A Query Language for XML

Alin Deutsch
University of Pennsylvania
Philadelpha, PA
adeutsch@gradient.cis.upenn.edu

Mary Fernandez
AT&T Labs
Florham Park, NJ
mff@research.att.com

Daniela Florescu
INRIA
Le Chesnay cedex, France
Daniela.Florescu@inria.fr

Alon Levy
University of Washington
Seattle, WA
levy@cs.washington.edu

Dan Suciu
AT&T Labs
Florham Park, NJ
suciu@resesarch.att.com

Abstract

An important application of XML is the interchange of electronic data (EDI) between multiple data sources on the Web. As XML data proliferates on the Web, applications will need to integrate and aggregate data from multiple source and clean and transform data to facilitate exchange. Data extraction, conversion, transformation, and integration are all well-understood database problems, and their solutions rely on a *query language*. We present a query language for XML, called *XML-QL*, which we argue is suitable for performing the above tasks. XML-QL is a declarative, ''relational complete'' query language and is simple enough that it can be optimized. XML-QL can extract data from existing XML documents *and* construct new XML documents.
Keywords: XML, query languages, electronic-data interchange (EDI)

1. Introduction

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 user-defined tags, nested elements, and an optional validation of document structure with respect to a *Document Type Descriptor* (DTD).

One important application of XML is the interchange of electronic data (EDI) between two or more data sources on the Web. Electronic data is primarily intended for computer, not human, consumption. For example, search robots could integrate automatically information from related sources that publish their data in XML format, e.g., stock quotes from financial sites, sports scores from news sites; businesses could publish data about their products and services, and potential customers could compare and process this information automatically; and business partners could exchange internal operational data between their information systems on secure channels. New opportunities will arise for third parties to add value by integrating, transforming, cleaning, and aggregating XML data. In this paper, 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*.

EDI applications require tools that support the following tasks:

- extraction of data from large XML documents,
- conversion of data between relational or object-oriented databases and XML data,
- transformation of data from one DTD to a different DTD, and/or
- integration of multiple XML data sources.

Data extraction, conversion, transformation, and integration are all well-understood database problems. Their solutions rely on a *query language*, either relational (SQL) or object-oriented (OQL). We present a query language for XML, called *XML-QL*, which we argue is suitable for performing the above tasks. XML-QL has the following features:

- It is declarative.
- It is ''relational complete''; in particular, it can express joins.
- It is simple enough that known database techniques for query optimization, cost estimation, and query rewriting could be extended to XML-QL.
- It can extract data from existing XML documents *and* construct new XML documents.
- It can support both *ordered* and *unordered* views on an XML document.

An initial draft of the query language is a W3C note, `http://www.w3.org/TR/NOTE-xml-ql/`.

One salient question is why not adapt SQL or OQL to query XML. The answer is that XML data is fundamentally different than relational and object-oriented data, and therefore, neither SQL nor OQL is appropriate for XML. The key distinction between data in XML and data in traditional models is that is XML is not rigidly structured. In the relational and object-oriented models, every data instance has a *schema*, which is separate from and independent of the data. In XML, the schema exists *with* the data as tag names. For example, in the relational model, a schema might define the relation `person` with attribute names `name` and `address`, e.g., `person(name, address)`. An instance of this schema would contain tuples such as `("Smith", "Philadelphia")`. The relation and attribute names are separate from the data and are usually stored in a database catalog.

In XML, the schema information is stored with the data. Structured values are called *elements*. Attributes, or element names, are called *tags*, and elements may also have *attributes* whose values are always atomic. For instance,
`<person><name>Smith</name><address>Philadelphia</address></person>.` is well-formed XML. Thus, XML data is self-describing and can naturally model irregularities that cannot be modeled by relational or object-oriented data. For example, data items may have missing elements or multiple occurrences of the same element; elements may have atomic values in some data items and structured values in others; and collections of elements can have heterogeneous structure. Even XML data that has an associated DTD is self-describing (the schema is always stored with the data) and, except for restrictive forms of DTDs, may have all the irregularities described above. Most importantly, this flexibility is crucial for EDI applications.

Self-describing data has been considered recently in the database research community. Researchers have found this data to be fundamentally different from relational or object-oriented data, and called it *semistructured data.* Semistructured data is motivated by the problems of integrating heterogeneous data sources and modeling sources such as biological databases, Web data, and structured text documents, such as SGML and XML. Research on semistructured data has addressed data models, query-language design, query processing and optimization, schema languages, and schema extraction. The key observation in this paper is that XML data is an instance of semistructured data.

In designing XML-QL, we drew from other query languages for semistructured data [1, 2, 5]: tutorials describing some of the work on semistructured data can be found in [3] and [4]. XML-QL includes most features found in these languages, but it differs from all of them in several important respects. Specifically, this paper makes the following contributions:

- We propose a data model for XML data that extends the semistructured-data model with order. This extension is necessary for XML documents, which are ordered.
- We design a syntax for XML-QL that combines elements of the XML syntax with traditional query-language syntax.
- We propose a novel semantics for XML-QL to support order in the input and output data.
- We combine two powerful data-construction mechanisms, nested queries and Skolem functions, in a novel way.
- We illustrate that XML-QL can be used for the tasks it has been designed for, such as data extraction, transformation, and integration.

In Sec. 2, we introduce XML-QL through examples, and in Sec. 3 , we describe formally its underlying data models. We show in Sec. 4 how XML-QL can be used for the tasks of data extraction, translation, and integration. Sec. 5 describes future extensions and discusses open issues.

2. Examples in XML-QL

The simplest XML-QL queries extract data from an XML document. Our example XML input is in the document `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 `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. 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 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 produces an XML document that contains a list of (`<firstname>,<lastname>`) pairs or (`<lastname>`) elements 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 XML data in Figure 1:

```
<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</title>
     <author><lastname>Date </lastname></author>
     <author><lastname>Darwen </lastname></author>
     <publisher><name>Addison-Wesley </name >  </publisher>
  </book>
</bib>
```

Figure 1. Example bibliographic data

When applied to the example data in Figure 1., 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</title>
</result>
<result>
   <author><lastname>Darwen </lastname></author>
   <title>Foundation for Object/Relational Databases</title>
</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</title>
   <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 `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
```

3. A Data Model for XML

To better understand XML-QL, it is necessary to describe its data models. We describe an *unordered* data model in detail first and then an *ordered* data model.

Unordered Model. To understand XML-QL's data model, consider the example `book` elements in Figure 1. 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 directly 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 the 1995 book precedes the 1998 book.

This information is not always relevant to an EDI application that ignores comments and treats its data as unordered structured values. We assume that a distinction can be made between information that is intrinsic to the data and information, such as document layout specification that is not. Our unordered model ignores comments and relative order between elements, but preserves all other essential data. These compromises simplify XML-QL's semantics and can effect the efficiency of a query interpretor.

Definition. An unordered *XML Graph* consists of:

- A graph, G, in which each node is represented by a unique string called an *object identifier* (OID),

- G's edges are labeled with element tags,
- G's nodes are labeled with sets of attribute-value pairs,
- G's leaves are labeled with one string value, and
- G has a distinguished node called the *root*.

For example, the example data would be represented by the XML graph in Figure 2. 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.

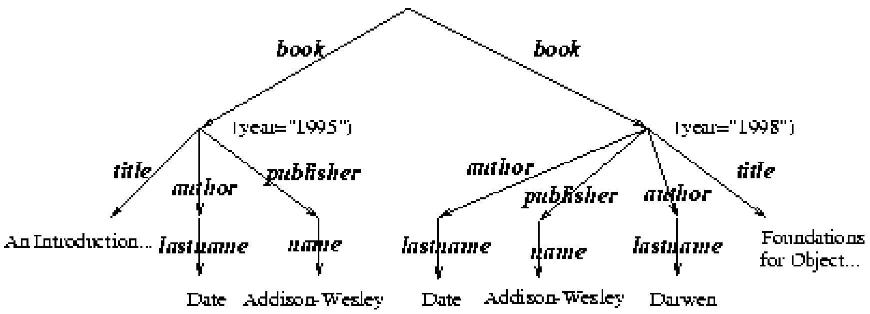

Figure 2: Unordered XML graph for example data

The model allows several edges between the same two nodes, but with the following restriction. A node cannot have two outgoing edges with the same labels and the same *values.* Here *value* means the string value in the case of a leaf node, or the oid in the case of a non-leaf node. Restated, this condition says that (1) between any two nodes there can be at most one edge with a given label, and (2) a node cannot have two leaf children with the same label and the same string value. The second condition means that in this data model, XML documents denote sets. For example, the XML graph for `<author> <lastname> Dates </lastname> <lastname> Dates </lastname> </author>` is isomorphic to the graph for `<author> <lastname> Dates </lastname> </author>`.

It is important to note that XML graphs are not only derived from XML documents, but are also generated by queries.

Ordered Model. An ordered XML graph is an XML graph in which there is a total order on all nodes in the graph. For graphs constructed from XML documents a natural order for nodes is their document order. Given a total order on nodes, we can enforce a local order on the outgoing edges of each node. In an ordered model, we may have arbitrarily many edges with the same source, same edge label, and same destination *value*. For example, the example data would be represented by the ordered graph in Figure 3. Nodes are labeled by their index (parenthesized integers) in the total node order and edge labels are labeled with their local order (bracketed integers). In Section 4.4, we discuss XML-QL's features that support the ordered model.

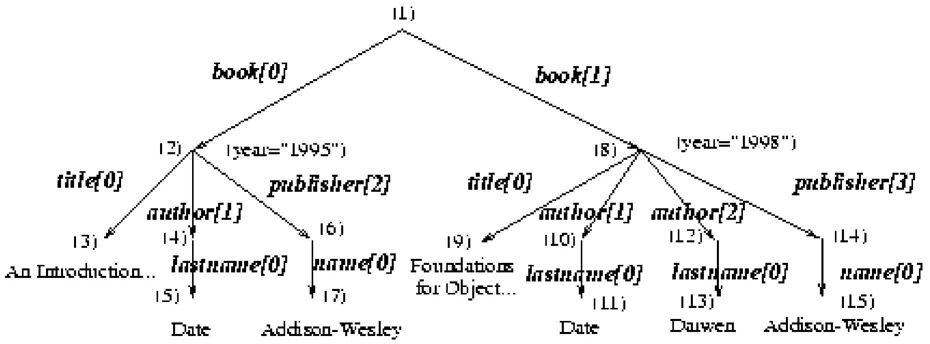

Figure 3: Ordered XML graph for example data.

Although XML documents are always ordered (e.g., DTDs may impose a particular order on elements), querying ordered data is harder than querying unordered data. The semantics of XML-QL is more complex on ordered data (see Sec.5), and a query optimizer will have fewer choices for generating good query-execution plans. For many EDI applications, element order is not significant, e.g., the tuples in a relational table can be represented by an unordered model without loss of information. In these cases, we can ignore order and use the subset of XML-QL that is order independent. For applications in which order is significant, e.g., any document-processing application must have access to section, paragraph, and sentence order, we can use the ordered model.

3.1 Element Identity, IDs, and ID References

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

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

For example, in the XML fragment below, the first element associates the keys `o123` and `o234` with the two <person> elements, and the second 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 by 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, because they become the node's OID. For example, the elements above are represented by the XML graph in Fig. 4:

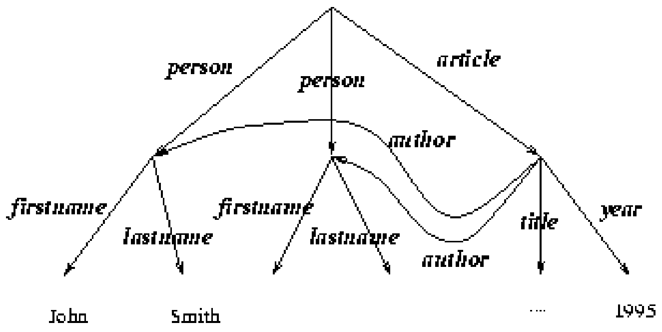

Figure 4: Representation of IDREF attributes in XML graph.

In this representation, it is makes sense to blur the distinction between attributes of type IDREF(S) and elements, because 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 write explicit join expressions on ID and IDREF 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
       </> IN "abc.xml"
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.

We note that although we choose to represent both elements and IDREF attributes by labeled edges it is possible, given an XML graph generated from an XML document, to recover the original document. To do this, we distinguish between the types lexically, by assuming that all attributes of type IDREF are distinct from all tags in the document; this restriction could be enforced by the DTD. Therefore, given a graph node with multiple incoming edges, we can distinguish between the node's unique element name and the (possibly) multiple IDREF attributes that refer to the node.

3.2 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 as an XML graph. In such cases, we introduce extra edges to enforce the invariant of one value per leaf node. The data above would be transformed into:

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

which is modeled by the XML graph:

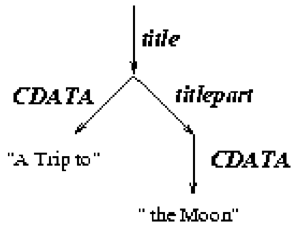

Figure 5. Representation of CDATA values.

By default, variables in patterns bind to internal nodes, not leaf values, i.e., the pattern <title><titlepart>$n</></> binds $n to the node referenced by <title><titlepart>, not its CDATA value. The value function produces the CDATA value of a node. This representation is usually transparent to users. For example, the pattern <title><titlepart>the Moon</></> is syntactic shorthand for <title><titlepart><CDATA>the Moon</></></>, which would match this graph.

3.3 Mapping XML Graphs into XML Documents

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
- shared nodes with multiple incoming edges do not have a unique representation in XML.

Both issues can be handled when the graph is emitted as an XML document. Given a DTD for a query's result, an XML graph can be emitted as an XML document that conforms to that DTD and therefore all elements will be ordered. We do not address the algorithmic issues of externalizing an XML graph here.

4. 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. We also illustrate how XML-QL can be used to integrate and transform XML data.

4.1 Tag Variables

XML-QL supports querying of element tags using tag variables, which support querying of the data's schema. 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>`.

4.2 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 zero or more edges, each of which is labeled `part`. The pattern:

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

is equivalent to the union of 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 `part`:

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

We abbreviate $* by *. Also, . denotes concatenation of regular expressions, hence the pattern <*.brand> Ford </> would match the brand Ford at any depth in the XML graph. 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. The + operator means ‘‘one or more’’, e.g., part+ is equivalent to part.part*.

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

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.

4.3 Transforming XML data with Skolem functions

An important use of XML-QL is transforming XML data, for example, from one DTD 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 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 value of ($fn,$ln). If the query binds $fn and $ln to the same values by finding another publication by the same author, the query does not create a new <person>, but instead *appends* information to the existing <person> element. Thus, all <publicationtitle>’s by one author will be grouped in the same <person> element. The <firstname> and <lastname> attributes are not duplicated since they refer to the same values. The <phone> and <address> values are not emitted, because they do not exist in the bibliography data.

Note that in our discussion of Skolem functions we assume an unordered data model. The ordered data model has a more complex and somewhat less intuitive semantics, which we describe in Sec. 4.5.

4.4 Integrating data from multiple 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> <ssn> $ssn </>  $n $i </>
```

Alternatively, we can use Skolem functions and write a related, but not equivalent, query in two fragments:

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

This query contains, in addition to the previous query, all persons that are not taxpayers, and vice versa. This is the *outer join* of `"www.a.b.c/data.xml"` and `"www.irs.gov/taxpayers.xml"`.

XML-QL queries are structured into *blocks* and *subblocks*: the latter are enclosed in braces (`{ ... }`), and their semantics is related to that of outer joins in relational databases. In this 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 and their incomes 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>`.

Query blocks are powerful. The following query retrieves all titles published in 1995 from the bibliography database, and it also retrieves the publication month 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 with their title. The first subblock 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). The second subblock checks if `e` is a book and has a `publisher`, and adds that `publisher` element to the output.

4.5 Support for ordered model

Recall that in the ordered data model, there exists a total order on nodes, usually document order, which induces a local order on outgoing edges of each node. In Figure 3, nodes are labeled by their index (parenthesized integers) in the total node order and edge labels are labeled with their local order (bracketed integers). The order of the outgoing edges is determined by the order of their destination nodes.

Under the ordered data model, the patterns in the `WHERE` clause are still interpreted as unordered. For example, the pattern`<a> <b> $x </b> <c> $y </c> </a>` will match both the XML data : `<a> <b> string1 </b> <c> string2 </c> </a>` and the data : `<a> <c> string2 </c> <b> string1 </b> </a>`. Although patterns are still unordered, variable bindings in the `WHERE` clause are ordered. Applying the example pattern to the XML data on the left-hand side below, the variables will be bound in the order given on the right-hand side:

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

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

Note that the order of the pattern is significant. If we change the pattern's definition to:

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

then the variables are bound in a different order, sorted first by `$y`, then by `$x`.

XML-QL supports element-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 local order of the edges. For example, the following query retrieves all persons whose `lastname` precedes the `firstname`:

```
WHERE <person> $p </> in "www.a.b.c/people.xml",
         <firstname[$i]> $x </> in $p,
         <lastname[$j]>  $y </> 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 $p
ORDER-BY value($y), value($z)
CONSTRUCT <result> $t </>
```

The `value` function produces the CDATA value of a node. In the absence of `value` function, the results would be ordered by the OIDs of the nodes bound to `$y, $z`. For a more complex example, the following query reverses the order of all `authors` in a publication:

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

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

4.6 Function definitions and DTD's

XML-QL supports functions, which take XML documents as arguments. Functions can increase query reuse, because the same query can be applied to multiple documents. This code defines a function that takes two XML documents as inputs:

```
 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 </> </>
 }
```

The following expression applies the function above to two XML sources:

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

The result of a function is always well-formed XML data. Function arguments and results can be restricted by DTDs. For example, the function above could be re-written as:

```
FUNCTION findDeclaredIncomes($Taxpayers:"www.irs.gov/tp.dtd",
                             $Employees:"www.employees.org/employees.dtd") :
                             "www.my.site.com/myresult.dtd" {
  WHERE ...
  CONSTRUCT ...
}
```

which specifies that the two inputs must conform to `tp.dtd` and `employees.dtd` respectively and that the function's result must conform to `myresult.dtd`. Determining that the output of a query conforms to the output DTD statically, i.e., at query compile time, is an open research problem and the subject of future work. Currently, the XML-QL query processor checks at runtime that the input documents conform to their respective DTDs.

5. Future work

Future work on XML-QL falls into two categories: extending the expressiveness of the language and providing better support for existing XML standards.

5.1 Query language features

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) </>
       </>
```

In the absence of `GROUP-BY`, the `CONSTRUCT` clause consumes a relation from the `WHERE` clause. Each tuple maps the variables in the `WHERE` clause to the values that satisfy the expression in the `WHERE` clause. The `GROUP-BY` clause maps this relation into a nested relation and groups the nested relations by the value(s) of the group-by variables(s). In the `CONSTRUCT` clause, all variables other than the group-by variable denote sets and therefore may be the arguments to aggregation functions.

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

As explained in Sec 4.5, compile-time type checking of a query is important if we expect XML-QL to be used in real applications. In particular, guaranteeing that a function's output conforms to a given DTD is important for applications that must transform data between various DTDs. Static type checking remains an open problem.

5.2 Support for XML standards

It would be useful to have an XML representation of XML-QL queries, because then applications that use XML-QL could embed XML-QL queries in any XML document. Existing XML-related standards, such as RDF [6] and XSL [8], support XML representations. In choosing an XML syntax for XML-QL, we intend to study related languages so that we can choose a similar notation for similar linguistic constructs.

A multitude of XML-related standards could influence the evolution of XML-QL. In particular, the XPointer and XLink [7] proposals support inter-document linking and provide a way of referring to data from multiple sources in one XML document. Given data with inter-document references and an XML-QL query that can query such data, an interesting open problem is how to decompose the XML-QL query into multiple queries that are distributed to the appropriate remote servers.

DTDs only impose syntactic, not semantic, restrictions on XML data. Several proposals for XML schemas have been proposed. These proposals are more precise than DTDs and can enforce strict types on XML data. Query optimizers rely on schemas to generate good query-execution plans. We expect that XML-QL's implementation would benefit from more precise schemas.

6. Summary

Given the expectation that XML data will become as prevalent as HTML documents, we anticipate an increased demand for search engines that can search both XML data's content *and* query 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 DTDs will be necessary.

XML-QL supports querying, constructing, transforming, and integrating XML data. XML data is very similar to *semistructured data*, which has been proposed by the database research community to model irregular data and support integration of multiple data sources. When designing XML-QL, we drew on our experience with other query languages for semistructured data. We have implemented an interpretor for XML-QL using the unordered data model; the prototype can be downloaded from `http://www.research.att.com/sw/tools/xmlql`, and an on-line demonstration of XML-QL is at `http://www.research.att.com/~mff/xmlql-demo/html`

Acknowledgements. The authors thank Serge Abiteboul, Catriel Beeri, Peter Buneman, Stefano Ceri, Ora Lassila, Alberto Mendelzon, Yannis Papakonstantinou, and the referees for their comments.

Bibliography

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

[2] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu *A query language and optimization techniques for unstructured data*, Proceedings of ACM-SIGMOD International Conference on Management of Data", 1996.

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

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

[5] M. Fernandez, D. Florescu, A. Levy, D. Suciu, *A Query Language for a Web-Site Management System,*, SIGMOD Record, vol. 26, no. 3, pp. 4-11, 9/1997.

[6] *Resource Description Framework (RDF) Model and Syntax*, `http://www.w3.org/TR/PR-rdf-syntax/`, 1999.

[7]*XML Linking Language*, `http://www.w3.org/TR/WD-xlink`, 1998.

[8] *Extensible Stylesheet Language (XSL)* , `http://www.w3.org/TR/WD-xsl`, 1998.

Appendix: Grammar for XML-QL

the grammar is changing as the language evolves and is incomplete in 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
                    '}'
          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 ::= '<' RegExpr Attribute* '>'
        RegExpr ::= RegExpr '*'  | RegExpr '+'  | RegExpr '.' RegExpr  | RegExpr '|' RegExpr  |
                    <VAR> ('['<INDEX-VAR>']')? | <ID>('['<INDEX-VAR>']')?
      BindingAs ::= 'ELEMENT_AS' <VAR>  | 'CONTENT_AS' <VAR>
      Predicate ::= Expression OpRel Expression
     Expression ::= <VAR> | <CONSTANT>
          OpRel ::= '<' | '<=' | '>' | '>=' | '=' | '!='
        OrderBy ::= 'ORDER-BY' <VAR>+ ('DESCENDING')?
     DataSource ::= <VAR> | <URI> | <FUN-ID>'('DataSource (',' DataSource)*')'
```

Vitae

Alin Deutsch is a PhD candidate in the Database Group at the University of Pennsylvania. He earned a M.Sc. degree in Computer Science from the Polytechnic Institute Bucharest, Romania, and one from the Technical University of Darmstadt, Germany.

Daniela Florescu received a master degree in computer science from University of Bucharest in 1990, Romania and the Ph.D. in computer science from University of Paris VI, France in 1996. After spending one year at AT&T Labs Research, she is now a researcher in INRIA, France.

Mary Fernandez is a researcher at AT&T Labs. Her primary research area is improving software development by designing very high-level languages and developing tools for their efficient implementation.

Alon Levy received his Ph.D. from Stanford University in 1993, and is now an assistant professor at the University of Washington's Computer Science and Engineering Department. His research interests are in database systems, artificial intelligence, data integration and web-site management.

Dan Suciu is a researcher at AT&T Labs. He works on query languages, query optimizations, semistructured databases, complex values, object-oriented databases, and database theory.