Essential XQuery - The XML Query Language

Monday, February 2, 2004

Summary
XQuery or XML Query Language is a W3C specification (http://www.w3.org/TR/xquery) designed to provide a flexible and standardized way of searching through (semi-structured) data that is either physically stored as XML or virtualized as XML (such as XML Views over relational data). To give you an analogy, SQL is used to query relational data, XQuery is a standard language for querying XML data. In addition, XQuery also allows general processing of XML (such as creating nodes). Currently in the last call working draft status, the W3C XQuery specification will hopefully get the recommendation status this year. This article is an introduction to XQuery language. It is based on November 2003 working draft. (**Click here to see the changes made to 23rd July 2004 XQuery 1.0 Working Draft.**)
(23 printed pages)

By Darshan Singh

Getting Ready

- This article assumes that you have some familiarity with XML 1.0, XPath 1.0, XML Namespaces, XML Schema, and XSLT. If you want to get a quick 15 minutes refresher on XPath, see my article on PerfectXML.com, XPath for .NET Developers; this article contains some introductory text on XPath.
- The examples presented in this article might not work with Microsoft SQL Server "Yukon" Beta 1 XQuery implementation, which is based on subset of XQuery drafts in range of December 20, 2001 to November 15, 2002.
- Download the latest version of Saxon from saxon.sourceforge.net to try out the examples. I am using version 7.8 (using its command line interface) in this article.

XQuery Info on W3C Web site

The XQuery effort at W3C is lead by the XML Query Working Group (http://www.w3.org/XML/Query). XQuery borrows a lot of features from XPath; actually, XQuery 1.0 is said to be an extension to XPath 2.0. Both, XPath 2.0 and XQuery 1.0 make use of the same data model (the abstract, parsed, logical structure of an XML document). XPath 2.0 and XQuery 1.0 are very closely related. The XML query working group works closely with XPath editors (part of XSL working group), and have published specifications together.

XQuery 1.0 is a strongly-typed language. The XQuery type system is based on XML Schema W3C recommendation (http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/). XQuery 1.0 supports XML Schema simple types (such as string, integer, dateTime, etc.), types from imported schemas (such as USAddress), and XML document structure types (such as document, element, attribute, node, processing instruction, comment, etc.).

There are total twelve documents published by XML Query Working Group (some in conjunction with XSL working group). These documents together define the XQuery 1.0 and XPath 2.0 languages. When writing XQuery queries, I often refer to specifications marked as **bold** in the following table:

Specification	Description
XML Query Requirements www.w3.org/TR/xquery-requirements	A document to define and track goals and requirements for the XQuery data model, algebra, and query language.
XQuery 1.0: An XML Query Language **www.w3.org/TR/xquery**	The actual XQuery 1.0 specification document. Defines the language grammar using Basic EBNF notation.
XML Query Use Cases **www.w3.org/TR/xquery-use-cases**	This document contains around 80 XQuery snippets solving specific problems in application areas (categorized in nine categories). If you prefer to learn by example, you can start with this document, and see how XQuery looks like and what it can be used for.
XQuery 1.0 & XPath 2.0 Data Model www.w3.org/TR/xpath-datamodel	The term Data Model refers to the abstract, logical, parsed structure of an XML document. It is an extension to the XML Infoset, XPath 2.0, XSLT 2.0, and XQuery 1.0 use the same data model, wherein, every document is represented as a tree of nodes. Input the XQuery is a data model, and output is also a data model.
XQuery 1.0 & XPath 2.0 Formal Semantics www.w3.org/TR/xquery-semantics	This document formally defines the XPath 2.0 and XQuery 1.0 languages (using underlying algebra).
XQuery 1.0 & XPath 2.0 Functions & Operators **www.w3.org/TR/xpath-functions**	This document is a catalog of about 230 functions and operators required by XPath 2.0, XQuery 1.0, and XSLT 2.0.
XPath Requirements Version 2.0 www.w3.org/TR/xpath20req	A relatively small document describing the XPath 2.0 requirements alongwith annotated resolution in XPath and XQuery specifications.
XML Path Language (XPath) 2.0 www.w3.org/TR/xpath20	The actual XPath 2.0 specification document. It uses Basic EBNF notation to describe the XPath grammar.
XSLT 2.0 and XQuery 1.0 Serialization www.w3.org/TR/xslt-xquery-serialization	The process of converting nodes from data model into a sequence of octets. XQuery implementation may or may not provide a serialization interface. This document describes the serialization of data model.
XML Query and XPath Full-Text Requirements www.w3.org/TR/xquery-full-text-requirements	An extension to XPath/XQuery languages, full text search facilitates searching tokenized text. This document defines this extension to XPath and XQuery languages.
XML Query and XPath Full-Text Use Cases www.w3.org/TR/xmlquery-full-text-use-cases	Around 120 examples to show full-text search can be performed using XQuery and XPath.

XML Syntax for XQuery 1.0 (XQueryX) www.w3.org/TR/xqueryx	XQuery 1.0 does not use XML syntax (however, it is important to note that XQuery is case-sensitive language). (XSLT uses XML syntax.). XQuery 1.0 is a functional language whose basic building block is an expression (like as in XPath). However, one of the XML Query 1.0 requirements mentions that XML query language may have more than one syntax binding. XQueryX is an XML representation of an XQuery. There is no tool/product support for XQueryX as of today. XQueryX uses the verbose XML syntax that is most suitable for machine consumption, mostly to parse, generate and interrogate the contents of a query.

Basics

You write XQuery queries to efficiently extract data from XML data/documents. These queries can be executed via XQuery processor implementations (like Yukon, SAXON 7.8, Qexo, X-Hive/DB and so on). You can execute these queries on command prompt, or from within your code. In this article, we'll execute all the queries on the command prompt using SAXON 7.8. SQL Server Yukon allows executing XQuery queries using SQL Workbench tool; it also includes XQuery Designer allowing you to graphically create and execute XQuery expressions. In addition, you can write .NET code to execute XQuery expressions.

The concept of querying XML documents is not new. Several past research papers, proposals, and specifications have influenced the design of XQuery. These include XQL, OQL, XML-QL, Lorel, Quilt and XPath 1.0 The last two entries in the previous list (Quilt and XPath 1.0) have the greatest impact on how XQuery looks today. If you are familiar with XPath concepts (expressions, location steps, XPath functions, and so on), XQuery would seem a natural extension to that with looping, sorting, conditional, and modularization constructs added in order to meet the requirements of a query language. XPath now can be treated as a core subset of XQuery.

Instead of digging deep into XQuery concepts here, the way I want to proceed is to list some facts that you should remember, and then show you some XQuery examples, followed by discussion on XQuery theoretical concepts and study the terminology. With this, here is a list of some important facts about XQuery that you should remember:

- XQuery 1.0 queries are not written using XML syntax (XSLT is written using XML syntax, for instance). Like XPath, Expressions form the basic building block of XQuery queries. Another W3C proposal, XQueryX, is in progress that makes use of XML syntax for XQuery queries.
- XQuery is a case-sensitive language. All the XQuery keywords are in lower-case.
- XQuery is a functional language comprised of several kinds of expressions that return values. Every query is an expression to be evaluated. These expressions can be nested and composed with full generality. We'll talk about this more, and see examples of this in following sections.
- XQuery is a strongly typed language with a type system based on W3C XML Schema (XSD) specification. Strong typing provides two primary benefits: static compilation for early error detection, and query optimization by devising efficient execution plan (as the engine knows beforehand about the object types). We'll see examples of this in following sections.
- XQuery 1.0 as such does not specify any mechanism to perform updates using XQuery queries. However, almost all the XQuery implementations have custom extension to allow updates using XQuery queries.
- XQuery allows locating nodes (like XPath), constructing and re-structuring XML (like XSLT, but XQuery has a different syntax), sorting, filtering, and so on. One of the most commonly used XQuery expressions is the FLWOR (for-let-where-order by-return) expressions, pronounced "flower expressions".
- XQuery makes some of the XPath axes as optional. In other words, it is optional for XQuery processor to support ancestor, ancestor-or-self, following, following-sibling, preceding, and preceding-sibling axes. If it does support these axes (in addition to other XPath axes), it is said to support Full Axis Feature.
- XQuery offers the ability to do JOINs across documents.
- XQuery offers a comprehensive set of functions and operators that you can use in your queries.

The next two sections are provided here to give you an idea of how XQuery queries look like. Don't worry too much about syntax at this point, just look at the queries, understand the generated output, and we'll look at the details in the subsequent sections.

Example Data

The series of examples in the next section are based on the following two sample XML documents: emp.xml (contains employee data) and timeoff.xml (contains absence requests data):

emp.xml

```
<?xml version="1.0" encoding="utf-8"
         standalone="yes" ?>
<employees>

  <emp id="1">
    <name>Kelly Firkins</name>
    <email>kelly</email>
    <ssn>111-11-1111</ssn>
    <managedEmail>dean</managedEmail>
    <dateOfJoining>1994-11-16</dateOfJoining>
  </emp>

  <emp id="2">
    <name>Dean Parker</name>
    <email>dean</email>
    <ssn>222-22-2222</ssn>
    <managedEmail>mark</managedEmail>
    <dateOfJoining>1992-12-24</dateOfJoining>
  </emp>

  <emp id="3">
    <name>Kim Fell</name>
    <email>kim</email>
    <ssn>333-33-3333</ssn>
    <managedEmail>mark</managedEmail>
```

```
    <dateOfJoining>1993-08-19</dateOfJoining>
  </emp>

  <emp id="4">
    <name>Katie Parise</name>
    <email>katie</email>
    <ssn>444-44-4444</ssn>
    <managedEmail>kim</managedEmail>
    <dateOfJoining>1996-07-11</dateOfJoining>
  </emp>

  <emp id="5">
    <name>Mark Robinson</name>
    <email>mark</email>
    <ssn>555-55-5555</ssn>
    <managedEmail></managedEmail>
    <dateOfJoining>1990-01-01</dateOfJoining>
  </emp>

</employees>
```

timeoff.xml

```
<?xml version="1.0" encoding="utf-8"
        standalone="yes" ?>
<timeoff>

  <request id="T1" empID_="2"
    startDate="2003-01-02" endDate="2003-01-10"
    hours="40" status="approved" absenceCode="vacation" />

  <request id="T2" empID_="4"
    startDate="2003-01-10" endDate="2003-02-10"
    hours="180" status="approved" absenceCode="vacation" />

  <request id="T3" empID_="1"
    startDate="2003-01-15" endDate="2003-01-31"
    hours="80" status="declined" absenceCode="vacation" />

  <request id="T4" empID_="1"
    startDate="2003-02-01" endDate="2003-02-01"
    hours="8" status="approved" absenceCode="sick" />

  <request id="T5" empID_="3"
    startDate="2003-02-12" endDate="2003-02-12"
    hours="8" status="approved" absenceCode="sick" />

  <request id="T6" empID_="1"
    startDate="2003-03-01" endDate="2003-03-05" hours="30"
    status="approved" absenceCode="vacation" />

  <request id="T7" empID_="4"
    startDate="2003-03-12" endDate="2003-03-15" hours="10"
    status="approved" absenceCode="unpaid" />

</timeoff>
```

15 Simple XQuery Examples

In this section, I'll use the above two sample XML documents and show you 15 different queries. As mentioned before, I'll be using SAXON 7.8 to run these sample queries. I'll assume that all the queries and data files (above two XML files) reside in the same folder.

1: All employees that directly report to Mark Robinson

Query:

```
doc("emp.xml")/employees/emp[managerEmail='mark']
```

Save the above text in a text file and call it 1.xq and then you can run the following on the command line:

```
java net.sf.saxon.Query 1.xq
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<emp id="2">
      <name>Dean Parker</name>
      <email>dean</email>
      <ssn>222-22-2222</ssn>
      <managerEmail>mark</managerEmail>
      <dateOfJoining>1992-12-24</dateOfJoining>
   </emp>
<?xml version="1.0" encoding="UTF-8"?>
```

```
<emp id="3">
      <name>Kim Fell</name>
      <email>kim</email>
      <ssn>333-33-3333</ssn>
      <managerEmail>mark</managerEmail>
      <dateOfJoining>1993-08-19</dateOfJoining>
</emp>
```

Description:
If you have XPath knowledge, the above path expression query should look quite familiar (except the doc function). XQuery, like XPath, supports both the abbreviated (., .., //, and so on) and non-abbreviated (self::node(), parent::node(), /descendant-or-self::node()/, and so on) syntax of path expression. We'll use the abbreviated path syntax in almost all the queries in this article.

The above query selects all the emp nodes from the emp.xml document, where managerEmail equals 'mark'. This example shows how XQuery can be as simple as XPath; yet it is much more powerful than that as the rest of the examples illustrate.

If you get the following error:

```
Error on line 1 of file:/C:/temp/emp.xml:
Error reported by XML parser: Document root element is missing.
Warning: org.xml.sax.SAXParseException: Document root element is missing.
Error on line 1 Failed to load document emp.xml
```

The above error means that the XML parser (Crimson) was not able to load the source XML document, mainly because your XML document contains byte order mark (for example EF BB BF for UTF-8 files) at the beginning of the file. You'll have to remove these initial bytes using any binary file editor (like Microsoft Visual Studio; open the file in binary mode).

2: Employees that do not report to anybody

Query:

```
doc("emp.xml")/employees/emp[managerEmail='' or count(managerEmail) < 1]
```

Save the above text in a text file and call it 2.xq and then you can run the following on the command line:

```
java net.sf.saxon.Query 2.xq
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<emp id="5">
      <name>Mark Robinson</name>
      <email>mark</email>
      <ssn>555-55-5555</ssn>
      <managerEmail/>
      <dateOfJoining>1990-01-01</dateOfJoining>
</emp>
```

Description:
This XQuery expression reads emp.xml document and selects the emp nodes where either managerEmail element value is an empty string or the element is not present at all.

3: All employees whose name starts with letter 'K'

Query:

```
doc("emp.xml")/employees/emp[upper-case(substring(name, 1, 1)) = 'K']
```

Save the above text in a text file and call it 3.xq and then you can run the following on the command line:

```
java net.sf.saxon.Query 3.xq
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<emp id="1">
      <name>Kelly Firkins</name>
      <email>kelly</email>
      <ssn>111-11-1111</ssn>
      <managerEmail>dean</managerEmail>
      <dateOfJoining>1994-11-16</dateOfJoining>
</emp>
<?xml version="1.0" encoding="UTF-8"?>
<emp id="3">
      <name>Kim Fell</name>
      <email>kim</email>
      <ssn>333-33-3333</ssn>
      <managerEmail>mark</managerEmail>
      <dateOfJoining>1993-08-19</dateOfJoining>
</emp>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<emp id="4">
      <name>Katie Parise</name>
      <email>katie</email>
      <ssn>444-44-4444</ssn>
      <managerEmail>kim</managerEmail>
      <dateOfJoining>1996-07-11</dateOfJoining>
</emp>
```

Description:
This XQuery expression illustrates some of the built-in string functions such as upper-case and substring.

4: What date did Kim Fell join?

Query:

```
data(doc("emp.xml")/employees/emp[name='Kim Fell']/dateOfJoining)
```

Results:

```
1993-08-19
```

Description:
This XQuery expression illustrates one of the ways in which you can get the atomic value from a node via the data() function.

5: Sum of Timeoff hours for employee with ID equals 4

Query:

```
sum(doc("timeoff.xml")/timeoff/request[@empID_=4 and
          @status="approved"]/@hours)
```

Results:

```
190
```

Description:
This XQuery expression illustrates accessing the attribute nodes and the aggregate function - sum.

6: List all unpaid requests

Query:

```
doc("timeoff.xml")/timeoff/request[@absenceCode="unpaid" and
          @status="approved"]
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<request id="T7" empID_="4" startDate="2003-03-12"
     endDate="2003-03-15" hours="10" status="approved"
     absenceCode="unpaid"/>
```

Description:
This XQuery expression returns all the request nodes having absenceCode attribute as unpaid.

7: List all Timeoff requests sorted by hours in descending order: Our first FLWOR query

Query:

```
for $req in doc("timeoff.xml")/timeoff/request
order by xs:double($req/@hours) descending
return $req
```

Results (abridged to save some space):

```
<?xml version="1.0" encoding="UTF-8"?>
<request id="T2" empID_="4" startDate="2003-01-10" endDate="2003-02-10"
         hours="180" status="approved" absenceCode="vacation"/>
<?xml version="1.0" encoding="UTF-8"?>
<request id="T3" empID_="1" startDate="2003-01-15" endDate="2003-01-31"
         hours="80" status="declined" absenceCode="vacation"/>
<?xml version="1.0" encoding="UTF-8"?>
<request id="T1" empID_="2" startDate="2003-01-02" endDate="2003-01-10"
         hours="40" status="approved" absenceCode="vacation"/>
```

Description:
The above query illustrates how iteration and sorting works in XQuery. The above query is our first example based on FLWOR expression (however it does not make use of let and where clauses). Also, we are assuming that there is no XSD schema associated or used here (this is the reason we

have to use xs:double constructor function in the order by statement, else the sorting would be based on text values).

The above query reads like this: *For each request (for $req in doc("timeoff.xml")/timeoff/request) order by the hours attribute in descending order (order by xs:double($req/@hours) descending) and then return matching (all in this case) requests (return $req)*. $req is a variable bound to request nodes.

8: Print names of employees whose requests were declined

Query:

```
for $emp in doc("emp.xml")/employees/emp,
    $req in doc("timeoff.xml")/timeoff/request[@empID_ = $emp/@id]
where $req/@status = 'declined'
return data($emp/name)
```

Results:

```
Kelly Firkins
```

Description:
The above query illustrates **JOIN**ing data from two XML sources.

The above query reads like this: *Join employee and timeoff requests on employee ID, filter by requests having status as declined, for all such employee return the name element value*.

There are many ways in which a query can be written to get the same desired results. For instance, you can get rid of the where condition from the above query by adding an and condition during the $req variable binding:

```
for $emp in doc("emp.xml")/employees/emp,
    $req in doc("timeoff.xml")/timeoff/request[@empID_ = $emp/@id
         and @status = 'declined']
return data($emp/name)
```

9: Employees and total timeoff hours: Generating XML

In this example, we are going to JOIN employee and timeoff XML sources, to create a new XML document that contains basic employee information and sum of their timeoff hours. The generated XML would list employees sorted by employee name.
Query:

```
<Report>
{
   for $emp in doc("emp.xml")/employees/emp
   let $req := doc("timeoff.xml")/timeoff/request[@empID_ = $emp/@id and
           @status="approved"]
   order by $emp/name ascending
   return
      <timeoffDetails
            empID="{$emp/@id}"
            empName="{$emp/name}"
            totalHours="{sum($req/@hours)}"
            />
}
</Report>
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<Report>
   <timeoffDetails empName="Dean Parker" empID="2" totalHours="40"/>
   <timeoffDetails empName="Katie Parise" empID="4" totalHours="190"/>
   <timeoffDetails empName="Kelly Firkins" empID="1" totalHours="118"/>
   <timeoffDetails empName="Kim Fell" empID="3" totalHours="8"/>
   <timeoffDetails empName="Mark Robinson" empID="5" totalHours="0"/>
</Report>
```

Description:
This query illustrates few important XQuery features such as element/attribute construction, the let statement, and using curly braces **{ }** to delimit the enclosed expressions.

The query starts with creating <Report> element. Enclosed within this element is an expression which reads: for each employee, bind $req to employee's timeoff requests, order by employee name, and return a new sub-element called timeOffDetails containing three attributes: employee ID, name, and sum of timeoff hours.

10: Employee(s) who have submitted maximum timeoff requests: User-Defined Functions

Query:

```
declare function local:count-of-requests()
   as item()*
{
```

```
            for $emp in doc("emp.xml")/employees/emp
            let $req := doc("timeoff.xml")/timeoff/request[@empID_ = $emp/@id]
            return count($req)
};

for $emp in doc("emp.xml")/employees/emp
let $req := doc("timeoff.xml")/timeoff/request[@empID_ = $emp/@id]
where count($req) =  max(local:count-of-requests())
return data($emp/name)
```

Results:

```
Kelly Firkins
```

Description:
The above query illustrates writing and calling user-defined functions. The query begins with a user-defined function named count-of-requests(), which returns sequence of zero or more items - in this case sequence of integers representing count of requests submitted by each employee. This user-defined function is called from within the following FLWOR expression. The FLWOR expression reads like this: *for each employee, bind the corresponding timeoff requests to $req variable where count of requests matches the maximum of requests of submitted by each employee. For such entry, return the atomized data value for the name node*.

11: Requests Grouped By Employees: Grouping

Query:

```
for $emp in doc("emp.xml")/employees/emp
let $req := doc("timeoff.xml")/timeoff/request[@empID_ = $emp/@id]
return
            <requestsByEmp>
                        <emp name="{$emp/name}">
                        {
                                    $req
                        }
                        </emp>
            </requestsByEmp>
```

Results (abridged to save some space):

```
<?xml version="1.0" encoding="UTF-8"?>
<requestsByEmp>
  <emp name="Kelly Firkins">
    <request id="T3" empID_="1" startDate="2003-01-15"
             endDate="2003-01-31" hours="80" status="declined"
             absenceCode="vacation"/>
    <request id="T4" empID_="1" startDate="2003-02-01"
             endDate="2003-02-01" hours="8" status="approved"
             absenceCode="sick"/>
    <request id="T6" empID_="1" startDate="2003-03-01"
             endDate="2003-03-05" hours="30" status="approved"
             absenceCode="vacation"/>
  </emp>
</requestsByEmp>
```

Description:
This query illustrates one of the ways in which you can group the data. For each employee, it filters the timeoff requests ($req), and returns the new XML containing employee name and requests submitted by that employee.

12: Show all overlapping requests (timeoff requests where dates overlap with any other request)

Query:

```
<overlappingReqs>
{
            for $req1 in doc("timeoff.xml")/timeoff/request

            let $req2 := doc("timeoff.xml")/timeoff/request[@id != $req1/@id
                 and ((xs:date($req1/@startDate) >= xs:date(@startDate)
                 and xs:date($req1/@startDate) <= xs:date(@endDate))
                 or ((xs:date($req1/@endDate) >= xs:date(@startDate)
                 and xs:date($req1/@endDate) <= xs:date(@endDate))))]
            where count($req2) > 0
            return $req1
}
</overlappingReqs>
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<overlappingReqs>
  <request id="T1" empID_="2" startDate="2003-01-02"
             endDate="2003-01-10" hours="40" status="approved"
             absenceCode="vacation"/>
  <request id="T2" empID_="4" startDate="2003-01-10"
             endDate="2003-02-10" hours="180" status="approved
```

```
               " absenceCode="vacation"/>
   <request id="T3" empID_="1" startDate="2003-01-15"
               endDate="2003-01-31" hours="80" status="declined"
               absenceCode="vacation"/>
   <request id="T4" empID_="1" startDate="2003-02-01"
               endDate="2003-02-01" hours="8" status="approved"
               absenceCode="sick"/>
</overlappingReqs>
```

Description:
For each timeoff request, it binds req2 variable to remaining requests (@id != $req1/@id) where startDate or endDate is between the other startDate and endDate. The T2 request starts on Jan 10th and ends on Feb 10th, and other three request in the above result sequence overlap certain dates in that period.

Note how the query starts with the XML generation (<overlappingReqs> tag) and that the actual expression is enclosed inside curly braces, making request elements as child elements to the <overlappingReqs> generated element.

13: Show all employees who will not be in the office between 1/2/2003 and 1/10/2003

Query:

```
for $req in doc("timeoff.xml")/timeoff/request[
           @status= "approved" and xs:date(@startDate)
           >=
           xs:date("2003-01-02")  and
           xs:date(@endDate) < = xs:date("2003-01-10") ]

let $emp := doc("emp.xml")/employees/emp[@id = $req/@empID_]

return
           element onVacation
           {
                      attribute employee {$emp/name},
                      attribute from {$req/@startDate},
                      attribute to {$req/@endDate}
           }
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<onVacation employee="Dean Parker" from="2003-01-02" to="2003-01-10"/>
```

Description:
This example illustrates an alternate way of constructing nodes (see example 9 above for other method). Note the commas after each attribute, except the last attribute creation.

14: Show all employees who will not be coming to office today

Query:

```
for $req in doc("timeoff.xml")/timeoff/request[
           @status="approved" and
           xs:date(@startDate) <= current-date()
           and xs:date(@endDate) >= current-date() ]

let $emp := doc("emp.xml")/employees/emp[@id = $req/@empID_]

return
           element onVacationToday
           {
                      attribute employee {$emp/name}
           }
```

Results (assuming today is 1/10/2003):

```
<?xml version="1.0" encoding="UTF-8"?>
<onVacationToday employee="Katie Parise"/>
```

Description:
This example is quite similar to the previous example - except that it uses built in current-date() function for comparison.

15: Conditional Expressions

Query:

```
for $req in doc("timeoff.xml")/timeoff/request[@status="approved"]
return
           <request>
           {
                      $req/@id
           }
           {
```

```
                    (:  Check if the request starts before or on 15th :)

                    if(get-day-from-date(xs:date($req/@startDate)) <= 15) then
                              "TimeOff starts in first 15 days of month."
                    else "TimeOff starts in second half of the month."
          }
          </request>
```

Results (abridged to save some space):

```
<?xml version="1.0" encoding="UTF-8"?>
<request id="T1">TimeOff starts in first 15 days of month.</request>
<?xml version="1.0" encoding="UTF-8"?>
<request id="T2">TimeOff starts in first 15 days of month.</request>
```

Description:
This query selects all approved requests and constructs result XML which contains the request id and some text indicating if request starts in first or second half of the month. This is done using the conditional if..then..else expression available in XQuery.

The **(:** and **:)** symbols are used to happily include comments inside XQuery expressions. Note that these are not the XML comments, and will not be present in the output.

XQuery in Detail

The above examples hopefully gave you some idea on how XQuery expressions look like and what they can be used for. Let's now have a closer look at some of the core XQuery concepts and terminology.

XQuery Terms That You Should Be Familiar With

XQuery processes XML documents as a logical tree of nodes, such as elements, attributes, and so on. This logical structure of XML document on which XQuery (and XPath 2.0) is based is termed as **Data Model**. In addition to nodes, the data model may also consist of XSD simple types, such as strings, booleans, integers, dates, and so on. These XSD 'simple types' single values are termed as **Atomic Values**. A single node or an atomic value is known as an **Item**, and series of such items is known as **Sequence**.

XQuery expression evaluation always results in a sequence. In other words, the result of XQuery expression is zero or more nodes or atomic values. Single item is essentially a sequence with length one. These are termed as **Singleton Sequences**. XQuery does not permit nesting of sequences. That means, you can not have a sequence within a sequence. An empty sequence does not contain any item and is represented as **()**.

Some sequence examples:

- Sequence containing three atomic values: (10, "Darshan", 30)
- Sequence containing one node and one atomic value: (<Person />, 1)
- Sequence containing 21 atomic values (10 through 30), a node, and an atomic value: (10 to 30, <Person />, 20)
- Singleton Sequence: (205)

In some circumstances, it is required to extract the value of an item, this process is known as **Atomization**. To give you an example, $req/@hours returns a node item, but $req/@hours + 1 returns an atomic value item. In the later case, because an arithmetic operation is involved, the XQuery processor needs to get the value of $req/@hours node, this task is termed as Atomization.

Node can be one of seven pre-defined categories: document, element, attribute, text, namespace, processing instruction, or comment. As per the XQuery data model, each node in the tree has unique identity.

The other important term you should be familiar is **Document Order**. As mentioned earlier, XQuery data model is defined in terms of abstract, logical tree of nodes. The in-order, depth-first traversal of such tree results in nodes in Document Order. In general, XQuery expressions result in sequences based on document order. The only way to change this ordering is to use the order by clause inside a FLWOR expression. There are many XQuery concepts that are based on document order. For example, many axis steps (such as preceding, following, following-sibling, etc.) return results based on document order, the << and >> sequence operators (discussed later) are based on document order, and so on.

Input Sources

In the above 15 examples, you have already seen one of the ways XQuery input source is defined - via the **doc** function. The other input source function is the **collection** function. Note that all these built-in XQuery 1.0/XPath 2.0 functions are defined in http://www.w3.org/2003/11/xpath-functions namespace, which by default has the prefix of fn:. That means you could call doc() function as fn:doc(), collection function as fn:collection, without specifically declaring the namespace prefix. The other predefined namespaces prefixes include:

- **xs** (http://www.w3.org/2001/XMLSchema): XML Schema
- **xsi** (http://www.w3.org/2001/XMLSchema-instance): XML Schema Instance
- **xdt** (http://www.w3.org/2003/11/xpath-datatypes): XPath data types
- **local** (http://www.w3.org/2003/11/xquery-local-functions): Local functions
- **xml** (http://www.w3.org/XML/1998/namespace) - XML

The above six namespaces prefixes can be used in XQuery expressions, without an explicit namespace declaration.

Note that the earlier drafts of the XQuery specification had third option for providing query input source: the input() function. This function allowed binding to a sequence of nodes provided by implementation-specific source. This function no longer exists in the current draft. It has been removed,

and you should now use the doc() function with specific URI.

Expressions

As mentioned before, Expressions are the XQuery building blocks. Each query is an expression or nested expressions composed with full generality. XQuery specification defines following categorization of expressions:

Primary Expressions
Literals, variable references, context item expressions, constructors, and function calls form the primary expressions set defined by XQuery.
Literals: A direct representation of an atomic value; can be a numeric literal or a string literal.

- "Hello World" (: string literal :)
- "This is Bob's book"
- 'Hello World'
- 12500.00 (: numeric literal, a double value indicating 12500 :)
- 125E2 (: numeric literal, a double value indicating 12500 :)
- "Next Page >>" (: string literal containing predefined entity reference :)
- "€100" (: €100 string literal containing character reference :)

Variable References: A variable reference is a QName preceded by a dollar (**$**) sign. It is important to note that variables are not really assigned some value; they are actually bound to the expression. The variable binding can be created via FLWOR expressions (for and let clauses), quantified expressions, typeswitch expressions, and during function body execution where formal parameters are bound to values. Let's look at variable reference in a FLWOR expression:

```
for $var in (10, 20, 30, 40, 50)
return $var
```

In the above query, $var is a variable that is bound to sequence of numbers. This query returns 10 20 30 40 50.

Guess what would be the output of following expression (note that same variable $var is used with the for and let clauses):

```
for $var in (10, 20, 30, 40, 50)
let $var := ('x', 'y', 'z')
return $var
```

Context Item Expressions: The item (node or atomic value) being currently processed in a path expression is termed as context item. The Context Item Expression (.) returns the current item being processed.

Constructors: XQuery 1.0 and XPath 2.0 Functions and Operators specification defines various constructor functions, such as xs:integer(), xs:date(), xdt:dayTimeDuration(), which allow constructing values of atomic types.

Function Calls: Built-in functions and user-defined functions can be called in XQuery expressions, by writing function QName followed by parenthesized list of zero or more expressions (arguments).

XQuery Comments: Informative annotations can be made part of query using the (: and :) delimiters. Unlike as in XML, XQuery comments can be nested.

```
(: This is an example of (: nested :) comments :)
```

Path Expressions
This is where XPath and XQuery meet. Like with XPath, the Path Expressions are used to locate nodes within trees. The path expression (example: /article/para[5]/text()) may consists of steps (separated by / or //), filter steps, axis steps, predicates, node test, and name test.

See examples 1 through 4 in above 15 Simple XQuery Examples section to see how Path Expressions look like.

One important thing to note here is that XQuery implementation is not required to support all the axes. XQuery makes ancestor, ancestor-or-self, following, following-sibling, preceding, preceding-sibling axes optional, and rules out namespace axis defined by XPath. XQuery supports both, abbreviated and unabbreviated syntax.

Sequence Expressions
As defined earlier, Sequence is an ordered collection of zero or more items; item being a node or an atomic value.

You can use **comma operator** (,) to separate items to construct a sequence. Examples:

```
("Darshan", "Singh", "PerfectXML", "YukonXML")      (: sequence containing 4 atomic values :)
(<agent id="1" score="94"/>, <agent id="2" score="97"/>)    (: sequence containing 2 nodes :)
```

You can use the Range Expression to construct a sequence of consecutive integers.

```
(20 to 100)           (: integers 20 through 100 :)
```

Guess what would be output of following sequence expressions:

```
(98 to 90)
("a" to "z")
```

XQuery provides three operators for combining nodes: union (or |), intersect, and except. The union operator takes two node sequences and returns a sequence containing all the nodes that occur in either of the operands. The intersect operator on the other hand returns only the nodes that are present in both the node sequences. The except operator returns sequence of nodes that occur in the first operand but not in the second operand.

Arithmetic, Comparison and Logical Expressions

XQuery provides arithmetic operators for addition (+), subtraction (-), multiplication (*), division (div or idiv), and modulus (mod), in their usual binary and unary forms.

XQuery clubs the comparison operators into three categories:

- Use the **Value** Comparison operators (eq, ne, lt, le, gt, ge) if you need to compare a single value.
- Use the **General** Comparison operators (=, !=, <, <=, >, >=) to compare operand sequences of any length.
- Use the **Node** Comparison operators (is, <<, >>) to compare two nodes by their identity or their document order to see if two nodes have exact same identity, or a node occurs before/after the other node.

XQuery provides two logical operators - and & or. You can use fn:**not**() function to negate the Boolean value. A logical expression always results in either true or false.

One important term that you should know about is **Effective Boolean Value** or EBV. It signifies that when implicitly or explicitly fn:boolean() function is used on an empty sequence, or on a zero-length xs:string or xdt:untypedAtomic type, or on a numeric zero value, or a NaN xs:double or xs:float type value, it will result in false. This does not apply if you cast to xs:boolean type.

Consider the sequence query:

```
fn:boolean(()),
fn:boolean(""),
fn:boolean(0),
fn:boolean(xs:double("NaN")),
fn:boolean(0),
1 and 0,
"" or fn:boolean(())
(: ,("") cast as xs:boolean :)
```

The above query returns seven times false. The first five items explicitly use fn:boolean(), whereas sixth and seventh item show an example of implicitly use of fn:boolean() function.

If you un-comment the cast statement in above query, it would produce the *"cannot be cast to a boolean"* error.

Constructors

In addition to querying XML data, XQuery allows create XML nodes. There are two ways in which you can generate XML using XQuery:

- **Direct**: Using this approach, you write XML directly, and use curly braces { } to enclose expressions that need to be evaluated. See examples 9 and 15 above.
- **Computed**: With this approach, you use keywords such as, element, attribute, document, text, processing-instruction, comment, or namespace to create the respective node. See examples 13 and 14 above.

You can write **declare xmlspace** in the prolog section (discussed later) in the query to indicate if you wish to preserve or strip the whitespace while using the Constructors.

FLWOR Expressions

FLWOR (for-let-where-order by-return), pronounced "**flower**", is the core XQuery expression that allows looping, variable binding, sorting, filtering and returning the results.

The for and let clauses allow binding variables to expressions. The for clause evaluates the bound expression and iterates for each item in the evaluated sequence (except those which were filtered out because of where clause). The let clause on the other hand binds a variable to the entire result of an expression at once (without iteration).

Consider following example:

```
let $var1 := (1, 2, 3)
return <ints>{$var1}</ints>
```

The above query will result in

```
<ints>1 2 3</ints>
```

Just change the let cause to for as shown below:

```
for $var1 in (1, 2, 3)
```

```
return <ints>{$var1}</ints>
```

And the output would now be:

```
<ints>1</ints>
<ints>2</ints>
<ints>3</ints>
```

The for and let clauses can contain one or more variables, each with an associated expression.

```
for $var1 in (1, 2, 3), $var2 in (10, 20, 30)
return <ints>{$var1, $var2}</ints>
```

The above query produces the following output:

```
<ints>1 10</ints>
<ints>1 20</ints>
<ints>1 30</ints>
<ints>2 10</ints>
<ints>2 20</ints>
<ints>2 30</ints>
<ints>3 10</ints>
<ints>3 20</ints>
<ints>3 30</ints>
```

Like nested loops, the entire second for clause set is evaluated for each item in the first for clause (Cartesian cross-product).

The for clause supports the notion of Positional Variable using the at clause. Positional variable is always of type xs:integer and as the bound variable iterates over items in the sequence, the value of the positional variable is incremented by 1 (like a counter), it starts with 1.

Try out the following example:

```
for $var1 at $count1 in (1, 2, 3),
    $var2 at $count2 in (10, 20, 30)
return <ints c1="{$count1}" c2="{$count2}">{$var1, $var2}</ints>
```

See examples 7 through 15 above for FLWOR expression samples.

Unordered Expressions

If the order of results is not important, you can specifically call **fn:unordered** function, and possibly get some performance benefits and the XQuery implementation might do some optimizations in such cases.

Try out following example:

```
fn:unordered
(
<MoreThanTwo>
{
           for $emp in doc("emp.xml")/employees/emp
           let $req := doc("timeoff.xml")/timeoff/request[@empID_ = $emp/@id]
           where count($req) > 2
           return
                     $emp
}
</MoreThanTwo>
)
```

The above FLWOR query returns following output:

```
<?xml version="1.0" encoding="UTF-8"?>
<MoreThanTwo>
   <emp id="1">
      <name>Kelly Firkins</name>
      <email>kelly</email>
      <ssn>111-11-1111</ssn>
      <managedEmail>dean</managedEmail>
      <dateOfJoining>1994-11-16</dateOfJoining>
   </emp>
</MoreThanTwo>
```

Conditional Expressions

You can use if, then, else keywords in your query and have expressions evaluated conditionally. The final example in the above 15 examples illustrates the conditional expressions.

Note that the "else" expression is mandatory in the if..then..else expression. If you do not really want to return anything in the else block, just return a pair of parentheses as an empty sequence.

Quantified Expressions

If you need to find out if any (existential quantification) or every (universal quantification) item in a sequence satisfies a condition, you can use Quantified Expressions, **some** or **every**, in such scenarios.

Consider the following sample:

```
<Report>
{
            for $t in doc("timeoff.xml")/timeoff
            where some $req in $t/request
                        satisfies ($req/@hours <= 4)
            return
                        "There are some half-day requests."
}

{
            for $t in doc("timeoff.xml")/timeoff
            where every $req in $t/request
                        satisfies ($req/@hours >= 8)
            return
                        "All requests are atleast for a day."
}
</Report>
```

Note that the above results can be obtained by many other ways in which query can be written; the above is just an example to illustrate quantified expressions.

Expressions on SequenceTypes

Let's first understand what SequenceType really means. As mentioned before, XQuery is a strongly typed language. It supports some built-in types (element node, attribute node, etc.), XSD basic types (xs:integer, xs:string, etc.), and types from imported XSD schema (how to import XSD is discussed later). Values in XQuery are generally sequences of such types. This is given a generic name SquenceType. In other words, any value in XQuery can be said to be of sequence type.

Following tables lists some of the examples of sequence type indicators:

element()+	One or more elements
attribute()*	Zero or more attributes
document-node()?	Zero or one document node
node()	Any node
text()	Any text node
processing-instruction()	Any processing instruction node
comment()	Any comment node
empty()	An empty sequence
item()	Any node or atomic value
xs:string?	Zero or one string
xs:anySimpleType	Any simple type
element(startDate, timeOffDates)	An element named startDate of type timeOffDates (defined in imported schema)
attribute(@*, requestTypes)	Any attribute of type requestTypes (defined in imported schema)
processing-instruction(xml-stylesheet)	Matches xml-stylesheet processing instruction

XQuery allows you to check if a value is of some specified sequence type, or do various things based on the type of the value, or cast the value from one type to the other. These facilities are provided via **instance of** boolean operator, **typeswitch**, **cast**, **castable**, and **treat** expressions.

Refer to XQuery specification SequenceType Syntax and Expressions on Sequence Types sections for complete details on this.

Validate Expressions

XQuery allows importing XSD schema and validating elements (using the validate expression) against that schema. Refer to XQuery specification for more details on this. As per the SAXON 7.8 XQuery Conformance notes, it does not support the validate expression.

Functions and Operators

The XQuery 1.0 and XPath 2.0 Functions and Operators document is a result of joint efforts of XSL and XQuery working groups. This document is a catalog of built-in functions and operators that are available via XPath 2.0, XQuery 1.0, and XSLT 2.0. For a complete listing of all these functions and operators, refer to the above link. In this article, I am listing few functions that I think will be used commonly. The description for the functions is taken mostly from the W3C Specification document.

Accessors	
fn:**node-name**($arg as node()?) as xs:QName?	Returns an expanded-QName for node kinds that can have names
fn:**string**() as xs:string fn:**string**($arg as item()?) as xs:string	Returns the value of context item or $arg represented as a xs:string
fn:**data**($arg as item()*) as xdt:anyAtomicType*	Returns a sequence of atomic values

Errors and Debugging	
fn:**error**() as none fn:**error**($arg as item()?) as none	Raise an error.
fn:**trace**($value as item()*, $label as xs:string) as item()*	To be used in debugging queries by providing a trace of their execution

Functions on Numeric Values and Aggregate Functions	
fn:**abs**($arg as numeric?) as numeric? fn:**ceiling**($arg as numeric?) as numeric? fn:**floor**($arg as numeric?) as numeric? fn:**round**($arg as numeric?) as numeric? fn:**round-half-to-even**($arg as numeric?) as numeric? fn:**round-half-to-even**($arg as numeric?, $precision as xs:integer) as numeric?	Get absolute, ceiling, floor, round values. The round-half-to-even function returns the nearest (that is, numerically closest) numeric to $arg that is a multiple of ten to the power of minus $precision.
fn:**count**($arg as item()*) as xs:integer fn:**avg**($arg as xdt:anyAtomicType*) as xdt:anyAtomicType? fn:**max**($arg as xdt:anyAtomicType*) as xdt:anyAtomicType? fn:**max**($arg as xdt:anyAtomicType*, $collation as string) as xdt:anyAtomicType? fn:**min**($arg as xdt:anyAtomicType*) as xdt:anyAtomicType? fn:**min**($arg as xdt:anyAtomicType*, $collation as string) as xdt:anyAtomicType? fn:**sum**($arg as xdt:anyAtomicType*) as xdt:anyAtomicType fn:**sum**($arg as xdt:anyAtomicType*, $zero as xdt:anyAtomicType?) as xdt:anyAtomicType?	Various aggregate functions.

Functions on Strings	
fn:**codepoints-to-string**($arg as xs:integer*) as xs:string fn:**string-to-codepoints**($arg as xs:string?) as xs:integer*	Code Point also known as Code Position refers to any value in the Unicode codespace; that is, the range of integers from 0 to 10FFFF. These functions allow conversion between string and sequence of code points. For example, fn:**string-to-codepoints**("DARSHAN") will return (68, 65, 82, 83, 72, 65, 78)
fn:**compare**($comparand1 as xs:string?, $comparand2 as xs:string?) as xs:integer? fn:**compare**($comparand1 as xs:string?, $comparand2 as xs:string?, $collation as xs:string) as xs:integer?	Returns -1, 0, or 1, depending on whether the value of the $comparand1 is respectively less than, equal to, or greater than the value of $comparand2, according to the rules of the collation that is used.
fn:**concat**($arg1 as xs:string?, $arg2 as xs:string?, ...) as xs:string	Concatenates two or more xs:strings
fn:**string-join**($arg1 as xs:string*, $arg2 as xs:string) as xs:string	Returns the xs:string produced by concatenating a sequence of xs:strings using an optional separator.
fn:**substring**($sourceString as xs:string?, $startingLoc as xs:double) as xs:string fn:**substring**($sourceString as xs:string?, $startingLoc as xs:double, $length as xs:double) as xs:string	Returns the xs:string located at a specified place in an xs:string.
fn:**string-length**() as xs:integer fn:**string-length**($arg as xs:string) as xs:integer	Returns the length of the argument.
fn:**normalize-space**() as xs:string fn:**normalize-space**($arg as xs:string?) as xs:string	Returns the whitespace-normalized value of the argument.
fn:**normalize-unicode**($arg as xs:string?) as xs:string fn:**normalize-unicode**($arg as xs:string?, $normalizationForm as xs:string) as xs:string	Returns the normalized value of the first argument in the normalization form specified by the second argument.
fn:**upper-case**($arg as xs:string?) as xs:string fn:**lower-case**($arg as xs:string?) as xs:string	Returns the upper/lower-cased value of the argument.
fn:**translate**($arg as xs:string?, $mapString as xs:string, $transString as xs:string) as xs:string	Returns the first xs:string argument with occurrences of characters contained in the second argument replaced by the character at the corresponding position in the third argument.
fn:**escape-uri**($uri-part as xs:string?, $escape-reserved as xs:boolean) as xs:string	Returns the string representing an xs:anyURI value with certain characters escaped as specified in RFC 2396 and RFC 2732.
fn:**contains**($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean fn:**contains**($arg1 as xs:string?, $arg2 as xs:string?, $collation as	Indicates whether one xs:string contains another xs:string. A collation may be specified.

xs:string) as xs:boolean	
fn:**starts-with**($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean fn:**starts-with**($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:boolean	Indicates whether the value of one xs:string begins with the collation units of another xs:string. A collation may be specified.
fn:**ends-with**($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean fn:**ends-with**($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:boolean	Indicates whether the value of one xs:string ends with the collation units of another xs:string. A collation may be specified.
fn:**substring-before**($arg1 as xs:string?, $arg2 as xs:string?) as xs:string fn:**substring-before**($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:string	Returns the collation units of one xs:string that precede in that xs:string the collation units of another xs:string. A collation may be specified.
fn:**substring-after**($arg1 as xs:string?, $arg2 as xs:string?) as xs:string fn:**substring-after**($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:string	Returns the collation units of xs:string that follow in that xs:string the collation units of another xs:string. A collation may be specified.

Regular Expressions Functions for Pattern Matching

fn:**matches**($input as xs:string?, $pattern as xs:string) as xs:boolean fn:**matches**($input as xs:string?, $pattern as xs:string, $flags as xs:string) as xs:boolean	Returns an xs:boolean value that indicates whether the value of the first argument is matched by the regular expression that is the value of the second argument.
fn:**replace**($input as xs:string?, $pattern as xs:string, $replacement as xs:string) as xs:string fn:**replace**($input as xs:string?, $pattern as xs:string, $replacement as xs:string, $flags as xs:string) as xs:string	Returns the value of the first argument with every substring matched by the regular expression that is the value of the second argument replaced by the replacement string that is the value of the third argument.
fn:**tokenize**($input as xs:string?, $pattern as xs:string) as xs:string+ fn:**tokenize**($input as xs:string?, $pattern as xs:string, $flags as xs:string) as xs:string+	Returns a sequence of one or more xs:strings whose values are substrings of the value of the first argument separated by substrings that match the regular expression that is the value of the second argument.

Functions on Dates

fn:**adjust-dateTime-to-timezone**($arg as xs:dateTime?) as xs:dateTime? fn:**adjust-dateTime-to-timezone**($arg as xs:dateTime?, $timezone as xdt:dayTimeDuration?) as xs:dateTime?	Adjusts an xs:dateTime value to a specific timezone, or to no timezone at all. If $timezone is the empty sequence, returns an xs:dateTime without a timezone. Otherwise, returns an xs:dateTime with a timezone.
fn:**get-year-from-date**($arg as xs:date?) as xs:integer? fn:**get-month-from-date**($arg as xs:date?) as xs:integer? fn:**get-day-from-date**($arg as xs:date?) as xs:integer?	Returns an xs:integer representing the year, month, or day in the localized value of $arg.

Functions on Nodes

fn:**name**() as xs:string fn:**name**($arg as node()?) as xs:string	Returns the name of a node, as an xs:string that is either the zero-length string, or has the lexical form of an xs:QName.
fn:**local-name**() as xs:string fn:**local-name**($arg as node()?) as xs:string	Returns the local part of the name of $arg as an xs:string that will either be the zero-length string or will have the lexical form of an xs:NCName.
fn:**namespace-uri**() as xs:string fn:**namespace-uri**($arg as node()?) as xs:string	Returns the namespace URI of the QName of $arg as a xs:string.
fn:**root**() as node() fn:**root**($arg as node()?) as node()?	Returns the root of the tree to which $arg belongs. This will usually, but not necessarily, be a document node.

Function on Sequences

fn:**zero-or-one**($arg as item()*) as item()? fn:**one-or-more**($arg as item()*) as item()+ fn:**exactly-one**($arg as item()*) as item()	Returns the input sequence if it contains zero or one, one or more items, or exactly one item. Raises an error otherwise.
fn:**index-of**($seqParam as xdt:anyAtomicType*, $srchParam as xdt:anyAtomicType) as xs:integer* fn:**index-of**($seqParam as xdt:anyAtomicType*, $srchParam as xdt:anyAtomicType, $collation as xs:string) as xs:integer*	Returns a sequence of xs:integers, each of which is the index of a member of the sequence specified as the first argument that is equal to the atomic value that is the value of the second argument.
fn:**empty($arg as item()*) as xs:boolean**	Indicates whether or not the provided sequence is empty.
fn:**exists**($arg as item()*) as xs:boolean	Indicates whether or not the provided sequence is not empty.
fn:**distinct-values**($arg as xdt:anyAtomicType*) as xdt:anyAtomicType* fn:**distinct-values**($arg as xdt:anyAtomicType*, $collation as xs:string) as xdt:anyAtomicType*	Returns a sequence in which all but one of a set of duplicate values, based on value equality, have been deleted. The order in which

	the distinct values are returned is implementation dependent.
`fn:`**`insert-before`**`( $target as item()*, $position as xs:integer, $inserts as item()*) as item()*`	Inserts an item or sequence of items at a specified position in a sequence.
`fn:`**`remove`**`($target as item()*, $position as xs:integer) as item()*`	Removes an item from a specified position in a sequence.
`fn:`**`reverse`**`($arg as item()*) as item()*`	Reverses the order of items in a sequence.
`fn:`**`unordered`**`($sourceSeq as item()*) as item()*`	Indicates that the given sequence may be returned in any order.
`fn:`**`deep-equal`**`($parameter1 as item()*, $parameter2 as item()*) as xs:boolean` `fn:`**`deep-equal`**`( $parameter1 as item()*, $parameter2 as item()*, $collation as string) as xs:boolean`	Returns true if the two arguments have items that compare equal in corresponding positions

Context Functions

`fn:`**`position`**`() as xs:integer`	Returns an xs:integer indicating the position of the context item within the sequence of items currently being processed.
`fn:`**`last`**`() as xs:integer`	Returns an xs:integer indicating the number of items in the sequence of items currently being processed.
`fn:`**`current-dateTime`**`() as xs:dateTime`	Returns the xs:dateTime (with timezone) that is current at some time during the evaluation of a query or transformation in which fn:**current-dateTime**() is executed.
`fn:`**`current-date`**`() as xs:date`	Returns the xs:date (with timezone) that is current at some time during the evaluation of a query or transformation in which fn:**current-date**() is executed.
`fn:`**`current-time`**`() as xs:time`	Returns the xs:time (with timezone) that is current at some time during the evaluation of a query or transformation in which fn:**current-time**() is executed.
`fn:`**`default-collation`**`() as xs:string`	Returns the value of the default collation property from the static context.

Functions to Generate Sequences

`fn:`**`doc`**`($uri as xs:string?) as document?`	Retrieves a document using an xs:anyURI supplied as an xs:string.
`fn:`**`collection`**`($arg as xs:string?) as node()*`	Takes a xs:string as argument and returns a sequence of nodes obtained by interpreting $arg as an xs:anyURI and resolving it.

User-Defined and External Functions

To create easy-to-use re-usable queries or to encapsulate complex queries, you can create you own functions. You can declare your own function by providing the name of the function, parameters, and the data type of the result. These functions are termed as **user-defined functions**.

In addition to this, if you want to do something outside the query environment, and if the XQuery implementation you are using supports it, there is a concept of **external functions**.

The function declaration tells if the function is local user-defined or is an external function. XQuery allows recursive (function calling itself) and mutually recursive (two functions calling each other) functions. However, the current version does not allow function overloading for user-defined functions.

See example 10 above for an example of writing and calling a user-defined function.

XQuery allows putting functions in library modules, so that they can be shared and imported by any query. XQuery defines module as a "fragment of XQuery that can independently undergo the analysis phase". So far we have seen just the query body in the XQuery examples, there is more to it, and we'll study that in the next section.

Structure of a Query: Modules and Prologs

In real life, an XQuery query will be quite complex than what we have seen in above 15 examples. It would be a good design to modularize to re-use and maintenance. An XQuery then would be composed of one **main module** and zero or more **library modules**.

All the above examples just contained the query body. In addition to this, an XQuery module can include things like **module declaration** and **prolog**

If a module includes prolog followed by query body, it is said to be main module. A query can have exactly one main module.

If a module includes module declaration followed by a prolog, it is called a library module. A library module can not be executed independently. It is created to provide services (in the form of functions and variable declarations) to the module that is including it. It is not allowed to have a module containing both, a module declaration and a query body.

I have used words and phrases such as prolog and module declaration without defining them. Let's look at what I mean by these terms.

Prolog Prolog is nothing but a series of declarations and imports that create the environment for query processing. Each declaration or import is followed by a semicolon.

```
Prolog    ::=
  ((NamespaceDecl
    XMLSpaceDecl
    DefaultNamespaceDecl
    DefaultCollationDecl
    BaseURIDecl
    SchemaImport
    ModuleImport
    VarDecl
    ValidationDecl
    FunctionDecl) Separator)*

    Separator    ::=    ";"
```

The things that can go as part of prolog include:

- **Namespace declaration**: Defines the namespace prefix to namespace URI association.
- **Xmlspace declaration**: Controls the whitespace handling
- **Default Namespace declaration**: Defines namespace for unprefixed elements or functions.
- **Default Collation declaration**: Defines name of the collation to be used by functions (such as fn:compare) and operators (such as gt).
- **Base URI declaration**: Defines base URI property used for resolving relative URIs within a module.
- **Schema Import**: Imports the element, attribute, and type definitions from a named schema.
- **Module Import**: Imports the function declarations and variable declarations from the Prolog of a library module.
- **Variable declaration**: Declares a typed variable, and optionally assigns a value to it.
- **Validation declaration**: Establishes a default validation mode for the query.
- **Function Declaration**: Declare user-defined or external functions.

Module Declaration

Library modules begin by a Module declaration. Followed by the module declaration, a library module consists of Prolog. The module declaration establishes an identity for the module, and it consists of the words module and namespace, followed by namespace prefix and a URI as a string. Library modules may import other library modules using the Module Import in its prolog section.

Refer to XQuery W3C document and/or resources at the bottom of this article for more information on prolog, and creating and using library modules.

Comparing XQuery 1.0 with XSLT 1.0

Before I conclude this article, I would like to list some bullet points that compare and contrast XQuery with XSLT.

- XSLT is primarily a language for describing XML transformation; XQuery is primarily a language to query XML data and documents.
- XSLT uses XML-based syntax; XQuery 1.0 does not use XML-based syntax.
- XPath is at the core for both, XSLT and XQuery.
- XSLT 1.0 turned W3C recommendation on November 16, 1999. XQuery 1.0 (as of Jan 28, 2004) is in Last Call Working Draft status. Many tools, APIs, and vendors have excellent support for XSLT. XQuery support is introduced by many vendors/toolkits; it is been rapidly improved and made complete.
- XSLT and XQuery, both, provide means to construct new nodes.
- XSLT being a rule-based language scores higher when it comes to handling unpredictable structures.
- XQuery 1.0 has a concept of user-defined functions, which can be modeled in XSLT 1.0 as named templates. XSLT 2.0 matches the XQuery 1.0 functionality by allowing writing and calling user-defined functions.
- XQuery 1.0 is strongly typed language, XSLT 1.0 is not.
- XQuery provides FLWOR expression for looping, sorting, filtering; XSLT 1.0's xsl:for-each instruction (and XSLT 2.0's for expression) allows to do the same.
- XQuery does not support all the XPath axes; XSLT does.

Summary

If your data is natively stored as XML or you are working with virtual XML views or XML documents, and if you need to query them, you can use XQuery as the flexible and standard solution and write XQuery expressions that loop, filter, sort, generate new nodes, and so on, to produce the results that you want. Hopefully, this article gave you some idea of as to what XQuery is, how the XQuery expressions look like, what various XQuery-related terms mean, and how it fits with other XML standards such as XPath and XSLT.

XQuery Implementations, API, and Tools

Following is a list of some XQuery tools/products/implementations that you might find interesting:

- SAXON (http://saxon.sourceforge.net/)
- X-Hive/DB (http://www.x-hive.com/)
- XQuantum™ XML Database Server (http://www.cogneticsystems.com/)

- BumbleBee (http://www.xquery.com/bumblebee)
- XQEngine (http://xqengine.sourceforge.net/)
- XStreamDB™ (http://www.bluestream.com/)
- JSR-000225 XQuery API for JavaTM (XQJ) (http://jcp.org/aboutJava/communityprocess/edr/jsr225/index.html)
- DataDirect XQuery™ (http://www.datadirect.com/techzone/xml/topics/xquerysupport/index.ssp

Links to Other XQuery Resources on the Web

- XQuery Tutorial
- Mark Logic xq:zone: where xquery gets to work
- http://www.xquery.com/
- XQuery: A Guided Tour - Chapter 1 from the book XQuery from the Experts
- Liquid Data: XQuery-Based Enterprise Information Integration

Next Steps

If you are interested in learning more about XQuery and/or using it in your projects, I would recommend the book XQuery from the Experts. Write some more queries and try out using SAXON. See how XQuery is implemented in Yukon. Go through XQuery help sections in Yukon Books Online.

About the author

Darshan Singh is the Managing Editor at PerfectXML.com - the XML community Web site. He has now taken up the challenge to create one of the premier Web sites on SQL Server "Yukon". Darshan can be reached at darshan@YukonXML.com.