Type Methods” in SQL Server Books Online.

The modify Method



***** Illegal for non-trainer use *****

Introduction

You can use the **modify** method to update XML data in an **xml** data type. The **modify** method uses three extensions to the XQuery language specification: **insert**, **replace**, and **delete**. These extensions are referred to as XML DML.

The insert statement

You can use the **insert** statement to add nodes to XML in an **xml** column or variable. The **insert** statement has the following syntax.

```
insert Expression1 (
  {as first | as last} into | after | before
  Expression2 )
```

The syntax parameters for the **insert** keyword are described in the following table.

Parameter	Description
<i>Expression1</i>	The node to be inserted. This can be literal XML—for example, <code><Item Product="5" Quantity="1"/></code> . It can also be an element expression to insert a text node—for example, <code>element SalesPerson { "Alice" }</code> . Finally, it can be an attribute expression to insert an attribute—for example, <code>attribute discount { "1.50" }</code> .
as first	Used to insert the new XML as the first sibling in the hierarchy.
as last	Used to insert the new XML as the last sibling in the hierarchy.
into	Used to insert <i>Expression1</i> into <i>Expression2</i> .
after	Used to insert <i>Expression1</i> after <i>Expression2</i> .
before	Used to insert <i>Expression1</i> before <i>Expression2</i> .
<i>Expression2</i>	An XQuery expression that identifies an existing node in the document.

The following example shows how to use the XQuery **insert** statement in the **modify** method.

```
SET @xmlDoc.modify('declare default element namespace
                  "http://schemas.adventure-works.com/InvoiceList";
                  insert element salesperson {"Alice"}
                  as first
                  into (/InvoiceList/Invoice)[1]')
```

The replace statement

You can use the **replace** statement to update an **xml** value. The **replace** statement has the following syntax.

```
replace value of
  Expression1
with
  Expression2
```

The parameters of the **replace** syntax are described in the following table.

Parameter	Description
<i>Expression1</i>	An XQuery expression identifying the node containing the value to be replaced.
<i>Expression2</i>	The new value for the node.

The following example shows how to use the **replace** statement in the **modify** method.

```
SET xmlCol.modify('declare default element namespace
                  "http://schemas.adventure-works.com/InvoiceList";
                  replace value of
                    (/InvoiceList/Invoice/SalesPerson/text())[1]
                  with "Holly"')
```

The delete statement

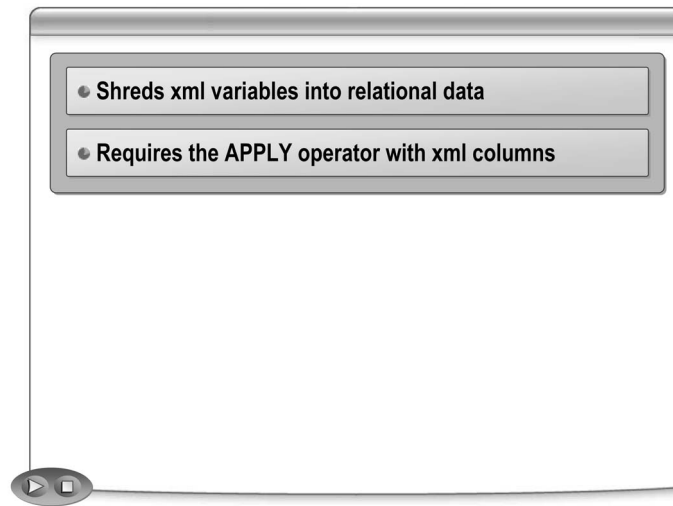
You can use the **delete** statement to remove a node from an **xml** value. The **delete** statement has the following syntax.

```
delete Expression
```

The *Expression* parameter is an XQuery expression identifying the node to be deleted. The following example shows how to use the **delete** statement in the **modify** method.

```
SET xmlCol.modify('declare default element namespace
                  "http://schemas.adventure-works.com/InvoiceList";
                  delete (/InvoiceList/Invoice/SalesPerson)[1]')
```


The nodes Method



***** Illegal for non-trainer use *****

Introduction

The **xml** data type provides the **nodes** method, which you can use to generate a relational view of the XML data. The **nodes** method returns a rowset in which each node identified by an XQuery expression is returned as a context node from which subsequent queries can extract data.

Syntax of the nodes method

The **nodes** method has the following syntax.

```
xmlvalue.nodes (XQuery) [AS] Table(Column)
```

The parameters of the nodes syntax are described in the following table.

Parameter	Description
<i>Xmlvalue</i>	An xml variable or column.
<i>XQuery</i>	The XQuery expression that identifies the nodes you want to return.
<i>Table(Column)</i>	The name of the table and column in which the results will be returned. This table can be used in subsequent queries to extract data from the context nodes.

How to use the nodes method with an xml variable

To extract data in a relational format from an **xml** variable, use the **query**, **value**, or **exist** method with the rowset returned from the **nodes** method. The following example shows how to extract order data in a relational format from an **xml** variable.

```
DECLARE @xmlOrder xml
SET @xmlOrder = '<?xml version="1.0" ?>
  <Order OrderID="1000" OrderDate="2005-06-04">
    <LineItem ProductID="1" Price="2.99" Quantity="3" />
    <LineItem ProductID="2" Price="3.99" Quantity="1" />
  </Order>'

SELECT nCol.value('@ProductID', 'integer') ProductID,
       nCol.value('@Quantity', 'integer') Quantity
FROM   @xmlOrder.nodes('/Order/LineItem') AS nTable(nCol)
```

This code returns the following results.

ProductID	Quantity

1	3
2	1

How to use the nodes method with an xml column

You can use the APPLY operator with the **nodes** method to return data in a relational format from an **xml** column. The following example shows how to use the **nodes** method to extract data from an **xml** column that contains order documents in the same format as the variable in the preceding example.

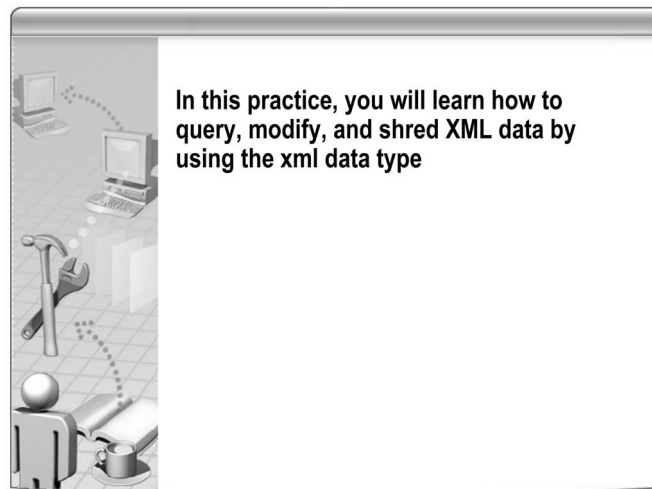
```
DECLARE @xmlOrder xml
SELECT nCol.value('..@OrderID[1]', 'int') OrderID,
       nCol.value('..@OrderDate[1]', 'datetime') OrderDate,
       nCol.value('@ProductID[1]', 'int') ProductID,
       nCol.value('@Price[1]', 'money') Price,
       nCol.value('@Quantity[1]', 'int') Quantity
FROM Orders_X
CROSS APPLY OrderDoc.nodes('/Order/LineItem') AS nTable(nCol)
```

This code produces results similar to the following example.

OrderID	OrderDate	ProductID	Price	Quantity

1000	2005-06-04 00:00:00.000	1	2.99	1
1000	2005-06-04 00:00:00.000	2	3.99	2
1001	2005-06-04 00:00:00.000	2	3.99	1
1002	2005-06-04 00:00:00.000	2	3.99	1

Practice: Using the xml Data Type



***** Illegal for non-trainer use *****

Goals

The goal of this practice is to enable you to use the **xml** data type and the methods that act on the type.

Preparation

Ensure that virtual machine 2779A-MIA-SQL-03 is running and that you are logged on as **Student**.

If a virtual machine has not been started, perform the following steps:

1. Close any other running virtual machines.
2. Start the virtual machine.
3. In the **Log On to Windows** dialog box, complete the logon procedure by using the user name **Student** and the password **Pa\$\$wOrd**.

To declare an xml column

► You must perform the following steps to create a table with an **xml** column:

1. In Windows Explorer, view the contents of the D:\Practices folder.
2. Double-click **xmlDataType.sql** to open it in SQL Server Management Studio.
3. In the **Connect to Database Engine** dialog box, specify the values in the following table, and then click **Connect**.

Property	Value
Server type	Database Engine
Server name	MIAMI
Authentication	Windows Authentication

4. Select the Transact-SQL under the comment **Create a table with an xml column**.
5. On the toolbar, click **Execute**. This creates a table with a column for untyped XML.

To implicitly cast a string to xml

- You must perform the following steps to cast a string to the **xml** data type:
1. Select the Transact-SQL under the comment **Use implicit casting to assign an xml variable and column.**
 2. On the toolbar, click **Execute**. This code assigns an **nvarchar** variable to an **xml** variable and inserts it into the table. The code then inserts a string constant into the **xml** column directly.
 3. Review the results, noting that the XML string has been stored in the table for both INSERT statements.

To insert a well-formed document

- You must perform the following steps to insert a well-formed document into the **xml** data type:
1. Select the Transact-SQL under the comment **Well-formed document. This will succeed.**
 2. On the toolbar, click **Execute**. This code inserts a well-formed XML document into the table.
 3. Review the results, noting that the XML string has been stored in the table. Note also that the XML version information is not stored.

To attempt to insert XML that is not well formed

- You must perform the following steps to attempt to insert XML that is not well formed into the **xml** data type:
1. Select the Transact-SQL under the comment **Not well-formed. This will fail.**
 2. On the toolbar, click **Execute**. This code attempts to insert a string that is not well-formed XML into the table.
 3. Review the results, noting that the insert failed.

To execute xml methods

- You must perform the following steps to use the methods of the **xml** data type:
1. In Microsoft Windows Explorer, view the contents of the D:\Practices folder.
 2. Double-click **xmlMethods.sql** to open it in SQL Server Management Studio.
 3. In the **Connect to Database Engine** dialog box, specify the values in the following table, and then click **Connect**.

Property	Value
Server type	Database Engine
Server name	MIAMI
Authentication	Windows Authentication

4. Examine the code under the comment **Create a table that includes XML data.** This code creates a table with an **xml** column named **Invoices** and populates the table with data.
5. On the toolbar, click **Execute** to execute the entire script.

To examine the results of the query method

- You must perform the following steps to examine the results of the **query** method:
1. In the query pane, examine the code under the comment **Use the query method.** This code uses the **query** method in a SELECT statement to retrieve XML data from the **Invoices** column.
 2. In the results pane, examine the results returned by the SELECT statement. (The result set has two columns, **StoreName** and **SoldItems**.)

To examine the results of the value method

- You must perform the following steps to examine the results of the **value** method:
 1. In the query pane, examine the code under the comment **Use the value method**. This code uses the **value** method in a SELECT statement to retrieve a single value from the **Invoices** column.
 2. In the results pane, examine the results returned by the SELECT statement. (The result set has two columns, **StoreName** and **FirstInvoice**.)

To examine the results of the exist method

- You must perform the following steps to examine the results of the **exist** method:
 1. In the query pane, examine the code under the comment **Use the exist method**. This code uses the **exist** method in a SELECT statement to find rows that contain an Invoice element in the **Invoices** column.
 2. In the results pane, examine the results returned by the SELECT statement. (The result set has one column named **StoresWithInvoices**.)

To examine the results of binding relational columns

- You must perform the following steps to examine the results of binding relational columns:
 1. In the query pane, examine the code under the comment **Bind a relational column**. This code uses the **query** method in a SELECT statement to retrieve XML data that includes the **StoreName** relational column.
 2. In the results pane, examine the results returned by the SELECT statement. (The result set has a single column, **InvoicesWithStoreName**.)

To examine the results of using the modify method to insert XML

- You must perform the following steps to examine the results of using the **modify** method to insert XML:
 1. In the query pane, examine the code under the comment **Use the modify method to insert XML**. This code uses the **modify** method in an UPDATE statement to insert a **SalesPerson** element into the **Invoice** column. The modified XML is then returned by a SELECT statement.
 2. In the results pane, examine the results returned by the SELECT statement. (The result set has a single column, **InsertedSalesPerson**.)

To examine the results of using the modify method to update XML

- You must perform the following steps to examine the results of using the **modify** method to update XML:
 1. In the query pane, examine the code under the comment **Use the modify method to update XML**. This code uses the **modify** method in an UPDATE statement to update a **SalesPerson** element in the **Invoice** column. The modified XML is then returned by a SELECT statement.
 2. In the results pane, examine the results returned by the SELECT statement. (The result set has a single column, **UpdatedSalesPerson**.)

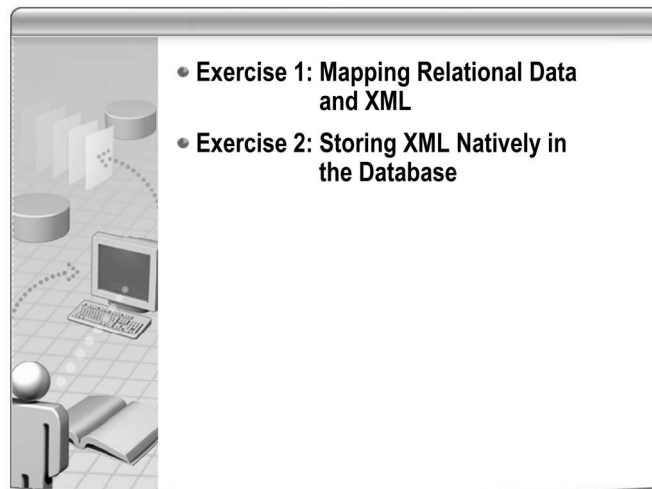
To examine the results of using the modify method to delete XML

- You must perform the following steps to examine the results of using the **modify** method to delete XML:
 1. In the query pane, examine the code under the comment **Use the modify method to delete XML**. This code uses the **modify** method in an UPDATE statement to delete a **SalesPerson** element in the **Invoice** column. The modified XML is then returned by a SELECT statement.
 2. In the results pane, examine the results returned by the SELECT statement. (The result set has a single column, **DeletedSalesPerson**.)

To examine the results of using the nodes method

- ▶ You must perform the following steps to examine the results of using the **nodes** method:
 1. In the query pane, examine the code under the comment **Use the nodes method to extract relational data**. This code uses the **nodes** method with the APPLY operator to extract relational data from an **xml** column.
 2. In the results pane, examine the results returned by the SELECT statement. (The result set has four columns; **InvoiceNo**, **ProductID**, **Price**, and **Quantity**.)
 3. Close SQL Server Management Studio and Windows Explorer. Click **No** if prompted to save files when closing SQL Server Management Studio.

Lab: Working with XML



***** Illegal for non-trainer use *****

Scenario

Adventure Works currently generates order manifests by performing a query in the **AdventureWorks** database. The organization intends to deploy a new Pocket PC-based application to its warehouse employees, which they will then use when picking items from the warehouse for an order delivery. The item-choosing application requires the order manifest to be downloaded as XML. You must modify the existing Transact-SQL query that is used to generate the order manifest so that it retrieves the data as XML.

Adventure Works is also creating a new ordering system for specific customers so that they can send orders as XML documents. This system is still in the initial testing phase. You must create the appropriate Transact-SQL required to store the XML order details in the existing database tables.

The senior database developer has provided you with the following requirements for the modifications:

- Two different FOR XML queries are required to produce two different XML formats.
- The first FOR XML query should produce attribute-centric XML, as shown in the following example.

```
<SalesOrder SalesOrderID="43659" OrderDate="2001-07-01T00:00:00"
  AccountNumber="10-4020-000676">
  <Item ProductID="776" OrderQty="1" />
  <Item ProductID="777" OrderQty="3" />
  <Item ProductID="778" OrderQty="1" />
  <Item ProductID="771" OrderQty="1" />
  <!-- remaining items go here. -->
</SalesOrder>
```

- The second FOR XML query should produce a mixture of element-centric and attribute-centric XML, as shown in the following example.

```
<SalesOrder SalesOrderID="43659" OrderDate="2001-07-01T00:00:00"
  AccountNumber="10-4020-000676">
  <Item>
    <ProductID>776</ProductID>
    <OrderQty>1</OrderQty>
  </Item>
  <Item>
    <ProductID>777</ProductID>
    <OrderQty>3</OrderQty>
  </Item>
  <Item>
    <ProductID>778</ProductID>
    <OrderQty>1</OrderQty>
  </Item>
  <Item>
    <!-- more items here -->
  </Item>
</SalesOrder>
```

- In the FOR XML queries, the **SalesOrder** attributes come from the **Sales.SalesOrderHeader** table. The **Item** attributes and elements come from the **Sales.SalesOrderDetail** table.
- Two OPENXML queries are required to shred the XML document into the appropriate tables. The first query will insert data into the **Sales.SalesOrderHeader** table and should use the following schema declaration.

Column name	Data type
CustomerID	int
DueDate	datetime
AccountNumber	nvarchar(15)
ContactID	int
BillToAddressID	int
ShipToAddressID	int
ShipMethodID	int
SubTotal	money
TaxAmt	money

- The second OPENXML query will insert data into the **Sales.SalesOrderDetail** table and should use the following schema declaration.

Column name	Data type
OrderQty	int
ProductID	int
UnitPrice	money

- All FOR XML queries should use sales order number 43659 for testing purposes.

Additional resources

Use the following additional information to perform the exercises in this lab:

- The existing Transact-SQL query scripts and XML files are located in the **XML.ssmssl** SQL Server Scripts solution in the D:\Labfiles\Starter folder.

Additional information

When performing database development tasks, it can be helpful to use SQL Server Management Studio to create a SQL Server Scripts project, and use it to document the Transact-SQL code necessary to re-create the solution if necessary.

Use the following procedure to create a SQL Server Scripts project:

1. Open SQL Server Management Studio, connecting to the server you want to manage.
2. On the **File** menu, point to **New**, and then click **Project**.
3. Select the **SQL Server Scripts** template and enter a suitable name and location for the project. Note that you can create a solution that contains multiple projects, but in many cases a single project per solution is appropriate.

To add a query file to a project:

1. Click **New Query** on the **Project** menu, or right-click the **Queries** folder in Solution Explorer and click **New Query**. If Solution Explorer is not visible, you can display it by clicking **Solution Explorer** on the **View** menu.
2. When prompted, connect to the server on which you want to execute the query. This will add a connection object to the project.
3. Change the name of the query file from the default name (**SQLQuery1.sql**) by right-clicking it in Solution Explorer and clicking **Rename**.

Although you can perform all database development tasks by executing Transact-SQL statements, it is often easier to use the graphical user interface in SQL Server Management Studio. However, you should generate the corresponding Transact-SQL scripts and save them in the project for future reference.

Often, you can generate the Transact-SQL script for an action before clicking **OK** in the **Properties** dialog box used to perform the action. Many **Properties** dialog boxes include a **Script** drop-down list with which you can script the action to a new query window, a file, the Clipboard, or a SQL Server Agent job. A common technique is to add a blank query file to a project, and then script each action to the Clipboard as it is performed and paste the generated script into the query file.

You can also generate scripts for many existing objects, such as databases and tables. To generate a script, right-click the object in Object Explorer and script the CREATE action. If Object Explorer is not visible, you can display it by clicking **Object Explorer** on the **View** menu.

Preparation

Ensure that virtual machine 2779A-MIA-SQL-03 is running and that you are logged on as **Student**.

If a virtual machine has not been started, perform the following steps:

1. Close any other running virtual machines.
2. Start the virtual machine.
3. In the **Log On to Windows** dialog box, complete the logon procedure by using the user name **Student** and the password **Pa\$\$w0rd**.

Exercise 1: Mapping Relational Data and XML

Retrieving XML from relational data

Task	Supporting information																				
Create FOR XML queries to return the order manifest documents.	<ol style="list-style-type: none"> 1. Start SQL Server Management Studio. Connect to MIAMI when prompted. 2. Open the D:\Labfiles\Starter\XML.ssmssln solution. 3. View the FORXML.sql query, and then execute it and review the results. 4. Add a FOR XML clause to produce the first, attribute-centric XML format. 5. Execute the modified query and review the results, confirming that the results match the preceding example. 6. Create a copy of the modified query. 7. Change the copied query to produce the second, attribute- and element-centric XML format. 8. Execute the modified query and review the results, confirming that the results match the preceding example. 9. Save the FORXML.sql query. 																				
Use the OPENXML function to insert the XML data into the tables.	<ol style="list-style-type: none"> 1. View the OPENXML.sql query in the SQL Server Management Studio solution. 2. Examine the existing Transact-SQL, but do not execute it yet. 3. Add a call to the sp_xml_preparedocument stored procedure to create the in-memory tree. 4. Add the OPENXML function to the first INSERT statement using the following values. <table> <tr> <th>Argument</th><th>Value</th></tr> <tr> <td><i>idoc</i></td><td>@dochandle</td></tr> <tr> <td><i>rowpattern</i></td><td>/SalesOrder</td></tr> <tr> <td><i>flag</i></td><td>1</td></tr> <tr> <td><i>schemaDeclaration</i></td><td>See lab scenario</td></tr> </table> 5. Add the OPENXML function to the second INSERT statement using the following values. <table> <tr> <th>Argument</th><th>Value</th></tr> <tr> <td><i>idoc</i></td><td>@dochandle</td></tr> <tr> <td><i>rowpattern</i></td><td>/SalesOrder/Item</td></tr> <tr> <td><i>flag</i></td><td>2</td></tr> <tr> <td><i>schemaDeclaration</i></td><td>See lab scenario</td></tr> </table> 6. Add a call to the sp_xml_removedocument stored procedure to clean up the in-memory tree. 7. Execute the entire script, and then review the results. Confirm that a new record was added to the SalesOrderHeader table and two records were added to the SalesOrderDetail table. 	Argument	Value	<i>idoc</i>	@dochandle	<i>rowpattern</i>	/SalesOrder	<i>flag</i>	1	<i>schemaDeclaration</i>	See lab scenario	Argument	Value	<i>idoc</i>	@dochandle	<i>rowpattern</i>	/SalesOrder/Item	<i>flag</i>	2	<i>schemaDeclaration</i>	See lab scenario
Argument	Value																				
<i>idoc</i>	@dochandle																				
<i>rowpattern</i>	/SalesOrder																				
<i>flag</i>	1																				
<i>schemaDeclaration</i>	See lab scenario																				
Argument	Value																				
<i>idoc</i>	@dochandle																				
<i>rowpattern</i>	/SalesOrder/Item																				
<i>flag</i>	2																				
<i>schemaDeclaration</i>	See lab scenario																				

Exercise 2: Storing XML Natively in the Database

Scenario

When a customer orders goods from Adventure Works, the goods are delivered by a delivery driver. To determine which orders they must deliver, delivery drivers consult a delivery list. This list specifies the route to be driven by the driver and the delivery details for each order he or she must deliver.

At present, the delivery list is compiled manually. Adventure Works would like to issue to drivers a mobile application that will retrieve XML delivery lists that conform to an XML schema defined by the logistics department. You must create a **DeliverySchedule** table and populate it with some test data.

The **Sales.DeliverySchedule** table structure should be as follows.

Column name	Data type
ScheduleID	int IDENTITY PRIMARY KEY
ScheduleDate	datetime
DeliveryRoute	int
DeliveryDriver	nvarchar(20)
DeliveryList	Untyped xml

You must write Transact-SQL code for a delivery scheduling application to test its functionality. The senior database developer has provided you with the following test cases for the application:

- Use the **xml query** method to retrieve a result set from the **Sales.DeliverySchedule** table. The result set should contain the **DeliveryDriver** column value and the **Delivery xml** column.
- Use the **xml value** method to retrieve a result set from the **Sales.DeliverySchedule** table. The result set should contain a single **nvarchar(100)** column named **DeliveryAddress** containing the address for the first delivery in the delivery list document for the record in the **Sales.DeliverySchedule** table with a **ScheduleID** of 1.
- Use the **xml exist** method to retrieve the **DeliveryDriver** column in the **Sales.DeliverySchedule** table, where the XML in the **DeliveryList** column includes a **Delivery** element with a **SalesOrderID** attribute of 43659.
- Use the **xml query** method to retrieve a delivery route document from the **DeliveryList** column in each row of the **Sales.DeliverySchedule** table.
- Use the **xml modify** method to update the **Sales.DeliverySchedule** table, setting the first **Address** element value in the delivery list document for the record with a **ScheduleID** of 1 to **7194 Fourth St. Rockhampton**.

A sample XML document named **DeliveryList.xml** is provided in the **Miscellaneous** folder of the SQL Server Scripts project.

**Working with the xml
data type**

Task	Supporting information										
Create the Sales.DeliverySchedule table, and then insert test data.	<ol style="list-style-type: none">1. On the Project menu, click New Query to create a new query file. When prompted, connect to MIAMI.2. In Solution Explorer, rename SQLQuery1.sql to xmlDataType.sql.3. Create the Sales.DeliverySchedule table in the AdventureWorks database using the structure specified in the scenario.4. Execute the query, and then verify that the command completes successfully.5. Open DeliveryList.xml in the Miscellaneous folder in Solution Explorer, and then examine it. This is an XML delivery list document.6. In the xmlDataType.sql query script, type the necessary INSERT statement to insert the following data into the Sales.DeliverySchedule table.<table><tr><th>Column</th><th>Value</th></tr><tr><td>Schedule</td><td>DateGetDate()</td></tr><tr><td>DeliveryRoute</td><td>3</td></tr><tr><td>DeliveryDriver</td><td>Alice</td></tr><tr><td>DeliveryList</td><td>The delivery list XML document</td></tr></table> <hr/> <p>Tip Copy and paste this from the DeliveryList.xml file.</p> <hr/>	Column	Value	Schedule	DateGetDate()	DeliveryRoute	3	DeliveryDriver	Alice	DeliveryList	The delivery list XML document
Column	Value										
Schedule	DateGetDate()										
DeliveryRoute	3										
DeliveryDriver	Alice										
DeliveryList	The delivery list XML document										
	<ol style="list-style-type: none">7. Execute the query, and then verify that the command completes successfully.										

Task	Supporting information				
Use xml data methods to query and modify the data.	<ol style="list-style-type: none"> On the Project menu, click New Query to create a new query file. When prompted, connect to MIAMI. In Solution Explorer, rename SQLQuery1.sql to xmlMethods.sql. Type the necessary Transact-SQL to select the following result set from the Sales.DeliverySchedule table in the AdventureWorks database by using the query method. <table> <tr> <th>Delivery Driver</th><th>Delivery</th></tr> <tr> <td>The DeliveryDriver column value</td><td>All Delivery XML elements in the DeliveryList document</td></tr> </table> Execute the query, and then verify that the desired results are returned. Type the necessary Transact-SQL to retrieve a single nvarchar(100) column named DeliveryAddress containing the address for the first delivery in the delivery list document for the record in the Sales.DeliverySchedule table with a ScheduleID of 1. Execute the query, and then verify that the desired results are returned. Type the necessary Transact-SQL to retrieve the DeliveryDriver column in the Sales.DeliverySchedule table, where the XML in the DeliveryList column includes a Delivery element with a SalesOrderID attribute of 43659. Execute the query, and then verify that the desired results are returned. Type the necessary Transact-SQL to retrieve a delivery route document in the following XML format from the DeliveryList column in each row of the Sales.DeliverySchedule table. Notice that each delivery route XML document contains a DeliveryRoute element, in which a RouteNo element contains the DeliveryRoute column value. The XML then contains the Address element for each Delivery element in the DeliveryList document. <pre> <DeliveryRoute> <RouteNo>3</RouteNo> <Address>7194 Fourth St. Rockhampton</Address> <Address>6445 Cashew Street, Rockhampton</Address> </DeliveryRoute> </pre> Execute the query, and then verify that the desired results are returned. Type the necessary Transact-SQL to update the Sales.DeliverySchedule table, setting the first Address element value in the delivery list document for the record with a ScheduleID of 1 to 7194 Fourth St. Rockhampton. Execute the query, and then verify that the desired results are returned. Save all files, and then close SQL Server Management Studio. 	Delivery Driver	Delivery	The DeliveryDriver column value	All Delivery XML elements in the DeliveryList document
Delivery Driver	Delivery				
The DeliveryDriver column value	All Delivery XML elements in the DeliveryList document				

Results checklist

You can use the following checklist of results to verify whether you have successfully performed this lab:

- Created two FOR XML queries
- Created a working set of OPENXML functions that insert data into the **Sales.SalesOrderHeader** and **Sales.SalesOrderDetail** tables
- Created a table named **Sales.DeliverySchedule** that contains an **xml** data type column
- Created various **xml** method queries, including **query**, **value**, and **modify**

