Real World XML processing in SQL Server 2005

Shredding typed XML data into SQL recordsets

It's increasingly difficult as a SQL DBA or developer to avoid having to deal with XML data from external sources. Since the introduction of the XML native datatype in SQL Server 2005 it's simple to store and query XML data within databases, but how do you go about transforming XML data with fixed schemas into SQL datasets?

The various commands and concepts involved are explained at considerable length in SQL Server Books Online[i], but when I needed to parse Excel 2002-generated XML data I found that the information and examples provided didn't address the problems I was facing, particularly those relating to the fixed schema imposed by Excel. In this article, I hope to explain the problems and solutions in some depth. It may look complicated at first glance but if you examine each part (and read the related Books Online topics) then it should quickly become clear. The general principles apply to all fixed-schema XML data.

The source data

The source data is stored in a staging table named tbStagingCsvImport. This is a simple staging table populated using an SSIS package with just two columns, an ID (PK, incrementing integer) and a column named CSVContent with datatype xml.

Each record contains a complete XML document as generated by Excel 2002. The source data in the example used comes from the Windows Performance monitor, but any Excel 2002 worksheet saved as XML will show a similar structure. The example used for this document is available here. The goal was to take each individual XML doc and shred it into a SQL recordset to be stored in another table, tbSQLData, with the following structure:

```
create      table tbSQLData
      (ID   int   identity (1, 1) not null primary key,
      DocFK int   not null,
      RowID int   not null,
      ColID int   not null,
      MPID  int   not null,
      Type  nvarchar(16) not null,
      Data  nvarchar(128) not null)
```

Let's examine the XML data for a moment. A very simplified representation of the structure looks like this:

```
<Workbook>
      <Worksheet>
            <Table ss:ExpandedColumnCount="14" ss:ExpandedRowCount="12>
                  <Row>
                        <Cell>
                              <Data ss:Type="(datatype)">Value</Data>
                        </Cell>
                        …
                        </Cell>
                  </Row>
                  …
                  </Row>
            </Table>
      </Worksheet>
<Workbook>
```

In our example, we've got an XML document containing 12 row elements, each containing 14 cell elements. We want to create one new SQL record in tbSQLData for each <Cell> element, so we should end up with 168 SQL records.

Testing it for yourself

If you want to try this example out for yourself, you'll need to create the tbSQLData table as shown above in a test database. In the same db, copy and run the script "Create test data" (download all files from http://www.360data.nl/en/docs/080123_XML.aspx) to create the staging table and populate it with test data.

The stored procedure

After much experimentation, I managed to create a stored procedure that did exactly what it needed to do. The entire SP script can be downloaded from http://www.360data.nl/en/docs/080123_XML.aspx; read on for a blow-by-blow explanation.

The SP has two input parameters, @p_DocNr and @p_List. The first is the ID number of the staging table record that is to be parsed, the second is used here for illustrative purposes only; we can use it to examine how the SP works in closer detail, but it's not strictly necessary for parsing to succeed.

Line 157

Skipping past all the boilerplate TSQL to the meat of the SP, we come to the first variable assignment:

```
select      @v_ImportXML =          CSVContent
from        tbStagingCsvImport
where       ID =                    @v_DocFK;
```

This quite simply populates an XML-typed variable with the XML data from the staging table. We'll use this variable for all subsequent processing.

Line 166

The next step is to populate the @v_NumRows variable with the number of rows in the document. We'll use this to loop through the XML doc one row at a time, from row 1 up to row @v_NumRows.

We could count the number of occurrences of the row element using the SQL XQuery **count** function, but in this instance the row count is an attribute of the <Table> element so we can query it directly.

```
<Table ss:ExpandedColumnCount="14" ss:ExpandedRowCount="12"
x:FullColumns="1" x:FullRows="1">
```

We'll need to declare the "ss" namespace to be able to parse this element correctly. To do this we'll use the WITH XMLNAMESPACES clause as follows:

```
with xmlnamespaces (default 'urn:schemas-microsoft-
com:office:spreadsheet','urn:schemas-microsoft-com:office:spreadsheet' as
ss)

select @v_NumRows =
@v_ImportXML.value('(/Workbook//Table/@ss:ExpandedRowCount)[1]', 'int')
```

What do these values mean and how do we know where to get them from? If we look at the beginning of the XML source document again we can see the namespaces in use:

```
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
xmlns:o="urn:schemas-microsoft-com:office:office" xmlns:x="urn:schemas-
microsoft-com:office:excel" xmlns:ss="urn:schemas-microsoft-
com:office:spreadsheet" xmlns:html="http://www.w3.org/TR/REC-html40">
```

So if we want to parse the <Table> element we need to declare the default namespace (urn:schemas-microsoft-com:office:spreadsheet) as <Table> has no namespace prefix and the "ss" namespace (urn:schemas-microsoft-com:office:spreadsheet) because we need it for the "ss:ExpandedRowCount" attribute; we can safely ignore the other namespaces for now.

Note the small but important differences in syntax between how the namespaces are defined in the XML document and how they appear in the WITH XMLNAMESPACES clause.

Note also that when you use WITH XMLNAMESPACES in a batch or stored procedure you must terminate the preceding statement with ';' or you'll get an error message.[ii]
We use the XQuery **value** method in the SELECT statement to get the value of the ExpandedRowCount attribute and return it as SQL datatype **int**. Even though the <Table> node only occurs once in our document, we're still required to explicitly declare that our path expression returns a single value, hence the "[1]" in the query string.

Line 180

The next important step is to pass the entire XML document into the SQL XML parser for further processing. To do this we call the system SP sp_xml_preparedocument[iii]:

```
exec sp_xml_preparedocument @v_Handle output, @v_ImportXML,
                  '<Workbook xmlns:w="urn:schemas-microsoft-
com:office:spreadsheet" xmlns:ss="urn:schemas-microsoft-
com:office:spreadsheet"/>'
```

Each in-memory XML document is assigned a handle value, so we'll populate our @v_Handle value with this. We also need to declare our namespaces again. This time, you'll notice that we've added the "w" suffix to what was the default namespace in the previous example. Why? Because sp_xml_preparedocument already uses the default value for metaproperties[iv] for this namespace (with the "mp" suffix – we'll use it later), so if we want to use something else we need to declare it ourselves. You'll notice that the syntax is different again, just to keep things interesting!

We've now got our XML doc in memory and ready to be parsed. The XQuery string per row looks like this:

```
/w:Workbook/w:Worksheet/w:Table/w:Row[(row number)]/w:Cell/w:Data
```

…which we can shorten to…

```
/w:Workbook//w:Row[(row number)]//w:Data
```

Line 193

This is where we start looping through the rows. For each row in the XML doc, we'll query the content using OPENXML to return a SQL rowset.

```
set   @v_XQStr =  N'/w:Workbook//w:Row[' +
            rtrim(cast(@v_CurrentRow as nvarchar(5))) +
                  ']//w:Data'
```

Line 210

This is where the row parsing takes place and is worth examining in fine detail.

```
select      @v_DocFK as DocFK, @v_CurrentRow as RowID,
            row_number() over (order by ID) as ColID,
            *
from        openxml (@v_Handle, @v_XQStr, 3)
            with  (ID    int              '@mp:id',
                   Type  nvarchar(16)     '@ss:Type',
                   Data  nvarchar(128)    '..//w:Data')
```

We want to be able to track the precise origin of each value that ends up in our SQL table, so we get the doc variable value and the current row number from the existing variables. We use the row_number() function to generate a column id (in this case a row because we're transposing the set).

So far, so normal. The "SELECT * FROM OPENXML"[v] part of the statement is where the real work takes place. The "3" in "`openxml (@v_Handle, @v_XQStr, 3)`" is a bitmask and it indicates that we'll be using a mixture of attribute- and element-centric mapping. In this example, the "Type" value is an attribute property, while the "Data" value is an element value, so we need to use the mixed mapping. We'll also store the metadata ID in our data table by querying the metaproperties using the "mp" namespace.[vi]

The WITH clause is where we define the SQL schema mapping for the parsed data; we declare a table in the format ColumnName, SQL Data type, and the XPath node mapping for the column. Once this is applied we've got a SQL dataset which we can insert into our table (as at line 208).

After that we increment the current row variable and loop through until all rows have been consumed and that's it: the data is now processed. The only thing remaining is to dump the XML doc out of memory to free resources. This is important as, according to Books Online, the SQLXML parser uses one-eighth of the total memory available to SQL. This is accomplished very easily as follows:

```
exec sp_xml_removedocument @v_Handle
```

Paul Clancy
360Data
http://www.360data.nl

References

[i] Using XML in SQL Server, http://msdn2.microsoft.com/en-us/library/ms190936.aspx

ii The GO Command and the Semicolon Terminator,
http://www.sqlservercentral.com/articles/SQL+Puzzles/thegocommandandthesemicolonterminator/2200/

[iii] sp_xml_preparedocument, http://msdn2.microsoft.com/en-us/library/ms187367.aspx

[iv] Specifying Metaproperties in OPENXML, http://msdn2.microsoft.com/en-us/library/aa226531.aspx

[v] Using OPENXML, http://msdn2.microsoft.com/en-us/library/aa226522(SQL.80).aspx

[vi] From Books Online: "The default value is **<root xmlns:mp="urn:schemas-microsoft-com:xml-metaprop">**. *xpath_namespaces* provides the namespace URIs for the prefixes used in the XPath expressions in OPENXML by means of a well-formed XML document. *xpath_namespaces* declares the prefix that must be used to refer to the namespace **urn:schemas-microsoft-com:xml-metaprop**; this provides metadata about the parsed XML elements. Although you can redefine the namespace prefix for the metaproperty namespace by using this technique, this namespace is not lost. The prefix **mp** is still valid for **urn:schemas-microsoft-com:xml-metaprop** even if *xpath_namespaces* contains no such declaration."