XQL (XML Query Language)

August 1999

This version:
http://metalab.unc.edu/xql/xql-proposal.xml
Latest version:
http://metalab.unc.edu/xql/xql-proposal.xml
Editor:
Jonathan Robie (Software AG) <jonathan.robie@sagus.com>
Contributors:
Eduard Derksen (CSCIO) <enno@att.com>
Peter Fankhauser (GMD-IPSI) <fankhaus@darmstadt.gmd.de>
Ed Howland (DEGA) <Ed@dega.com>
Gerald Huck (GMD-IPSI) <huck@darmstadt.gmd.de>
Ingo Macherius (GMD-IPSI) <macherius@darmstadt.gmd.de>
Makoto Murata (Fuji Xerox) <murata@apsdc.ksp.fujixerox.co.jp>
Michael Resnick (Object Design, Incorporated) <resnick@odi.com>
Harald Schöning (Software AG) <harald.schoening@softwareag.de>

Abstract

As more and more information is either stored in XML, exchanged in XML, or presented as XML through various interfaces, the ability to intelligently query our XML data sources becomes increasingly important. XML documents are structured documents – they blur the distinction between data and documents, allowing documents to be treated as data sources, and traditional data sources to be treated as documents.

XQL is a query language designed specifically for XML. In the same sense that SQL is a query language for for relational tables and OQL is a query language for objects stored in an object database, XQL is a query language for XML documents. The basic constructs of XQL correspond directly to the basic structures of XML, and XQL is closely related to XPath, the common locator syntax used by XSL and XPointers. Since queries, transformation patterns, and links are all based on patterns in structures found in possible XML documents, a common model for the pattern language used in these three applications is both possible and desirable, and a common syntax to express the patterns expressed by that model simplifies the task of the user who must master a variety of XML-related technologies. Although XQL originated before XSL Patterns, Patterns, there were strong similarities between the two languages, and we have adopted XPath syntax for the constructs which differed. Not all constructs found in XPath were needed for queries, and some constructs used in XQL are not found in XPath, but the two languages share a common subset.

The XQL language described in this paper contains several features not found in previously published versions of the language, including joins, links, text containment, and extensible functions. These new features are inspired in large part by discussions stemming from the W3C QL '98 Workshop, and make it possible to combine information from heterogeneous data sources in powerful ways. Great care has been made to maintain the fundamental simplicity of XQL while adding these features.

This paper is intended as input for the upcoming W3C Query Language Activity, and for the further development of XPath.

Table of contents

Appendices

---

1. XML Query Language

Traditionally, structured queries have been used primarily for relational or object oriented databases, and documents were queried with relatively unstructured full-text queries. Although quite sophisticated query engines for structured documents have existed for some time, they have not been a mainstream application. In the last year, a number of very different approaches to querying XML have been proposed, with several distinct perspectives on what constitutes a query. Several particularly interesting proposals have come from the semi-structured database community, including XML-QL and Lorel, and adopt semi-structured approaches to XML. This proposal incorporates several ideas from those languages into XQL.

XQL was designed to be used in a number of different XML environments, using a syntax that may be used in XML attributes, embedded in programming languages, or incorporated in URIs. From the beginning, we have endeavored to keep the language simple and small, and we have been careful not to add functionality that would make it difficult to implement XQL. During the last year, we have been persuaded to add several powerful new features that allow users to combine information from multiple sources, use the relationships expressed in links as part of a query, and search based on text containment. Queries that can make use of information in multiple documents allow the information contained in those documents to be reused in ways not foreseen by the people who created the original documents. This is extremely useful when many documents or data sources may each contain part of the information needed on a given topic. For instance, suppose one document contains a set of recommended books for a given course of study, another lists books and prices for a store, and third contains a set of reviews of books. A query can be constructed to list recommended books, their prices, and the reviews they have received.

XQL is closely related to XPath, and we hope to be able to maintain compatibility with XPath as it evolves. We see XQL as complementary to XSLT, which may be used for sophisticated reshaping and formatting of query results.

1.1 XML as a Data Model

An important motivation for the design of XQL is the realization that XML has its own implied data model, which is neither that of traditional relational databases nor that of object oriented or object-relational databases. In XQL, a document is an ordered, labelled tree, with nodes to represent the document entity, elements, attributes, processing instructions, and comments. The model is compatible with the XML Information Set (http://www.w3.org/XML/Group/1999/04/WD-xml-infoset-19990428.html).

It is important to note that the relationships among data contain a large proportion of the information contained in a document, which is one of the reasons that structured document formats like XML are useful in the first place. The original formulation of XQL was based completely on the tree structure of XML documents:

1. Hierarchy

    1. parent/child
    2. ancestor/descendant
2. Sequence (within a sibling list or in document order)
3. position (within a sibling list or in document order)
    1. absolute
    2. relative
    3. ranges

These relationships have long been basic to the XPointer model, and are now reflected in XPath in the form of axes. In XQL, all queries use the child axis, so we will speak in terms of parent/child and ancestor/descendant relationships rather than use the term **Locator Path** from the XPath Working Draft.

The current draft extends this model to support the following:

1. Ad-hoc relationships established via joins
2. Dereferencing of links

Joins allow subtrees of documents to be combined in queries; links allow queries to support references as well as tree structure.

1.2 What is an XML Query?

In XQL, a query returns XML document nodes from one or more XML documents. To examine the characteristics of an XQL query, it is useful to consider four basic questions about the environment in which a query takes place:

1. What is a database?
2. What is the query language?
3. What is the input to a query?
4. What is the result of a query?

The following table provides a brief answer to each of these questions, including a comparison with the SQL query language, which is widely used for querying relational databases:

SQL	XQL
The database is a set of tables.	The database is a set of one or more XML documents.
Queries are done in SQL, a query language that uses the structure of tables as a basic model.	Queries are done in XQL, a query language that uses the structure of XML documents as a basic model.
The FROM clause determines the tables which are examined by the query.	A query is given a list of input nodes from one or more documents.
The result of a query is a table containing a set of rows; this table may serve as the basis for further queries.	The result of a query is a list of XML document nodes, which may serve as the basis for further queries.

From the preceding table, it should be clear that document nodes play a central role in XQL queries. These nodes are an abstraction. Any real XQL implementation will find some concrete way to implement the

nodes used in queries. For instance, XQL engines may represent the input to a query via DOM nodes, XSL nodes, index structures, or XML text. Any of these might also be used to represent the results of queries; in addition, hyperlinks or other references into the original document might be used, a new virtual document might be created, or DOM Level Two TreeWalkers or Iterators might be used.

The nodes which form the input to a query may come from a variety of different sources. They may be the result of a prior query, the contents of a document repository, the nodes from a Document Object Model Nodelist, or any other source that identifies nodes from one or more documents. XQL does not specify how these nodes are brought to the query. Current XQL implementations take a variety of approaches, including the following: using Document Object Model subtrees as the basis for a query, querying whole documents supplied as the input to a Unix-style pipe, reading a document from the command line, using data dictionaries or repository directory structures to identify nodes to be queried, and identifying documents using a URL. This proposal adds support for merging information from heterogeneous data sources using joins.

In XQL, nodes have identity, and they retain their identity, containment relationships, and sequence in query results. Grouping operators allow levels of a tree to be omitted, while still retaining the relative sequence and containment of the nodes which are returned by a query. Joins allow subtrees from one data source to be inserted into another document subtree, subject to the join conditions. Link functions are similar to joins, allowing a hypertext link in a document to be replaced by the node or nodes to which it refers. Some functions in XQL return values, which may be boolean, integer, or string. These values are also treated as nodes in the query model.

1.3 XQL Tutorial

Before going into further detail, we feel it would be helpful to present some typical XQL queries to help convey a feeling for the language. This tutorial discusses the simplest XQL queries, which are also likely to be the most common. In this tutorial, we will present a quick overview of XQL without taking the time to be precise. For more precise definitions of the expressions used in these examples, see the section on XQL Expressions.

A simple string is interpreted as an element name. For instance, this query specification returns all <table> elements:

```
table
```

The child operator ("/") indicates hierarchy. This query specification returns <author> elements that are children of <front> elements:

```
front/author
```

The root of a document may be indicated by a leading "/" operator:

```
/novel/front/author
```

> **Ed. Note:** In XQL, the root of a document refers to the document entity, in the technical XML sense, which is basically equivalent to the document itself. It is not the same as the root element, which is the element that contains the rest of the elements in the document. The document root always contains the root element, but it may also contain a doctype, processing instructions, and comments. In this example,<novel> would be the root element.

Paths are always described from the top down, and unless otherwise specified, the right-most element on the path is returned. For instance, in the above example, <author> elements would be returned.

The content of an element or the value of an attribute may be specified using the equals operator ("="). The following returns all authors with the name "Theodore Seuss Geisel" that are children of the <front> element:

```
front/author='Theodore Seuss Geisel'
```

Attribute names begin with "@". They are treated as children of the elements to which they belong:

```
front/author/address/@type='email'
```

The descendant operator ("//") indicates any number of intervening levels. The following shows addresses anywhere within <front>:

```
front//address
```

When the descendant operator is found at the start of a path, it means all nodes descended from the document. This query will find any address in the document:

```
//address
```

The filter operator ("[ ]") filters the set of nodes to its left based on the conditions inside the brackets. The following query returns addresses; each of these addresses must have a nattribute called "type" with the value "email":

```
front/author/address[@type='email']
```

Note that"address[@type='email']" returns addresses, but"address/@type='email'" returns type attributes.

Multiple conditions may be combined using Boolean operators.

```
front/author='Theodore Seuss Geisel'[@gender='male' and @shoesize='9EEEE']
```

Brackets are also used for subscripts, which indicate position within a document. The following refers to sections 1, 3, 4, 5, and 8, plus the last section:

```
section[1,3 to 5, 8, -1]
```

Conditions and subscripts may not both occur in the same brackets, but both uses of brackets may occur in the same query. The following refers to the first three sections whose level attributes have the value "3"; in other words, it returns the first three "level3" sections:

```
section[@level='3'][1 to 2]
```

Now that we know the basics, let's take a look at a document and try some XQL queries on it. The following is an invoice document. Traditionally, invoices are often stored in databases, but invoices are both documents and data. XQL is designed to work on both documents and data, provided they are represented via XML through some interface. This document will be the basis for the sample queries that follow:

```
<?xml version="1.0"?>
<invoicecollection>
  <invoice>
    <customer>
              Wile E. Coyote, Death Valley, CA
    </customer>
         <annotation>
                  Customer asked that we guarantee return rights
                  if these items should fail in desert conditions.
                  This was approved by Marty Melliore, general
        manager.
    </annotation>
         <entries n="2">
              <entry quantity="2"  total_price="134.00">
                <product maker="ACME" prod_name="screwdriver" price="80.00"/>
      </entry>
              <entry quantity="1" total_price="20.00">
                <product maker="ACME" prod_name="power wrench" price="20.00"/>
      </entry>
    </entries>
  </invoice>

  <invoice>
    <customer> Camp Mertz  </customer>
```

```
            <entries n="2">
               <entry quantity="2" total_price="32.00">
                       <product maker="BSA" prod_name="left-handed smoke shifter" price="1
               </entry>
               <entry quantity="1" total_price="13.00">
                       <product maker="BSA" prod_name="snipe call" price="13.00"/>
               </entry>
        </entries>
      </invoice>
    </invoicecollection>
```

Now let's look at some sample queries. For these examples, we will present query results as text, using a serialization approach described in the section "Query Results and Serialization". In general, XQL queries return lists of nodes, which may be represented in any way convenient to the environment in which the query is performed, e.g. as DOM nodes, serialized XML text, XPointers, hyperlinks, or by creating an iterator to navigate the results. Since XML text is easily read, we find it suitable as a way of representing results in our examples.

Suppose we wanted to see just the customers from the database. We could do the following query:

Query:

```
//customer
```

Result:

```
    <xql:result>
      <customer> Wile E. Coyote, Death Valley, CA </customer>
      <customer> Camp Mertz </customer>
    </xql:result>
```

We might want to look at all the products manufactured by BSA. This query would do the trick:

Query:

```
//product[@maker='BSA']
```

Result:

```
    <xql:result>
      <product maker="BSA" prod_name="left-handed smoke shifter" price="16.00"/>
      <product maker="BSA" prod_name="snipe call" price="13.00"/>
    </xql:result>
```

Filters are particularly useful when specifying conditions on paths that are not the same as what is returned. For instance, the following query returns the products ordered by Camp Mertz:

Query:

```
//invoice[customer='Wile E. Coyote, Death Valley, CA']//product
```

Result:

```
    <xql:result>
      <product maker="ACME" prod_name="screwdriver" price="80.00"/>
      <product maker="ACME" prod_name="power wrench" price="20.00"/>
    </xql:result>
```

This is the end of the tutorial, which covers only the most basic features of XQL. For examples illustrating newer or more advanced features, such as return operators, sequence, joins, references, and user-defined functions, see the appropriate parts of the next section.

1.4 XQL Expressions

An XQL query is always evaluated for a **context**, which is a list of document nodes. The initial context for a query is known as the **start context**. In XQL, the nodes in a start context may come from different documents, and even if they are in the same document, there is no assumption that they come from contiguous portions of the document. Some XQL operators establish a new context in which a subexpression will be evaluated; for instance, in the expression "author/name", "author" is evaluated in the start context. For each author, the "/" operator establishes a new context consisting of the children of that author, and "name" is evaluated in that context. The operators that establish a new context are /, //, and [].

> **Ed. Note:** In XSL, expressions are evaluated with respect to a node which is called the context node. Our use of the term "context" is intended to allow semantic consistency with XSL Patterns without imposing unecessary restrictions on the query language. As a consequence, XSL Patterns are defined in terms of children of the context node, and XQL queries are defined in terms of the context node directly. We maintain the correspondence of XSL Pattern definitions and XQL definitions by constructing an imaginary context node that contains the nodes of the context, and allowing the XSL term "." to map to this context node.

1.4.1 Terms

The following expressions are **terms**, which select particular nodes from the context based on the type or name of the node:

n	element name	All nodes in the context where the node type is element and the node name is "n".
*	element name with wildcards	All nodes in the context where the node type is element.
@n	attribute name	All nodes in the context where the node type is attribute and the node name is "n".
@*	attribute name with wildcards	All nodes in the context where the node type is attribute.
text()	text node	All nodes in the context where the node type is text.
comment ()	comment	All nodes in the context where the node type is comment.
pi()	processing instruction	All nodes in the context where the node type is processing instruction.
pi("v")	processing instruction with target	All nodes in the context where the node type is processing instruction and the target is "v".
.	context node	The node which is the parent to the nodes in the context - this node may be real or imaginary.

1.4.2 Namespaces and names

In XML expressions, names may be associated with namespace prefixes. A namespace prefix can be declared using a variable declaration. In the following query, the first line declares "b" to be a variable equivalent to the namespace URI "http://www.TwiceSoldTales.com". The second line of the query searches for all <book> elements belonging to this namespace:

```
b := "http://www.TwiceSoldTales.com";
//b:book
```

An XML document may well use a different namespace prefix for the same namespace URI. Matching is done on the basis of the namespace URI, not the prefix associated with it in the document or in the XQL query.

XQL expressions can explicitly state whether namespaces should be taken into account when matching node names:

table	Any element named <table>, regardless of the namespace to which it belongs.
html:table	Any element named <table> that belongs to the namespace indicated by the prefix "html".
*	Any element, regardless of the namespace to which it belongs.

:table	Any element named <table> for which no namespace has been declared.
*:table	Any element named <table> for which a namespace has been declared.
html:*	Any element belonging to the namespace associated with the prefix "html".
:*	Any element for which no namespace has been declared.
:	Any element for which a namespace has been declared.

The same conventions apply to attribute names. In attribute names, the attribute prefix comes before the namespace prefix:

```
@lib:isbn
```

Namespaces are preserved in the output of a query. To change the namespaces of nodes in the output, use the Renaming Operator.

1.4.3 Comparisons

Comparisons add constraints based on the content or value of nodes. Consider the following examples:

```
author="Washington Irving"

@id="id-sec-0203"

text() = "Whan that Aprille with his shoures soughte"
```

Regardless of the node type on the left hand of the comparison, it is compared to the value on the right. For systems that use a schema that supports data types, they are used in comparisons:

```
books[pub_date < date("1990-01-01")]
```

Since some environments in which XQL is used have restricted character sets, e.g. URIs or queries stored in attribute values, many comparisons have an alternative syntax that meets the syntactic constraints of these environments. For instance, the following two queries are equivalent:

```
books[pub_date < date("1990-01-01")]

books[pub_date lt date("1990-01-01")]
```

The following comparison operators are available in XQL:

Equality	n="value"
	n eq "value"
Case insensitive comparison	n ieq "value"
Inequality	n !="value"
	n ne "value"
Text containment	n contains "value"
Case insensitive text containment	n icontains "value"

Text comparisons support the wildcard characters "*" and "?". Consider the following example:

Data:

```
<editor>
  <name>
    <first> Ramesh </first>
          <last> Lekshmynarayanan </last>
  </name>
</editor></customer>
```

Query:

```
//(editor contains "Leksh*")
```

The value "Leksh*" matches the name "Lekshmynarayanan", and the <editor> element is returned.

The following operators may be defined in XQL environments that support data types:

Less than	n < value
	n lt value
Less than or equals	n <=value
	n lte value
Greater than	n > value
	n gt value
Greater than or equals	n >=value
	n gte value

1.4.4 Hierarchy and Filters

These operators establish a new search context and evaluate a subexpression within that context. In this table, Q1 and Q2 are used to denote arbitrary XQL expressions.

Q1/Q2	parent/child	Children of nodes that satisfy Q1, evaluated in the current context, such that the children satisfy Q2. Q2 is evaluated separately for the child list of each node in Q1; the nodes to which each child list evaluates are unioned together.
Q1//Q2	ancestor/descendant	Descendants of nodes that satisfy Q1, evaluated in the current context, such that the descendants satisfy Q2. Q2 is evaluated separately for each child list of each node in Q1, and recursively for each node in the child list; the nodes to which each child list evaluates are unioned together.
Q1[Q2]	filter	Nodes that satisfy Q1, evaluated in the current context, containing children that satisfy Q2. Q2 is evaluated separately for the child list of each node in Q1; the nodes to which each child list evaluates are unioned together.
Q1 [poslist]	subscript	Nodes that satisfy Q1, evaluated in the current context, whose position in the evaluation list is contained in the poslist.

1.4.5 Boolean and Set Operators

Terms or other XQL expressions may be combined using **boolean operators**and **set operators**:

not(q)	negation	All nodes in the context for which the expression q evaluates to null.
q1 union q2	union	The union of q1 and q2, evaluated in the context.
q1 intersect q2	intersection	The intersection of q1 and q2, evaluated in the context.
q1 | q2	union	The union of q1 and q2, evaluated in the context.
q1 ~ q2	both	If both q1 and q2 are non-empty, returns q1 union q2; if either is empty, returns the empty list.
q1 or q2	or	(Boolean) If the union of q1 and q2, evaluated in the context, is non-empty, returns true; else, returns false.
q1 and q2	and	(Boolean) If the intersection of q1 and q2, evaluated in the context, is non-empty, returns true; else, returns false.

The "both" operator was introduced because we found that many queries use filters to express constraints on the same data that is returned outside the filter, resulting in expressions that are rather redundant. For instance, the following query uses filters to express that only invoices for the customer named "Wile E. Coyote"that also contain products are of interest, and both the customer name and the set of products should be returned:

```
//invoice[customer[name='Wile E. Coyote'] and .//product]/(customer | .//product)
```

Using the "both" operator, this same query can be expressed more concisely:

```
//invoice/(customer[name='Wile E. Coyote'] ~ .//product)
```

Note that the "both" operator is neither the boolean "or" operator nor the set intersection operator. The expression "customer intersect product" always returns an emtpy result since no element is ever simultaneously a <customer> element and a <product> element. The "both" operator is used to specify conditions which must simultaneously be satisfied for the context.

1.4.6 Grouping Operator

It is often useful to group results using the structure of the original document. For instance, a query that lists the products on invoices might want to group products by invoice, placing each group of products within an invoice tag. XQL provides a grouping operator that provides exactly this functionality. In the following query, the element to the left of the curly braces (the Grouping Element) is used to group the results of the query within the braces:

```
//invoice { .//product }
```

For each grouping element matched by the query, the grouping operator creates an empty element with the same name. The results of the query contained within the curly braces are then appended to this new node as children. If we apply this query to the invoice data presented in the tutorial, we obtain these results:

```
<xql:result>
  <invoice>
    <product maker="ACME" prod_name="screwdriver" price="80.00"/>
    <product maker="ACME" prod_name="power wrench" price="20.00"/>
  </invoice>
  <invoice>
    <product maker="BSA" prod_name="left-handed smoke shifter" price="16.00"/>
    <product maker="BSA" prod_name="snipe call" price="13.00"/>
  </invoice>
</xql:result>
```

Complex queries that use the grouping operator can be made more readable by the appropriate use of whitespace, eg:

```
invoice {
         .//customer[name contains "Coyote"] {
                  name | address
         } ~
   entries {
            .//product[@maker="ACME"]
         }
}
```

1.4.7 Sequence

XQL defines the following operators for sequence:

before	a before b	Returns a list of all "a"s that precede a "b".
after	a after b	Returns a list of all "a"s that occur after a "b".
list concatenation	a, b	Returns a list containing all "a"s, followed by all "b"s. Useful for specifying order in return lists.

The list concatenation operator is used to specify order in return lists. In general, XQL operators maintain document order; the concatenation operator allows an order to be specified within a return list. For instance, instance, the following query specifies that the order of the returned results should be author, then title, then isbn:

```
//book//(author, title, isbn)
```

If there is more than one author, all authors will be listed before the title.

In systems where XML is used mainly to represent data from object oriented systems or relational databases, sequence may not be particularly important. However, sequence is important in documents, and it also can be useful in data-oriented applications where the markup does not clearly indicate the role of each element. Consider the following table, which lists the latest scores for some fictitious sport:

Western League	
Aardvarks 12	Weasels 10
Mosquitos 17	Slugs 2
Southern League	
Tortoises 25	Hares 0
Platypii 17	Amoebae 16

The markup for this table looks like this:

```
<table width="50%" border="1">
  <tbody>
    <tr>
      <td colspan="2"><emph>Western League</emph> </td>
    </tr>
    <tr>
      <td colspan="1">Aardvarks 12</td>
      <td>Weasels 10</td>
    </tr>
    <tr>
      <td colspan="1">Mosquitos 17</td>
      <td>Bulls 2</td>
    </tr>
    <tr>
      <td colspan="2"><emph>Southern League</emph></td>
    </tr>
    <tr>
      <td colspan="1">Tortoises 25</td>
      <td>Hares 0</td>
    </tr>
    <tr>
      <td colspan="1">Platypii 17</td>
      <td>Amoebae 16</td>
    </tr>
  </tbody>
</table>
```

Purists may object that this is not particularly good markup, since it does not clearly distinguish the leagues from the scores. We agree, and when we write our own documents, we would write them differently; however, there is a lot of mediocre markup in the real world, and when querying documents, we do not have the luxury of rewriting them first. Therefore, we feel that a query language should be able to manage data like that shown above.

To find all the latest scores for the Western League, we can use the following query:

```
table//((tr after (tr contains "Western League")) before (tr contains "Southern League"
```

> **Ed. Note:** Sequence is handled by axes in XPath. We believe that an XML query language should provide some means for allowing sequence in queries, and that various approaches should be considered. The approach discussed here has advantages in expressing relationships among multiple nodes, especially when comparisons are to be made only within the descendants of a particular node.

1.4.8 Functions

Most of the functions of XQL have been taken directly from XSL Pattern Language. A few functions have been added, many more have been omitted because we found them to be less relevant in a pure query environment than in a general purpose transformation environment.

1.4.8.1 Collection functions

attribute(), attribute('name')	Returns the attributes in the context. If a name argument is supplied, returns the attribute attribute with the given name.
comment()	Returns the comments in the context.
element(), element('name')	Returns the elements in the context. If a name argument is supplied, returns the elements with the given name.
entity-ref()	Returns the entity references in the context. XQL operates on a view of the document in which all entity references are expanded; this function is the only way to locate entity references in XQL.
node()	Returns all nodes in the context.
pi(), pi('target')	Returns the processing instructions in the context. If a target argument is supplied, returns the processing instructions with the given target.
text()	Returns the text nodes in the context. For the sake of text nodes, XQL assumes that CDATA sections are treated as text, adjacent text nodes are merged, and entity references are expanded.

1.4.8.2

count()
id()
idref()
position()

1.4.9 Extensible Functions

Many XQL implementations are part of a programming environment. In these environments, it is helpful to allow users to write their own functions, which may be used in queries. This must be done in a language-independent manner, since XQL implementations have been done in a variety of languages, including C++, Java, Haskell, and Perl. To allow user-defined functions to be written, XQL provides a function called "function()".

Suppose a user wanted to add a function that computes the average for a list of values. The user could write a function called "average" and call it in an XQL query like this:

```
average(property//price)
```

User-defined functions are typically written in the language environment of the XQL implementation; for instance, if the XQL implementation is written in Java, user-defined functions are generally written as Java functions. All XQL functions are passed the list of nodes in the current context. If the function has parameters, these are passed as strings to the XQL function. Typically, the function will evaluate these parameters as queries against the current context; for instance, the user code that implements the "average" function might first execute the query "property//price" for the current context to obtain a set of <price> elements, then compute the average of these elements.

The result of a function call is also a nodelist. If a single value is to be returned, such as a string or a number, it should be returned as an element node of that type:

```
<xql:number> 112,000.47 </xql:number>
```

The available set of types that may be returned by functions is described in the section "Query Results and Serialization", which follows the current section. If a function is called with the wrong parameters, this may be communicated by returning an <xql:warning> element in the result:

```
<xql:warning> "average" requires numeric values for the nodes to be averaged
```

```
</xql:warning>
```

Ed. Note: Some vendors have asked that extensible operators be provided as well. This would be a useful feature; so far, we have not found a clean design for extensible operators in XQL.

Issue (function-namespace): There are differing opinions as to whether namespaces add significant value as vendors and users add functions to XQL.

1.4.10 References

Ed. Note: The ideas in this section are exploratory, and have not yet been incorporated into XQL.

There is currently no syntax for dereferencing links in XQL, but this is clearly needed in many applications. XSL provides the "id()" function, which returns the element containing a given id. For instance, the following would evaluate to the node pointed to by an HREF attribute in an <A> element:

```
A/id(@HREF)
```

From an XQL perspective, this is actually a kind of join. However, the above syntax is less complex than the the equivalent join syntax:

```
A/id[$h = @HREF]/(//*[id=$h])
```

We need functionality similar to id(), extending this functionality to incorporate any kind of link, not just ID/IDREF. Let's create a function called ref() which returns the node or nodes to which an XPointer or HTML HREF points

```
A/ref(@HREF)
```

One advantage of the join syntax is that it allows the type of the referenced node to be specified. It may be useful to be able to specify this as a further parameter to the function. Let's allow the type of the referenced node to be specified as a second parameter to the function. For instance, the following will return the referenced node only if it is a 'table" element; otherwise, it will return null:

```
A/ref(@HREF, "table")
```

It may also be helpful to specify further parameters, e.g. to limit the scope of the reference to the current document, the local repository, or some other identifiable scope.

It is frequently useful to be able to identify the references to a particular node from other nodes. For instance, if we are thinking of deleting something from a document, we may want to know if it is referenced. For this purpose, it may be useful to introduce another function that returns all nodes that reference a particular node. If we call this function "backref()", it might look like this:

```
A/backref(table[0])
```

Issue (ref-scope): Backwards references will also need to be scoped somehow, and not all systems will want to support them, due to implementation overhead.

References can also be used to specify the URLs of documents used in queries:

```
ref("http://www.amazon.com")//book[.//title contains "Alhambra"]
```

1.4.11 Joins

Ed. Note: Joins are a new feature in XQL. The approach to joins discussed in this section comes largely from Peter Fankhauser of the GMD-IPSI and Harald Schöning of Software AG. Gerald Huck of the GMD-IPSI has been particularly helpful in refining the initial model. There

is some preliminary implementation experience with this approach.

In many environments, it is useful to be able to combine information from multiple sources to create one unified view. For instance, suppose we have a source of books and a source of reviews:

```
<book>
  <isbn> 84-7169-020-9 </isbn>
  <title> Tales of the Alhambra </title>
  <author> Washington Irving </author>
</book>

<review>
  <isbn> 84-7169-020-9 </isbn>
  <title> Tales of the Alhambra </title>
  <reviewer> Ricardo Sanchez </reviewer>
  <comments>
         A romantic and humorous account of the time that
         the author of "The Legend of Sleepy Hollow" lived
         in an Arabian palace in Spain.
  </comments>
</review>
```

We may want to combine these to create a view of the book that includes the comments found in reviews:

```
<book>
  <isbn> 84-7169-020-9 </isbn>
  <title> Tales of the Alhambra </title>
  <author> Washington Irving </author>

  <review>
     <reviewer> Ricardo Sanchez </reviewer>
     <comments>
              A romantic and humorous account of the time that
              the author of "The Legend of Sleepy Hollow" lived
              in an Arabian palace in Spain.
     </comments>
  </review>
</book>
```

This amounts to inserting information from the review into the book. If we had a database that consisted only of this one book and this one review, we could obtain the desired result with this query:

```
/book {
                  isbn | title | author | //review { reviewer | comments }
}
```

If we are using a database with many books and many reviews, the above query would include the whole list of reviews in every single book, not just the reviews for the book in question. We need some way to restrict our reviews to those that have the same ISBN number as the book. We will do this by introducing correlation variables. In the following example, "$i := isbn" assigns the variable "$i" to the evaluation of isbn in the context of each book. The expression "//review[isbn=$i]" restricts the reviews to those that match "$i":

```
/book[$i:=isbn] {
           isbn | title | author | //review[isbn=$i] { reviewer | comments }
}
```

Ed. Note: Although filters and variable bindings both use square bracket notation, variable bindings do not filter results. For instance, the expressions "/book" and "/book[$i:=isbn]" will always return the same set of books, whether or not any <isbn> elements are present.

Variable bindings propogate as new search contexts are created; when a new context is created, e.g. as the result of a child or descendant operator, it inherits all variable bindings that are active. This allows bindings declared high in the document hierarchy to be used for joins performed lower down.

If a correlation variable is bound to a subexpression that evaluates to more than one result, any value in the list of results will be used as the basis for a join. To be precise, "list1 relop list2" evaluates to "all e1 in list1

such that for some e2 in list2, e1 relop e2 is satisfied".

The following query returns books whether or not they have an isbn; reviews are returned only if they have a matching isbn:

```
/book[$i:=isbn] {
    $i  | title  | author
    | //review[isbn=$i] { reviewer | comments }
}
```

> **Ed. Note:** In this example, it seems intuitive to say that you can't join on null - a book with no isbn does not match all reviews that have no isbn. On the XQL mailing list, there is some difference of opinion as to whether it should be possible to join on null.

In XQL, square brackets are used for three distinct things that can not be mixed: subscripts, filters, and variable bindings. If you want both a filter and a variable binding, you must use separate sets of brackets:

```
/book[isbn][$i:=isbn] {
    $i  | title  | author
    | //review[isbn=$i] { reviewer | comments }
}
```

1.4.12 Renaming Operator

The nodes in a list may be renamed using the renaming operator "->". In joins, this can be used to reflect a meaningful name that describes the synthesized result:

```
/book[isbn][$i:=isbn] -> BookWithReviews {
    $i  | title  | author
    | //review[isbn=$i] { reviewer | comments }
}
```

The renaming operator may also be used to adjust namespaces in query results. Since renaming changes the name of a node, it also changes the namespace. For instance, suppose <book> is in the namespace of "http://www.TwiceSoldTales.com", and we rename the <book> element to <livre>:

```
//book->livre
```

We can assume that <livre> is not defined in the namespace associated with "http://www.TwiceSoldTales.com". Since renaming often creates element names that do not exist in the original namespace, renaming in XQL does not keep the namespace of the original node name. This property of the renaming operator can be used to remove namespaces; for instance, the following query places <book> elements in the default namespace, regardless of their original namespace:

```
//book->book
```

New namespace prefixes may be explicitly applied with the rename operator:

```
//book->a:book
```

1.4.13 Precedence of Operators

XQL expressions are evaluated from left to right. The following table shows the precedence of operators in XQL:

Query Operators by Decreasing Precedence	
Grouping	()
Filter	[]
Renaming	->
Grouping	{ }

Path	/ //
Comparison, Assignment	= != < <= > >= eq ne lt le gt ge contains ieq ine ilt ile igt ige icontains :=
Intersection	intersect
Union	union |
Negation	not()
Conjunction	and
Disjunction	or
Sequence	before after
End of Statement	;

Parentheses may be used for grouping:

```
(author | editor)/name

author | (editor/name)
```

1.5 Query Results and Serialization

In some environments, the results of a query are returned as XML text. XQL defines a serialization format to allow the results of queries to be returned as well-formed XML documents. Namespaces are used to distinguish tags belonging to the serialization format from tags returned by the query. When query results are serialized, they are wrapped in an <xql:result> element:

```
<xql:result xmlns:xql="http://www.metalab.unc/xql/serialization">
  <customer> Wile E. Coyote, Death Valley, CA </customer>
  <customer> Camp Mertz </customer>
</xql:result>
```

The reason for this is that a well-formed XML document may have only one root element, and queries may return any number of results. Other XQL serialization elements are used to return values from functions, provide additional information about a query, or indicate errors or warnings. The following elements are defined in the XQL serialization namespace:

<xql:result>	Surrounds the serialized results of the query.
<xql:query>	Optional. Contains the original query string. Useful for debugging.
<xql:true>	Returned by boolean functions.
<xql:false>	Returned by boolean functions.
<xql:number>	Returned by numeric functions.
<xql:text>	Returned by text functions.
<xql:attribute name="attributeName" value="attributeValue">	Used to return attributes when they are returned outside of the attribute list of an element.
<xql:declaration>	Used to return the XML declaration when it is returned in a query.
<xql:error>	Used to indicate an error in the query. The content of this element explains the error.
<xql:warning>	Used to indicate a warning. The content of this element explains the warning.

A. Appendix A: XQL and XML Pattern Expression Language

B. Appendix C: References