NOTE-xml-ql-19980819

XML-QL: A Query Language for XML

Submission to the World Wide Web Consortium 19-August-1998

This version:
    http://www.w3.org/TR/1998/NOTE-xml-ql-19980819
    http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.html
Latest version:
    http://www.w3.org/TR/NOTE-xml-ql
Authors:
    Alin Deutsch , University of Pennsylvania, USA
    Mary Fernandez , AT&T Labs, USA
    Daniela Florescu, INRIA, France
    Alon Levy , University of Washington, USA
    Dan Suciu , AT&T Labs, USA

Status of this document

This document is a submission to the World Wide Web Consortium (see Submission Request, W3C Staff Comment). It is intended for review and comment by W3C members and is subject to change.

This document is a NOTE made available by the W3 Consortium for discussion only. This indicates no endorsement of its content, nor that the Consortium has, is, or will be allocating any resources to the issues addressed by the NOTE.

Abstract

XML is a new standard that supports data exchange on the World-Wide Web. It is likely to become as important and as widely used as HTML.

The availability of large amounts of data on the Web raises several issues that the XML standard does not address. In particular, what techniques and tools should exist for extracting data from large XML documents, for translating XML data between different ontologies (DTD's), for integrating XML data from multiple XML sources, and for transporting large amounts of XML data to clients or for sending queries to XML sources.

We propose a query language for XML, called XML-QL, as one possible answer to these questions. The language has a SELECT-WHERE construct, like SQL, and borrows features of query languages recently developed by the database research community for semistructured data.

Table of Contents

Introduction

The SGML working group of the W3 Consortium has proposed a new standard, called XML (eXtensible Markup Language), which is a subset of SGML. The goal of XML is to provide many of SGML's benefits not available in HTML and to provide them in a language that is easier to learn and use than complete SGML. These benefits include arbitrary extension of a document's tags and attributes, support for documents with complex structure, and validation of document structure with respect to an optional document-structure grammar, called a *Document Type Descriptor* (DTD). Inherited from SGML, a DTD specifies what elements may occur and how the elements may nest in an XML document that conforms to the DTD.

There are many potential applications of XML. One important application is interchange of electronic data (EDI) between two or more data sources on the Web. This problem is increasingly important as more businesses choose to provide access to their databases and to exchange data with related businesses and organizations. In this note, we focus on XML's application to EDI. Specifically, we take a database view, as opposed to document view, of XML: we consider an XML document to be a database and a DTD to be a database *schema*.

One of XML's benefits is its simplicity. An XML document is a sequence of elements, each consisting of free text and/or other elements. The only restriction is that element tags must match, e.g., each <ADDRESS> must have a matching </ADDRESS>, and must nest properly. An XML document that has matching and properly nested tags is called *well-formed*. The elements in XML loosely correspond to *objects* in object-oriented or object-relational databases. For example, a <PERSON> ... </PERSON> would correspond to an object of type `class PERSON { ... }`. Nested XML elements correspond to an object's *fields*, e.g., <NAME>, <PHONE> and <ADDRESS> elements in <PERSON> would correspond to the `NAME, PHONE,` and `ADDRESS` fields of a `PERSON` object.

This simplicity allows users to produce XML data with complex structure without having to first define a schema. In EDI applications, this is particularly important, because the data's schema may evolve over time. It can be useful, however, to have some specification of XML data's structure, especially for a user community to define its own ontology for data exchange; in this case, DTDs can be used to specify the data's known structure. While DTDs are similar to schemas in object-oriented or object-relational databases, they are less restrictive and permit more variation in the data. For example, DTD's can specify that some fields are optional and that others may occur multiple times, and DTDs do not require that the type of a reference (IDREF) be specified.

Given its flexibility, its likely that XML will facilitate exchange of huge amounts of data on the Web, just as HTML enabled vast numbers of documents. Dozens of applications of XML already exist, including a Chemical Markup Language for exchanging data about molecules and the Open Financial Exchange for exchanging financial data between banks or banks and customers. However, the availability of huge amounts of XML data poses several technical questions that the XML standard does not address. In particular,

- How will data be extracted from large XML documents?
- How will XML data be exchanged, e.g., by shipping XML documents or by shipping queries?
- How will XML data be exchanged between user communities using different but related ontologies (or DTD's)?
- How will XML data from multiple XML sources be integrated?

Data extraction, transformation, and integration are all well-understood database problems. Their solutions often rely on a *query language*, either relational (SQL) or object-oriented (OQL). These query languages do not apply immediately to XML, because the XML data differs from traditional relational or object-oriented data. XML data, however, is very similar to a data model recently studied in the research community: the *semistructured data* model. Several research query languages have been designed and implemented for semistructured data.

In this note, we propose a *query language* for XML data, called **XML-QL**, to address some of the above questions. XML-QL can express *queries*, which extract pieces of data from XML documents, as well as *transformations*, which, for example, can map XML data between DTDs and can integrate XML data from different sources.

Throughout this document we use the term *data* as in *database* or in *electronic data interchange*, and not as in XML Data. We mention here that XML-QL is different from XSL, which is intended primarily for specifying style and layout of XML documents. Although XML-QL shares some functionalities with XSL, XML-QL supports more data-intensive operations, such as joins and aggregates, and has better support for constructing new XML data, which is required by transformations.

We believe that there is a very essential need for a standard query language for XML. We go ahead by presenting XML-QL as our proposal. This proposal has already created some interest in a large community. We are proposing to use the current proposal and its current group of authors as the seed to establish a working group to develop a language that could gain even wider support.

This document introduces XML-QL through example queries. The examples show how XML-QL can extract data from XML documents and how it can construct new XML data. We then introduce XML-QL's underlying *data model*. All query languages have an underlying data model that abstracts away from the physical representation of the data; for example relational query languages operate on relations and object-oriented query languages on objects. Therefore, it is necessary to define precisely a data model for XML. For XML, a variant of the semistructured data model is appropriate. Given this model, we then describe more advanced features of XML-QL, such as transforming and integrating XML data. Finally, we provide XML-QL's full grammar.

In designing XML-QL we applied existing research in the design and implementation of query languages for semistructured data: see, e.g., [1], [2], and [3]. Tutorials describing some of the work on semistructured data can be found in [4] and [5].

Examples in XML-QL

We introduce XML-QL through examples. The simplest XML-QL queries extract data from an XML document. Our example XML input is in `www.a.b.c/bib.xml`, and we assume that it contains *bibliography* entries that conform to the following DTD:

```
<!ELEMENT book (author+, title, publisher)>
<!ATTLIST book year CDATA>
<!ELEMENT article (author+, title, year?, (shortversion|longversion))>
<!ATTLIST article type CDATA>
<!ELEMENT publisher (name, address)>
<!ELEMENT author (firstname?, lastname)>
```

This DTD specifies that a `book` element contains one or more `author` elements, one `title`, and one `publisher` element and has a `year` attribute. An `article` is similar, but its `year` element is optional, it omits the publisher, and it contains one `shortversion` or `longversion` element. An `article` also contains a `type` attribute. A `publisher` contains `name` and `address` elements, and an `author` contains an optional `firstname` and one required `lastname` We assume that name, address, firstname, and lastname are all CDATA, i.e., string values.

Matching Data Using Patterns. XML-QL uses *element patterns* to match data in an XML document. This example produces all authors of books whose publisher is Addison-Wesley in the XML document at `www.a.b.c/bib.xml`. Any URI (uniform resource identifier) that represents an XML-data source may appear on the right-hand side of `IN`.

```
WHERE <book>
        <publisher><name>Addison-Wesley</name></publisher>
        <title> $t</title>
        <author> $a</author>
      </book> IN "www.a.b.c/bib.xml"
CONSTRUCT $a
```

Informally, this query matches every `<book>` element in the XML document `www.a.b.c/bib.xml` that has at least one `<title>` element, one `<author>` element, and one `<publisher>` element whose `<name>` element is equal to `Addison-Wesley`. For each such match, it binds the variables `t` and `a` to *every* title and author pair. The result is the list of authors bound to `a`. Note that variable names are preceded by `$` to distinguish them from string literals in the XML document (like `Addison-Wesley`).

For convenience, we abbreviate </element> by </>. This abbreviation is a relaxation of the XML syntax that will be quite handy later, when we introduce tag variables and regular expressions over tags. Thus the query above can be rewritten as:

```
WHERE <book>
        <publisher><name>Addison-Wesley</></>
        <title> $t</>
        <author> $a</>
      </> IN "www.a.b.c/bib.xml"
CONSTRUCT $a
```

Constructing XML Data. The query above returns a list of all authors (`<firstname>`,`<lastname>` pairs) from the input document. Often it is useful to construct new XML data in the result (i.e., data that did not exist in the input document). The following query returns both `<author>` and `<title>`, and groups them in a new `<result>` element:

```
WHERE <book>
        <publisher><name>Addison-Wesley</></>
        <title> $t</>
        <author> $a</>
      </> IN "www.a.b.c/bib.xml"
CONSTRUCT <result>
```

```
              <author> $a</>
              <title> $t</>
           </>
```

For example, consider the following XML data:

```
<bib>
  <book year="1995">
     <!-- A good introductory text -->
     <title> An Introduction to Database Systems </title>
     <author> <lastname> Date </lastname> </author>
     <publisher> <name> Addison-Wesley </name >  </publisher>
  </book>

  <book year="1998">
     <title> Foundation for Object/Relational Databases: The Third Manifesto </t
     <author> <lastname> Date </lastname> </author>
     <author> <lastname> Darwen </lastname> </author>
     <publisher> <name> Addison-Wesley </name >  </publisher>
  </book>
</bib>
```

When applied to the example data, our example query would produce the following result:

```
<result>
   <author> <lastname> Date </lastname> </author>
   <title> An Introduction to Database Systems </title>
</result>

<result>
   <author> <lastname> Date </lastname> </author>
   <title> Foundation for Object/Relational Databases: The Third Manifesto </tit
</result>

<result>
   <author> <lastname> Darwen </lastname> </author>
   <title> Foundation for Object/Relational Databases: The Third Manifesto </tit
</result>
```

Grouping with Nested Queries. The query above ungroups the authors of a book, i.e., different authors of the same book appear in different `<result>` elements. To group results by book title, we use a nested query, which produces one result for each title and contains a list of all its authors:

```
WHERE <book > $p</> IN "www.a.b.c/bib.xml",
      <title > $t</>,
      <publisher><name>Addison-Wesley</>> IN $p
CONSTRUCT <result>
             <title> $t </>
             WHERE  <author> $a </> IN $p
             CONSTRUCT <author> $a</>
          </>
```

Notice that the pattern beginning with `<book>` has been unnested, so that we can bind `$p` to the element's content. Once a content variable has been bound, it may appear on the right-hand side of any `IN` expression in the same `WHERE` expression or in any nested `CONSTRUCT` expressions.

Since binding the content of an element and matching patterns in the bound element is a common idiom, we introduce a syntactic shorthand that preserves the original nesting. The `CONTENT_AS $p` following a pattern binds the content of the matching element to the variable `$p`:

```
WHERE <book>
        <title> $t </>
        <publisher><name>Addison-Wesley </> </>
      </> CONTENT_AS $p IN "www.a.b.c/bib.xml"
CONSTRUCT <result><title> $t </>
                  WHERE  <author> $a</> IN $p
                  CONSTRUCT <author> $a</>
          </>
```

On the example data, the query would produce:

```
<result>
   <title> An Introduction to Database Systems </title>
   <author> <lastname> Date </lastname> </author>
</result>

<result>
   <title> Foundation for Object/Relational Databases: The Third Manifesto </tit
   <author> <lastname> Date </lastname> </author>
   <author> <lastname> Darwen </lastname> </author>
</result>
```

Joining Elements by Value. XML-QL can express ‘‘joins” by matching two or more elements that contain the same value. For example, the following query retrieves all articles that have at least one author who has also written a book since 1995. Here, we assume that an author uses the same first and last names (denoted by the variables `$f` and `$l`) for both articles and books.

```
WHERE <article>
        <author>
          <firstname> $f </> // firstname $f
          <lastname> $l </>  // lastname $l
        </>
      </> CONTENT_AS $a IN "www.a.b.c/bib.xml"

      <book year=$y>
        <author>
          <firstname> $f </> // join on same firstname $f
          <lastname> $l </>  // join on same lastname $l
        </>
      </> IN "www.a.b.c/bib.xml",
      y > 1995
CONSTRUCT <article> $a </>
```

The query above uses another common idiom: the first `CONSTRUCT` expression produces the same element that occurs in the input element. To avoid recreating an element, we can use the `ELEMENT_AS $e` expression, which binds `$e` to the complete preceding element:

```
WHERE <article>
        <author>
          <firstname> $f</> // firstname $f
          <lastname> $l</>  // lastname $l
        </>
      </> ELEMENT_AS $e IN "www.a.b.c/bib.xml"
...
CONSTRUCT $e
```

A Data Model for XML

The example queries above make implicit assumptions about how the XML data is modeled. For example, patterns that match sibling elements can appear in any order, i.e., this pattern expression:

```
<author> <firstname> $f</> <lastname> $l</> </author>
```

is equivalent to:

```
<author> <lastname> $l</> <firstname> $f</>  </author>
```

The data model does not require sibling elements to be ordered even though they may have a fixed order in the XML document. These assumptions are defined formally by the data model. To describe XML-QL's data model, consider the example `book` elements.

The first record has three elements (one title, one author, and one publisher) and the second record has two authors. The XML document, however, contains additional information that is not relevant to the data itself, such as,

- the comment at the beginning of the first book element,
- the fact that that title precedes authors, and
- the fact that firstname precedes lastname in every author.

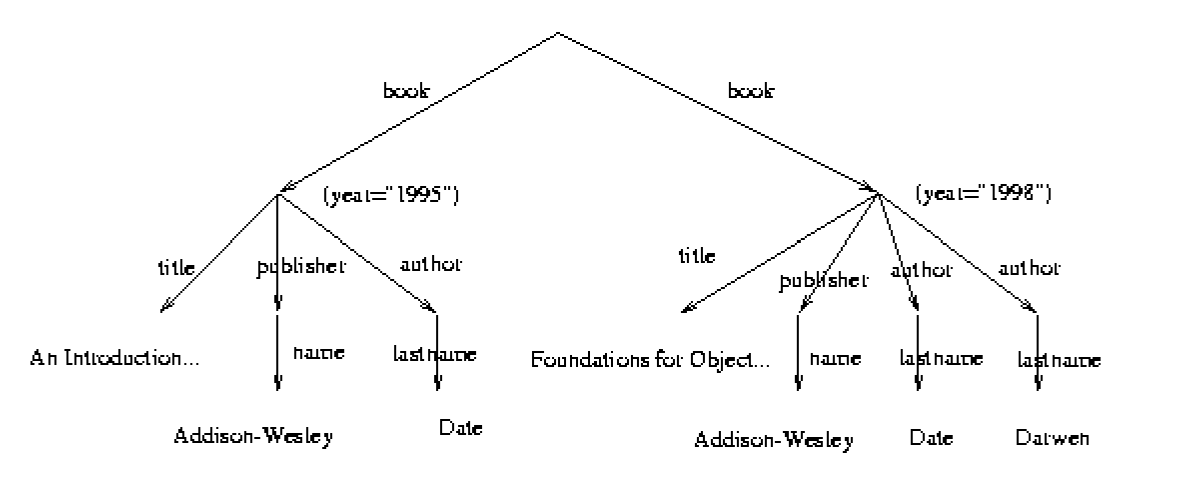
We assume that a distinction can be made between information that is intrinsic to the data and information, such as document layout specification, which is not. We propose a model that expresses only information that is essential to the data. We made some compromises for reasons having to do with the semantics and efficiency of the query language.

The proposed XML data model is a variation of the semistructured data model[5]. An XML document is modeled by an *XML graph*. A formal definition follows.

Definition. An *XML Graph* consists of:

- A graph, G, in which each vertex is represented by a unique string called an *object identifier* (OID),
- G's edges are labeled with element tag identifiers,
- G's nodes are labeled with sets of attribute-value pairs,
- G's leaves are labeled with values (strings), and
- G has a distinguished node called the *root*.

For example, the example data would be represented by the graph below. Attributes are associated with nodes and elements are represented by edge labels. We use the terms *node* and *object* interchangeably. We omit object identifiers to reduce clutter.



XML graph for example XML `book` elements

The model allows several edges between the same two nodes, but they must carry distinct labels: in other words we collapse two edges with the same source, same destination, and same label.

Element Identity, IDs, and ID References

To support element sharing, XML reserves an attribute of type ID (often called ID), which allows a unique key to be associated with an element. An attribute of type IDREF allows an element to refer to another element with the designated key, and one of type IDREFS may refer to multiple elements. The data model treats attributes of these types differently from all others. For example, assume attributes `ID` and `author` of types `ID` and `IDREFS` respectively:

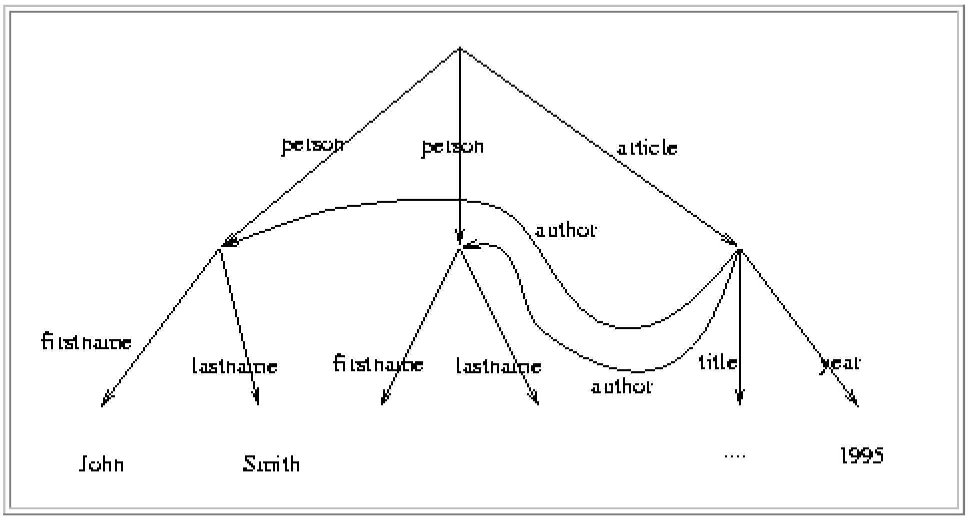
```
<!ATTLIST person  ID ID #REQUIRED>
<!ATTLIST article author IDREFS #IMPLIED>
```

Then in the XML fragment below the first element associates the keys `o123` and `o234` with the two `<person>` elements, and the defines an `<article>` element whose authors refer to the these persons.

```
<person ID="o123">
    <firstname>John</firstname>
    <lastname>Smith<lastname>
</person>
<person ID="o234">
    . . .
</person>
<article author="o123 o234">
    <title> ... </title>
    <year> 1995 </year>
</article>
```

In an XML graph, every node has a unique object identifier (OID), which is the ID attribute associated with the corresponding element or is a generated OID, if no ID attribute exists. Elements can refer to other elements using IDREF or IDREFS attributes.

Unlike all other attributes, which are name-value pairs associated with nodes, an IDREF attribute is represented by an edge from the referring element to the referenced element; the edge is labeled by the attribute name. ID attributes are also treated specially: they become the node's oid. For example, the elements above are represented by the XML graph:



At this point is makes sens to blur the distinction between IDREF(S) attributes and elements, since both logically ‘‘point to” other objects. This choice also allows users to write simple edge traversal (a.k.a. ‘‘pointer chasing”) queries, such as:

```
WHERE  <article><author><lastname> $n</></></> IN "abc.xml"
```

It is also possible to stay within the XML syntax and write explicit join expressions on ID values to access referenced objects. For example, the following query produces all lastname, title pairs by joining the author element’s `IDREF` attribute value with the person element’s `ID` attribute value.

```
WHERE  <article author=$i>
         <title>  </> ELEMENT_AS $t
       </>,
       <person ID=$i>
         <lastname>  </> ELEMENT_AS $l
       </>
CONSTRUCT <result> $t $l</>
```

We note here that the idiom `<title></> ELEMENT_AS $t` binds `$t` to a `<title>` element with *arbitrary* contents. The element expression `<title/>` matches a `<title>` element with *empty* contents.

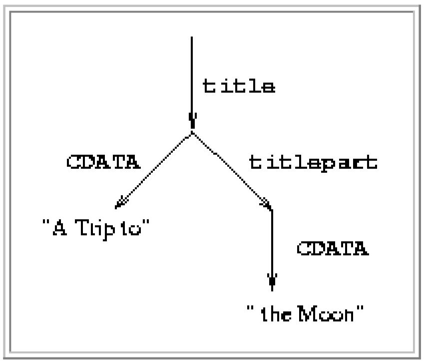
Scalar Values. Only leaf nodes in the XML graph may contain values, and they may have only one value. As a consequence, the XML fragment:

```
<title>A Trip to <titlepart> the Moon </titlepart></title>
```

cannot be represented directly in the data model. However, the following fragment:

```
<title><CDATA> A Trip to </CDATA><titlepart><CDATA> the Moon</CDATA></titlepart><
```

is modelded by the XML graph:



The value of a leaf node is its oid.

Element Order. So far we have described an unordered XML data model, i.e., in which the order in which XML elements appear in a containing element is ignored. This choice simplifies the data model, the query language, and allows order-independent optimizations in the query processor.

In fact, XML-QL supports two distinct data models: an unordered one and an ordered one. An ordered XML graph is like an unordered one, but includes, for each node, a total order on its successors. Under the ordered data model, XML-QL provides additional constructs that support element order. The price for an ordered model is a more complex (and ocasionally ad-hoc) semantics of the query language, and potentially less efficient processors. Although the semantics of the unordered data model is simpler, we still must choose an order when an unordered XML graph is rendered into a document, for example, if it must conform to a DTD. We discuss this mapping next.

Mapping XML Graphs into XML Data. XML graphs may result from parsing an XML document or from querying or transforming an existing XML graph. In general, XML graphs do not have a unique representation as an XML document, because

- element order is unspecified (in the unordered model), i.e., we have to choose an order for the outgoing edges of each graph node, and
- sharing of nodes, i.e., a node may have several incoming edges.

Both issues can be handled outside the data model and query processing when the graph is emitted as an XML document. Given a DTD for a query's result, an XML graph can be emitted as XML data that conforms to that DTD and therefore all elements will be ordered. We do not address algorithmic issues in this proposal.

Advanced Examples in XML-QL

The following examples introduce advanced features of XML-QL. These include *tag variables*, which allow a variable to be bound to any edge (element) in an XML graph, and *regular path expressions*, which can specify arbitrary paths through an XML graph.

Tag Variables. XML-QL supports querying of element tags using tag variables. For example, the following query finds all publications published in 1995 in which `Smith` is either an author, or an editor:

```
WHERE <$p>
        <title> $t </title>
        <year>1995</>
        <$e> Smith </>
```

```
      </> IN "www.a.b.c/bib.xml",
      $e IN {author, editor}
CONSTRUCT <$p>
             <title> $t </title>
             <$e> Smith </>
          </>
```

`p` is a tag variable that is bound to any top-level tag, e.g., <book>, <article>, etc. Note that the `CONSTRUCT` clause constructs a result with the same tag name. `e` is also a tag variable; the query constrains it to be bound to one of <author> or <editor>.

Regular Path Expressions. XML data can specify nested and cyclic structures, such as trees, directed-acyclic graphs, and arbitrary graphs. Element sharing is specified using element IDs and IDREFs as described above.

Querying such structures often requires traversing arbitrary paths through XML elements. For example, consider the following DTD that defines the self-recursive element `part`:

```
<!ELEMENT part (name brand part*)>
<!ELEMENT name  CDATA>
<!ELEMENT brand CDATA>
```

Any `part` element can contain other nested `part` elements to an arbitrary depth. To query such a structure, XML-QL provides *regular path expressions*, which can specify element paths of arbitrary depth. For example, the following query produces the name of every `part` element that contains a `brand` element equal to ‘‘Ford”, regardless of the nesting level at which `r` occurs.

```
WHERE <part*> <name>$r</> <brand>Ford</> </> IN "www.a.b.c/bib.xml"
CONSTRUCT <result>$r</>
```

Here `part*` is a *regular path expression*, and matches any sequence of edges, all of which are labeled `part`. The pattern:

```
<part*> <name>$r</> <brand>Ford</> </>
```

is equivalent to the following infinite sequence of patterns:

```
<name>$r</> <brand>Ford</>
<part> <name>$r</> <brand>Ford</> </>
<part> <part> <name>$r</> <brand>Ford</> </> </>
<part> <part> <part> <name>$r</> <brand>Ford</> </> </> </>
  .  .  .
```

The wildcard `$` matches any tag and can appear wherever a tag is permitted. For example, this query is like the one above, but matches *any* sequence of elements, not just `parts`:

```
WHERE <$*> <name>$r</> <brand>Ford</> </> IN "www.a.b.c/bib.xml",
CONSTRUCT <result>$r</>
```

We abbreviate `$*` by `*`. Also, `.` denotes concatenation of regular expression, hence a pattern looking only for the brand `Ford`, no matter how deep, would be written as:

```
<*.brand>Ford</>
```

The notation `*.brand` is reminiscent of Unix-like wild-cards in file names: it means "anything followed by `brand`".

XML-QL's regular path expressions provide the alternation (`|`), concatenation (`.`), and Kleene-star (`*`) operators, similar to those used in regular expressions. A regular path expression is permitted wherever XML permits an element. For example, this query considers any sequence of nested `<part>` elements and produces any nested `<subpart>` element or any `<piece>` element contained in a nested `<component>` element.

```
WHERE <part+.(subpart|component.piece)>$r</> IN "www.a.b.c/parts.xml"
CONSTRUCT <result> $r</>
```

The + operator means ‘‘one or more”: `part+` is equivalent to `part.part*`.

Tag variables and regular path expressions make it possible to write a query that can be applied to two or more XML data sources that have similar but not identical DTDs, because these expressions can handle the variability of the data's format from multiple sources.

Transforming XML Data An important use of XML-QL is transforming XML data. For example, we may want to translate data from one DTD (a.k.a. ontology) into another. To illustrate, assume that in addition to our bibliography DTD, we have a DTD that defines a *person*:

```
<!ELEMENT person (lastname, firstname, address?, phone?, publicationtitle*)>
```

We can write a query to transform data that conforms to the bibliography DTD into data that conforms to the person DTD. This query extracts authors from the bibliography database and transforms them into `<person>` elements conforming to the person DTD:

```
WHERE <$> <author> <firstname> $fn </>
                   <lastname> $ln </>
          </>
          <title> $t </>
      </> IN "www.a.b.c/bib.xml",
CONSTRUCT <person ID=PersonID($fn, $ln)>
              <firstname> $fn </>
              <lastname> $ln </>
              <publicationtitle> $t </>
          </>
```

This query uses object identifiers (OID's) and *Skolem functions* to control how the result is produced and grouped. Whenever a `<person>` is produced, its associated OID is `PersonID($fn,$ln)`. `PersonID` is a Skolem function, and its purpose is to generate a new identifier for every distinct values of `$fn,$ln`. If, at a later time, the query binds `$fn` and `$ln` to the same values again (e.g., by finding another publication by the same author), then the query will not create another `<person>`, but *append* information to the existing `<person>` element. Thus, all `<publicationtitle>`s of that author will be grouped in the same `<person>` element. The `<firstname>` and `<lastname>` attributes are not duplicated since they lead to the same values (see here and here). We cannot fill in the `<phone>` and `<address>`, since we do not have them in the bibliography data.

Integrating data from different XML sources In XML-QL, we can query several sources simultaneously and produce an integrated view of their data. In this example, we produce all pairs of names and social-security numbers by querying the sources `www.a.b.c/data.xml` and `www.irs.gov/taxpayers.xml`. The two sources are joined on the social-security number, which is bound to `ssn` in both expressions. The result contains only those elements that have both a `name` element in the first source and an `income` element in the second source.

```
WHERE <person>
        <name></> ELEMENT_AS $n
        <ssn> $ssn</>
      </> IN "www.a.b.c/data.xml",

      <taxpayer>
        <ssn> $ssn</>
        <income></> ELEMENT_AS $i
      </> IN "www.irs.gov/taxpayers.xml"
CONSTRUCT <result> $n $i </>
```

Alternatively, we can use Skolem functions and write the query in two fragments:

```
{ WHERE <person>
          <name> </> ELEMENT_AS $n
          <ssn> $ssn </>
        </> IN "www.a.b.c/data.xml"
  CONSTRUCT <result ID=SSNID($ssn)> $n </>
}
{ WHERE <taxpayer>
          <ssn> $ssn </>
          <income> </> ELEMENT_AS $i
        </> IN "www.irs.gov/taxpayers.xml"
  CONSTRUCT <result ID=SSNID($ssn)> $i </>
}
```

XML-QL queries are structured into *blocks* and *subblocks*: the latter are enclosed in braces (`{ ... }`), and their semantics is related to that of semi-joins in relational databases. In our example the root block is empty and there are two subblocks. In the first subblock, all persons' names are produced. Each result has a unique OID given by that person's SSN. In the second subblock, persons with their income are produced. Wherever there is a match with a previously generated OID, the `<income>` will be appended to the object rather than included in a new `<person>`. This query is not equivalent to the previous: it contains, in addition to the previous query, all persons which are not taxpayers, and vice versa. This is the *outer join* of `"www.a.b.c/data.xml"` and `"www.irs.gov/taxpayers.xml"`.

Query blocks are a versatile and powerful feature. The following query retrieves all titles published in 1995 in the publication database; in addition, it retrieves the month of publication for journal articles, and the publisher for books.

```
WHERE <$e> <title> $t </>
           <year> 1995 </> </> CONTENT_A $p
                IN "www.a.b.c/bib.xml"
CONSTRUCT <result ID=ResultID($p)> <title> $t </> </>
  { WHERE $e = "journal-paper",
          <month> $m </> IN $p
    CONSTRUCT <result ID=ResultID($p)> <month> $m </> </>
  }
  { WHERE $e = "book",
          <publisher>$q </> IN $p
    CONSTRUCT <result ID=ResultID($p)> <publisher>$q </> </>
  }
```

The outer block runs over all publications in 1995, and produces a `result` element for their title. The first inner block checks if `e` is `journal-paper` and has a `month` tag: if so, it adds the `month` element to the *same* publication (this is controlled by the Skolem function). Finally, the second inner block checks if `e` is a book and has a `publisher`, and adds that `publisher` element to the output. **Order** Under the ordered data model, XML-QL is extended as follows:

- The patterns in the `WHERE` clause are now interpreted as ordered. For example, the pattern

  ```
  <a> <b> $x </b> <c> $y </c> </a>
  ```

  will match the XML data :

  ```
   <a> <b> string1 </b> <c> string2 </c> </a>
  ```

  but not the data :

  ```
   <a> <c> string2 </c> <b> string1 </b> </a>
  ```

- Variable bindings in the `WHERE` are now ordered. Continuing with the example above, if the XML data is:

  ```
  <a>
    <b> string1 </b>
    <c> string2 </c>
    <b> string3 </b>
    <c> string4 </c>
  </a>
  ```

  then the variables will be bound in the following order:

  ```
  $x      $y
  string1 string2
  string1 string4
  string3 string4
  ```

  (Note that `string3, string2` is not a valid binding.) In more complex cases, when the data graph has sharing, two bindings may be in different orders due to different paths to those nodes. XML-QL preprocesses the data graph and enumerates the vertices in a depth-first graph traversal. Variable bindings are ordered by the node numbers, eliminating duplicate bindings in the process. In the case of simple queries this amounts to preserving the order of the input graph.
- XML-QL supports order variables, which extract the index of some element. For example, the patterns:

  ```
  <a> ... </>[$i]
  <$x> ... </>[$j]
  ```

  bind $i or $j to an integer 0, 1, 2, ... that represents the index in the total order of the edges. For example, the following query retrieves all persons whose `lastname` precedes the `firstname`:

  ```
  WHERE <person> $p </> in "www.a.b.c/people.xml",
           <firstname> $x </>[$i] in $p,
           <lastname> $y </>[$j] in $p,
           $j < $i
  CONSTRUCT <person> $p </>
  ```

- An optional `ORDER-BY` clause specifies element order in the output. For example, the following query retrieves publication titles ordered by year and month and preserves the order in the original document for publications within the same year/month:

```
WHERE <pub> &p </> in "www.a.b.c/bib.xml",
      <title> $t </> in $p,
      <year> $y </> in $p
      <month> $z </> in $p
ORDER-BY $y,$z
CONSTRUCT $t
```

For a more complex example, the following query reverses the order of all `authors` in a publication:

```
WHERE <pub> $p </> in "www.a.b.c/bib.xml"
CONSTRUCT <pub> WHERE <author> $a </>[$i] in $p
                ORDER-BY $i DESCENDING
                CONSTRUCT <author> $a </>

                WHERE <$e> $v </> in $p,
                      $e != "author"
                CONSTRUCT <$e> $v </>
          </pub>
```

Embedding queries in data XML-QL can be embedded in XML. For example the following query constructs two disinct parts: `articles` and `books`.

```
<result>
   <articles>
      WHERE <article< <title> $t </>
                       <year> $y </>
            </> in "www.a.b.c/bib.xml",
            $y > 1995
      CONSTRUCT <title>  $t </>
   </>
   <books>
      WHERE <book< <title> $t </>
                       <year> $y </>
            </> in "www.a.b.c/bib.xml",
            $y > 1995
      CONSTRUCT <title>  $t </>
   </>
</>
```

Function definitions and DTD's The following code defines a function expecting two input XML documents:

```
FUNCTION findDeclaredIncomes($Taxpayers, $Employees)
   WHERE <taxpayer> <ssn> $s </>
                    <income> $x </> </> IN $Taxpayers,
         <employee> <ssn> $s </>
                    <name> $n </> </> IN $Employees
   CONSTRUCT <result> <name> $n </>
                      <income> $x </> </>
END
```

The function can be invoked like in:

```
findDelcaredIncomes("www.irs.gov/taxpayers.xml", "www.a.b.c/employees.xml")
```

A more complex issue is related to parameter's DTDs. The function could be written like:

```
FUNCTION findDeclaredIncomes($Taxpayers:"www.irs.gov/tp.dtd", $Employees:"www.emp
   WHERE ... CONSTRUCT ...
```

The query processors would (a) check at compile time that the query indeed produces a result conforming to `"myresult.dtd"` , given that the inputs conform to `"tp.dtd"` and `employees.dtd` respectively, and (b) check at run-time that the input documents conform to `"tp.dtd"` and `employees.dtd` respectively.

Extensions and Open Issues

The following features are not included in the current version of this document, but are planned for future versions.

Entities. We will recognize entity references, e.g., `&M`, in XML-QL queries. Entities are defined in DTDs, for example, the entity `M` has the string value ``Mary''.

```
<!ENTITY % M "Mary">
```

User-defined predicates. User-defined predicates would allow users to apply arbitrary predicates, written in any external language such as Java or Perl, to element and tag values.

String regular expressions. The following query produces any person whose first name is equal to the replacement text of the entity `M`, whose last name matches the regular expression 'Fern*', and whose address satisfies the external predicate defined by `ATTAddress`.

```
WHERE  <person>
         <firstname>&M</>
         <lastname>'Fern*'</>
         <address>$a</>
       </> ELEMENT_AS $x in abc.xml,
       ATTAddress($a)
CONSTRUCT $x
```

Name spaces This would allow tag values to be qualified by XML name-space prefixes. XML name spaces support creation of data whose structure is defined by multiple schemas. For example, assume addresses can be defined according to multiple DTDs, e.g., one for US, one for Europe, etc. We can qualify an element name by the name space in which it is defined. This query matches only those persons whose addresses are in the `US` namespace.

```
WHERE  <person>
         <lastname>'Fern*'</>
         <US:address>$a</>
       </> ELEMENT_AS $x in abc.xml,
       ATTAddress($a)
CONSTRUCT $x
```

Aggregates We expect to support aggregates like those provided in SQL. For example, the following query retrieves the lowest and highest price for each publisher:

```
WHERE <book>
        <publisher> </> ELEMENT_AS $p
```

```
        <price> $x </>
      </>
GROUP-BY $p
CONSTRUCT <result ID=f($p)> $p
            <lowest-price> $min($x) </>
            <highest-price> $max($x) </>
       </>
```

Semijoins Subblocks offer only a limited form of semijoins. A more general semijoin construct would connect optional parts in the `WHERE` clause with optional components in the `CONSTRUCT` part: this can be currently expressed in a rather cumbersome way, either as nested `WHERE-CONSTRUCT` queries or with subblocks and Skolem functions.

XML Syntax Some applications may need to process XML-QL programs as data and could benefit from an encoding of XML-QL queries as XML documents. Other XML-related standards have done precisely that (e.g. XML Data, XSL). We view this as an orthogonal effort to the design of XML-QL.

Extensions to other XML-related standards Here we have in mind RDF(Resource Description Framework), XPointer, XML-Link.

Summary

Given the expectation that XML data will become as prevalent as HTML documents, we anticipate an increased demand for engines that can query both XML data's content *and* its structure. We also expect that XML data will become a primary means for electronic data interchange on the Web, and therefore high-level support for tasks such as integrating data from multiple sources and transforming data between ontologies will also be necessary.

XML-QL provides support for querying, constructing, transforming, and integrating XML data. XML data is very similar to *semistructured data*, which has been proposed in the database research community as a data model for data sources that have irregular or rapidly evolving structure. We have designed XML-QL based on our past experience with other query languages (UnQL and Strudel) for semistructured data. We have implemented an interpretor for XML-QL and currently are experimenting with the language.

Acknowldegments The authors would like to thank Serge Abiteboul, Catriel Beeri, Peter Buneman, Stefano Ceri, Ora Lassila, Alberto Mendelzon, and Yannis Papakonstantinou for their comments.

Bibliography

[1] S. Abiteboul and D. Quass and J. McHugh and J. Widom and J. Wiener, *The Lorel Query Language for Semistructured Data*, International Journal on Digital Libraries, vol. 1, no. 1, pp. 68-88, 4/1997.

[2] Peter Buneman, Susan Davidson, Gerd Hillebrand, Dan Suciu *A query language and optimization techniques for unstructured data*, Proceedings of ACM-SIGMOD International Conference on Management of Data", 1996.

[3] Mary Fernandez and Daniela Florescu and Alon Levy and Dan Suciu, *A Query Language for a Web-Site Management System,*, SIGMOD Record, vol. 26, no. 3, pp. 4-11, 9/1997.

[4] Serge Abiteboul, *Querying semistructured data*, Proceedings of the International Conference on Database Theory, 1997.

[5] Peter Buneman, *Tutorial: Semistructured data*, Proceedings of the ACM SIGMOD Symposium on Principles of Database Systems, 1997.

Appendix: Grammar for XML-QL

Note The grammar is still being developed, as the language evolves, and is incomplete in the current version of this document. Terminal symbols are shown in angular brackets and their lexical structure is not further specified.

XML-QL Grammar

```
           XML-QL ::= (Function | Query) <EOF>
         Function ::= 'FUNCTION' <FUN-ID> '(' (<VAR>(':' <DTD>)?)* ')' (':' <DTD>)?
                        Query
                      'END'
            Query ::= Element | Literal |  <VAR> | QueryBlock
          Element ::= StartTag  Query  EndTag
         StartTag ::= '<'(<ID>|<VAR>) SkolemID? Attribute* '>'
         SkolemID ::= <ID> '(' <VAR> (',' <VAR>)* ')'
        Attribute ::= <ID> '='  ('"' <STRING> '"'  |  <VAR> )
           EndTag ::= '<' / <ID>? '>'
          Literal ::= <STRING>
       QueryBlock ::= Where  Construct ('{' QueryBlock '}')*
            Where ::= 'WHERE' Condition (',' Condition)*
        Construct ::= OrderedBy? 'CONSTRUCT'  Query
        Condition ::= Pattern BindingAs*  'IN' DataSource  |  Predicate
          Pattern ::= StartTagPattern Pattern* EndTag
  StartTagPattern ::= '<' RegularExpression Attribute* '>'
RegularExpression ::= RegularExpression '*' |
                      RegularExpression '+' |
                      RegularExpression '.' RegularExpression |
                      RegularExpression '|' RegularExpression |
                      <VAR> |
                      <ID>
        BindingAs ::= 'ELEMENT_AS' <VAR>  | 'CONTENT_AS' <VAR>
        Predicate ::= Expression OpRel Expression
       Expression ::= <VAR> | <CONSTANT>
            OpRel ::= '<' | '<=' | '>' | '>=' | '=' | '!='
        OrderedBy ::= 'ORDERED-BY' <VAR>+
```

```
DataSource ::=  <VAR> | <URI> | <FUN-ID>(DataSource (',' DataSource)*)
```
