XML Argument Protocols for SQL 2005 Stored Procedures

This article describes using some of the new XML functionality provided by MS SQL Server 2005 as applied to argument passing to T-SQL stored procedures. One of the more tedious areas of application support is managing arguments passed to and from T-SQL stored procedures. Unlike other RDBMS engines such as Oracle, T-SQL does not support the notion of passing in record structures as arguments from a client directly to a T-SQL stored procedure. Realizing that the new XML functionality might offer some relief in this area, I did some searching to see what anyone else had come up with. I found some articles related to XML, but the articles I did find used SQL 2000 and had the primary focus of using XML for bulk Inserts with no preliminary data validations or other processing steps (see References below). As I had some different objectives in mind, and I found no articles highlighting the new SQL 2005 XML features, I decided to present my ideas and results here.

Background

The use of MS SQL Server as a web back-end storage system is well understood. Maintaining the stored procedures that interact with web clients can be tedious, with the lack of record structure support to and from stored procedures a contributing factor. There are several consequences to this, including:

- Commonly used procedures have multiple copies created with slightly different arguments
- There are many instances where passing numerous arguments and/or varying number of arguments to stored procedures can be awkward or nearly impossible
- Altering the arguments to a commonly used procedure can be a high-maintenance task any dependent code snippets must be changed in parallel
- T-SQL does not include a formal mechanism to pass in record structures from a client application, much less support passing in a varying number of records

Transaction support is another labor intensive effort: In a web back end, the database context, or session, only exists as long as that one procedure call is active. Thus, a transaction that requires multiple procedure calls requires some way to wrap the transaction/rollback around the multiple invocations. For example, storing an e-commerce transaction can involve a dozen tables and multiple, varying number of rows per table (e.g. sales line items) which is really impossible to pack into standard procedure call arguments. Some people use this difficulty as a reason to embed SQL in their application code instead of placing it into a stored procedures.

Another interesting case arises when you want to send a stored procedure the minimal number of columns in a row to update. First, we have the issue of variable number of arguments (which columns are getting updated?); next, we are faced with future maintenance issues as columns are added to (or deleted from) the table; then the update code has to decipher a tri-state assertion for each column in the table: Do we leave the column alone, update it to a value, or update it to Null? Traditional argument calling sequences might involve two arguments per column, one to pass the new value, and an indicator variable to say if we are really updating or not - not pretty. The open structure of XML lets us define such things easily.

Finally, there is the reverse interface issue of sending back to the calling program multiple values, possibly structured; for this discussion I will use error messages. Suppose we have a row insert procedure, and we do a number of validations on the column values before executing the Insert command. What we do not want to do is bail out after detecting the first error, go back to the user with one error, they fix it and resubmit, we bail out on the next error, etc. until the user gets dizzy. We would like to do all our data validations and pass back all the conditions and messages in one pass, so the user can fix them in one try. Here again we need to pass multiple results (independent of a dataset), which the basic T-SQL Procedure argument list makes difficult if not impossible.

Thus the issues at hand are:

- Number of arguments (complexity), and maintenance issues (add/delete arguments)
- Tendency to create multiple copies of the same procedure for slightly varying input conditions
- Passing a record structure
- Passing a varying number of multiple records
- Adding processing criteria to arguments (e.g. no update; update with value; update to Null)
- Pass back a varying number of (potentially structured) values, e.g. error messages.

XML As A Solution

T-SQL has native support for manipulating XML in SQL 2000, but with the release of SQL 2005, XML support is even better. T-SQL arrives with additional functions and an actual XML native datatype [essentially an nvarchar(max) with storage in "optimised UTF-16 characters" per BOL]. There are some curious comments in the BOL about the overhead of XML processing, but I have not yet tried to measure it. My general rule is, if a certain syntax or process makes for increases in programming efficiency and reliability as well as reducing long term maintenance, I care less about a little additional overhead on the machine.

Client Code

On the application (e.g. web page) side, instantiating the MSXML object to create an XML document certainly has overhead, especially as it supports a lot of esoteric XML syntax that may be seldom used. With the new SQL Server 2005 enhancements, some of the client-side processing can be entirely relieved: XML arguments can be treated as a string and passed to a stored procedure without any need for complicated objects to help us. You should certainly use the MSXML object if it fills other requirements the client-side application needs. Note: If you do not use the MSXML object, then you need to handle the translation of reserved characters into their alternate representations, a.k.a. 'entities' . For example the XML reserved character "<" [less than sign/open angle bracket] in user data must be replaced with the XML Entity "`&lt`;". The same holds for ">", "'", """, and "&". You could write a small function to handle those [hint: process "&" first].

Using XML makes the calling convention to stored procedures very simple: As the XML is by its nature structured, when we extract values out of the XML from inside the procedure, the code need make no assumptions about ordering of individual data items; handling data element existence or non-existence is also handled gracefully. The presence of possibly obsolete data items is no longer a bother to the stored procedure - it simply ignores them. All of this helps reduce maintenance.

Web Server Application Code

We utilize the fact that an XML document can be represented by a character string, and simply append and build it. Note, if you really wanted, you could instantiate the MSXML object and build your XML with that, but this seems far too much for the job at hand. My presumption is that all of the user input has been collected and validated to some extent by the application program, and is then to be passed to the database as a single transaction.

Example: Insert Customer Row [vbscript]

```
xcmd = "<arg><first_name>" & firstName & "</first_name><last_name>" & lastName &_
"</lastname></arg>"
```

where arg is the root tag for the whole document, and firstName, lastName etc. are the variables holding the user input. If a data item is missing, the stored procedure can fill in a default or raise an error, depending on the business rules involved.

Example: Update Customer Row [vbscript]

```
xcmd = "<arg><customerid>74285</customerid><first_name>Robert</first_name>" &_

 "<company_name null='yes'/></arg>"
```

This could be sent to an Update procedure. It provides a primary key, a new first name, and an explicit instruction to set the company name to Null; all other columns are unmodified.

Example: Create Sales Transaction [vbscript]

```
xcmd = "<arg><customer><first_name>Mary</first_name>. . . .</customer>" &_
 "<order_header><order_date>08/15/2006</order_date>. . . .</order_header>" &_
 "<order_lines count='6'><line productid='7294' . . .>" &_
      "<line productid='8241' . . .>. . . .</order_lines>" &_
 "<payment><type>Visa</type>. . . .</payment></arg>"
```

Here we consolidate multiple record types, and multiple records for sales line items, all into one procedure call. This can then easily be wrapped in a single Transaction in case of unexpected database errors. The line count is optional, but if you know it then why not send it along, the procedure can make use of it. Otherwise the procedure will have to loop on searches for order_lines until it runs out of them. Remember that in most cases the non-significant white space (blanks, new lines, etc. in between elements) are ignored and often disappear when the XML document is stored in a canonical form, such as an XML variable.

I have written web page scripts that do just this kind of processing. First you call a procedure to add the customer. Then call another procedure to add the order header. Then a big loop call to insert each sale line item. Yet another procedure to post the payment. And if one of these burps along the way? Go write another procedure to try to clean up a half-baked transaction! That kind of processing is where this interface really shines.

Example: Error Message Return Results

```
<err>
 <error>
 <err_code>APP001</err_code><err_msg>First Name is a required field.</err_msg>
 </error>
 <error>
 <err_code>APP002</err_code><err_msg>Last Name is a required field.</err_msg>
 </error>
</err>
```

This might be returned by a procedure call. Multiple messages can easily be handled, as well as a record structure (shown here) composed of both an error code as well as the error text message. Successful execution is indicated by simply returning Null.

Stored Procedure Interface and Code

The calling program (a server-side web page script, for our examples) makes a standard procedure call, passing the string that has been built as an input XML argument. plus an output XML argument for the error status return. The variable types could typically be double-wide character, e.g. Unicode.

Here is an excerpt of a stored procedure to validate values then Insert a customer row.

```
/*********************************************************************************
file is xtest.sql Copyright (c) 2006 Jack A. Hummer
*/
CREATE PROCEDURE [dbo].[xtest]
/*
Purpose
 Example procedure demonstrating XML input and output arguments.
 Processing to validate and insert a Customer row.
Input */
@xin
XML, /* input values for columns
 <arg>
 <fname>...</fname> First Name, required
 <lname>...</lname> Last Name, required
 </arg>
 Missing fields, or fields passed as <xxxx
null="yes"/> are considered
 Null and will have defaults applied as needed.
Output */
@id_customer
INT = NULL OUTPUT, -- new primary key
@xout
XML = NULL OUTPUT /* error message(s), or Null = good return
 <err>

<error><err_code>nnn</err_code><err_msg>xxxxxxx</err_msg></error>
 </err>
Notes

Revision History
14aug06 jah created
*********************************************************************************
AS
DECLARE @first_name NVARCHAR(50), @last_name NVARCHAR(50), @nvl VARCHAR(3)
SET NOCOUNT ON;
SET @xout = NULL -- default is good return

-- process first name - required field
SET @nvl = @xin.value('(/arg/fname/@null)[1]',
'NVARCHAR(3)')
SET @first_name = @xin.value('(/arg/fname)[1]', 'NVARCHAR(50)')
IF @nvl = 'yes' OR @first_name IS NULL BEGIN

SET @xout = dbo.xerr(@xout, 'E001', 'First Name cannot be Null')
END

-- process last name - required field
SET @nvl = @xin.value('(/arg/lname/@null)[1]',
'NVARCHAR(3)')
SET @last_name = @xin.value('(/arg/lname)[1]', 'NVARCHAR(50)')
IF @nvl = 'yes' OR @last_name IS NULL BEGIN

SET @xout = dbo.xerr(@xout, 'E002', 'Last Name cannot be Null')
```

```
END
-- more field validations.....
-- Finished field validations

IF @xout IS NOT NULL RETURN
-- continue processing.....
-- A Try/Catch around other statements could also append an error.
RETURN
```

For convenience as well as modularity, I created a small T-SQL function to insert one error record into the XML error document. This will get enhanced in the future, for example returning the error message in the users language, or optionally writing it to a log file.

```
/*********************************************************************************
file is xerr.sql Copyright (c) 2006 Jack A. Hummer
*/
CREATE FUNCTION [dbo].[xerr]
/*
Purpose
 Example function demonstrating XML error arguments.
 Add one error record to the XML error structure.
Input */
(@xmsg XML, -- current error msg(s), or Null
@err_code
NVARCHAR(20), -- new error code
@err_msg
NVARCHAR(1000) -- new error message
)
-- Output
RETURNS XML /* updated document
 <err>

<error><err_code>nnn</err_code><err_msg>xxxxxxx</err_msg></error>
 </err>
Notes

Revision History
14aug06 jah created
*********************************************************************************
AS
BEGIN

-- Initialize the document if needed
IF @xmsg IS NULL SET @xmsg = '<err></err>'

-- Insert this error sequentially after any others
SET @xmsg.modify('
insert (
 <error>

<err_code>{sql:variable("@err_code")}</err_code>,

<err_msg>{sql:variable("@err_msg")}</err_msg>
 </error>
 )
as last into (/err)[1] ')

-- Return the result of the function
RETURN @xmsg
END
```

A few notes on this code and related programming issues:

- The XML data type acts very much like an object, a new twist for T-SQL. It is also very flexible, as it is valid as a variable, argument, or function return. However, from ASP, ADO did not like it as an output variable type.
- Many of the new methods/arguments require lower case only, an abrupt change from most vendors SQL syntax which has usually been case insensitive. Not my preference but, hey, they forgot to ask me.
- The .value method utilizes XQuery syntax. If you have not seen this before, it is a complex language used to find things inside an XML document. Don't try to learn it from BOL, look for an introductory tutorial on the web (but, most of the introductory material I found progressed very quickly into advanced language features without completely explaining the fundamentals).
- You will need to experiment a bit to see what happens when you search for element values or attribute values and they do or do not exist.
- The .modify method uses both standard XQuery plus new extensions that Microsoft had to invent. The standards people have not yet addressed XML update syntax. Re-read the BOL articles a few times until it starts to make sense. In particular, the first argument must litterally be a string litteral, which seems rather limiting at first, unless you manage to stumble across the "sql:variable()" work-around.
- Contrary to popular belief, apostrophes (a.k.a. single quote) may optionally be used around XML attribute values.
- The .value method returns the reserved entities back into their original characters.
- Within T-SQL, CAST or CONVERT an XML variable to a character string type leaves the special entities intact.
- The .value method will right-trim the return string according to the data type specification, with no error.

Special Note: when putting together these ideas into some real production code, I found the procedure return argument as an XML data type did not work coming back through ADO to vbScript, so I changed it to an NVARCHAR. This worked just fine when loading it into an MSXML object. The .Net infrastructure may provide enhanced support for passing XML.

Stored Procedure - XML Code Loop

So you may ask, How do I exactly process an unknown number of records in the input XML structure? Given the restrictive syntax on the built-in XML methods, the only solution I have come up with is sp_executesql. You will customize this for your situation, but here is a small example:

```
SET @n = 1
SET @params = N'@x XML, @value INT OUTPUT'
WHILE 1 = 1
 BEGIN
  SET @cmd = N'SET @value = @x.value(''(/arg/cat)[' +     CAST(@n AS NVARCHAR(2))
  EXECUTE sp_executesql @cmd, @params, @x = @xarg, @value = @id_cat OUTPUT

  IF @id_cat IS NULL
   BREAK
   -- do some processing
 SET @n = @n + 1
END -- while
```

Example: ASP Calls the Stored Procedure [vbscript]

```
Dim xin, xout,id_customer, conn, sqlcmd
xin = "<arg><fname>Mary</fname></arg>"

Set conn = Server.CreateObject("ADODB.Connection")
Set sqlcmd = Server.CreateObject("ADODB.Command")
conn.Open dsn
sqlcmd.ActiveConnection = conn
sqlcmd.CommandText = "dbo.xtest"
sqlcmd.CommandType = adCmdStoredProc
sqlcmd.Parameters.Append
sqlcmd.CreateParameter _
 ("@xin", adVarChar, adParamInput, 1000, xin)
sqlcmd.Parameters.Append
sqlcmd.CreateParameter _
 ("@id_customer", adInteger, adParamOutput)
sqlcmd.Parameters.Append
sqlcmd.CreateParameter _
 ("@xout", adVarWChar, adParamOutput, 2000)
sqlcmd.Prepared = True
sqlcmd.CommandTimeout = 40
sqlcmd.Execute , , adExecuteNoRecords
xout = sqlcmd.Parameters("@xout")

If Not IsNull(xout) Then
 Dim oDoc, loadOk, oNodes, oNode, errCode, errMsg
 Set oDoc = Server.CreateObject("MSXML2.DOMDocument.4.0")
 oDoc.async = False
 loadOk = oDoc.loadXML(xout)

 Set oNodes = oDoc.selectNodes("/err/error")

 For Each oNode In oNodes
 Response.Write oNode.nodeType & "<br>"
 errCode = oNode.selectSingleNode("err_code").nodeTypedValue
 errMsg = oNode.selectSingleNode("err_msg").nodeTypedValue

Response.Write("<tr><td>Error Code: " & errCode & "</td><td>" &_

 errMsg & "</td></tr>" & vbCrLf)
 Next

 Set oNode = Nothing
 Set oNodes = Nothing
 Set oDoc = Nothing
End If
```

The above is a very simple example, but demonstrates everything we have discussed. The error display to the web page could be wrapped in a small function, and as an alternative could be handled nicely with some XSLT code as well.

What About Schemas?

Yes, for added data validation you could add schema definitions. But for something as transient as argument passing, it seems like too much extra overhead. But for some very critical processing it may be appropriate.

Speaking of Overhead

The extra overhead of converting native data types into XML and back, as well as the overhead of instantiating the MSXML object seems to be the only noticable downside to this approach. For me, the added capability of passing variable amounts of data plus any reduction in long term maintenance makes it a winner. But probably no worse than the creation of a recordset to return structured data. Certainly many if not most procedure calls will not need complex variable input arguments, and the XML output processing is only invoked if there is an error, presumably only a small per centage of the time.

In Conclusion

MS SQL Server T-SQL does not provide a robust interface for passing in varying numbers of arguments or any kind of record structure. On output we may also desire a logical multiple record structure, separate from a data recordset. And the more arguments a procedure takes, the bigger the maintenance chore.

An XML input argument provides a simple way to send from application code a variable number of arguments, including arbitrarily complex logical record structures. An XML output argument likewise is a good container for multiple return values, again including optional record structures. Since the stored procedure just ignores any input elements it does not need, and can often provide defaults for elements not passed, you have a greater chance of being able to make changes to the stored procedure and/or underlying database structure and have relatively little or no impact on existing programs that use it.

References:

These all reference SQL 2000, and mostly use the XML input to just do a direct bulk Insert without preliminary data validations.

http://www.devx.com/dotnet/Article/16155/0/page/2
http://www.sql-server-performance.com/jb_openxml.asp
http://msdn.microsoft.com/msdnmag/issues/05/06/DataPoints/
http://www.devx.com/dotnet/Article/16032/1954?pf=true

A Microsoft article on XML functionality overview in SQL Server, emphasis on SQL 2005.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql2k5xmloptions.asp

Introduction to XQuery in SQL Server 2005

http://msdn.microsoft.com/SQL/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5_xqueryintro.asp

5
The biggest tool belt in the business!