An Early Look at XQuery

Andrew Eisenberg
IBM, Westford, MA 01886
andrew.eisenberg@us.ibm.com

Jim Melton
Oracle Corp., Sandy, UT 84093
jim.melton@acm.org

Introduction

XQuery is a query language for real and virtual XML documents and collections of these documents. Its development began in the second half of 1999. With roughly 3 years of work completed, it's high time that we provided an initial description of this language, and a sense of where it is in its development cycle.

XQuery is being developed within W3C. Every consortium of this type has its own rules and its own ways of getting its work done. W3C provides visibility to the public by making available drafts of the specifications that it has under development at relatively frequent intervals. Mailing lists are established for each specification to allow the public to provide feedback on these drafts. Unfortunately, the W3C process does not allow us to publicly discuss the internal workings of the XML Query WG, including schedules, proposals that are being considered, and discussions that have taken place.

Even so, with the amount of material that is contained in the most recent public drafts of these specifications, we have more than enough to discuss.

W3C Process

In Dec. 1998, W3C held QL'98, the W3C Query Languages Workshop. Interest in this area was great enough that the XML Query WG was formed in August 1999. The group published XML Query Requirements in Jan. of 2000, and has updated it several times since then (most recently in Feb. 2001 [1]).

Within the W3C Process, specifications progress through several stages of maturity:

- Working Draft (WD)
- Last Call Working Draft
- Candidate Recommendation (CR)
- Proposed Recommendation (PR)
- Recommendation (REC)

The XQuery specifications that we will be discussing are all Working Drafts. As we said, we are not able to comment on when XQuery will move forward to these more advanced stages.

At the time that a specification reaches CR, a call for implementations goes out. In order to become a PR, the group must be able to demonstrate that each feature has been implemented, preferably by two interoperable implementations.

The Set of XQuery Documents

The XML Query WG has produced the following documents:

- XML Query Requirements [1]
- XQuery Use Cases [4]
- XQuery 1.0: An XML Query Language [2]
- XML Syntax for XQuery 1.0 (XQueryX) [8]

The XML Query WG has worked jointly with the XSL WG and produced the following documents:

- XQuery 1.0 and XPath 2.0 Data Model [5]
- XQuery 1.0 and XPath 2.0 Formal Semantics [6]
- XQuery 1.0 and XPath 2.0 Functions and Operators 1.0 [7]

Taken as a whole, these specifications define XQuery and XQueryX.

We've already discussed the Requirements document, the first document that the XML Query WG produced.

The Use Cases document provides a number of specific usage scenarios for XQuery. Each use case is focused on a specific application area, and contains a DTD or XML Schema, example input data, and a number of queries. Each query is presented with a prose description, an XQuery expression, and the expected output from the query.

The Data Model specification defines the data model on which queries will operate and return as a result. This data model is an extension of the XML Infoset and the Post-Schema-Validation Infoset (PSVI).

The XQuery Requirements document calls for both a human-readable query syntax and an XML query syntax. The XQuery specification defines the human-readable syntax and provides a high level description of its semantics. It contains numerous examples of this language. The XQueryX specification defines the XML syntax for a query.

The Formal Semantics specification defines the semantics of XQuery in much greater detail than is contained in the XQuery specification.

XQuery and XPath

XPath 2.0 has as one of its requirements the manipulation of XML Schema-based content. XPath 1.0, you'll remember, created its own model consisting of nodes, strings, numbers, and boolean values. The XML Query WG and the XSL WG have been working together so that XPath 2.0 can use the same Data Model, Formal Semantics, and Functions and Operators specifications as XQuery.

XQuery is largely a superset of XPath 2.0. However, XQuery does not support all of the XPath 2.0 axes:

Supported by XQuery and XPath	Supported by XPath only
child	ancestor
descendent-or-self (//)	following-sibling
parent (..)	preceding-sibling
attribute (@)	following
self	preceding
descendent	namespace
	ancestor-or-self

XQuery is not defined by referring to XPath 2.0; the two specifications each stand alone. The XQuery specification contains the syntax and semantics of the parts of XPath that it supports. This has required that the editors of the two specifications work together very closely. In order to ensure consistency, they maintain a common source from which the two documents are generated.

With these preliminaries out of the way, the rest of this article will discuss the Data Model, XQuery, XQueryX, and Functions and Operators in greater depth. The Formal Semantics specification is discussed only briefly.

The XQuery 1.0 and XPath 2.0 Data Model

The XQuery/XPath Data Model is an abstract data model. It is based on both the Infoset and PSVI, extended to cover sequences of both nodes and atomic values.

The documents that XQuery can operate on, Schema-validated documents, DTD-validated documents, and simply well-formed documents, all produce instances of this data model. Collections of documents produce instances of the data model as well (sequences of document nodes).

XQuery is a functional language, with each expression operating on instances of the data model and producing an instance of the data model. As you would expect, XQuery is closed with respect to its data model.

A data model instance consists of a sequence of zero or more items. An item is either a node or an atomic value, but not another sequence. An item is indistinguishable from a sequence containing just that one item. A node is one of several types of nodes; document, element, attribute, text, namespace, processing instruction, and comment.

Elements nodes, attribute nodes, and atomic values are all labeled with their type name, expressed as an expanded QName (target namespace URI and local name). A type name identifies either an XML Schema built-in type or a named type in an XML Schema. XML Schema anonymous types are identified by either `xs:anyType` or `xs:anySimpleType`.

An atomic value is either a value in the value space of an XML Schema atomic type that has been labeled with its atomic type, or it is a string labeled with `xs:anySimpleType`. An atomic type can be an XML Schema primitive type, or a type that has been derived from a primitive type only by restriction. XML Schema defines 19 primitive types, such as `xs:string` and `xs:decimal`, and built-in derived types such as `xs:integer` and `xs:NMTOKEN`. `xs:anySimpleType` is used to represent the value that is the typed value of a node whose type is unknown. The atomic values taken from a well-formed or DTD-valid XML document will have a type of `xs:anySimpleType`.

Nodes have identity, with the identity of a node established at the time that the node is created. Document order is defined on all of the nodes of a document. The document node is first, with element nodes, comment nodes, and processing instruction nodes ordered based on their location in the XML document. Element nodes are ordered prior to their children nodes. The namespace nodes of an element immediately follow the element node, and the attribute nodes of an element immediately following the namespace nodes of that element. The relative order of these namespace nodes and attribute nodes is implementation-dependent. The order of multiple documents is implementation-dependent and stable.

XQuery

XQuery, as we've just said, operates on instances of its data model and produces instances of its data model. The documents that are used to create the data model instance may be physical or virtual documents. A virtual document might be created by an application, or it might be derived from a database.

XQuery defines the transformation from a data model instance to an infoset, but it does not define how to transform it all the way back to a serial, character-oriented, representation.

XQuery is a functional language. Expressions do not have side-effects and can be freely composed. XQuery is also a language with a strong notion of typing.

Query Processing

An XQuery expression is processed in two phases; the analysis phase and the evaluation phase.

The analysis phase depends on the static context and the query itself. The static context contains scoped namespaces (and default namespaces), schema definitions, variable definitions, functions, and collations.

During the analysis phase, each expression is given a static type. If an operand to an expression contains an inappropriate static type, then a static type error is raised.

The evaluation phase depends upon the evaluation context. This contains the focus, dynamic variables, current date and time, and the input sequence. The focus contains:

context item	item currently being processed
context position	position of the context item in the sequence of items being processed
context size	number of items in the sequence being processed

Focuses can be nested. This occurs in an expression such as `$d/employee/phone`. The outermost focus is the sequence of items in `$d`. For each of these items an inner focus of `employee` elements is created.

The input sequence is a sequence of nodes that can be accessed by the `xf:input` function. The value of the input sequence is provided in some implementation-dependent way.

In addition to the `input` function, an XQuery can use the `xf:document` and `xf:collection` functions. These functions accept a URI and produce document nodes and sequences of document nodes (and possibly other node types).

```
xf:document('employees.xml')
   //employee[@id='444378']

→
<employee id='444378'>
   ...
</employee>
```

The result of every expression has a dynamic type, which may be the same as the static type of the expression, or it may be a more specific type.

Element, attribute, and text nodes have both a string value and a typed value that are returned by the `xf:string` and `xf:data` functions, respectively. The string value of an element is the concatenated values of each of the text nodes contained in the element in document order. The typed value of a node is a sequence of atomic values. The typed value of an element of type `xs:anyType` is the same as its string value. If the element is of a type that allows complex content, then the `xf:data` function raises an error.

input:

```
   <employee id='661008'>
      <name>Albert Jones</name>
      <position>Accountant</position>
      <HSYears>4</HSYears>
      <Veteran status='true'/>
   </employee>

string(input()//employee[@id='661008'])

→
'Albert JonesAccountant4' (xs:anySimpleType)

data(input()
     //employee[@id='661008']/HSYears
    )

→
4 (xs:integer)
```

In this example we're assuming that the `HSYears` element was defined to be of type `xs:integer`, which allowed data to return the integer value of 4. If this element had been obtained from a document without an XML Schema, then data would have returned the value '4' (a character string value) with a type of `xs:anySimpleType`.

XQuery Prolog and Body

An XQuery consists of a prolog section, followed by body section that consists of a sequence expression. Either of these sections can be left out.

The prolog allows namespace declarations, a declaration of whitespace handling, and a declaration of the default collation, and it allows XML Schemas to be imported. Following these declarations, it can contain functions definitions. The following is an example of an XQuery prolog:

```
declare xmlspace = preserve

import schema
   namespace
   myco="http://www.myco.com/personnel"
   at "http://www.myco.com/personnel.xsd"

declare namespace
   xhtml = "http://www.w3.org/1999/xhtml"

define function compensation
                (element myco:employee $emp)
   returns xs:decimal
{
   $emp/salary + $emp/bonus
}
```

Expressions

XQuery defines a number of different types of expressions.

expression type	expression syntax
sequence	*expr*, *expr*, ...
literal	e.g. 7, 'XQuery', "literal"
variable	e.g. $x, $po:backOrders
constructor	*see below*
numeric	+, -, *, div, idiv, mod
value comparison	eq, ne, lt, le, gt, ge
general comparison	=, !=, <, <=, >, >=
node comparison	is, isnot
order comparison	<<, >>
logical	and, or
conditional	if *expr* then *expr* else *expr*
range	*expr* to *expr*
quantified	some/every *$var* in *expr* satisfies *expr*
set	union, intersect, except
typeswitch	typeswitch *expr* case type *$var* return *expr* default *$var* return *expr*
instance of	*expr* instance of *type*
validate	validate { *expr* } validate *schemaContext* { *expr* }
cast	cast as *type* (*expr*)
treat	treat as *type* (*expr*)
path	e.g. $emp/phone[@type='home']
FLWR	for *type* *$var* in *expr*, ... let *$var* := *expr*, ... where *expr* return *expr*
function	*QName* (*expr*, ...)

Value comparison expressions (such as `eq`) compare single atomic values. General comparison expressions (such as =) compare two sequences of values, and provide existential semantics (return true if the comparison is true for some value in the first sequence and some value in the second sequence).

Order comparison determines whether one node appears earlier than or later than another node in document order.

A range expression produces a sequence of integers, starting at the value of its first argument and ending at its second argument.

```
(1 to 4) → (1, 2, 3, 4)
```

Many of these expressions accept the empty sequence, written as `()`, as an argument. Many of them return an empty sequence when they are given an empty sequence as an argument.

```
4.2e5 + () → ()
```

Nodes and node sequences may be used where atomic values are expected. For a sequence of one node, the typed value of the node will be used. A sequence of more than one value will raise an error. In the example below, plus (+) is being applied to two nodes of type `xs:integer`.

input:

```
   <employee id='575090'>
      <HSYears>4</HSYears>
      <CollegeYears>6</CollegeYears>
      <veteran discharge='honorable'/>
   </employee>

count(input()
      //employee
      [HSYears + CollegeYears ge 8]
     )

→

1
```

Nodes and node sequences may be used where a Boolean value is expected. If a node contains a single value of type `xs:boolean`, then this value is used. If the node sequence is empty, then `false` is returned. If the node sequence contains at least one node, then `true` is returned.

```
if (input()
    //employee[@id eq '575090']/veteran)
then 'served'
else 'did not serve'

→

'served'
```

Types

Some of the expressions that we listed above are specified with a type designator. Rather than going through BNF, we'll just look a number of examples:

`xs:integer?`	a sequence of zero or one integer
`element+`	a sequence of one or more elements
`node*`	a sequence of zero or more nodes
`item+`	one or more items
`attribute`	an attribute (single) of any name and type
`element myco:address`	an element with name `myco:address`
`element of type myco:addrType`	an element of any name, with type `myco:addressType`
`element zip in type myco:addrType/zipplus`	an element named `zip`, with the type of the `zip` element that occurs within an `zipplus` element in the `myco:addrType` complex type

A type designator might be used as follows:

```
input()
//employee
  [address
   instance of element myco:USAddress)
  ]
```

Constructors

The construction of new XML content is central to XQuery. XQuery contains constructors for elements, attributes, CDATA sections, processing instructions, and comments using a syntax that is largely the same as XML itself. Element content and attribute values can contain enclosed expressions, denoted by {}'s, that contain expressions that will be evaluated. This allows the following:

```
{-- demonstrate enclosed expressions --}
let $x := 5
return
   <let x="{$x}">
      <!-- assignment -->
      $x := {$x}
   </let>

→

<let x="5">
   <!-- assignment -->
   $x := 5
</let>
```

An element may contain a namespace declaration. These namespaces will be added to the in-scope namespaces for the scope of the element.

```
<myco:employee xmlns:myco="..."
               xmlns:sk="...">
   <myco:skills>
      <sk:skill> ... </sk:skill>
      ...
   <myco:skills>
</myco:employee>
```

The names of elements and attributes can be computed as well, but doing so requires a different syntax. The following function uses the element provided in an argument to construct a new element. The first child element of the supplied element is determined and its name is used for the name of the new element; the value of its first attribute is used for the new element's content:

```
define function first (element $f)
          returns element
{
   let $firstChild := $f/*[1]
   return
      element { xf:name($firstChild) }
              { xf:data($firstChild/@*[1]) }
}

first(<employee id='998359'>
         <status='retired'>
         <name>Alan Greene</name>
      </employee>)

→

<status>retired</status>
```

A constructed element has its own identity. The nodes that are used to construct this element are copied, so that every node in the constructed element has a new identity. The type of a constructed element is `xs:anyType` … all of the type information of its contents is discarded. Even if an attribute such as `xsi:type='myco:USAddressType'` is specified, the type of the constructed element will still be `xs:anyType` until it has been validated.

Validation

The validate expression applies schema validation to its argument (of type `element*`). Its argument is converted into an infoset, discarding any type annotations that it might have contained. The result of validation is a new element (with new contents) with type annotations. If validation is not successful, then a dynamic type error is raised.

Type annotations can be applied to a constructed element using the validate expression:

```
validate (<myco:employee id='440612'>
            <name>Augustus Child</name>
            ...
          </myco:employee>
         )
```

In this case, the "myco" schema must contain a globally defined element employee. The name element in the constructed element has type xs:anyType, while in the validate result it might have type myco:nameType. Validation can also be specified for locally defined elements:

```
validate in myco:employee/contact
   (<phone>666-555-1212</phone>)
```

FLWR Expression

The FLWR (for, let, where, return) expression provides for iteration over the items in one or more sequences. It is as central to XQuery as SELECT is to SQL.

The "for" clause binds variables to the items in its sequences, generating a Cartesian product among the bound variables. The "let" clause binds a variable to a value. The "where" clause filters the elements of the Cartesian product, leaving those that remain to contribute to the result of the FLWR. Finally, the "return" clause creates a sequence, concatenating the values produced by the evaluation of its expression.

The best that we can do in this limited space is to provide some examples of this type of expression.

- Return within a single element the names of the employees that have more than 8 years of education:

```
<educated>
   { for $e in document('employees.xml')
                //employee
     where $e/HSYears + $e/CollegeYears gt 8
     return $emp/name
   }
</educated>
```

→

```
<educated>
   <name>Albert Jones</name>
   <name>Joe Cody</name>
</educated>
```

- Return as a sequence of strings the names of the employees that have more than 8 years of education:

```
for $e in document('employees.xml')
          //employee
where $e/HSYears + $e/CollegeYears gt 8
return data($emp/name)
```

→

```
Albert Jones
Joe Cody
```

- Produce an element structure with employees contained within their departments:

```
let $dept := document('depts.xml')
let $emp := document('employees.xml')
for $d in $dept//department
return
   <department name='{$d/name}'>
      { for $e in $emp//employee
        where $e/dept eq $d/name
        return
           <employee>
              { $e/name }
           </employee>
      }
   </department>
```

→

```
<department name='accounting'>
   <employee>
      <name>Albert Jones</name>
   </employee>
   .
   .
   .
</department>
.
.
.
```

- Produce a structure with employee names and the names of their departments provided in attributes:

```
let $dept := document('depts.xml')
let $emp := document('employees.xml')
for $e in $emp//employee,
    $d in $dept//department
where $e/dept eq $d/name
return
   <employee name='{data($e/name)}'
             dept='{data($d/dept)}'/>
```

→

```
<employee name='Albert Jones'
          dept='accounting'/>
<employee name='Gloria French'
          dept='accounting'/>
<employee name='Clark Hill'
          dept='security'/>
```

XQuery Updates

You may have noticed that we have not discussed the ability to modify XML documents through the use of the XQuery language. The XML Query WG has not published any documents that provide facilities to update any XML elements in an existing document, to insert new elements into such a document, or to delete elements from such a document.

This omission has been discussed on the public XQuery mailing lists and is recognized to be a

deficiency. It is possible that this omission will be corrected, either in a future version of XQuery or perhaps in a separate document.

XQuery Conformance

Basic XQuery is the minimal level of conformance that can be claimed for XQuery. Basic XQuery encompasses all of the XQuery functionality, with the following restrictions:

- the Query prolog must not import XML Schemas
- instances of the XQuery data model that are built from a PSVI will map atomic data types to their nearest XML Schema built-in data type. Nodes with complex data types will be given the type `xs:anyType`.
- static type errors do not have to be raised during the analysis phase

A conforming XQuery implementation may support the Schema Import Feature, which removes the first of these restrictions. It may support the Static Typing Feature, which removes the remaining restrictions.

XQueryX

XQueryX defines an XML representation of XQuery. It defines an element structure that mirrors the abstract syntax of XQuery. Let's look at a simple XQuery and the corresponding XQueryX representation:

```
for $b in document("bib.xml")//book
return $b/title
```

→

```
<q:query xmlns:q
       ="http://www.w3.org/2001/06/xqueryx">
  <q:flwr>
    <q:forAssignment variable="$b">
      <q:step axis="SLASHSLASH">
        <q:function name="document">
          <q:constant datatype="CHARSTRING">
             bib.xml
          </q:constant>
        </q:function>
        <q:identifier>book</q:identifier>
      </q:step>
    </q:forAssignment>
    <q:return>
      <q:step axis="CHILD">
        <q:variable>$b</q:variable>
        <q:identifier>title</q:identifier>
      </q:step>
    </q:return>
  </q:flwr>
</q:query>
```

While XQueryX is harder for a human to read and write than XQuery, it does have several useful properties. It is easily generated by tools and layered applications, it is easily embedded within larger XML documents, and it allows "queries on queries".

XQuery 1.0 and XPath 2.0 Functions and Operators

The "F&O" specification defines a large number of functions and operators. While these functions and operators are being written to support XQuery, XPath, and XSLT, they could also be used by other XML specifications.

The "F&O" specification defines a number of string functions, numeric functions, node functions, date and time functions, aggregate functions such as `xf:avg`, `xf:min`, and `xf:max`, and the `xf:document` and `xf:collection` functions for accessing the content of XML documents. It defines what the allowable casts are, and what their behavior is.

The operators are functions that have been defined in a different namespace from the others and are not intended for invocation directly by users. They exist to define the semantics of operators such as "+" in the XQuery and XPath specifications.

Functions, as you would expect, are defined by providing their signature, a description of their semantics, and some examples. The parameters and the return types of functions use the XQuery/XPath data model that we discussed earlier. Functions have a local name, and are defined to be in the following namespace (which will be updated in each draft):

```
http://www.w3.org/2002/08/xquery-functions
```

The method of invoking these functions is left to the XQuery, XQueryX, and XPath specifications.

For the most part, functions with the same name have a different number of parameters and the parameters in the "smaller" function and "larger" function that occupy the same position have the same data types. A small number of functions with the same name and same number of parameters have different data types, for the purpose of backward compatibility with XPath 1.0 functions.

By way of example, let's look at a specific function:

```
xf:translate(string? $srcval,
             string? $mapString,
             string? $transString)
            => string?
```

This function constructs an xs:string result by considering each character in `$srcval` in turn. If the character is not found in `$mapString`, then it is appended to the result. If the character is found in

`$mapString`, then the character in the corresponding position in `$transString` is appended to the result. This might be used as follows:

```
xf:translate("abcdabc", "abc", "AB")
```

→

```
"ABdAB"
```

Many of the string functions that are defined have two variants, one with an `xs:anyURI` collation parameter, and one without. If a collation argument is supplied, then it is used to determine how two strings compare. If the collation argument is not supplied, then the default collation in the static context is used. If that default does not exist, then Unicode codepoint collation is used.

This specification defines two subtypes of the XML Schema `xs:duration` data type, `xf:yearMonthDuration` and `xf:dayTimeDuration`. As their names imply, they contain only the year and month components and the day through seconds components of the `xs:duration` data type, respectively. The restrictions on these subtypes give them the desirable property of full ordering. The "F&O" specifications provide a number of functions on these new subtypes, but provide only equals and cast to and from string for `xs:duration`. It is possible that these two subtypes will be added to a future version of XML Schema.

Formal Semantics

The Formal Semantic specification defines the normalization of a query, which expresses some XQuery constructs in terms of simpler XQuery constructs (the XQuery Core language). The static type of every expression in the core language is rigorously defined (via inference rules and judgements), as are the values produced by the dynamic evaluation of these expressions.

Many people find Formal Semantics difficult to read because of its mathematical notation and its firm focus on formality. However, in order to properly express the semantics of a complex language such as XQuery, such formality is mandatory. Feel free to read it if you wish, but you don't need to do so in order to program using XQuery.

Future Work

It is frustrating not to be able to say more about how we expect this work to progress. Having tried to give you a sense of the features and syntax of the XQuery language, we'll leave you with the reminder that this work has not yet reached W3C's Last Call Working Draft status. It is possible that quite a bit of what we have described will change. In fact, by the time you read this, new drafts of these specifications may already have been published. A quick look at the W3C Technical Reports page will let you know if this is the case.

References

[1] *XML Query Requirements*, Don Chamberlin, Peter Fankhauser, Massimo Marchiori, Jonathan Robie, Feb. 15, 2001, http://www.w3.org/TR/xmlquery-req.

[2] *XQuery 1.0: An XML Query Language*, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, Scott Boag, Aug. 16, 2002, http://www.w3.org/TR/xquery/.

[3] *XML Path Language (XPath) 2.0*, Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, Jérôme Siméon, Aug. 16, 2002, http://www.w3.org/TR/xpath20/.

[4] *XML Query Use Cases*, Don Chamberlin, Daniela Florescu, Peter Fankhauser, Massimo Marchiori, Jonathan Robie, Aug. 16, 2002, http://www.w3.org/TR/xmlquery-use-cases.

[5] *XQuery 1.0 and XPath 2.0 Data Model*, Mary Fernández, Jonathan Marsh, Marton Nagy, Aug. 16, 2000, http://www.w3.org/TR/query-datamodel/.

[6] *XQuery 1.0 Formal Semantics*, Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, Philip Wadler, Aug. 16, 2002, http://www.w3.org/TR/query-semantics/.

[7] *XQuery 1.0 and XPath 2.0 Functions and Operators Version 1.0*, Ashok Malhotra, Norman Walsh, Jim Melton, Jonathan Robie, Aug. 16, 2002, http://www.w3.org/TR/xquery-operators/.

[8] *XML Syntax for XQuery 1.0 (XQueryX)*, Ashok Malhotra, Jonathan Robie, Michael Rys, June 11, 2001, http://www.w3.org/TR/xqueryx.

Web References

W3C http://www.w3.org

XML Query WG
http://www.w3.org/XML/Query

QL'98 - The Query Languages Workshop
http://www.w3.org/TandS/QL/QL98/