XML with Virtuoso and XQuery

Introduction

XML has passed the point of ubiquity by establishing a standard format for data interchange. XML is today's foundation for building interoperability into new systems and applications; information architects use XML, XML data definitions (schemas) and messaging formats (SOAP, XML-RPC) as the clay with which to form IT system sculptures of Michaelangelian proportions.

In an imperfect world, XML's ideal descriptive constructs must make allowances for format translation and transformation. When fetching data from relational data sources, or receiving flat files over direct HTTPS, one must have tools to bend that which is given, to that which is actually needed.

XSLT has been, for the past several years, the Swiss army knife of XML transformation. Every XML processor and application server has some method to apply an XSLT transform to an instance of XML; for once XML is 'document en model', standard XPATH 2.0 is used within the framework of XSLT language.

With XML well adopted as a data type in the database universe, and native XML storage systems growing in market share, the recently (W3C) blessed XQuery is emerging as a popular method of manipulating and transforming XML data.

XSLT vs. XQuery in Virtuoso

OpenLink Software's Virtuoso Universal Server is a virtual database integration engine joined to a highly integrated web application platform . Virtuoso unifies data storage systems and provides a powerful suite of application delivery services. These features allow Virtuoso to act as a catalyst for the creation of composite applications based on Web Services standards.

Virtuoso allows the use of both XSLT and XQuery to suit the requirements of your application. XQuery and XSLT are functionally equivalent - programmers will be able to garner desirable results with either language.

XQuery is, of course, optimized for queries, whereas XSLT is optimized for reporting and transformation. If a task requires database retrieval, XQuery is the natural choice

as a SQL-like declarative dialect.
XSLT is the better choice to copy or transform a presently instantiated document within the context of an application data space. Extracting a data subset from a document may be accomplished with either XSLT or XQuery.

For those familiar with SQL, XQuery may be somewhat easier to learn, in a subjective sense, as XQuery was designed to manipulate highly structured data.

XQuery or XSLT' This issue boils down to the programmer's perspective -how is the data stored, how does one wish to report or transform' Virtuoso provides both XSLT and XQuery; pick your poison.

XQuery in Virtuoso

XQuery can also be used to retrieve elements and fragments from XML documents stored in the Virtuoso WebDAV virtual file system, from relational data in Virtuoso's core DBMS system, or from other databases attached via the VDB.

Virtuoso's XQuery benefits shine when a need arises to compose queries in order to return XML from an attached database sans XQuery support, or from a multi-vendor DBMS hell farm that does not support XML as a data type.

Becoming Familiar with XQuery

W3C defines XQuery as a language meant to "...provide flexible query facilities to extract data from real and virtual documents on the World Wide Web." XQuery is simply another XML transformation language, like XSLT with a dash of SQL style. Both XSLT and XQuery use the same XPATH 2.0 inner syntax for matching and transforming XML.

One may also wonder how XQuery is related to SQL, however loosely.

SQL, a declarative language, is optimized for relational queries, and generating tabular results. Vendor specific renderings of SQL have been type extended to support the generation of XML, encompassing some of capabilities of XQuery. A good example of the use of XQuery is the formation of hierarchical expressions, not an easy SQL task.
XQuery is semi-programmatic;. The query engine allows fine-grained control over document processing at the behest of the programmer. XQuery is therefore ideal for programmers transitioning from structured programming languages.

Common XQuery Usage Scenarios

Document Querying

XQuery's mission as defined by the W3C is XML document querying. Such a world-view of XQuery is provincial, when considering XQuery's capabilities beyond the realm of documents.
SQL implementations have always been organically suited to their underlying storage or retrieval systems. XQuery has a more limited native scope, but the loyal IT tools community is creative, and continues to work around limitations.

Native XML Databases

XQuery's first adoption was in XML Databases, a common milieu for XQuery. Relational database vendors now advocate XML hybrid data types, and XQuery will become a coin of the realm for XML and relational data queries.

Querying of Relational Data

Early XML support in relational databases consisted of simple row set serialization, amply adequate for queries without the need for hierarchical results. As XML's primary value is to represent hierarchical elements and attributes as self-describing information., this would not endure.

The SQL/XML (SQLX) specification is implemented as extended SQL functions, and affords fine-grained control of XML creation from your relational store. In either case, further processing may be required, and XQuery can fill that role.

Message Transformation

Message transformation is XQuery's tour de force. Making dissimilar systems interoperate is a full time job for the IT department; XQuery may attain popularity by helping with the heavy lifting of Integration tasks

It's de'ja' vu all over again [1]. Much like the era of terminal-based applications making the conversion to web-based services, today's two-tier applications are bound to relational data trapped in crystallized applications. Far too expensive for wholesale overhaul, the remake of these applications to web services will require supreme flexibility to translate XML to and from relational data services and storage systems. Xquery stands in as the refit mechanic's tool for information mutability until contemporary applications catch up by replacing columnar data with native XML constructs.XQuery is the engine of choice for conversion to formats better tolerated by web services clients.

The Capabilities of XQuery

The following is a list of some of the high-level capabilities of XQuery.

- Path Expressions
- Element Construction
- FLWR Expressions
- Operators and Functions
- Conditional Expressions

Path Expressions

XPath is the core selection model that drives XQuery. XPath includes full-depth searches, navigational predicates, alternative axes, functions, and the ability to address more than one node at a time with a single path expression. The selfsame XPath V2.0 is used by XSLT in a different language construct.

Here is an example of a simple XPath expression:

```
/Emps/Emp[FirstName="Nancy" and LastName="Davolio"]
```

This query instructs the XPath processor to navigate to the 'Emps' element of the working context, in this case, the root element of the document, and select all 'Emp' child elements matching the predicate criteria included within square brackets. This predicate requires that the first 'FirstName' element of the 'Emp' element be equal to 'Nancy', and the first 'LastName' element of the 'Emp' element equals 'Davolio'.

Note "the first 'FirstName' element";.an important point regarding XPath processing is the exposure of nodes as sequences. Certain comparison constructs, such as the '=' operator, act solely on the first node of a sequence. In the case of an 'Emp' element with two 'FirstName' elements, the XPath expression would fail to return the expected element, though logically it should. If the order of 'Mary' and 'Nancy' elements were reversed, the query would return that 'Emp' element. Work around this problem by placing the tests values in predicates of their own:

```
/Emps/Emp[FirstName[text()="Nancy"] and LastName[text()="Davolio"]]
```

The previous query introduces the concept of XPath function calls. XPath functions are executed in the context of the current node. In this case, the text() function is called repeatedly for each element in the context of the 'FirstName' and 'LastName' sequences; only those elements returning text() function as queried value results are returned from a given sequence.

To query a document for 'LastName' elements, ignoring the 'Emp' element context, one would issue this path statement:

```
//LastName
```

This traverses the depth of the document searching for 'LastName', returning each element as part of a sequence. Predicates can also be applied in order to query documents with deep hierarchies.

Element Construction

The ability to query a document and return element nodes is no guarantee of a well-formed document result. XQuery allows output of XML elements and attributes as a well-formed document, using XML or computed syntax.

An example of Element construction:

```
<LastNames>
{
document("EmployeeOrders.xml")//LastName
}
</LastNames>
```

This uses the XML-syntax of construction. It could also be expressed using the computed syntax:

```
element LastNames {
   document("EmployeeOrders.xml")//LastName
}
```

The computed syntax allows creation of data types within the XQuery model, such as timestamps, strings, floating point numbers, and integers.[2]

FLWR Expressions

The power of XQuery exists in its FLWR expressions.

FLWR expressions allow XQuery to iterate a sequence of nodes containing conditional operations.. FLWR is the abbreviation of the syntactic operator constructs within an expression: For, Let, Where, and Return. To better understand these keywords, think of it this way:

- **For** each variable that I assign from this expression
- **Let** these values equal the result of these expressions
- **Where** this expression results in a boolean true result
- **Return** this node

The following is a simple XQuery example demonstrating For, Where, and Return keywords. Notice that the keywords may be omitted from the expression depending on the query-processing context:

```
<LastNames>
{
   for $i in document("EmployeeOrders.xml")//LastName
   where starts-with($i/text(), "Da")
   return
      <ln>
      {
         $i/text()
      }
      </ln>
}
</LastNames>
```

This example selects the 'LastName' elements from EmployeeOrders document. For each node in the sequence, test to see that the text() function returns a value that starts with ('Da'). If so, return the constructed element 'ln', placing in it only the textual value of each node, rather than the entire element. Another way to express this query using a Let keyword follows:

```
let $lastnames := document("EmployeeOrders.xml")//LastName
return
   <LastNames>
   {
      for $i in $lastnames
      where starts-with($i/text(), "Da")
      return
         <ln>
         {
            $i/text()
         }
         </ln>
   }
   </LastNames>
```

Operators and Functions

Operators and functions are an important part of the conditional processing of node sequences, and the dynamic construction of XML content in XQuery. Comparison operators and functions can be used to filter processed node sequences. String and numeric manipulation methods can be used to alter element values and perform numeric aggregation on sequences of nodes.

The most common operators in XQuery are:

Arithmetic Operator	**Description**
+	Add the value of the first expression to the second.
-	Subtract the value of the second expression from the first.
*	Multiple the value of the first expression by the second.
div	Divide the value of the first expression by the second.
idiv	Divide the value of the first expression by the second, returning only the integral portion of the result.
mod	Divide the value of the first expression by the second, returning only the remainder of the result.
Value Comparison	**Description**
eq	Returns true if the first scalar expression is equal to the second.
ge	Returns true if the first scalar expression is greater than or equal to the second.
gt	Returns true if the first scalar expression is greater than the second.
le	Returns true if the first scalar expression is less than or equal to the second.
lt	Returns true if the first scalar expression is less than the second.
ne	Returns true if the first scalar expression is not equal to the second.
General Comparison	**Description**
=	Returns true if the first scalar expression is equal to the second.
>=	Returns true if the first scalar expression is greater than or equal to the second.
>	Returns true if the first scalar expression is greater than the second.
<=	Returns true if the first scalar expression is less than or equal to the second.
<	Returns true if the first scalar expression is less than the second.

!=	Returns true if the first scalar expression is not equal to the second.
Logical Operator	**Description**
or	Returns true if either the first or the second expression are true.
and	Returns true only if the first and the second expression are true.

Notice that Value Comparisons perform similarly to General Comparisons, with a subtle difference. Value Comparisons will operate on two operands or sequences representing a single node. If a sequence with more than one node is passed to a Value Comparison, an exception will be flagged. General Comparisons performs implicit conversions depending on the operation, such as using only the first node in the sequence.
More common functions:

Function	**Description**
concat	Concatenates two or more character strings.
starts-with	Indicates whether the value of one string begins with the characters of the value of another string.
ends-with	Indicates whether the value of one string ends with the characters of the value of another string.
contains	Indicates whether the value of one string contains the characters of the value of another string. A collation may optionally be specified.
substring	Returns a string located at a specified place in the value of a string.
count	Returns the number of items in the sequence.
avg	Returns the average of a sequence of numbers.
max	Returns the object with maximum value from a collection of comparable objects.
min	Returns the object with minimum value from a collection of comparable objects.
sum	Returns the sum of a sequence of numbers.
current-dateTime	Returns the current dateTime.
current-date	Returns the current date.
current-time	Returns the current time.

Conditional Expressions

XQuery allows the return of one of two values depending on the result of an expression. This behavior can be chained with nested tests as well. The syntax to perform these actions is found in the 'if' operator.

```
let $doc := document("EmployeeOrders.xml")//Emp
return
   <Emps>
   {
      for $i in $doc
      return
         <Emp>
         {
           $i/FirstName,
           if ( $i/LastName/text() = "Davolio" ) then
              <LastName davolio="true" >
              {
                 $i/LastName/text()
              }
              </LastName>
           else
              <LastName>
              {
                 $i/LastName/text()
              }
              </LastName>
         }
         </Emp>
   }
   </Emps>
```

This query outputs the 'FirstName' and 'LastName' elements of each 'Emp' element found in the document, adding a special attribute called 'davolio' to the resulting 'LastName' elements if the source value is equal to 'Davolio'.

The Virtuoso Demo Database

For this demonstration, use the Virtuoso demo database included in the standard installation.

Type the following URL into your web browser:

```
http://localhost:8890/admin/
```

This will prompt for the DBA user name and password. After correctly entering username and password, you will be presented with the Virtuoso Administrator interface.

Issuing Queries in Virtuoso Administrator using XQuery

In Conductor, Virtuosos System Administrator, navigate to the XML Service tab, select the XQuery sub tab and then the XQuery Basic option of the XQuery Wizard interface.

To test if all is well, enter the XQuery Context mode for the DAV resource and then type "/DAV/xmlsql/EmployeeOrders.xml" as the Path. Enter the NEXT button and enter in the example document in to this query window. After that, cut and paste the "Conditional Expressions" example from this document into the query editing area and then click the "Execute" button to check the query The result should look like this:

```
<Emps>
<Emp>
    <FirstName>Nancy</FirstName>
    <LastName davolio="true">Davolio </LastName>
</Emp>
<Emp>
    <FirstName>Andrew</FirstName>
    <LastName>Fuller </LastName>
</Emp>
<Emp>
    <FirstName>Janet</FirstName>
    <LastName>Leverling </LastName>
</Emp>
<Emp>
    <FirstName>Margaret</FirstName>
    <LastName>Peacock </LastName>
</Emp>
<Emp>
    <FirstName>Steven</FirstName>
    <LastName>Buchanan </LastName>
</Emp>
<Emp>
    <FirstName>Michael</FirstName>
    <LastName>Suyama </LastName>
</Emp>
<Emp>
    <FirstName>Robert</FirstName>
    <LastName>King </LastName>
</Emp>
<Emp>
    <FirstName>Laura</FirstName>
    <LastName>Callahan </LastName>
</Emp>
<Emp>
    <FirstName>Anne</FirstName>
    <LastName>Dodsworth </LastName>
</Emp>
</Emps>
```

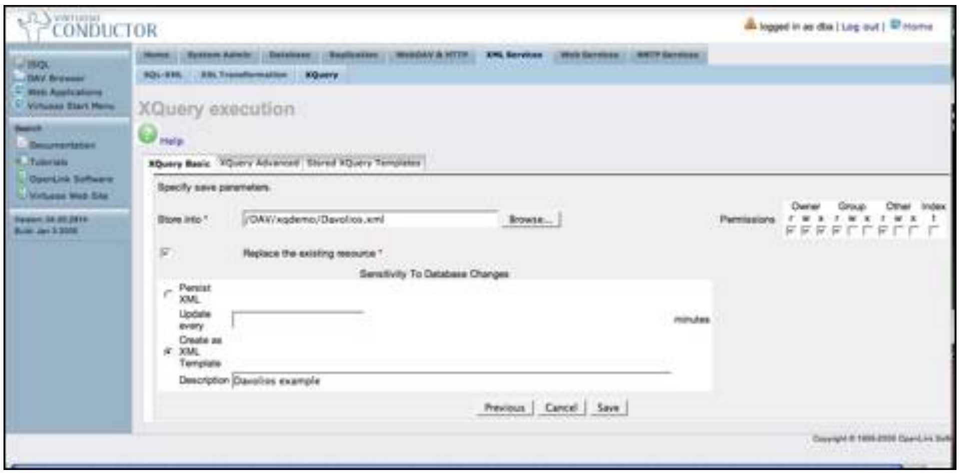
Hit the NEXT button in the XQuery wizard to save.   You can then continue cutting, pasting, and executing the other examples. Try to make modifications to the queries, such as renaming the elements that are produced, or promoting all of the elements to attributes. Virtuoso will indicate syntax errors, and this is good way to learn.

Saving Queries as HTTP Resources

Queries used for periodic generation of XML results may be saved asa text file and used via Virtuoso's administrative interface, but there are better alternatives for repetitive use of saved queries. In order to create an XQuery that is available by web browser, simply click the 'Save' button instead of the 'Execute' button on the XQuery interface.

XQueries may be stored in Virtuoso's WebDAV virtual directory. To access XML query results, point your browser or XML tool to the WebDAV URI storing the saved XQuery. Virtuoso may be configured to generate this document on demand, or periodically re-generate the document.

The screen will look like this:



Specify save parameters.

Store into * /DAV/xqdemo/Davolios.xml

Permissions: Owner r w x, Group r w x, Other r w x, Index t

Replace the existing resource *

Sensitivity To Database Changes

Persist XML

Update every minutes

Create as XML Template

Description Davolios example

Set the following fields to their respective values:

Field	***Value***
File of XML template	*/DAV/xqdemo/Davolios.xml*
Replace Existing Resource	*Selected*
XML Template	*Checked (Davolios example)*
Permissions	*Owner: rwx - Group: r - Other: r*

Set the 'Query' field to the following:

```
for $i in document("EmployeeOrders.xml")//Emp
return
   <Emp>
   {
     $i/FirstName,
     if ( $i/LastName/text() = "Davolio" ) then
        <LastName davolio="true" >
        {
           $i/LastName/text()
        }
        </LastName>
     else
        <LastName>
        {
           $i/LastName/text()
        }
        </LastName>
   }
   </Emp>
```

Notice that this query omits the root-level 'Emps' element. Virtuoso's XQuery template system will automatically generate a root element, and in this case, we've specified that it should generate a root element named 'Emps'.

Click the 'Save' button. There will be no indication that the query has been saved, however, if you browse Virtuoso's WebDAV repository, you will find a document called 'Davolios.xml' under the /DAV/xqdemo/ collection.

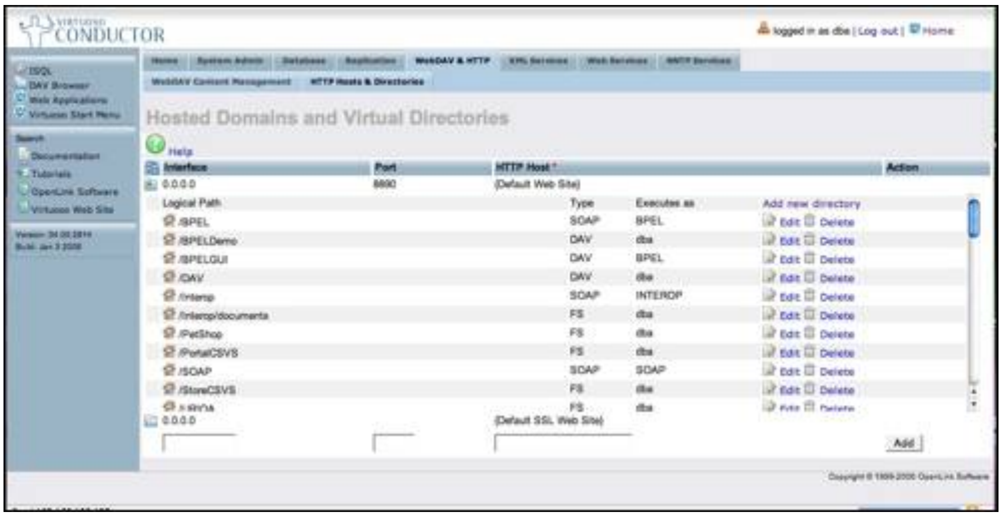
You may now browse to the following URL to view the query results:

```
http://localhost:8890/DAV/xqdemo/Davolios.xml
```

Virtuoso WebDAV requires user authentication to browse stored queries. This is a prudent security practice, but there may be situations that call for a query to be made publicly available.
Creating an open virtual directory as part of Virtuoso's default HTTP server is a simple task.

Navigate to the 'HTTP Hosts & Directories' sub-tab under the WebDAV & HTTP.In the 'HTTP Hosts & Directories' screen, expand the directories left of the default port and Default Web Site. by clicking on the folder icon. Once the directories are expanded click on the 'Add New Directory' Link at the top of the directory to create a new mapping.

The screen will then look like this:



Hosted Domains and Virtual Directories

Interface	Port	HTTP Host *	Action
0.0.0.0	8890	(Default Web Site)	

Logical Path	Type	Executes as	Add new directory
/BPEL	SOAP	BPEL	Edit Delete
/BPELDemo	DAV	dba	Edit Delete
/BPELGUI	DAV	BPEL	Edit Delete
/DAV	DAV	dba	Edit Delete
/interop	SOAP	INTEROP	Edit Delete
/interop/documents	FS	dba	Edit Delete
/PetShop	FS	dba	Edit Delete
/PortalCSVS	FS	dba	Edit Delete
/SOAP	SOAP	SOAP	Edit Delete
/StoreCSVS	FS	dba	Edit Delete
[illegible]	FS	dba	Edit Delete

Interface	Port	HTTP Host *	Action
0.0.0.0		(Default SSL Web Site)	
			Add

Copyright © 1999-2006 OpenLink Software

In the Virtual Directory Wizard, select 'None' for the Virtual Directory Type and hit the NEXT button. In the next screen in the wizard, set the following fields to their respective values, and click the 'Save Changes' button.

Field	Value

Logical Path	/xquery-test
WebDAV source	Checked
Physical Path or URL	/DAV/xqdemo/

Browse to the following URL to view the query results:

```
http://localhost:8890/xquery-test/Davolios.xml
```

This URL is open whereas the WebDAV URL required authentication.

Virtuoso-Specific Extensions

The official XQuery specification is limited to an agnostic feature implementation. XQuery makes no assumptions regarding context, environment, or supporting infrastructure. Vendors commonly extend standards-based languages with additional system specific functions.

Learn More

- **Virtuoso Documentation**
- **Virtuoso Tutorials**

[1] Yogi Berra

[2] Note: The computed syntax is not yet supported by Virtuoso's XQuery implementation.