

Understanding MS-SQL Xml Capabilities By Example

Prepared and Presented by Eduardo Sobrino, Open Knowledge
Last Update October 29, 2006

1.0 Abstract

MS-SQL 2005 does further support Xml trough various components, interfaces and features through the Native Client, T-SQL and the .Net CLR. Xml is a first class citizen in this MS-SQL 2005 release. This presentation will go through each of the interfaces available to access the Xml features and tools. Various demonstrations will be provided to show how to manage definition schemas, create valid Xml document instances, and query and manipulate those through XPath, XQuery and XSLT.

2.0 Learning Strategy

A lot has been written about Xml, for starters you will find tons of documentation just for the standard specifications (see XML in "References/Web Site" section) in top of that there is a lot more describing how it is supported in MS-SQL 2005 (S2k5). Since there is enough to learn to get "proficient" in Xml and in its implementation in S2k5 I propose to use the following learning work agenda:

XML 101 Basic Training

- Understand Xml Schemas Definition (XSD)
- Learn XPath 2.0 Expression Language
- Learn XQuery 1.0 Query Language
- Learn XSLT 2.0 Style Sheet Transformation

XML 102 MS-SQL 2005 Basic Training

- MS-SQL 2005 Xml Features Overview
- Accessing XML: Client Side
- Working with Images

If you already understand some of the above just jump into those that you need to review, still I hope that you find interesting material in all. This document doesn't pretend to be an "In-Depth" knowledge of each topic but to get you started.

3.0 Preparing Xml Schemas

The purpose of this section is to review a few details that will help you to read and understand a definition schema and get the basics so you write your own. If you open a new "Xml Schema" (xsd) file in VS 2005 you will get something as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="http://tempuri.org/XMLSchema.xsd"
           elementFormDefault="qualified"
           xmlns="http://tempuri.org/XMLSchema.xsd"
           xmlns:mstns="http://tempuri.org/XMLSchema.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
</xs:schema>
```

Sample 3.0.1. Schema root element as auto defined in VS 2005 for a new xsd document.

Since it is not clear what you need to do to complete a schema definition let me suggest the following:

1. Declare the root “<xs: schema>” element with its related and needed URI’s, related aliases and namespaces (see Sample 3.0.1). Most of the tools that help you writing “xsd” files output similar “<?xml >” and root schema elements as show in Sample 3.0.1. (see the root schema element discussion in 3.0.1 that follows)
2. Declare required “<xs: import>” elements that will be used to reference types, attributes and elements on other xsd’s.
3. Define all needed “Types” by using the “<xs: complexType>” or “<simpleType>” elements.
4. Define needed attributes and elements with the “<xs: attributeGroup>”, “<xs: attribute>” and “<xs: element>” elements.

3.1 Defining root “<xs: schema>”.

The first line defines the version and encoding and is followed by the “<xs: schema>” root element with the following attributes:

Attribute	Description
targetNamespace	Specify the namespace for the schema constructs such as element declarations, attribute declaration and type definitions contained in this file. For additional discussion and alternatives while working with default namespaces read http://www.xfront.com/DefaultNamespace.pdf .
elementFormDefault	Indicates whether or not locally declared elements must be qualified by the target namespace in an instance document. If the value of this attribute is 'unqualified', then locally declared elements should not be qualified by the target namespace. If the value of this attribute is 'qualified', then locally declared elements must be qualified by the target namespace. For a discussion on when to hide or expose namespaces read http://www.xfront.com/HideVersusExpose.html .
xmlns=	Specify the default namespace. (see the “targetNamespace” attribute)
xmlns:mstns=	Define the “mstns” alias for the “ http://tempuri.org/XMLSchema.xsd ” namespace.
xmlns:xs=	Define the “xs” alias for the “ http://www.w3.org/2001/XMLSchema ” namespace.

3.1.1 TODO: declare a URI that identify your definitions.

The “<http://tempuri.org/XMLSchema.xsd>” is not a URL (a resource Locator) but a (temporary) URI (a resource Identifier). As a URI it does not need to point to any Web resource since its objective is to identify your self. Therefore replace it to a URI of your own.

In Sample 3.0.1 schema the “targetNamespace”, namespace (“xmlns:mstns=“...””) declarations and related alias (“mstns”) are used to identify the document. Therefore the use of the “mstns” alias will refer to types and elements defined in this document.

3.2 “<xsd:import >” needed definitions.

You can organize your definitions in different files. To help you understand let's assume that you need that valid values are always used to define a customer (or any other entity) kind. In Sample 3.2.1 schema the “KindType” is used to declare the accepted valid values. This means that any other value will invalidate this restriction and therefore will render submitted xml file as invalid.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://openk/EntityCodesSchema">

    <xsd:simpleType name="KindType">
        <xsd:restriction base="xsd:token">
            <xsd:enumeration value="Corporate"/>
            <xsd:enumeration value="Person"/>
            <xsd:enumeration value="Institution"/>
        </xsd:restriction>
    </xsd:simpleType>

</xsd:schema>
```

Sample 3.2.1.Schema defining valid (entity) kinds.

To reference the “KindType” from another file you will use the “import” element as shown in Sample 3.2.2.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema"
            xmlns:opkc="http://openk/EntityCodesSchema"
            xmlns:opk="http://openk/EntitySchema"
            targetNamespace="http://openk/EntitySchema">

    <xsd:import namespace="http://openk/EntityCodesSchema"
              schemaLocation="SampleCodes1.xsd" />

</xsd:schema>
```

Sample 3.2.2.Referencing the “KindType” using the “import” element.

3.3 Define your types.

Reviewing the Sample 3.2.1 you see the use of the “simpleType” element, in Sample 3.3 you see the use of the “complexType”. You will use both tags to define all your types.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema"
            xmlns:opkc="http://openk/EntityCodesSchema"
            xmlns:opk="http://openk/EntitySchema"
            targetNamespace="http://openk/EntitySchema">

    <xsd:import namespace="http://openk/EntityCodesSchema"
              schemaLocation="SampleCodes1.xsd" />

</xsd:schema>
```

```

<xsd:complexType name="EntityType">
  <xsd:sequence>
    <xsd:element name="EntityNo" type="xsd:integer" />
    <xsd:element name="AlternateId" type="xsd:string" />
    <xsd:element name="Name" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="Kind" type="opkc:KindType" />
</xsd:complexType>

<xsd:element name="Entity" type="opk:EntityType" />

</xsd:schema>

```

Sample 3.3.1. Define the “EntityType” type.

3.4 Define your supported attributes and elements.

Define your needed element definitions as shown in Sample 3.3.1 with the “Entity” declaration based on the “opk:EntityType” type.

4.0 Understanding XPath v2.0 Expression Language

The good news about XPath is that it is easily learned and simple to use. Let’s say you have an Xml document as shown in Sample 4.0.1. XPath samples that will follow will use this Xml document to demonstrate the possible expressions.

```

<?xml version="1.0"?>
<entities xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <entity kindno="2">
    <entityno>0</entityno>
    <alternateid>0001</alternateid>
    <name>Rivera Rivera, Maria</name>
    <phone>787 764-1942</phone>
  </entity>
  <entity kindno="2">
    <entityno>1</entityno>
    <alternateid>0002</alternateid>
    <name>Rivera Martinez, Carlos</name>
  </entity>
  <entity kindno="2">
    <entityno>2</entityno>
    <alternateid>0003</alternateid>
    <name>Rivera Sanchez, Coral</name>
  </entity>
  <entity kindno="2">
    <entityno>3</entityno>
    <alternateid>0004</alternateid>
    <name>Rivera Roman, Raul</name>
  </entity>
  <entity kindno="1">
    <entityno>4</entityno>
    <alternateid>C001</alternateid>
    <name>Juancho's Inc.</name>
  </entity>
  <entity kindno="1">
    <entityno>5</entityno>

```

```
<alternateid>C002</alternateid>
<name>Carros y Algo Mas</name>
</entity>
</entities>
```

Sample 4.0.1. Xml document that defines a list of “entities”.

4.1 Basic XPath Expressions.

A common XPath expression to access Xml nodes is similar to accessing a node in a file system by using a file path. For example the following expression:

```
/entities/entity/name
```

The expected output is:

```
name: Rivera Rivera, Maria
name: Rivera Martinez, Carlos
name: Rivera Sanchez, Coral
name: Rivera Roman, Raul
name: Juancho's Inc.
name: Carros y Algo Mas
(6 row(s) affected)
```

You could also submit the expression as “//entity/name” since the “//” references the “root” (in this case the “entities”) element.

To get the above output I submit the following request to MS-SQL 2005:

```
declare @xmldoc as nvarchar(max)
set @xmldoc = '<...>'

declare @h int
exec sp_xml_preparedocument @h output, @xmldoc

select 'name: '+cast([text] as varchar)
from openxml(@h, '//entity/name') x where [text] is not null
```

Note that “<...>” represents the Xml as show in Sample 4.0.1 and was not displayed here to conserve space.

You could get the values of an attribute as follows:

```
//entity/@kindno
```

In the previous case you could use a wild card as “//entity/@*” producing the same result since there is only one attribute. When using the wild card as show and if there were more than one attribute in the “entity” element then all of those will have been outputted.

By submitting an expression like “//entity” you will get a list of “entity” elements with all their associated attributes and elements.

4.2 XPath Filtering Queries.

There are various ways to filter the results as follows:

Expression	Description
//entity [phone]	Returns all nodes that contain the “phone” element.
/entities/entity [@kindno="2"]	Returns all nodes where the “@kindno” attribute

	value is 2.
<code>//entity [@kindno >= "1"]</code>	Returns all nodes whose "@kindno" attribute value are greater than or equal too 1.

4.3 XPath Functions.

XPath provide a useful collection of functions including:

Expression	Description
<code>//entity [substring(name,1,3) = "Riv"]</code>	Returns all nodes that begin the "//entity/name" begin with "Riv".
<code>//entity [string-length(name) = 14]</code>	This will return just one node where the length of the name is equal to 14.

Although only few samples were given there are quite few more functions available.

5.0 Understanding XQuery v1.0 Language

XQuery is used for querying data from Xml documents. Within XQuery you can write XPath expressions and it also used to transform Xml documents therefore competing with XSLT.

To startup let's submit the following XQuery through MS-SQL 2005 that without too much thought you will understand:

```

declare @xdoc xml
set @xdoc = ''

select @xdoc.query('(2+2)')           -- add too numbers
select @xdoc.query('"que pasa"')    -- output a literal
select @xdoc.query('concat(xs:string(2+2),".")')
select @xdoc.query('
    for $a in (1,2,3)
    return $a') -- output numbers
select @xdoc.query('
    for $a in ("manzana","calabaza")
    return $a') -- output words

set @xdoc='<root><data>test</data></root>'
select @xdoc.query('
    for $a in (xs:string( "hola"), xs:double( "10.123" ),
    data(/root/data ))
    return $a') -- output some more stuff

declare @price money
set @price=10.123
select @xdoc.query('<value>{sql:variable("@price")}</value>')

set @xdoc = '<root><a>1</a><a attr="1">2</a></root>'
select @xdoc.query('/root/a[./@attr]')

-- bad query will result in an empty sequence
set @xdoc='<a>i-am-not-a-number</a>'
select @xdoc.query('xs:double(/a[1])')

-- OUTPUT (the result of all the above will be)
4
que pasa
4.
1 2 3

```

```
manzana calabaza
hola 10.123 test
<value>10.123</value>
<a attr="1">2</a>
```

5.1 The FLWOR Expressions.

The bread and butter of XQuery are the FLWOR expressions that are sort of the SQL SELECT statement of this language named after five clauses for, let, where, order by, return. By using a sample file "SampleEntity1.xml" Xml file whose content can be seen in 4.0.1 lets write a FLWOR expression:

```
let $doc := doc("./SampleEntity1.xml")
for $v in $doc//entity/name
where ends-with($v, 'Coral')
return $v
(: outputs "<name>Rivera Sanchez, Coral</name>" :)
```

Another example:

```
let $doc := doc("./SampleEntity1.xml")
for $v in $doc//entity/name
where substring($v, 1,1) = "R"
order by $v
return $v
(: outputs:
<name>Rivera Martinez, Carlos</name>
<name>Rivera Rivera, Maria</name>
<name>Rivera Roman, Raul</name>
<name>Rivera Sanchez, Coral</name>
:)
```

At the same time we filter data we can also transform the file into something else as follows:

```
<customers>
{
  let $doc := doc("./SampleEntity1.xml")
  for $v in $doc//entity
  where $v [@kindno="2"]
  order by $v/name
  return
    <customer id="{ $v/alternateid }">
      { $v/name }
    </customer>
}
</customers>
(: outputs: (all kind-no = 2 are customers)
<customers>
<customer id="0002"><name>Rivera Martinez, Carlos</name></customer>
<customer id="0001"><name>Rivera Rivera, Maria</name></customer>
<customer id="0004"><name>Rivera Roman, Raul</name></customer>
<customer id="0003"><name>Rivera Sanchez, Coral</name></customer>
</customers>
:)
```

```

<?xml version="1.0"?>
<sales>
  <order orderid="0001">
    <orderdate>1/1/2006</orderdate>
    <customerid>0001</customerid>
    <items>
      <item>
        <sku>TOY-0001</sku>
        <description>Big Doll</description>
        <unit-cost>10.01</unit-cost>
        <quantity>1</quantity>
        <extended-amount>10.01</extended-amount>
      </item>
      <item>
        <sku>FOOD-0002</sku>
        <description>Rice</description>
        <unit-cost>3.52</unit-cost>
        <quantity>2</quantity>
        <extended-amount>7.04</extended-amount>
      </item>
    </items>
  </order>
  <order orderid="0002">
    <orderdate>1/1/2006</orderdate>
    <customerid>0002</customerid>
    <items>
      <item>
        <sku>TOY-0001</sku>
        <description>Big Doll</description>
        <unit-cost>10.01</unit-cost>
        <quantity>2</quantity>
        <extended-amount>20.02</extended-amount>
      </item>
    </items>
  </order>
</sales>

```

Document 5.1.0. Sales Data File.

Assuming we do have sales information in an Xml file as shown in Document 5.1.0 and submitting the FLWOR expression that follows we will get the sales details by customer:

```

<sales>
{
  let $doc := doc("./SampleEntity1.xml"),
      $sal := doc("./SampleSales1.xml")
  for $s in $sal//order,
      $v in $doc//entity
  where $s [customerid=$v/alternateid]
  order by $sal/customerid
  return
    <customer id="{ $s/customerid }">
      { $v/name }
      <items>
        { $s/items/item }
      </items>
    </customer>

```

```
}  
</sales>
```

```
(: outputs:  
<sales>  
  <customer id="0001">  
    <name>Rivera Rivera, Maria</name>  
    <items>  
      <item>  
        <sku>TOY-0001</sku>  
        <description>Big Doll</description>  
        <unit-cost>10.01</unit-cost>  
        <quantity>1</quantity>  
        <extended-amount>10.01</extended-amount>  
      </item>  
      <item>  
        <sku>FOOD-0002</sku>  
        <description>Rice</description>  
        <unit-cost>3.52</unit-cost>  
        <quantity>2</quantity>  
        <extended-amount>7.04</extended-amount>  
      </item>  
    </items>  
  </customer>  
  <customer id="0002">  
    <name>Rivera Martinez, Carlos</name>  
    <items>  
      <item>  
        <sku>TOY-0001</sku>  
        <description>Big Doll</description>  
        <unit-cost>10.01</unit-cost>  
        <quantity>2</quantity>  
        <extended-amount>20.02</extended-amount>  
      </item>  
    </items>  
  </customer>  
</sales>  
:)
```

Another FLWOR expression (that follows) will also get the sales details by customer:

```
for $v in doc("./SampleEntity1.xml")//entity  
return  
  <customer id="{ $v/alternateid }">  
    { $v/name }  
    { for $a in doc("./SampleSales1.xml")  
      //order [customerid=$v/alternateid]  
      return  
        <purchases>  
          { $a/items/item }  
        </purchases>  
    }  
  </customer>
```

```
(: outputs:  
<customer id="0001"><name>Rivera Rivera, Maria</name>
```

```

<purchases>
  <item>
    <sku>TOY-0001</sku>
    <description>Big Doll</description>
    <unit-cost>10.01</unit-cost>
    <quantity>1</quantity>
    <extended-amount>10.01</extended-amount>
  </item>
  <item>
    <sku>FOOD-0002</sku>
    <description>Rice</description>
    <unit-cost>3.52</unit-cost>
    <quantity>2</quantity>
    <extended-amount>7.04</extended-amount>
  </item>
</purchases>
</customer>
<customer id="0002"><name>Rivera Martinez, Carlos</name>
  <purchases>
    <item>
      <sku>TOY-0001</sku>
      <description>Big Doll</description>
      <unit-cost>10.01</unit-cost>
      <quantity>2</quantity>
      <extended-amount>20.02</extended-amount>
    </item>
  </purchases>
</customer>
<customer id="0003"><name>Rivera Sanchez, Coral</name></customer>
<customer id="0004"><name>Rivera Roman, Raul</name></customer>
<customer id="C001"><name>Juancho's Inc.</name></customer>
<customer id="C002"><name>Carros y Algo Mas</name></customer>

:~)

```

You could calculate the total sales as follows:

```

let $subtotals :=
  for $s in doc("SampleSales1.xml")//order/items/item
  return $s/quantity * $s/unit-cost
return
  <total-sales>
  { sum( $subtotals ) }
  </total-sales>

(:
<total-sales>37.07</total-sales>
:~)

```

5.2 Working with User-Defined Functions.

You can create your own UDF ("User Define Functions") and use them as follows:

```
declare function local:extended(
  $q as xs:decimal, $u as xs:decimal) as xs:decimal
{
  let $r := $q * $u
  return ($r)
};
for $a in doc("./SampleSales1.xml")//order/items/item
return
  <Price sku="{ $a/sku}" quant="{ $a/quantity}" ucost="{ $a/unit-cost}">
    {local:extended($a/quantity,$a/unit-cost)}
  </Price>

(:
<Price quant="1" sku="TOY-0001" ucost="10.01">10.01</Price>
<Price quant="2" sku="FOOD-0002" ucost="3.52">7.04</Price>
<Price quant="2" sku="TOY-0001" ucost="10.01">20.02</Price>
:)
```

Let's list the sale items:

```
declare function local:SoldItems($order as element())
as element()*
{
  for $i in $order/items/item
  return
    <sale sku="{ $i/sku}" description="{ $i/description}" />
};
<sales>
{
for $s in doc("SampleSales1.xml")//order
return local:SoldItems($s)
}
</sales>

(:
<sales>
<sale description="Big Doll" sku="TOY-0001"/>
<sale description="Rice" sku="FOOD-0002"/>
<sale description="Big Doll" sku="TOY-0001"/>
</sales>
:)
```

We can get the orders total sale as follows:

```
declare function local:OrderTotals(
  $o as xs:string, $i as element()) as element()*
{
  let $subtotals :=
    for $s in $i/item
    return $s/quantity * $s/unit-cost
  return
    <total-sale orderid="{ $o}">
      { sum( $subtotals ) }
    </total-sale>
```

```

};

<sales>
{
  for $o in doc("SampleSales1.xml")//order
  return
    local:OrderTotals($o/@orderid,$o/items)
}
</sales>

(:
<sales>
  <total-sale orderid="0001">17.05</total-sale>
  <total-sale orderid="0002">20.02</total-sale>
</sales>
:.)

```

Calculate total sales as:

```

declare function local:TotalSales($i as element()) as xs:double
{
  let $subtotals :=
    for $s in $i/order/items/item
    return $s/quantity * $s/unit-cost
  return
    sum( $subtotals )
};

let $o := doc("SampleSales1.xml")/sales
return
<total-sales>
{
  xs:string(local:TotalSales($o))
}
</total-sales>

(:
<total-sales>37.07</total-sales>
:.)

```

5.3 Using the “collection” function for Database Access.

Various XQuery processors use the “collection” function to access the database. Although I have seen differences in various products the general idea is that you provide the database the schema and particular table resources as follows:

```
collection("KIFv1.Applications.Sessions")/Sessions [StatusNo=0]
```

As you guess, “KIFv1” is the database, “Applications” the schema and “Sessions” the table. The previous query return all the sessions who’s “StatusNo” is 0.

Understand that the XQuery implementation in MS-SQL 2005 does not implement all the language functionality. This implementation does not implements the “fn:doc()” and “fn:collection()” functions.

6.0 Understanding XSLT

XSLT (or “eXtensible Stylesheet Language: Transformations”) is a language designed for transforming one Xml document into another. As you will soon see with XSLT you can transform an Xml document into another Xml document, HTML and any other “text-based” formats.

Working with XSLT is “relatively simple” you can do a lot just concentrating on how you will like to manage (transform) each of the elements of your document. For example, let’s use the previous Document 5.1.0 sales data and write an XSLT file to create an HTML file that display sales information as shown in Figure 6.0.1.

Order#: 0001 Customer#: 0001 Date: 1/1/2006				
Sku	Description	Unit-Cost	Quantity	Extended
FOOD-0002	Rice	3.52	2	7.04
TOY-0001	Big Doll	10.01	1	10.01
				17.05
Order#: 0002 Customer#: 0002 Date: 1/1/2006				
Sku	Description	Unit-Cost	Quantity	Extended
TOY-0001	Big Doll	10.01	2	20.02
				20.02
2 Orders for a Total of 37.07 (USD) Sales.				

Figure 6.0.1. Sales Data HTML document preview.

Most of the work done in the 6.0.1 transformation was done by “matching” each Xml document elements as follows:

```
<?xml version = "1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:output method="html"/>
<xsl:template match="/"> ...
<xsl:template match="order"> ...
<xsl:template match="items"> ...
<xsl:template match="item"> ...
</xsl:stylesheet>
```

Note that in the previous listing the “...” represent the “template” details that I will show a bit later but is hidden here to clearly reveal each template based on matching an element of the Sales document.

Now let's see each matching element details starting with the document "root" (/):

```
<xsl:template match="/">
  <html><head>
    <title>Stylesheet Example</title></head>
    <body>

      <table align="center" cellpadding="2" cellspacing="0" border="1px"
        style="border-color:black; width:500px">
        <tr style="background-color:#eeeeee"><td>
          <xsl:apply-templates select="sales/order" />
        </td></tr>
        <tr><td align="right">
          <span>
            <xsl:value-of select="count(sales/order)"/> Orders for a Total of
            <xsl:value-of select="sum(sales/order/items/item/extended-amount)"/>
            (USD) Sales.
          </span>
        </td></tr>
      </table>

    </body>
  </html>
</xsl:template>
```

The previous template do output the HTML document and transfer the output of the "sales/order" details as defined in another template as follows:

```
<xsl:template match="order">
  <span>Order#: <xsl:value-of select="attribute::orderid"/>
  Customer#: <xsl:value-of select="customerid"/>
  Date: <xsl:value-of select="orderdate"/>
</span>
<xsl:apply-templates select="items">
  <xsl:sort select="orderdate"/>
</xsl:apply-templates>
<hr />
</xsl:template>
```

The previous template do output some of the "order" details and transfer the output of the order "items" details to another template as follows:

```
<xsl:template match="items">
  <table align="center" cellpadding="2" cellspacing="0" border="1px"
    style="border-color:black" width="100%">
    <tr style="background-color:lightskyblue">
      <th style="width:90px">Sku</th>
      <th style="width:160px">Description</th>
      <th style="width:70px">Unit-Cost</th>
      <th style="width:60px">Quantity</th>
      <th style="width:60px">Extended</th></tr>
    <xsl:apply-templates select="item">
      <xsl:sort select="sku"/>
    </xsl:apply-templates>
    <tr>
      <td colspan="4"></td>
      <td align="right" style="background-color:white">
        <xsl:value-of select="sum(item/extended-amount)"/></td>
    </tr>
```

```
    </table>
</xsl:template>
```

Finally another template is used to output the item details as follows:

```
<xsl:template match="item">
  <tr style="background-color:white">
    <td><xsl:value-of select="sku"/></td>
    <td><xsl:value-of select="description"/></td>
    <td align="right"><xsl:value-of select="unit-cost"/></td>
    <td align="right"><xsl:value-of select="quantity"/></td>
    <td align="right">
      <xsl:value-of select="extended-amount"/>
    </td>
  </tr>
</xsl:template>
```

As can easily be realized you output the values of each elements by using the “xsl:value-of” element. The rest of the template details is obvious and I will let you figure it out.

The previous example uses one method to go through each Xml document hierarchy, still the language do offer other methods (such as the “for-each”) to do more or less the same and quite a few other nice XSLT elements, expressions, patterns and functions to help you get your work done.

7.0 MS-SQL 2005 Xml Features Overview

MS-SQL 2005 further extends the support for XML way beyond the previous (2000) version. Among others, the enhancements include:

- New “XML Data Type”
- Typed XML Offering: Cataloging of XML schema collections
- Indexing in an XML Column
- XML Type Methods
- SELECT . . . FOR XML Enhancements
- OpenXML Enhancements

7.1 XML Data Type

New in MS-SQL the XML data type can be used a table column or variable as follows:

```
declare @xdoc xml
set @xdoc = '<xml/>'
print cast(@xdoc as varchar)

-- Outputs: "<xml/>"

create table SampleTable (
  RecordNo int,
  XDocument xml
)
go

insert into SampleTable values (0, '<xml/>')
select * from SampleTable

-- Output:
```

```

-- RecordNo      XDocument
-- -----
-- 0              <xml />
--
-- (1 row(s) affected)

```

7.2 Typed XML: Cataloging XML Schemas

“Garbage in garbage out” have you heard the previous? Of course you have, and MS-SQL 2005 provides you the tool to control what is stored in an “xml” column or variable by providing the name of a “Cataloged Schema”.

7.2.1 Cataloging Schemas

While cataloging you could cram all related schemas all at once as follows:

```

CREATE XML SCHEMA COLLECTION SampleEntity1Collection AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://openk/EntityCodesSchema">
  <xsd:simpleType name="KindType">
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="Corporate"/>
      <xsd:enumeration value="Person"/>
      <xsd:enumeration value="Institution"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sql="urn:schemas-microsoft-com:mapping-schema"
  xmlns:opk="http://openk/EntityCodesSchema"
  targetNamespace="http://openk/EntitySchema">
  <xsd:import namespace="http://openk/EntityCodesSchema" />
  <xsd:element name="Entity">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="EntityNo" type="xsd:integer" />
        <xsd:element name="AlternateId" type="xsd:string" />
        <xsd:element name="Name" type="xsd:string" />
      </xsd:sequence>
      <xsd:attribute name="Kind" type="opk:KindType" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

```

Or submit them independently as follows:

```

CREATE XML SCHEMA COLLECTION SampleEntity1Collection AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetSchema="http://openk/EntityCodesSchema">
  <xsd:simpleType name="KindType">
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="Corporate"/>
      <xsd:enumeration value="Person"/>
      <xsd:enumeration value="Institution"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>'
GO

```

```
ALTER XML SCHEMA COLLECTION SampleEntity1Collection ADD
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sql="urn:schemas-microsoft-com:mapping-schema"
  xmlns:opk="http://openk/EntityCodesSchema"
  targetNamespace="http://openk/EntitySchema">
  <xsd:import namespace="http://openk/EntityCodesSchema" />
  <xsd:element name="Entity">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="EntityNo" type="xsd:integer" />
        <xsd:element name="AlternateId" type="xsd:string" />
        <xsd:element name="Name" type="xsd:string" />
      </xsd:sequence>
      <xsd:attribute name="Kind" type="opk:KindType" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

You decide how to submit those for cataloging.

7.2.2 Reviewing Cataloged Schemas

To review your schemas catalog you will use the following TSQL commands:

```
SELECT * FROM sys.xml_schema_namespaces
SELECT * FROM sys.xml_schema_collections
GO
```

That will output the following:

```
xml_collection_id name                                xml_namespace_id
-----
1                http://www.w3.org/2001/XMLSchema                    1
1                http://schemas.microsoft.com/sqlserver/2004/sqltypes 2
1                http://www.w3.org/XML/1998/namespace      3
65536           http://openk/EntitySchema                          2
65536           http://openk/EntityCodesSchema                     1

(5 row(s) affected)

xml_collection_id schema_id principal_id name                create_date modify_date
-----
1                4                NULL                sys                2005-10-14 2005-10-14
65536           1                NULL                SampleEntity1Collection 2006-10-22 2006-10-22

(2 row(s) affected)
```

(Note: output was altered by hand to fit it on this page)

In the previous you see your schemas (in the submitted schemas the “EntitySchema” and “EntityCodesSchema”). To view the cataloged schemas execute the following TSQL command:

```
SELECT xml_schema_namespace(N'dbo', N'SampleEntity1Collection')
GO
```

7.2.3 Referencing Cataloged Schemas

To reference a cataloged schema just specify it on the declaration of your “xml” types as follows:

```
exec Helper.DropTable 'SampleTable2'
GO
```

```

create table SampleTable2 (
    RecordNo int primary key,
    XDocument xml(SampleEntity1Collection)
)
GO

declare @doc as xml(SampleEntity1Collection)
set @doc = '<?xml version="1.0"?>
<p1:Entity Kind="Corporate" xmlns:p1="http://openk/EntitySchema">
    <EntityNo>1</EntityNo>
    <AlternateId>0001</AlternateId>
    <Name>Juancho''s Inc.</Name>
</p1:Entity>'

print cast(@doc as varchar(max))
insert into SampleTable2 values (0, @doc)
select * from SampleTable2
GO

```

If you submit the following xml that don't follow the schema:

```

declare @doc as xml(SampleEntity1Collection)
set @doc = '<?xml version="1.0"?>
<p1:Entity Kind="Corporate" xmlns:p1="http://openk/EntitySchema">
    <EntityNo>1</EntityNo>
    <AlternateId>0001</AlternateId>
    <Name>Juancho''s Inc.</Name>
    <Phone>787 764-1942</Phone>
</p1:Entity>'

```

Then you will get:

```

Msg 6923, Level 16, State 1, Line 2
XML Validation: Unexpected element(s): Phone. Location: /*:Entity[1]/*:Phone[1]

```

7.3 Indexing in an XML Column

To extract information from your stored Xml instances you will submit a (XQuery) query such as the following:

```

with xmlnamespaces ('http://openk/EntitySchema' as "p1")
select XDocument.query('/p1:Entity/Name') NameElement
    from SampleTable2
    where XDocument.exist('/p1:Entity/EntityNo[.=1]') = 1
go

with xmlnamespaces ('http://openk/EntitySchema' as "p1")
select XDocument.value('/p1:Entity/Name[1]', 'varchar(60)') Name
    from SampleTable2
    where XDocument.exist('/p1:Entity/EntityNo[.=1]') = 1
go

```

As you can imagine the number of instances can be quite large, and the amount of data submitted within each can be as large as 2GB. Without an index, these binary large objects are shredded at run time to evaluate a query that can be time-consuming. To

help in the performance of queries over your XML instances you can create a “Primary” or “Secondary” XML Index.

7.3.1 Primary XML Indexes

Using the above TSQL samples we can create a “Primary XML Indexes” as follows:

```
create primary xml index pi_samp01 on SampleTable2(XDocument)
go
```

What happens is that a “Node Table” is created containing the following columns:

- Tag name such as an element or attribute name.
- Node value.
- Node type such as an element node, attribute node, or text node.
- Document order information, represented by an internal node identifier.
- Path from each node to the root of the XML tree. This column is searched for path expressions in the query.
- Primary key of the base table. The primary key of the base table is duplicated in the primary XML index for back join with the base table, and the maximum number of columns in primary key of the based table is limited to 15.

7.3.2 Secondary XML Indexes

To enhance search performance, you can create secondary XML indexes. You must first create a primary XML index before you can create secondary indexes. The available types are:

- *PATH secondary XML index.*
If your queries generally specify path expressions on xml type columns, a PATH secondary index may be able to speed up the search.
- *VALUE secondary XML index.*
If your workload involves querying for values from XML instances without knowing the element or attribute names that contain the values, VALUE index may be useful.
- *PROPERTY secondary XML index.*
Queries that retrieve one or more values from individual XML instances may benefit from a PROPERTY index.

Using the above TSQL samples we can create a “Secondary XML Indexes” as follows:

```
-- create secondary indexes (PATH, VALUE, PROPERTY)
create xml index si_samp01 ON SampleTable2(XDocument)
using xml index pi_samp01 for PATH
go
create xml index si_samp02 ON SampleTable2(XDocument)
using xml index pi_samp01 for VALUE
go
create xml index si_samp03 ON SampleTable2(XDocument)
using xml index pi_samp01 for PROPERTY
go
```

7.4 XML Type Method

The “xml” type does have associated methods including:

Method	Description
query()	Specifies an XQuery against an instance of the xml data type. The result is of xml type. The method returns an instance of untyped XML.
value()	You typically use this method to extract a value from an XML instance stored in an xml type column, parameter, or variable. In this way, you can specify SELECT queries that combine or compare XML data with data in non-XML columns.
exist()	Returns a bit that represents one of the following conditions: <ul style="list-style-type: none">• 1, representing True, if the XQuery expression in a query returns a nonempty result. That is, it returns at least one XML node.• 0, representing False, if it returns an empty result.• NULL if the xml data type instance against which the query was executed contains NULL.
modify()	This method takes an XML DML statement to insert, update, or delete nodes from xml data. The modify() method of the xml data type can only be used in the SET clause of an UPDATE statement.
nodes()	The nodes() method is useful when you want to shred an XML data type instance into relational data. It allows you to identify nodes that will be mapped into a new row.

7.4.1 The “query()” Method

Although we have already provided samples of the query method here are some other:

```
-- DECLARE XmlDocument1:
declare @doc as xml(SampleEntity1Collection)
set @doc = '<?xml version="1.0"?>
<p1:Entity Kind="Corporate" xmlns:p1="http://openk/EntitySchema">
  <EntityNo>1</EntityNo>
  <AlternateId>0001</AlternateId>
  <Name>Juancho''s Inc.</Name>
</p1:Entity>'

select @doc.query(
  'declare namespace p1="http://openk/EntitySchema"; /p1:Entity/Name')
go
-- outputs: <Name>Juancho's Inc.</Name>

-- DECLARE XmlDocument2:
declare @doc as xml
set @doc = '<?xml version="1.0"?>
<Entities>
<Entity Kind="Corporate">
  <EntityNo>1</EntityNo>
  <AlternateId>0001</AlternateId>
  <Name>Juancho''s Inc.</Name>
</Entity>
<Entity Kind="Personal">
```

```

    <EntityNo>2</EntityNo>
    <AlternateId>0002</AlternateId>
    <Name>Hernandez Rivera, Maria</Name>
</Entity>
</Entities>'

select @doc.query('/Entities/Entity/Name')
go

-- outputs:
-- <Name>Juancho's Inc.</Name><Name>Hernandez Rivera, Maria</Name>

```

Listing 6.4.1. “query()” method samples.

7.4.2 The “value()” Method

Using the previous “XmlDocument1” (see Listing 6.4.1) we will use this method as follows:

```

select @doc.value(
    'declare namespace pl="http://openk/EntitySchema";
    (/pl:Entity/Name)[1]',
    'varchar(80)')
go

```

Using the same listing and the “XmlDocument2” sample:

```

select @doc.value('/Entities/Entity/Name')[2], 'varchar(80)')

```

7.4.3 The “exist()” Method

In addition to previous samples provided you can use the “exists” method as follows:

```

declare @x xml
set @x = '<root PresentationDate="2006-11-02Z"/>'
select @x.exist(
    '/root[(@PresentationDate cast as xs:date?) eq
    xs:date("2006-11-02Z")]' )

set @x = '<PresentationDate>2006-11-02Z</PresentationDate>'
select @x.exist(
    '/PresentationDate[(text()[1] cast as xs:date ?) =
    xs:date("2006-11-02Z") ]' )

```

Notes From [5]:

Values of type *xs:datetime*, *xs:date* and *xs:time* must be specified in ISO 8601 format and include a time zone. Otherwise the data validation for these values fails. Thus, 2005-05-27T14:11:00.943Z is valid as a value of type *xs:datetime* but the following are not: 2005-05-27 14:11:00.943Z (missing date and time separator "T"), 2005-05-27T14:11:00.943 (missing time zone) and 2005-05-27 14:11:00.943 (missing time separator and time zone). Similarly, 2005-05-27Z is a valid *xs:date* value but 2005-05-27 is not since no time zone is specified.

Untyped XML data may contain date, time and datetime values that an application may wish to convert to the SQL types *dateTime* or *smallDateTime*. These date, time and datetime values may not conform to ISO 8601 format or contain a time zone. Similarly, typed XML may contain such values as types other than *xs:date*, *xs:time* and *xs:dateTime* (e.g., *xs:string*). In both cases, the values should be converted first to [n]varchar and then to SQL *datetime* or *smalldatetime*.

7.4.4 The “modify()” Method

With the modify method you can insert or delete nodes or replace values. Here are some samples:

```
declare @doc as xml
set @doc = '<?xml version="1.0"?>
<Entities>
<Entity Kind="Corporate">
    <EntityNo>1</EntityNo>
    <AlternateId>0001</AlternateId>
    <Name>Juancho''s Inc.</Name>
</Entity>
<Entity Kind="Corporate">
    <EntityNo>2</EntityNo>
</Entity>
<Entity Kind="Corporate">
    <EntityNo>3</EntityNo>
</Entity>
</Entities>'

-- INSERT Age element in "EntityNo = 2"
set @doc.modify('insert <Age>30</Age> into (/Entities/Entity)[2]')
select @doc

SET @doc.modify('
    replace value of (/Entities/Entity/Age[1]/text())[1]
    with "35"')
select @doc

set @doc.modify('delete /Entities/Entity/Age[1]')
select @doc
```

7.4.5 The “nodes()” Method

Using the XML “Entities” submission from the previous section we can use the “nodes” method to shred the XML document into new rows as follows:

```
select T.c.query('.') result
from @doc.nodes('/Entities/Entity') T(c)
go

result
-----
<Entity Kind="Corporate">
    <EntityNo>1</EntityNo>
    <AlternateId>0001</AlternateId><Name>Juancho's Inc.</Name></Entity>
<Entity Kind="Corporate"><EntityNo>2</EntityNo></Entity>
<Entity Kind="Corporate"><EntityNo>3</EntityNo></Entity>

(3 row(s) affected)
```

The nodes() function cannot be applied directly to the results of a user-defined function. To use the nodes() function with the result of a scalar user-defined function, you can either assign the result of the user-defined function to a variable or use a derived table to assign a column alias to the user-defined function return value and then use CROSS APPLY to select from the alias.

From MS-SQL 2005 Books On Line:

(ms-help://MS.SQLCC.v9/MS.SQLSVR.v9.en/udb9/html/0208b259-7129-4d9f-9204-8445a8119116.htm)

The APPLY operator allows you to invoke a table-valued function for each row returned by an outer table expression of a query. The table-valued function acts as the right input and the outer table expression acts as the left input. The right input is evaluated for each row from the left input and the rows produced are combined for the final output. The list of columns produced by the APPLY operator is the set of columns in the left input followed by the list of columns returned by the right input.

There are two forms of APPLY: CROSS APPLY and OUTER APPLY. CROSS APPLY returns only rows from the outer table that produce a result set from the table-valued function. OUTER APPLY returns both rows that produce a result set, and rows that do not, with NULL values in the columns produced by the table-valued function.

Using the same XML document lets use the “CROSS APPLY”:

```
select T2.entity.value('EntityNo[1]','int') id
from @doc.nodes('/Entities') T1(entities)
cross apply T1.entities.nodes('./Entity') T2(entity)
where T2.entity is not null
go

id
-----
1
2
3

(3 row(s) affected)
```

And an “OUTER APPLY”:

```
select T1.entity.value('@Kind','varchar(20)') id
from @doc.nodes('/Entities/Entity') T1(entity)
outer apply T1.entity.nodes('./AlternateId') T2(ids)
where T2.ids is not null
go

id
-----
Corporate

(1 row(s) affected)
```

The previous “OUTER APPLY” sample will give you the same result as using “CROSS APPLY” so don’t take it as a great sample.

7.5 SELECT ... FOR XML Enhancements

The “SELECT ... FOR XML” is not new to MS-SQL 2005. As you may recall you previously could use it with the “RAW”, “AUTO” and “EXPLICIT” modes. Beyond the previous capabilities new in MS-SQL 2005 there are quite a few enhancements and some will be discussed next.

7.5.1 FOR XML Results as xml Type

You can apply the FOR XML and get the result as an “xml” type as follows:

```
declare @xdoc xml
set @xdoc = (select * from entity.registry
             for xml auto, type)
select @xdoc
```

7.5.2 FOR XML RAW Enhancements

Now you can do the following:

- Specify a row element name
- Retrieve element-centric XML
- Specify the root element for the resulting XML

The following sample shows how to specify the 'row' element name as 'root', specify result to be element-centric XML and request the root to be 'result':

```
declare @xdoc xml
set @xdoc = (select * from entity.registry
             for xml raw('root'), elements, type, root('result'))
select @xdoc
```

7.5.3 Inline XSD Schema Generation

To generate an inline XSD Schema:

```
declare @xdoc xml
set @xdoc = (select * from entity.registry
             for xml auto,
             xmlschema('http://openk/entities'),
             root('data'))
select @xdoc
```

Since the result is rather bulky I will not show it.

7.5.4 FOR XML PATH Mode

From MS-SQL 2005 Books On Line:

(ms-help://MS.SQLCC.v9/MS.SQLSVR.v9.en/udb9/html/a685a9ad-3d28-4596-aa72-119202df3976.htm)

The PATH mode provides a simpler way to mix elements and attributes. PATH mode is also a simpler way to introduce additional nesting for representing complex properties. You can use FOR XML EXPLICIT mode queries to construct such XML from a rowset, but the PATH mode provides a simpler alternative to the potentially cumbersome EXPLICIT mode queries. PATH mode, together with the ability to write nested FOR XML queries and the TYPE directive to return xml type instances, allows you to write queries with less complexity.

In PATH mode, column names or column aliases are treated as XPath expressions. These expressions indicate how the values are being mapped to XML. Each XPath expression is a relative XPath that provides the item type., such as the attribute, element, and scalar value, and the name and hierarchy of the node that will be generated relative to the row element.

Using the SampleDatabase "Entity.Registry" table we can output and XML document as:

```
select KindNo "Customer/@KindNo",
       EntityNo "Customer/EntityNo",
       AlternateId "Customer/AlternateId",
       Name "Customer/Name"
from Entity.Registry
for xml path('Orders'), root('Transactions')
```

Using the "Documents.Invoices" table we can output all transactions per each customer as:

```
select KindNo "Customer/@KindNo",
       EntityNo "Customer/EntityNo",
       AlternateId "Customer/AlternateId",
       Name "Customer/Name",
```

```

cast(
  (select CreatedDate "@OrderDate",
    InvoiceNo,
    PoId,
    TotalAmount
    from Documents.Invoices
    where customerno = e.entityno
    for xml path('Invoice'), root('Invoices')) as xml)
from Entity.Registry e
for xml path('Orders'), root('Transactions')

```

7.6 OpenXML Enhancements

The TSQL OPENXML provides a rowset over in-memory XML documents that is similar to a table or a view and also allows access to XML data as though it is a relational rowset. It does this by providing a rowset view of the internal representation of an XML document. The records in the rowset can be stored in database tables.

Preparing an xml document using the “Entity.Registry” sample table we can do the following:

```

-- Step 1. Get the source "XML Document"
declare @xdoc xml, @idoc int
set @xdoc = (select *
  from entity.registry
  for xml path('Entity'), root('Entities'))

-- Step 2. Create an internal representation of the XML document.
exec sp_xml_preparedocument @idoc OUTPUT, @xdoc

-- Step 3. Exec. a SELECT statement using the OPENXML rowset provider.
select *
from openxml(@idoc, '/Entities/Entity',2) -- 2 = element centric
with (KindNo smallint, EntityNo int,
  AlternateId varchar(20), Name varchar(40))
go

-- The previous outputs:
--
-- KindNo EntityNo AlternateId Name
-- -----
-- 2 0 0001 Rivera Rivera, Maria
-- 2 1 0002 Rivera Martinez, Carlos
-- 2 2 0003 Rivera Sanchez, Coral
-- 2 3 0004 Rivera Roman, Raul
-- 1 4 C001 Juancho's Inc.
-- 1 5 C002 Carros y Algo Mas
--
-- (6 row(s) affected)

```

The 3rd argument of the “openxml” command is a flag that can be:

Flag	Description
0	Defaults to attribute-centric mapping.
1	Use the attribute-centric mapping. Can be combined with XML_ELEMENTS. In this case, attribute-centric mapping is applied first, and then element-centric mapping is applied for all columns that are not yet dealt with.

2	Use the element-centric mapping. Can be combined with XML_ATTRIBUTES. In this case, attribute-centric mapping is applied first, and then element-centric mapping is applied for all columns not yet dealt with.
8	Can be combined (logical OR) with XML_ATTRIBUTES or XML_ELEMENTS. In the context of retrieval, this flag indicates that the consumed data should not be copied to the overflow property @mp:xmltext.

8.0 Accessing XML: Client Side

Let's review how we can access the MS-SQL 2005 Xml capabilities from a remote Client.

8.1 SQL Native Client

We will begin with the "SQL Native Client". This is a new data access technology on Microsoft SQL Server 2005, and it is a stand-alone data access application programming interface (API) used for both OLE DB and ODBC. It combines the "SQL OLE DB" provider and the "SQL ODBC" driver into one native dynamic-link library (DLL).

SQL Native Client provides you the following advantages:

- Multiple active result sets (MARS);
- User-defined data types (UDT);
- Query notifications;
- Snapshot isolation;
- XML data type support.

On a recent project I use the "Kif.Win32.Data.OdbcClient" from the "Kif.Win32" framework that supports unmanaged code (available at <http://www.openk.com/Community.htm>) to developed an application that needed the "SQL Native Client" new capabilities. The project specifically needed to write to a table "xml" column.

8.2 SQLXML 4.0

Microsoft SQLXML 4.0 supports features that allow you to write applications to access XML data from an instance of SQL Server, bring the data into the Microsoft .NET Framework environment, process the data, and send the updates back to SQL Server.

Microsoft provides SQLXML 4.0 manage classes to support the access to an SQL instance. The namespace for those is "Microsoft.Data.SqlXml". To simplify the use of this class I have created the "SqlXmlDataSource" class (see Listing 8.2.1).

```

/// <summary>
/// Helper class that support Sql Xml commands.
/// </summary>
/// <author>Eduardo Sobrino</author>
/// <date>Oct/2006</date>
public class SqlXmlDataSource
{
    /// <summary>
    /// Command instance that will be used to execute submitted request.
    /// </summary>
    public Microsoft.Data.SqlXml.SqlXmlCommand Command;

    /// <summary>
    /// Default connection string.
    /// </summary>
    public static String ConnectionString =
        "Provider=SQLOLEDB;Server=(local);database=SampleDb;" +
        "Integrated Security=SSPI";

    /// <summary>
    /// Release allocated resources.

```

```

/// </summary>
public void Close()
{
    Command = null;
} // end of Close

/// <summary>
/// Create an Sql command.
/// </summary>
/// <param name="connectionString">connection string</param>
/// <returns>return an SqlXmlCommand instance</returns>
public Microsoft.Data.SqlXml.SqlXmlCommand CreateCommand(
    String connectionString)
{
    if (connectionString == null)
        connectionString = ConnectionString;
    else
        if (connectionString.Length == 0)
            connectionString = ConnectionString;

    Command =
        new Microsoft.Data.SqlXml.SqlXmlCommand(connectionString);
    return(Command);
} // end of CreateCommand

/// <summary>
/// By using current prepared command get the Xml result by using
/// ExecuteStream.
/// </summary>
/// <returns>the resulted Xml document is returned</returns>
public String GetXmlResultFromStream()
{
    if (Command == null)
        return null;

    System.IO.Stream s = Command.ExecuteStream();
    s.Position = 0;

    System.IO.StreamReader sr = new System.IO.StreamReader(s);
    String result = sr.ReadToEnd();
    sr.Close();
    sr.Dispose();
    s.Close();
    s.Dispose();
    return(result);
} // end of GetXmlResultFromStream

/// <summary>
/// Create an command from file.
/// </summary>
/// <param name="connectionString">connection string</param>
/// <param name="commandFilePath">command file path.</param>
/// <returns>an SqlXmlCommand is returned</returns>
public Microsoft.Data.SqlXml.SqlXmlCommand CreateCommandFromFile(
    String connectionString,
    String commandFilePath)
{
    // get updateGram from file
    String updateGram;
    System.IO.StreamReader sr =
        new System.IO.StreamReader(commandFilePath);
    updateGram = sr.ReadToEnd();
    sr.Close();
    sr.Dispose();

    // prepare command and execute request...
    CreateCommand(connectionString);
    if (Command == null)
        return(null);

    Command.CommandType =

```

```

        Microsoft.Data.SqlXml.SqlXmlCommandType.UpdateGram;
        Command.CommandText = updateGram;

        return(Command);
    } // end of CreateCommandFromFile

    // TODO: following sections will define remainder listing
} // end of SqlXmlDataSource

```

Listing 8.2.1. SqlXmlDataSource class.

You should know about the commands that you can submit. The options are:

Command Type	Description
Sql	Executes an SQL command (for example, "SELECT * FROM Entity.Registry FOR XML AUTO").
XPath	Executes an XPath command (for example, "Entities/Entity[@KindNo=1]").
TemplateFile	Executes a template file at the specified path.
UpdateGram	Executes an updategram.
Diffgram	Executes a DiffGram.

8.2.1 Sql Command Sample

After the "TODO:" in Listing 8.2.1 add the following code:

```

// -----
// Sql Command

/// <summary>
/// Create an Sql command.
/// </summary>
/// <param name="connectionString">connection string</param>
/// <param name="commandText">Sql query that will output an Xml document
/// . For example "select * from yourTable for xml auto".</param>
/// <returns></returns>
public Microsoft.Data.SqlXml.SqlXmlCommand CreateSqlCommand(
    String connectionString,
    String commandText)
{
    CreateCommand(connectionString);
    if (Command == null)
        return(null);
    Command.CommandType = Microsoft.Data.SqlXml.SqlXmlCommandType.Sql;
    Command.CommandText = commandText;
    return(Command);
} // end of CreateSqlCommand

/// <summary>
/// Create an Sql command using default connection string.
/// </summary>
/// <param name="commandText">Sql query that will output an Xml document
/// . For example "select * from yourTable for xml auto".</param>
/// <returns></returns>
public Microsoft.Data.SqlXml.SqlXmlCommand CreateSqlCommand(
    String commandText)
{
    return(CreateSqlCommand(ConnectionString,commandText));
} // end of CreateSqlCommand

// TODO: following sections will define remainder listing

```

Listing 8.2.2. SqlXmlDataSource class Sql command support.

Here is an "Sql" example:

```
static void TestSqlCommand()  
{  
    String cmdString =  
        "select EntityNo, AlternateId, Name from Entities for xml auto";  
    SqlXmlDataSource ds = new SqlXmlDataSource();  
    ds.CreateCommand(cmdString);  
    Console.WriteLine(ds.GetXmlResultFromStream());  
} // end of TestSqlCommand
```

The result for the above code was:

```
<Entities EntityNo="0" AlternateId="Ent1" Name="Entity 0"/>  
<Entities EntityNo="1" AlternateId="Ent1" Name="Entity 1"/>  
<Entities EntityNo="2" AlternateId="Ent1" Name="Entity 2"/>  
<Entities EntityNo="3" AlternateId="Ent1" Name="Entity 3"/>  
Press any key to continue . . .
```

8.2.2 XPath Command Sample

After the "TODO:" in Listing 8.2.2 add the following code:

```
// -----  
// XPath Command  
  
/// <summary>  
/// Create an XPath command.  
/// </summary>  
/// <param name="connectionString">connection string</param>  
/// <param name="annotatedSchemaPath">annotated schema path</param>  
/// <param name="xpathCommandText">XPath command that will be used  
/// . For example "/Customer[KindNo='1']".</param>  
/// <returns>an SqlXmlCommand is returned</returns>  
public Microsoft.Data.SqlXml.SqlXmlCommand CreateXPathCommand(  
    String connectionString,  
    String annotatedSchemaPath,  
    String xpathCommandText)  
{  
    CreateCommand(connectionString);  
    if (Command == null)  
        return(null);  
  
    Command.RootTag = "ROOT";  
    Command.CommandText = xpathCommandText;  
    Command.CommandType = Microsoft.Data.SqlXml.SqlXmlCommandType.XPath;  
    Command.SchemaPath = annotatedSchemaPath;  
  
    return(Command);  
} // end of CreateXPathCommand  
  
/// <summary>  
/// Get an Adapter for given xpath command.  
/// </summary>  
/// <param name="connectionString">connection string</param>  
/// <param name="annotatedSchemaPath">annotated schema path</param>  
/// <param name="xpathCommandText">XPath command that will be used  
/// . For example "/Customer[KindNo='1']".</param>  
/// <returns>an SqlXmlCommand is returned</returns>  
public Microsoft.Data.SqlXml.SqlXmlAdapter GetXPathAdapter(  
    String connectionString,  
    String annotatedSchemaPath,  
    String xpathCommandText)  
{  
    Microsoft.Data.SqlXml.SqlXmlAdapter ad;  
    Microsoft.Data.SqlXml.SqlXmlCommand cmd =  
        CreateXPathCommand(  
            connectionString, annotatedSchemaPath, xpathCommandText);
```

```

        ad = new Microsoft.Data.SqlXml.SqlXmlAdapter(Command);
        return(ad);
    } // end of GetXPathAdapter

    // TODO: following sections will define remainder listing

```

Listing 8.2.3. SqlXmlDataSource class XPath command support.

For the XPath command we need a schema as follows:

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
  <xsd:element name="Customers" sql:relation="Entity.Registry" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerId"
                    sql:field="AlternateId"
                    type="xsd:string" />
        <xsd:element name="CustomerName"
                    sql:field="Name"
                    type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

Here is an "XPath" example:

```

static void TestXPathCommand()
{
    String path = System.IO.Directory.GetCurrentDirectory();
    Microsoft.Data.SqlXml.SqlXmlAdapter ad =
        SqlXmlDataSource.GetXPathAdapter(null,
        path+"/SampleEntitySchema1.xsd", "Customer");
    System.Data.DataSet ds = new System.Data.DataSet();
    ad.Fill(ds);

    Int32 i;
    System.Data.DataTable dt = ds.Tables[0];
    for (i=0; i < dt.Rows.Count; i++)
    {
        Console.WriteLine(dt.Rows[i][0].ToString()+ " " + dt.Rows[i][1].ToString());
    }
} // end of TestXPathCommand

```

The result for the above code was:

```

0001 Rivera Rivera, Maria
0002 Rivera Martinez, Carlos
0003 Rivera Sanchez, Coral
0004 Rivera Roman, Raul
C001 Juancho's Inc.
C002 Carros y Algo Mas
Press any key to continue . . .

```

8.2.3 TemplateFile Command Sample

After the "TODO:" in Listing 8.2.3 add the following code:

```
// -----  
// TemplateFile Command  
  
/// <summary>  
/// Create an TemplateFile command.  
/// </summary>  
/// <param name="connectionString">connection string</param>  
/// <param name="templateFile">XPath command that will be used  
/// . For example "/Customer[KindNo='1']".</param>  
/// <returns>an SqlXmlCommand is returned</returns>  
public Microsoft.Data.SqlXml.SqlXmlCommand CreateTemplateFileCommand(  
    String connectionString,  
    String templateFile)  
{  
    CreateCommand(connectionString);  
    if (Command == null)  
        return(null);  
  
    Command.CommandText = templateFile;  
    Command.CommandType =  
        Microsoft.Data.SqlXml.SqlXmlCommandType.TemplateFile;  
  
    return(Command);  
} // end of CreateTemplateFileCommand  
  
// TODO: following sections will define remainder listing
```

Listing 8.2.4. SqlXmlDataSource class XPath command support.

In this case we need a Template file that is an XML document that contains an SQL or XPATH query such as the following:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">  
  <sql:query>  
    SELECT AlternateId CustomerId, Name  
    FROM Entity.Registry Entity  
    FOR XML AUTO  
  </sql:query>  
</ROOT>
```

Here is an "XPath" example:

```
static void TestTemplateFile()  
{  
    String path = System.IO.Directory.GetCurrentDirectory();  
    String cmdString = path+"/TemplateFileSample1.xml";  
    SqlXmlDataSource ds = new SqlXmlDataSource();  
    ds.CreateTemplateFileCommand(null, "Entities", cmdString);  
    Console.WriteLine(ds.GetXmlResultFromStream());  
} // end of TestTemplateFile
```

The result for the above code was:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">  
  <Entity CustomerId="0001" Name="Rivera Rivera, Maria"/>  
  <Entity CustomerId="0002" Name="Rivera Martinez, Carlos"/>  
  <Entity CustomerId="0003" Name="Rivera Sanchez, Coral"/>  
  <Entity CustomerId="0004" Name="Rivera Roman, Raul"/>  
  <Entity CustomerId="C001" Name="Juancho's Inc."/>  
  <Entity CustomerId="C002" Name="Carros y Algo Mas"/>  
</ROOT>  
Press any key to continue . . .
```

8.2.4 UpdateGram Command Sample

After the “TODO:” in Listing 8.2.4 add the following code:

```
// -----  
// UpdateGram Command  
  
/// <summary>  
/// Create an UpdateGram command.  
/// </summary>  
/// <param name="connectionString">connection string</param>  
/// <param name="annotatedSchemaPath">annotated schema path</param>  
/// <param name="updateGramPath">UpdateGram command file path</param>  
/// <returns>an SqlXmlCommand is returned</returns>  
public Microsoft.Data.SqlXml.SqlXmlCommand ExecUpdateGramCommand(  
    String connectionString,  
    String annotatedSchemaPath,  
    String updateGramPath)  
{  
    CreateCommandFromFile(connectionString, updateGramPath);  
  
    Command.CommandType =  
        Microsoft.Data.SqlXml.SqlXmlCommandType.UpdateGram;  
    Command.SchemaPath = annotatedSchemaPath;  
    Command.ExecuteNonQuery();  
  
    return(Command);  
} // end of ExecUpdateGramCommand  
  
// TODO: following sections will define remainder listing
```

Listing 8.2.5. SqlXmlDataSource class UpdateGram command support.

In this case we need an UpdateGram file that is an XML document such as the following:

```
<ROOT xmlns:updg="urn:schemas-microsoft-com:xml-updategram">  
<updg:sync>  
<updg:before>  
    <Customer><CustomerId>C001</CustomerId></Customer>  
</updg:before>  
<updg:after>  
    <Customer><CustomerName>Juan-Chon Tron Inc.</CustomerName></Customer>  
</updg:after>  
</updg:sync>  
</ROOT>
```

Here is an “UpdateGram” test example:

```
static void TestUpdateGram()  
{  
    String path = System.IO.Directory.GetCurrentDirectory();  
    String schemaPath = path+"/SampleEntitySchema1.xsd";  
    String updateGramPath = path+"/UpdateGramSample1.xml";  
  
    SqlXmlDataSource ds = new SqlXmlDataSource();  
    ds.CreateUpdateGramCommand(null, schemaPath, updateGramPath);  
    WriteDashLine();  
} // end of TestTemplateFile
```

In this case just take a look at the table and you will see that the “CustomerName” was update as requested.

8.2.5 DiffGram Command Sample

After the "TODO:" in Listing 8.2.5 add the following code:

```
// -----  
// DiffGram Command  
  
/// <summary>  
/// Execute / Submitt an DiffGram command.  
/// </summary>  
/// <param name="connectionString">connection string</param>  
/// <param name="annotatedSchemaPath">annotated schema path</param>  
/// <param name="updateGramPath">DiffGram command file path</param>  
/// <returns>an SqlXmlCommand is returned</returns>  
public Microsoft.Data.SqlXml.SqlXmlCommand ExecDiffGramCommand(  
    String connectionString,  
    String annotatedSchemaPath,  
    String diffGramPath)  
{  
    CreateCommandFromFile(connectionString, diffGramPath);  
  
    Command.CommandType =  
        Microsoft.Data.SqlXml.SqlXmlCommandType.DiffGram;  
    Command.SchemaPath = annotatedSchemaPath;  
    Command.ExecuteNonQuery();  
  
    return(Command);  
} // end of ExecDiffGramCommand
```

Listing 8.2.6. SqlXmlDataSource class DiffGram command support.

Let's update the previous schema as follows:

```
<?xml version="1.0"?>  
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
    xmlns:sql="urn:schemas-microsoft-com:mapping-schema"  
    elementFormDefault="qualified">  
    <xsd:element name="Customer" sql:relation="Entity.Registry">  
        <xsd:complexType>  
            <xsd:sequence>  
                <xsd:element name="RecordNo"  
                    sql:field="EntityNo"  
                    type="xsd:integer" />  
                <xsd:element name="TypeNo"  
                    sql:field="KindNo"  
                    type="xsd:integer" />  
                <xsd:element name="CustomerId"  
                    sql:field="AlternateId"  
                    type="xsd:string" />  
                <xsd:element name="CustomerName"  
                    sql:field="Name"  
                    type="xsd:string" />  
            </xsd:sequence>  
        </xsd:complexType>  
    </xsd:element>  
</xsd:schema>
```

In this case we need a DiffGram file that is an XML document such as the following:

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">  
    <diffgr:diffgram  
        xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"  
        xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">  
        <DataInstance>
```

```

    <Customer diffgr:id="Customer1" msdata:rowOrder="0"
      diffgr:hasChanges="inserted">
      <RecordNo>10</RecordNo>
      <TypeNo>1</TypeNo>
      <CustomerId>C010</CustomerId>
      <CustomerName>Hernandez, Teresa</CustomerName>
    </Customer>
  </DataInstance>
</diffgr:diffgram>
</ROOT>

```

Here is a “DiffGram” test example:

```

static void TestDiffGram()
{
    String path = System.IO.Directory.GetCurrentDirectory();
    String schemaPath = path+"/SampleEntitySchema2.xsd";
    String diffGramPath = path+"/DiffGramSample1.xml";

    SqlXmlDataSource ds = new SqlXmlDataSource();
    ds.ExecUpdateGramCommand(null, schemaPath, diffGramPath);
    WriteDashLine();
} // end of TestDiffGram

```

In this case just take a look at the table and you will see that a new record was inserted as requested.

9.0 Working with Images

Let’s update our “SampleDb” as follows:

```

CREATE TABLE Documents.Medias (
    CreatedDate DATETIME NOT NULL DEFAULT getdate(),
    EntityNo    INTEGER NOT NULL,
    MediaImage  IMAGE NOT NULL,

    CONSTRAINT fk_docmed001 FOREIGN KEY (EntityNo)
    REFERENCES Entity.Registry(EntityNo)
)
GO

```

We just added a table that has an “IMAGE” column type, no big deal. Now let’s add a new entity record for “John Doe” and insert an image into the database as follows:

```

INSERT INTO Entity.Registry VALUES ( 2, 6, '0006', 'Doe, John')
GO

INSERT INTO Documents.Medias (EntityNo, MediaImage)
SELECT 6, x.*
FROM OPENROWSET (BULK 'C:\temp\johndoe.jpg', SINGLE_BLOB) AS x
GO

```

Using our knowledge about XML in MS-SQL 2005 we can extract the image field as Base64 column by just issuing any “FOR XML” options such as in:

```

SELECT EntityNo,
       MediaImage 'MediaImage/Base64Data'
FROM Documents.Medias
FOR XML PATH('Medias'), ROOT('Documents')

```


9.1 Displaying XML Image in WinForm

To display the XML based image in a "PictureBox" control write a Windows Forms VB.Net sub as follows:

```
Public Shared Sub DisplayImageFromXml( _
    ByVal outputPictureBox As System.Windows.Forms.PictureBox, _
    ByVal xmlText As String)

    Dim sr As New System.IO.StringReader(xmlText)
    Dim buffLen As Int64 = 1024 * 10
    Dim readBuff As Int32
    Dim imgBuff(buffLen) As Byte
    Dim xdocr As System.Xml.XmlTextReader

    xdocr = New System.Xml.XmlTextReader(sr)
    If xdocr.ReadToFollowing("Base64Data") Then
        readBuff = xdocr.ReadElementContentAsBase64(imgBuff, 0, buffLen)
        Array.Resize(imgBuff, readBuff)
    End If
    sr.Close()
    sr.Dispose()
    xdocr.Close()

    Dim mstr As System.IO.MemoryStream = New System.IO.MemoryStream(imgBuff)
    Dim img As Drawing.Image = System.Drawing.Image.FromStream(mstr)

    outputPictureBox.Image = img
End Sub
```

You will note that the above nasty VB.Net code does leave you with some issues related to the size of the image that I will leave it to you to solve. Yes I hate the VB.Net language so much to write so ugly.

9.2 Displaying XML Image in WebForm

To display the XML based image in a "asp:img" html control use the following C# function:

```
public class Media
{
    public static String LoadXmlData(String filePath)
    {
        System.IO.TextReader tr = System.IO.File.OpenText(filePath);
        String data = tr.ReadToEnd();
        tr.Close();
        tr.Dispose();
        return(data);
    }

    public static void DisplayImageFromXml(
        System.Web.HttpResponse response,
        String xmlText)
    {
        System.IO.StringReader sr =
            new System.IO.StringReader(xmlText);
        Int64 buffLen = 1024 * 10;
        Int32 readBuff;

        Byte [] imgBuff = new Byte[buffLen];
        System.Xml.XmlTextReader xdocr;

        xdocr = new System.Xml.XmlTextReader(sr);
        if (xdocr.ReadToFollowing("Base64Data"))
        {
```

```

        readBuff = xdocr.ReadElementContentAsBase64(imgBuff, 0, (int)buffLen);
        System.Array.Resize(ref imgBuff, readBuff);
    }

    sr.Close();
    sr.Dispose();
    xdocr.Close();

    response.BinaryWrite(imgBuff);
}
}

```

Then on the WebForm page load write:

```

public partial class MediaLoad : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            String xmlData = Media.LoadXmlData("c:/temp/MediaFile.xml");
            Media.DisplayImageFromXml(Response, xmlData);
        }
    }
}

```

The actual "img" control that is referencing the "MediaLoad.aspx" page looks like this:

```

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
<title>XML Image Load Sample</title>
</head>
<body>
<form id="form1" runat="server">
<table cellpadding="0" cellspacing="0" width="100%">
<tr style="height:600px">
<td style="background-color:Gray" valign="middle" align="center">
    <asp:Image ID="Image1" ImageUrl="~/MediaLoad.aspx"
        runat="server" Width="200" />
</td>
</tr>
</table>
</form>
</body>
</html>

```

10.0 Terminology

Common terminology used includes:

Term	Description
IRI	International Resource Identifier. (see URI)
URI	Uniform Resource Identifier. A Uniform Resource Identifier (URI) is compact sequence of characters that identifies an abstract or physical resource. A URI can be further classified as a locator, a name, or both. (see [2])
URL	Uniform Resource Locator. The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location"). (see [2])
URN	Uniform Resource Name. The term "Uniform Resource Name" (URN) has been used historically to refer to both URIs under the

	"urn" scheme [RFC2141], which are required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable, and to any other URI with the properties of a name. (see [2])
XSD	Xml Schema Definition (W3C final recommendation)
XDR	Xml Data Reduced (<i>Deprecated</i> Microsoft prior [interim] XSD recommendation)

11.0 References and Suggested Readings

- [1] Malcom, Graeme (2002) "Programming Microsoft Sql Server 2000 with Xml", Microsoft Press
- [2] Network Working Group (2005) "Uniform Resource Identifier (URI): Generic Syntax", (see Web Page: <http://www.rfc-editor.org/rfc/rfc3986.txt>)
- [3] Beauchenmin, B., Berglund, N., and Sullivan, D (2004) "A First Look at SQL Server 2005 for Developers", Addison Wesley
- [4] Kay, Michael (2000) "XSLT: Programmer's Reference", WROX
- [5] Shankar, Pal, et.al. "XML Best Practices for Microsoft SQL Server 2005", Microsoft Corporation. Web Page: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsq190/html/sql25xmlbp.asp>
- [6] Schittko, Christoph, "SQLXML in .Net [for MS-SQL 2000]", TopXML. Web Page: <http://www.topxml.com/sqlxml/sqlxml.PDF>

11.1 Web Sites

PRPass	http://prpass.nagnoi.com/
SqlXml	http://www.sqlxml.org/default.aspx
XML	http://www.w3.org/XML/
XPath	http://www.w3.org/TR/1999/PR-xpath-19991008.html
XMLA	http://www.xmla.org/ http://www.xmlforanalysis.com/