Processing XML Documents in Transact-SQL

Objectives

- Learn how SQL Server can process XML documents.
- Use OpenXML to parse XML and create relational rowsets.
- Control the results that are returned from OpenXML.
- Extract data from an XML document and insert it into SQL Server tables.
Understanding the XML Document Object Model (DOM)

Any XML document can be represented as a tree of hierarchically nested nodes. The document as a whole is the root node that branches out to include all of the elements, attributes, and processing instructions contained in the document.

The World Wide Web Consortium (W3C) has formalized a standard object model for processing XML documents, called the Document Object Model (DOM). You can learn more about the DOM specification by visiting http://www.w3.org/DOM.

Microsoft’s DOM Implementation in MSXML

Microsoft’s implementation of the DOM is part of MSXML, the XML parser that is included with Internet Explorer (updated versions of the MSXML parser are also available as separate downloads from Microsoft at http://msdn.microsoft.com/xml).

Programs and scripts can use an implementation of the DOM to dig into an XML document, pull out selected information from it, and even to modify the document or build a new one.

As new XML standards have evolved, Microsoft has added support for them in MSXML. You can use MSXML methods to transform XML documents using stylesheets or to validate them using XML Schema.

Support in SQL Server 2000

SQL Server has not added full support for all the objects and methods described in the Document Object Model. However, there is a new system stored procedure, sp_xml_preparedocument, that allows you to load an XML document into a tree structure in memory from within Transact-SQL.

After you load your XML into memory, you can use XPath expressions to select a subset of nodes from the document, and you can format the data in those nodes into a relational rowset, by using the new rowset function OpenXML. You can then filter, sort, and join to the results you get back from OpenXML just like you would with the result set from a standard SELECT statement. You can also easily use the extracted data to update database tables. When you finish working with the XML data, you use the
sp_xml_removedocument system stored procedure to remove the XML tree from memory.

**Limitations of the DOM**

The biggest limitation when using an implementation of the DOM to process XML is that it requires the entire document to be represented in a memory structure. A 10 KB XML document can easily occupy 10 MB of memory when loaded into a DOM.

The DOM is great if you need to navigate about freely in a document. However, if you only need to extract a few pieces of data from a large XML document, you will pay a high performance penalty if you must load the entire document into memory first.

When performance and efficiency are your primary concerns, another model for working with XML is often a better choice. One commonly used alternative that hasn’t been formalized as a standard by the W3C is the Simple API for XML (SAX). SAX is an event-based model for parsing XML by streaming through it in one pass and responding to the data that is encountered.

In Web Release 1 for SQL Server 2000, Microsoft added an XML Bulk Load utility for importing large XML documents into a SQL Server database. Although this utility doesn’t use the SAX model, it follows a similar pattern of maximizing performance by streaming through the data efficiently in a single pass. If you need to process large XML imports, XML Bulk Load may be a better choice than OpenXML. You’ll learn how to install and use XML Bulk Load in a later chapter.
Parsing XML Data with OpenXML

In order to use the OpenXML rowset function, you first create an in-memory tree representation of the XML document, using the system stored procedure `sp_xml_preparedocument`. This procedure parses the XML document and loads it into a DOM memory structure that mirrors the structure of the original XML. The OpenXML function reads that DOM structure and provides you with a rowset that you can use like a table or a view.

A Data Input Scenario

Imagine that your Northwind head office has sent you a data list describing your customers demographically, listing each one under a heading such as European, North American, South American, and so on. You want to feed the info into Northwind’s demographics table, `CustomerCustomerDemo`.

The data you will be processing came to you as an XML file containing information on customers and their demographic types. You’ll need to parse this XML data and insert it in the CustomerCustomerDemo table, which contains CustomerID and CustomerTypeID columns. The data you’ve received looks like the XML shown here:
<?xml version="1.0" ?>
<demographics>
    <CustType CustomerTypeID="EUROP" CustomerDesc="European">
        <Cust CustomerTypeID="EUROP" CustomerID="DRACD" />
        <Cust CustomerTypeID="EUROP" CustomerID="MAISD" />
        <Cust CustomerTypeID="EUROP" CustomerID="VINET" />
    </CustType>
    <CustType CustomerTypeID="NAMER" CustomerDesc="North American">
        <Cust CustomerTypeID="NAMER" CustomerID="BOTTM" />
        <Cust CustomerTypeID="NAMER" CustomerID="HUNGC" />
    </CustType>
    <CustType CustomerTypeID="SAMER" CustomerDesc="South American">
        <Cust CustomerTypeID="SAMER" CustomerID="CACTU" />
        <Cust CustomerTypeID="SAMER" CustomerID="GROSR" />
        <Cust CustomerTypeID="SAMER" CustomerID="RANCH" />
    </CustType>
</demographics>

You plan to load the data into the CustomerCustomerDemo table. But first you need to know how to read the data from the XML document.

OpenXML Fundamentals

Load OpenXML .sql in Query Analyzer

You decide that you’ll first perfect your skill at reading the data. Later you’ll write it out to the appropriate table. You prepare a stored procedure, designed simply to test whether you can return rowset data from the XML document, as shown in Figure 1.
The procOpenXML_read procedure expects to be handed an XML document as input, and it includes an ntext parameter, @xmlDoc, for that purpose. You must also declare an integer variable to hold a pointer to the in-memory representation of the XML document that the stored procedure sp_xml_preparedocument will create. You execute the stored procedure, handing it @idoc, the integer variable, and @xmlDoc, the XML document. After sp_xml_preparedocument runs, @idoc contains a handle to the internal DOM representation that was created from the XML. This allows you to work...
with multiple DOM representations at once and to use the integer handle to identify the one you want to use.

Next, you issue a SELECT statement using the OpenXML rowset function, referencing @idoc as your data source. Specify the rowpattern (/Demographics/CustType/Cust), which is an XPath expression indicating the branch in the XML tree that you want to use as your data-retrieval starting point. The flags parameter, 0, means that the default is for SQL Server to look at attributes rather than elements when locating data. You’ll learn more about that flags parameter later.

**Using XPATH to Locate Data**

The XPath rowpattern parameter of OpenXML begins from the document root of the XML tree, and traces a path through the top-level element *Demographics*, and on through *CustType*, and settles at *Cust*, meaning that you intend to use that set of Cust nodes as the context for your rowset.

The WITH clause (called the *schema declaration*) lets you specify the exact data that you want to return, as well as the names, data types, and sizes of the columns in the rowset that you will be creating.

Following each of the data type declarations, the expressions in single quotes (column patterns) are also XPath expressions. These column patterns can override the flags parameter you specified earlier, so that you can get data from elements rather than attributes, and these XPath expressions also allow you to move through the tree of XML to retrieve data as needed. In this example, all the data is being pulled from the Cust element, but you can use the XPath patterns to get data from any element in the document, as you’ll see later.

The column pattern expressions in this call to OpenXML mean: “Retrieve the attributes named CustomerID and CustomerTypeID from the currently defined XPath level.” Each “@” sign indicates that you’re looking for an attribute rather than a sub-element.

**Releasing Memory**

After you get the data you’re interested in, clean up after yourself by removing the XML document’s internal tree from memory, using `sp_xml_removedocument`. The call to `sp_xml_removedocument` takes only one parameter—the @idoc handle that you got from `sp_xml_preparedocument` earlier.
Testing the Procedure

The next step is to test your stored procedure. You can do this using the Transact-SQL shown in Figure 2, which assigns the text of an XML document as a local variable, and then feeds it to procOpenXML_read.

```
DECLARE @xmlDoc NCHAR(4000)
SET @xmlDoc =
'<?xml version="1.0"?>
<demographics>
  <CustType CustomerTypeID="EUROP" CustomerDesc="European">
    <Cust CustomerTypeID="EUROP" CustomerID="DRACD"/>
    <Cust CustomerTypeID="EUROP" CustomerID="MAISD"/>
    <Cust CustomerTypeID="EUROP" CustomerID="VINET"/>
  </CustType>
  <CustType CustomerTypeID="NAMER" CustomerDesc="North American">
    <Cust CustomerTypeID="NAMER" CustomerID="BOTTM"/>
    <Cust CustomerTypeID="NAMER" CustomerID="HUNGSC"/>
  </CustType>
  <CustType CustomerTypeID="SAMER" CustomerDesc="South American">
    <Cust CustomerTypeID="SAMER" CustomerID="CACTU"/>
    <Cust CustomerTypeID="SAMER" CustomerID="GROSRS"/>
    <Cust CustomerTypeID="SAMER" CustomerID="RANCH"/>
  </CustType>
</demographics>'

-- Call stored proc to retrieve XML data as rowset
EXEC procOpenXML_read @xmlDoc
```

Figure 2. The result of running the procOpenXML_read stored procedure, which returns data from an XML document.
The stored procedure has performed exactly as advertised, providing a set of rows and columns from the XML document. Before you move on to insert the data into the database, take a closer look at the choices you have when shaping the rowset that OpenXML returns.
Defining the OpenXML Rowset

The WITH clause that you use when calling OpenXML gives you complete freedom to include any data from the XML document in the result set that is returned.

Schema Declarations

In the previous example, you used what is called a schema declaration in the WITH clause:

```
SELECT * FROM OpenXML
(@idoc, '/demographics/CustType/Cust', 0)
WITH (CustomerID NCHAR(5) '@CustomerID',
     CustomerTypeID NCHAR(10) '@CustomerTypeID')
```

Each item in the schema declaration contains up to four parts, separated by spaces. The first two parts, called ColName and ColType, contain the column name and data type, and they are required in any schema declaration. The third part, which can hold an XPath expression, is called a ColPattern, and it is optional. The fourth part, which is also optional and is missing in this example, is called MetaProperty, and it allows you to include information from the XML document that isn’t contained in elements or attributes, such as relative position or namespace.

The ColName and ColType need little explanation. They allow you to create aliases for your columns and to define the SQL Server data types used to hold the data. SQL Server will attempt to coerce the XML data into the specified types.

Using Column Patterns

The XPath RowPattern ('/demographics/CustType/Cust' in this example) already identifies a subset of the entire document. The ColPatterns are only needed if you want to narrow down or branch out from that subset.
In the example, the ColPattern values that appear in the schema declaration are actually unnecessary. You could alter the procedure and leave them out without changing the results. The RowPattern establishes the Cust elements as the default set of nodes to look at, and the ColNames exactly match attributes found in the Cust elements. So, you get the same results even if you leave out the optional ColPattern XPath expressions like this:

```sql
SELECT * FROM OpenXML (@idoc, '/demographics/CustType/Cust', 0)
WITH (CustomerID NCHAR(5),
     CustomerTypeID NCHAR(10))
```

You only need to use ColPatterns to specify data selections that aren’t already defined by the other parameters. For example, suppose you wanted to add a column for the CustomerDesc attribute values that appear in the CustType elements, and suppose you want to call it Description.

CustType is one level above the Cust node context defined in the RowPattern. In XPath (as in DOS), two dots are used to move up a level in the tree hierarchy. Figure 3 shows the new schema declaration and the results. There are XPath syntax options that allow you to select any data in the document.

See *OpenXML.sql*
Using Table Names

If the results that you want exactly match column names and data types that already exist in a table in your database, then you can use the name of that table rather than a schema declaration in your WITH clause.

In the Northwind example, there is a table called CustomerCustomerDemo that contains the same two columns of data that you want to extract from the XML document. The name of the table is all you need to get the desired results, as shown in Figure 4.
We’ve barely scratched the surface of all the options OpenXML provides for defining the rowset. You can vary the Flag values to control the mix of elements and attributes that are queried by default. You can use metaproperty values in schema declarations to extract additional data from your XML document. You can even leave out the WITH clause completely and get a special result set, called an edge table, that gives you a relational view of all the data in the XML tree.

If you’re interested in digging deeper, you’ll find that Microsoft’s SQL Server Books Online contains thorough coverage of all these possibilities.
Using OpenXML to Modify Data

Inserting Data from XML into SQL Server with OpenXML

Before you can insert the customer demographics data into the Northwind CustomerCustomerDemo table, there is one little piece of business you need to take care of. Northwind has established referential integrity between this table and CustomerDemographics, which holds the IDs and descriptions of valid customer demographic types. Run the following Transact-SQL to insert the types you will be working with. We’ll assume that you already have these types, and that all you’re looking for from the XML file is the assignment of types to customers.

```sql
INSERT INTO CustomerDemoGraphics
VALUES('EUROP', 'European')
INSERT INTO CustomerDemoGraphics
VALUES('NAMER', 'North American')
INSERT INTO CustomerDemoGraphics
VALUES('SAMER', 'South American')
```

Now you’re ready to extract the data you need from the XML file and insert it into Northwind. This is a good opportunity to make use of one of SQL Server 2000’s new features: the table data type. You can use a table variable to build an in-memory table structure to hold the rowset that OpenXML returns. The in-memory table can then serve as the source for inserting rows into the Northwind tables. This technique is especially useful if you need to make several passes at several XML files to build up the table structure you want, which won’t be necessary for this simple example.

Here’s the Transact-SQL code to create procOpenXML_write. It is very similar to the code for reading XML. The only difference is that this time the data is cached in a table variable and then inserted into the database:
CREATE PROCEDURE procOpenXML_write @xmlDoc NTEXT
AS
BEGIN

-- Integer to hold XML document handle
DECLARE @idoc INT

-- Declare in-memory table to hold data from XML doc
DECLARE @demog TABLE (
    customer_id      NCHAR(5),
    customertype_id  NCHAR(10) )

-- Use stored proc to build internal representation
-- of the XML doc
EXEC sp_xml_preparedocument @idoc OUTPUT, @xmlDoc

-- Insert XML data into table variable
INSERT INTO @demog
SELECT * FROM OpenXML
    (@idoc, '/demographics/CustType/Cust', 0)
WITH (  
    CustomerID      NCHAR(5) '@CustomerID',
    CustomerTypeID  NCHAR(10) '@CustomerTypeID'
)

-- Clean up: clear the XML document out of memory
EXEC sp_xml_removedocument @idoc

-- Create new entries in CustomerCustomerDemo
INSERT INTO CustomerCustomerDemo
SELECT DISTINCT customer_id, customertype_id
FROM @demog
END
To execute the procOpenXML_write procedure that inserts the customer data into CustomerCustomerDemo, use the OpenXML_WriteTest.sql query. This is very similar to the earlier OpenXML_ReadTest.sql query, except that it calls procOpenXML_write.

Finally, you can test that the demographic data was correctly inserted in CustomerCustomerDemo by running this ad hoc query:

```sql
SELECT * FROM CustomerCustomerDemo
```
Summary

- The Document Object Model (DOM) is a standard model for processing XML documents.
- Microsoft’s implementation of the DOM is part of MSXML, the XML parser that is included with Internet Explorer.
- Programs and scripts can use an implementation of the DOM to navigate within an XML document, pull out selected information from it, and even to modify the document or build a new one.
- You can use MSXML methods to transform XML documents using stylesheets or to validate them using XML schema.
- SQL Server 2000 has a system stored procedure, sp_xml_preparedocument, that allows you to load an XML document into a tree structure in memory from within Transact-SQL.
- When you finish working with the XML data, use the sp_xml_removedocument system stored procedure to remove the XML tree from memory.
- The DOM requires the entire document to be represented in a memory structure when it processes XML.
- SAX is an event-based model for parsing XML by streaming through it in one pass and responding to the data that is encountered.
- In order to use the OpenXML rowset function, you first create an in-memory tree representation of the XML document, using the system stored procedure sp_xml_preparedocument.
- The WITH clause used when calling OpenXML lets you specify the exact data that you want to return, as well as the names, data types, and sizes of the columns in the rowset that you will be creating.
- You can use schema declarations or table names in the OpenXML WITH clause.
- The data returned by OpenXML can be inserted into XML tables using standard Transact-SQL insert statements.
(Review questions and answers on the following pages.)
Questions

1. What is a DOM?

2. What is the name of Microsoft’s XML parser, which implements the DOM and is included with Internet Explorer?

3. What does the SQL Server 2000 system stored procedure, `sp_xml_preparedocument`, do?

4. How is the DOM different from SAX in the way it handles data?

5. Why do you use the WITH clause when calling OpenXML?
Answers

1. What is a DOM?
   The DOM, or Document Object Model, is a standard model for processing XML documents by loading them into an in-memory tree structure.

2. What is the name of Microsoft’s XML parser, which implements the DOM and is included with Internet Explorer?
   MSXML

3. What does the SQL Server 2000 system stored procedure, sp_xml_preparedocument, do?
   It allows you to load an XML document into a tree structure in memory from within Transact-SQL.

4. How is the DOM different from SAX in the way it handles data?
   The DOM requires the entire document to be represented in a memory structure when it processes XML, whereas SAX is an event-based model for parsing XML by streaming through it in one pass and responding to the data that is encountered.

5. Why do you use the WITH clause when calling OpenXML?
   The WITH clause allows you to use schema declarations or table names to define the columns that will be returned.
Lab 5: Processing XML Documents in Transact-SQL

TIP: Because this lab includes some typed code, we’ve tried to make it simpler for you. You’ll find all the code in ProcessingXML.txt. To avoid typing the code, you can copy/paste it from the text file instead.
In this lab you’ll learn to create a relational result set from XML using OpenXML.

To complete this lab, you’ll need to work through one exercise:

- Using OpenXML

The exercise includes an “Objective” section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.
Using OpenXML

Objective

Use OpenXML in Transact-SQL to create a result set from the data in the XML file named cactuorders.xml.

Things to Consider

• What system stored procedures will you need to use with OpenXML?
• How will you specify the data that you want included in your results?

Step-by-Step Instructions

1. Open the SQL Query Analyzer and declare the necessary variables.

   DECLARE @idoc INT
   DECLARE @xml DOC NCHAR(4000)
2. Assign the XML data to `@xmlldoc`:

```sql
SET @xmlldoc =
  '<root>
    <Orders CustomerID="CACTU" OrderID="10521">
      <OrderDetails ProductID="35" Quantity="3"/>
      <OrderDetails ProductID="41" Quantity="10"/>
      <OrderDetails ProductID="68" Quantity="6"/>
    </Orders>
    <Orders CustomerID="CACTU" OrderID="10782">
      <OrderDetails ProductID="31" Quantity="1"/>
    </Orders>
    <Orders CustomerID="CACTU" OrderID="10819">
      <OrderDetails ProductID="43" Quantity="7"/>
      <OrderDetails ProductID="75" Quantity="20"/>
    </Orders>
  </root>'
```

3. Prepare the XML document as a tree structure in memory:

```sql
EXEC sp_xml_preparedocument @idoc OUTPUT, @xmlldoc
```

4. Select the columns required using OpenXML and XPath expressions:

```sql
SELECT * FROM OpenXML
(@doc, '/root/Orders/OrderDetails', 0)
WITH {
  CustomerID NCHAR(5) '.../@CustomerID',
  OrderID INT '.../@OrderID',
  ProductID INT '@ProductID',
  Quantity SMALLINT '@Quantity'
}
```

5. Remove the XML document from memory:

```sql
EXEC sp_xml_removedocument @idoc
```