XML Query Languages in Practice: An Evaluation

Zachary G. Ives[1] and Ying Lu[2]

[1] University of Washington, Box 352350, Seattle WA 98195-2350, USA,
zives@cs.washington.edu
[2] University of Wisconsin, 1210 W. Dayton St., Madison WI 53706, USA,
luy@cs.wisc.edu

Abstract. The popularity of XML as a data representation format has led to significant interest in querying XML documents. Although a "universal" query language is still being designed, two language proposals, XQL and XML-QL, are being implemented and applied. Experience with these early implementations and applications has been instructive in determining the requirements of an XML query language. In this paper, we discuss issues in attempting to query XML, analyze the strengths and weaknesses of current approaches, and propose a number of extensions. We hope that this will be helpful both in forming the upcoming XML Query language standard and in supplementing existing languages.

1 Introduction

With the advent of the Internet and World Wide Web as mediums for electronic commerce and information exchange, the eXtensible Markup Language, XML, has emerged as a topic of great interest in the database community. XML provides a universal format for essentially any type of data, and it is rapidly being adopted as a replacement for proprietary formats in many applications. Shortly after XML's emergence, a number of researchers identified the need for a query language over this data representation format. The result has been a series of proposed languages, the most prominent of which are XML-QL [10] and XQL [17]. In an effort to provide a standard, the World Wide Web Consortium is developing a language called XML Query, for which they hope to have a working draft sometime this year; it is expected to be primarily derived from XML-QL and XQL.

Meanwhile, market demand for storing and querying XML has encouraged companies to develop and market commercial query processors for XML today. Object Design's eXcelon XML repository [18] maps XML documents to objects and supports queries over them based on the XQL language. Both Oracle and IBM support XML export of query results from their relational database systems. In these products and projects, XML is generally treated as a *protocol* for describing relational or object-based information — the focus is on encapsulating traditional data in an XML container, rather than on the unique representational aspects implicit in XML. By contrast, a number of research projects [16, 2, 12, 13] utilize XML as the basis for a data model, exploring the storage, integration, and processing of tree- and graph-structured XML data. A third perspective is that of users attempting to exploit XML in custom application domains, to obtain functionality not provided by more traditional database systems. One such

```
<db>
 <lab ID="baselab" manager="smith1">
  <name>Seattle Bio Lab</name>
  <location>
    <city>Seattle</city>
    <country>USA</country>
  </location>
 </lab>
 <lab ID="lab2">
  <name>Philadelphia Lab</name>
  <location>
    <city>Philadelphia</city>
    <country>USA</country>
  </location>
 </lab>
 <paper ID="Smith991231"
                  source="baselab">
  <title>Automatic Record...</title>
   <biologist>smith1</biologist>
 </paper>
 <biologist ID="smith1">
   <lastname>Smith</lastname>
 </biologist>
</db>
```

Fig. 1. Sample XML document representing biology labs and publications

project is the Cell Systems Initiative [8], an effort to define an ontology and experiment capture system for cellular research. Data in this application is too complex in structure to be effectively stored in a relational system, and requires a semi-structured data model and query language, such as those for XML.

Each of these domains has different needs for querying XML from a data management perspective. One sees XML as a standard exchange protocol; another uses XML as an intrinsic data model; the final applies XML to real-world application domains that would otherwise be unmanageable. In this paper, we discuss the strengths and weaknesses of the query languages applied to these domains, and we propose a number of improvements. Our goal is to present not simply a survey of language features, but an evaluation of how useful these languages are for data management. We do not attempt to address information-retrieval-style queries, which have been the focus of work such as [11]. We hope that our analysis will be useful in defining the standard XML Query language and for extending existing languages in the interim.

The structure of this paper is as follows. We begin in Section 2 with the basics of XML and how it is modeled, then continue in Section 3 with an overview of XML query languages (focusing on XQL and XML-QL). Section 4 describes issues in querying XML trees and graphs. In Section 5, we examine how input data is restructured into an output document. Section 6 discusses the application of XML models and query languages to the Cell Systems Initiative domain, and what difficulties this presents. In Section 7, we discuss related work and conclude with some recommendations.

2 XML Data

An XML document consists of pairs of matching open- and close-tags (elements), each of which may enclose additional elements or data values (in the form of "character data" strings). Every document must be contained within a single set of enclosing tags, known as the *root element*. Additionally, an element tag may include single-valued attributes further describing the element. XML documents may include embedded references to other XML documents in the form of XPointers [9]. See Figure 1 for the sample XML document upon which we shall frequently base our examples throughout this paper (note that it has no XPointers).

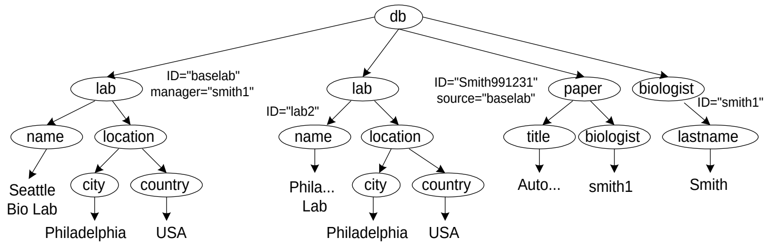

Fig. 2. XQL representation for Fig. 1. Edges indicate subelements.

An optional companion to the XML document, the Document Type Descriptor or DTD, adds a "schema" to which an XML document must conform to be considered *valid*. The DTD constrains the nesting of elements and assigns typing information to attributes. Attributes of type `ID` are element identifiers; those of type `IDREF` or `IDREFS` are references to other elements, by `ID`, within the XML document. `ID`s are guaranteed to be unique within a document and `IDREF`s may not dangle. For the example of Figure 1, we shall assume that an associated DTD (not shown) defines the `ID` attribute of the various elements to be of type `ID`, the `manager` attribute of the `lab` element to be an `IDREF`, and the `paper` element's `source` to also be a reference.

2.1 XML Data Models

One of the most important differences between XML query languages is in their data models. We begin here with an overview of the approaches to modeling an XML document.

Tree Models XQL [17] and XMAS [14] use the XML Document Object Model (DOM [1]) tree as the basis of their data models (Figure 2). This parse tree represents elements as nodes, contained subelements as edges to nodes, and all attributes as fields accessible from their elements.

Graph Models Languages such as XML-QL [10], XML-GL [5], and Lorel [12] treat the XML input document as a graph (Figure 3), where both subelements and `IDREF`s are mapped to edges. Each element is represented as an edge (labeled with the element name) directed to a node (given the element's `ID` if one exists, otherwise a unique identifier). `IDREF` edges are labeled with the `IDREF` name, directed to the referenced element node. In order to allow for intermixing of string data and nested elements within each element, XML-QL creates a `PCDATA` edge to each string. Conventional attributes (not shown in the example) are fields accessible from their element nodes.

The use of `IDREF`s as graph edges allows for modeling of any arbitrary structured or semi-structured data. However, current graph data models do not fully specify ordering within the data graph; to clarify order mappings, we propose the following correspondence. Given an XML data graph, we can take all `IDREF` edges and replace them with identically-named attributes, whose values are the `ID`s of the destination nodes; this will result in a tree equivalent to the DOM tree. A left-to-right depth-first traversal will generate the equivalent XML document. In this mapping, attributes and `IDREF` edges are ordered *before* subelement edges.

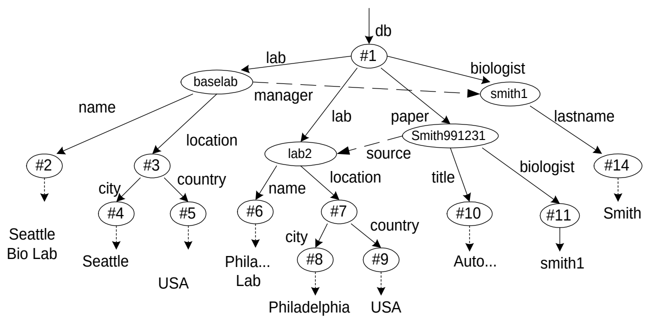

Fig. 3. XML-QL graph for Fig. 1. Dashed edges are `IDREF`s; dotted edges are `PCDATA`.

2.2 Traversing the Document

The key to querying an XML document lies in selecting the desired data from the input. Most XML query languages use *regular path expressions*, describing paths to be taken from the document "root node" to the data values. A regular path expression enumerates a sequence of node or edge labels to be followed; since XML may have recursively nested elements or irregular structure, it may include regular-expression operators such as the wildcard, the Kleene-star (for repetition), and choice (for alternate sub-paths).

For a tree-structured query language, there is one unique path from the root to a given node. If the data model is graph-structured, however, there may be multiple paths to a given node; here, language semantics generally specify that a path expression will only return each node once.

Even under the graph-structured data model, there are cases where we would like to traverse only subelements in the document, or to query `IDREF`s as attributes instead of edges. XML-QL does not differentiate between these different edge types; the Lorel [12] query language supports this by allowing the user to switch between a graph-structured and a tree-structured mode. We believe that Lorel's modes can be too coarse-grained — the query author may want to maintain the input graph structure, but simply traverse subelements along certain portions of the path. We suggest extending path expressions so they can restrict the type of a given traversal edge to be an `IDREF` or a subelement.

3 Query Language Basics

A number of query languages have characteristics of note. XMAS [14], the language of the MIX mediator system, is essentially a simplification of XML-QL for tree-structured data; its simplified model makes query processing and writing less complex. Lorel, the language for the Lore semi-structured database system, has been extended to support XML, and includes update as well as query capabilities (though these are only supported in Lore's original OEM model). Finally, XML-GL [5] uses diagrams rather than commands to express queries. While these languages have some novel features, we focus on XQL [17, 19] and XML-QL [10], the languages that are being implemented for real applications and the greatest influences on the W3C query language specification.

3.1 XQL

XQL is the "parent" of the W3C XPath [6] document navigation standard and the basis of the XSLT [20] transformation standard. However, XQL was recently extended with features not found in its "child"; we shall focus on XQL rather than XPath in this paper. XQL queries are very simple: they extract nodes and subtrees from a single input document and return these in a new XML document.

The XQL tree data model is closely matched to the XML physical format, as described in Section 2.1. Queries return elements and their children in the same order as they appear in the input document. This model makes XQL well suited to finding and returning XML document fragments; yet the data model is considerably less flexible and expressive than a graph model.

An XQL query is divided into a *path expression* and an optional *filter expression*. The path expression defines the nodes to be returned in the query result, and the filter expression selects only nodes meeting specified criteria. A query is of the form *path* `[`*filter*`]`, where *path* is typically a series of element names separated by the slash (`/`) character, and *filter* is a sequence of boolean conditionals, e.g. tests for sub-path existence. A sub-path is a sequence of element names, optionally followed by an attribute (indicated by a prepended `@` sign). XQL paths and sub-paths are expressed relative to a "current location" (defaulting to the entire document for the outermost query). A leading slash restarts the path at the document level; a star (`*`) is a wildcard representing any edge; two consecutive slashes (`//`) specify any number of wildcard edge traversals. Once a node has been selected by its path, it can be returned, or various methods can be called on it. Note that an XQL path expression is not as powerful as a full regular path expression, as it does not include true Kleene-star or choice operators.

Example XQL queries over Figure 1 include:

- `/db/lab[@manager="smith1"]` returns only `labs` with `manager` attributes containing the value "`smith1`" — namely the Seattle lab. Note that we return the `lab` subelement and all of its children; the filter within the brackets does not affect the query path.
- `/db/lab` { `location/city | name` } returns each `lab` element, with only its `city` and `name` subelements within. The { } are "grouping" operators and `|` forms a union of element results.

3.2 XML-QL

XML-QL was the first database-style language proposed for querying XML, and has had perhaps the largest impact on the W3C's vision of an "ideal" XML query language. XML-QL uses the graph data model of Section 2.1, and is a full graph-to-graph query and transformation language.

XML-QL uses a `WHERE` *pattern1* `IN` *source1*, ... `CONSTRUCT` *result* syntax, in which each *pattern* template is matched against an input XML data graph from its corresponding *source* (a URI, view, or variable) and the *result* defines the desired structure of the query output graph. XML-QL supports multi-document queries and relational-like operations such as joins and grouping.

```
WHERE <db>                          CONSTRUCT <result><combo>
        <lab><name>$l</></>                   <laboratory>$l</>
        <biologist>$b</>                      <person>$b</>
      </> IN "fig1.xml"                     </></>
```

Fig. 4. Example XML-QL query for Figure 1

An XML-QL pattern is expressed as a set of nested tags with embedded variable names (prefixed by leading dollar-signs) that specify *bindings* of graph nodes to variables. An example XML-QL query appears in Figure 4. We abbreviate each close-tag with a `</>`. The WHERE template is a tree structure of path expressions that get "matched" across the input graph. Each variable (l and b above) is bound to the matching node at the end of the path. In this case, we take a `db` edge from the document root. From here, we find a `lab` edge and then a `name` edge to a node we assign to variable l. Now, from the same `db` edge traversed earlier, we find a `biologist` edge to a node we shall call `b`. During query execution, we apply the template and form every possible combination of path expression matches. Each tuple of bindings to variables is evaluated much like a tuple in a relational database.

The `CONSTRUCT` clause specifies a tree structure to add to the output graph. Wherever an input variable appears in the `CONSTRUCT` clause, its associated node is inserted into the output. We also "carry forward" all other nodes transitively connected by edges radiating from the original node. In essence, an XML-QL variable bound to an XML graph node represents the *entire subgraph* to which the node transitively connects via "forward-pointing" edges.

4 Querying XML Data

In this section, we examine some of the important considerations in querying XML documents.

4.1 SQL-like Features

With the simple language elements discussed previously, we can express search-style queries that traverse an XML document and return portions. However, one of the applications of XML is as a "container" for relational data. In this type of application, the query language must support SQL-like relational operations.

Join XQL can join different path expressions within the *same* document. In order to support join predicates, dollar-sign-prefixed *correlation variables* are associated with sub-paths. Figure 5(a) takes `paper biologist` subelements and saves their values in the `b` variable; then finds `db biologist` elements with matching `ID`s. The returned result is the `paper` elements with additional `biologist lastname` subelements inserted within. Note that this is a left outer join, as papers may appear with no biologists. XQL also supports a limited inner join operation that returns the contents of *one* of the two join subtrees.

The XML-QL equivalent to Figure 5(a), in Figure 5(b), is slightly more powerful, as it can combine the `paper` and `author` elements from different sources. If the same variable occurs more than once within a query, it is constrained to have the same value in both places (thus forming an equijoin); or we can add a test such as `$b1 < $b2` to the `WHERE` clause to establish a range constraint on the

```
/db/paper[$b=biologist] {
   * | /db/biologist[@ID=$b]
                /lastname
}
```

(a) XQL

```
WHERE <db><paper>
         <biologist>$b</>
       </> CONTENT_AS $p
      </> IN "fig1.xml",
      <db><biologist ID=$b>
         <lastname>$n</></>
      </> IN "fig1.xml"
CONSTRUCT
     <result><paper>$p<lastname>$n</>
     </></>
```

(b) XML-QL

Fig. 5. Join query for data of Figure 1

variables' values. The CONTENT_AS specifier in the query expresses a binding of a variable to the contents (the node) of the previous element — in this case, it sets p equal to the paper node. The constructed output consists of paper elements with additional biologist lastname subelements, as in the XQL query, except that we have performed an inner join. XML-QL also supports outer join queries, but these are more complex to express and less commonly used.

Null values A problem can arise when mapping relational tables with null attributes to XML. Typically, the subelements corresponding to null relational attributes are simply omitted from the document. Neither XQL nor XML-QL support queries with optional elements that are not required in matching a pattern. If XML is to be effectively used to store relational data, it seems critical that the query language have this capability.

Universal quantification and negation Two important capabilities for certain classes of queries are universal quantification and negation. XQL includes all and not keywords in the filter expression for this purpose; XML-QL is missing these important features.

Aggregation Aggregate functions such as average and max are very commonly used for summarization and other purposes in SQL. XQL supports a count function, but no other aggregation operations. The original XML-QL specification describes a model for supporting aggregate functions, and this model has been further developed (see Section 5.4).

4.2 References

When XML is used as a graph-structured data format, following references becomes vital. Both XQL and XML-QL include mechanisms for managing IDREFs. XML-QL models both IDREF and IDREFS attributes as edges, so we can follow references with path expressions. In the latest XQL proposals, a global method, id, dereferences IDREF attributes by value. Unfortunately, XML IDREFS attributes consist of a string values with *multiple* ID references separated by delimiter characters; XQL does not include a mechanism for separating these into sub-components, and thus there is no way to de-reference an IDREFS attribute.

Neither XML language handles XPointers, but they can fairly easily be extended to do so. A proposed extension to XQL adds a ref global method that returns the contents of an XPointer. In XML-QL, one can create a user-defined "function" that does the equivalent, namely takes a URI or XPointer and returns the corresponding XML graph.

```
/db/lab[@manager="smith1"]
                  -> smithlab
```

(a) XQL

```
WHERE <db>
        <lab manager="smith1"></>
                  CONTENT_AS $lab
      </> IN "fig1.xml"
CONSTRUCT
      <result><smithlab>$lab</></>
```

(b) XML-QL

Fig. 6. Renaming an element

4.3 Querying Document Order

Both XQL and XML-QL allow queries to reference a subelement's *index*, i.e. its numeric ordering as a child of its parent element: we can specifically request the *i*-th child of an element, or we can query a node for its index value. However, XML-QL lacks the ability to query for the *i*-th element with label *L* (e.g. the second `<p>` element), which we can do in XQL.

In Section 2.1, we proposed a mapping between an XML graph model and document in which `IDREF`and `IDREFS` edges are ordered prior to sibling subelements. Under this ordering, we can now extend XML-QL in a useful way for graph data — to allow querying for the *i*-th *out-edge* with a particular name. (Note that we do not propose an ordering between edges of different names, since different `IDREF` attributes are unordered with respect to one another.)

5 Special Query Output Behavior

In the previous section, we analyzed the capabilities of the XML query languages on input documents. Both languages support simple "copying" of input subtrees to the output. In this section we discuss how portions of this output can be modified, and also discuss how graph-structured XML can be created in XML-QL.

5.1 Pruning XML output

An interesting aspect of nearly all XML query languages, including XQL and XML-QL, is that they allow for the selection of a portion of the input document (subtree or subgraph) via a path expression, but support no projection-like operations on it. An XQL path expression will return the matching subtree, however deep it might be, as output. An XML-QL node variable, when it is used in the CONSTRUCT clause, outputs the node plus the entire subgraph to which it is connected. A missing capability would allow the query to restrict the portions of the subtree or subgraph that get copied, i.e. prune the data.

5.2 Modifying elements

Often, queries need to rename the element labels from the original query. Examples of how to do this in XQL and XQL are in Figure 6, where we rename the `lab` element to `smithlab`. In the XQL query, we use the `->` renaming operator to change the name of the outermost tag of the query result; in XML-QL, we bind to the source element's node content using the `CONTENT_AS` specifier, and later "wrap" the node with a new enclosing tag.

```
WHERE <db>                          { WHERE <db>
        <paper>                               <biologist ID=$b>
          <biologist>$b</>                      <lastname>$l</>
        </> CONTENT_AS $p                     </>
      </> IN "fig1.xml"                     </> IN "fig1.xml"
CONSTRUCT                             CONSTRUCT <lastname>$l</>
      <db>                          }
        <paper>$p                   </>
                                  </>
```

Fig. 7. Nesting in XML-QL

5.3 Nesting Subqueries

XML is fundamentally a tree-structured format, so one of the most common operations is to take elements from one subquery and nest them under elements from a different query (perhaps "matching" parent and child query results with a join condition). Both XQL and XML-QL support this operation quite elegantly: essentially, the subquery is embedded within the portion of the parent query that constructs the result. The XQL query of Figure 5(a) outputs `paper`s as parents of a nested `biologist` subquery; its XML-QL equivalent appears in Figure 7. For both languages, the entire subquery is executed and embedded for each set of bindings in the outer query, producing a *1:n* nesting relationship.

5.4 XML Graphs in XML-QL

XQL is a single-document, tree-oriented language, whereas XML-QL is multi-document and graph-oriented. The extra features of XML-QL provide considerably more flexibility, but also add new concerns with respect to the output XML representation. Note that several people have proposed extending XQL to a graph model, so while this discussion is focused on XML-QL, it will also be relevant to such extended versions of XQL.

Skolem functions A fundamental concept in XML-QL is that of node identity in the output graph. If a query attempts to create an output node with the same node ID more than once, i.e. for more than one tuple of variable bindings, each iteration refers to the same node in the output graph — the output node will only be created once. This enables a query to refer to and extend an existing node. This is where the XML-QL *Skolem function* is useful. Each Skolem function creates a perfect hash value for its arguments, and its values will not collide with those of any other Skolem function.

Skolem functions are used to group elements based on data value and to create multiple references to the same node. For instance, in the query of Figure 8, we take any `paper` elements in Figure 1 and, using the Skolem function `Sk1`, re-group them by `biologist` instead of by paper. For each new value of `b`, we will output a `biologist` node and a nested `paper` node with a reference back to its parent. Each time a duplicate `b` value is bound, we insert a new `paper` node underneath the existing `biologist` node.

Skolem functions also form the basis of aggregation operations: functions such as `average` or `count` can be applied across the sets of values that get

```
WHERE <db>                               CONSTRUCT
        <paper ID=$i source=$s>                <result>
          <title>$t</>                           <biologist ID=Sk1($b)>
          <biologist>$b</>                         <paper ID=$i source=$s
        </>                                            ref=Sk1($b)>$t</>
      </> IN "labs.xml"                          </> </>
```

Fig. 8. Grouping with Skolem functions

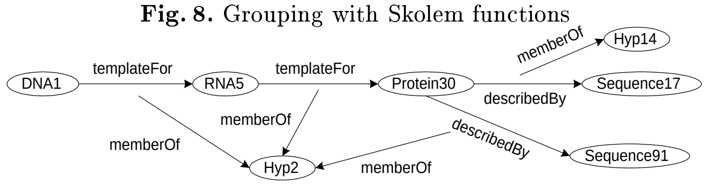

Fig. 9. Model of DNA-RNA-Protein interactions according to hypotheses

consolidated together by the Skolem function. Skolem functions can even perform duplicate removal and force XML-QL to output graph-structured rather than tree-structured data.

XML Graph Irregularities At times, graph-structured data may map into irregular and "ugly" XML. For example, it is possible to use Skolem functions to consolidate nodes such that they have multiple in-edges like the `Hyp2` node in Figure 9. The initial set of variable bindings will create the `Hyp2` node as an XML subelement under some parent element. For future bindings, however, referencing parent nodes must connect to `Hyp2` via `IDREF`s. Thus one of the node's "parents" will be a parent *element*, and all others will be *referrers* — despite the fact that the source query did not distinguish between any of the "parent" nodes.

Another XML mapping artifact arises because XML-QL outputs not only the nodes bound to variables, but also all nodes that are transitively connected to these. This feature is often convenient, but it has indeterminate output when the referenced element has a parent not in the query output. In this case, the most logical approach is to "fold" the referenced node under its first "parent" as an XML subelement.

6 An Application of XML

The goal of the Cell Systems Initiative (CSI) project at the University of Washington is to provide an online, web-like knowledge base representing all aspects of biological data — from experiments to hypotheses to publications. The CSI knowledge base is a complex graph structure with edges, "conditional edges," and various types of nodes. An example graph appears in Figure 9. Note that the `templateFor` and `describedBy` edges indicate relationships, but that these relationships are conditional on the validity of the hypotheses, as expressed by the `memberOf` edges originating from these edges.

A graph like this can be represented relationally, or even in an XML tree model, but querying it would be highly unintuitive and inefficient. We chose to use the XML-QL graph model, after making one transformation: since XML does not allow edges to originate from other edges, we must "split" each conditional edge into a pair of edges with an intermediate node, providing a source for each `memberOf` edge.

Proposed queries for the CSI domain have demonstrated the need for a several extensions to XML-QL. A significant problem occurs because of XML-QL's policy of "carrying over" *all* transitively connected nodes; this may result in "extra" output. A pruning feature, as suggested in Section 5.1, would solve this problem. Additionally, the CSI database is expected to consist of large numbers of interlinked XML documents, each using XPointers to reference other portions of the overall structure. XML-QL must be able to support XPointers to make this work. Overall, however, preliminary designs and results suggest that, with these extensions, XML-QL is are fairly well suited to this application.

7 Related Work and Conclusions

In this paper, we have described the two most widely accepted XML query languages, XQL and XML-QL, and examined how they can be applied to three different domains: relational queries, queries over arbitrary XML data, and graph-structured scientific applications. While we believe this to be the first analysis of XML query languages' applicability, issues in designing an XML query language have been frequently discussed in the literature. In particular, the W3C's 1998 Query Language Workshop included numerous papers describing the the motivations and requirements for querying XML [4, 7, 15], as well as several important language proposals [17, 10, 12, 5, 14]. The application of XML-QL to information retrieval queries was discussed in [11].

Recently, Bonifati and Ceri presented a survey of five major XML query languages [3] that compared the features present in each. The goal of this paper is more than to provide a feature comparison: we hope to promote a greater understanding of XML query semantics, and to detail some of the problems encountered in trying to apply these languages. While a query language containing the "union" of the features present in XQL and XML-QL will go a large way towards solving the needs of querying XML, we also propose a number of extensions that we feel are necessary:

– An XML graph model with defined order between `IDREF`s and subelements
– Regular path expression extensions for subelement, `IDREF`, or arbitrary edges
– Support for "optional" path expression components and null values
– Support for following XPointers
– Pruning of query output
– Clearer semantics for copying subgraphs to query output

In general, XML-QL nearly meets our needs, and it could fairly naturally be extended with the missing capabilities. In particular, if we add universal quantification, negation, and the features listed above, it should be well-suited for our domains of interest. XQL can also be further developed, but the required extensions fit less cleanly into its single-document query model.

While some of these operations may increase the complexity of an XML query processor, all should be possible using well-studied techniques. XML querying is still a young field, but the database community's experience in querying other data models has given it a solid foundation.

References

1. V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM) level 1 specification. http://www.w3.org/TR/REC-DOM-Level-1, October 1998.
2. C. K. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 597–599, 1999.
3. A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–79, March 2000.
4. A. Bosworth, A. Levy, J. Widom, R. Goldman, J. McHugh, A. Layman, A. Ardelwanu, and D. Schach. Position paper for the W3C query language workshop, December 3, 1998. W3C Query Language Workshop, http://www.w3.org/TandS/QL/QL98/pp, December 1998.
5. S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A graphical language for querying and reshaping XML documents. W3C Query Language Workshop, http://www.w3.org/TandS/QL/QL98/pp/xml-gl.html, December 1998.
6. J. Clark and S. DeRose. XML path language (XPath) recommendation. http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999.
7. P. Cotton and A. Malhotra. Candidate requirements for XML query. W3C Query Language Workshop, http://www.w3.org/TandS/QL/QL98/pp, December 1998.
8. Cell Systems Initiative. http://cellworks.washington.edu, 2000.
9. S. DeRose, R. D. Jr., and E. Maler. XML pointer language (XPointer) working draft. http://www.w3.org/TR/1999/WD-xptr-19991206, December 1999.
10. A. Deutsch, M. F. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the International Word Wide Web Conference, Toronto, CA*, 1999.
11. D. Florescu, D. Kossman, and I. Manolescu. Integrating keyword search into xml query processing. In *Proceedings of the 9th WWW Conference, Amsterdam, NL*, May 2000.
12. R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD Workshop on the Web (WebDB), Philadelphia, PA*, pages 25–30, 1999.
13. Z. G. Ives, A. Y. Levy, and D. S. Weld. Efficient evaluation of regular path expressions over streaming XML data. Submitted for publication, 2000.
14. B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. A brief introduction to XMAS. http://www.db.ucsd.edu/Projects/MIX/docs/XMAS-intro.pdf, February 1999.
15. D. Maier. Database desiderata for an XML query language. W3C Query Language Workshop, http://www.w3.org/TandS/QL/QL98/pp/maier.html, December 1998.
16. J. Naughton, D. DeWitt, D. Maier, J. Chen, L. Galanis, K. Tufte, J. Kang, Q. Luo, N. Prakash, F. Tian, J. Shanmugasundaram, C. Zhang, R. Ramamurthy, B. Jackson, Y. Wang, A. Gupta, and R. Chen. The Niagara internet query system. Submitted for publication, 2000.
17. J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, September 1998.
18. eXcelon: The XML application development environment. http://www.odi.com/excelon/main.htm.
19. XQL (XML Query Language). http://metalab.unc.edu/xql/xql-proposal.html, August 1999.
20. XSL Transformations (XSLT), version 1.0. http://www.w3.org/TR/xslt, 13 August 1999. W3C Working Draft.