XML Query Languages: Experiences and Exemplars

Editors		
Mary Fernandez	AT&T Labs – Research	mff@research.att.com
Jérôme Siméon	Bell Labs, Lucent Technologies	simeon@research.bell-labs.com
Philip Wadler	Bell Labs, Lucent Technologies	wadler@research.bell-labs.com
Contributors		
Sophie Cluet	INRIA Roquencourt	Sophie.Cluet@inria.fr
Alin Deutsch	Univ. of Pennsylvania	adeutsch@gradient.cis.upenn.edu
Daniela Florescu	INRIA Rocquencourt, France	Daniela.Florescu@inria.fr
Alon Levy	University of Washington, Seattle	alon@cs.washington.edu
David Maier	Oregon Graduate Institute	maier@cse.ogi.edu
Jason McHugh	Stanford University	mchughj@db.stanford.edu
Jonathan Robie	Software AG	jonathan.robie@sagus.com
Dan Suciu	AT&T Labs – Research	suciu@research.att.com
Jennifer Widom	Stanford University	widom@db.stanford.edu

Abstract

This paper identifies essential features of an XML query language by examining four existing query languages: XML-QL, YAT$_L$, Lorel, and XQL. The first three languages come from the database community and possess striking similarities. The fourth comes from the document community and lacks some key functionality of the other three.

This document:
`http://www-db.research.bell-labs.com/user/simeon/xquery.html`
`http://www-db.research.bell-labs.com/user/simeon/xquery.ps`
`http://www-db.research.bell-labs.com/user/simeon/xquery.txt`

1 Introduction

Over the years, the database community has learned a thing or two about how to process queries. There has been an evolution from relational databases through object-oriented databases to semistructured databases, but many of the principles have remained the same. From the semistructured community, three languages have emerged aimed at querying XML data: XML-QL [?], YAT$_L$ [?, ?], and Lorel [?, ?]. These languages were developed independently by research groups thousands of miles apart, yet they show striking similarities of approach.

Over the years, the document community has also learned a thing or two about searching and formatting documents. The document processing community has developed models of structured text and search techniques such as region algebras [?]. From this community, one language that has emerged for processing XML data is XQL [?, ?].

The two communities address different application areas. The database community is concerned with large repositories of data, integrating data from heterogenous sources, exporting new views of legacy data, and transforming data into common data-exchange formats. The document community is concerned with full-text search, queries of structured documents, integrating full-text and structured queries, and deriving multiple presentations from a single underlying document.

The majority of authors of this document come from the database camp. This community has a great deal of experience studying what expressive power is necessary to support the application areas listed above, and what query language features provide this expressive power. We wish to argue that what is known regarding the expressive power of query languages should play a central role in the design of a query language for XML.

Of course, what the document community has learned is also relevant, but we don't feel competent to advance those lessons here, and hope they will do so elsewhere. The database community also has learned about query complexity, algebras, and techniques for implementing these query languages efficiently, but these subjects are also outside the scope of this paper.

The database query languages listed above have several features that we believe are especially important:

- Queries that consist of three parts: a *pattern* clause, a *filter* clause, and a *constructor* clause. The information passed between these clauses can be modeled as a *relation*, which has a flat and unordered structure.

- Constructs to impose nesting and order upon the relations. These may retain the structure of the original document, or may allow complete restructuring of the document — this is the key advantage of this approach. These constructs include *nested queries*; grouping related data items together via *Skolem functions* or explicit *grouping operators*; *indexing* and *sorting*.

- Use of a *join* operator to combine data from different portions of documents, corresponding to the join operation on relations.

- Use of *tag variables* or *path expressions* to support querying without precise knowledge of the document structure and access to arbitrarily nested data.

They also provide other useful features:

- Constructs to process *alternatives* in different ways and constructs to check for the *absence* of information, e.g., missing fields.

- Use of arbitrary *external functions*, such as aggregation functions, string comparison functions, etc.

- Use of *navigation* operators, which simplify handling data with references.

We illustrate these points by a collection of exemplars: we consider typical queries for a database of books, and show how to express these in XML-QL, YAT$_\text{L}$, Lorel, and XQL. The first three languages almost always use the same structure for the same query, while XQL often uses a different structure. In some cases, the query may not be expressible in XQL. (Of course, since XQL grew out of the needs of the document community, there are also many queries that can be expressed in XQL but not in XML-QL, YAT$_\text{L}$, or Lorel.)

We wrote this paper for the XML Query Working Group to highlight the database research community's experience designing and implementing XML query languages. We are not suggesting that XML-QL, YAT$_\text{L}$, or Lorel be adopted as the working group's initial language. But in light of these languages' striking similarities, one cannot ignore the lessons learned by the database research community. Therefore, we do suggest that the common ideas and features of these languages be considered a starting point for the working group. and serve as a yardstick against which the working group's recommended language is compared.

In this paper, we do not address several important but orthogonal issues, such as the environment in which an XML query language will be executed. Instead, we refer the reader to a comprehensive list of desirable language features and related issues [?]. We also refer the reader to a substantial body of research, including the motivation for and typical applications of semistructured data, [?, ?, ?], data models for semistructured data [?], query-language design [?, ?, ?], query processing and optimization [?], schema languages [?, ?, ?], and schema extraction [?].

The next section presents what we consider to be ten essential queries. Section ?? presents other useful, but less crucial, features.

2 Ten Essential Queries

Here, we present example queries that illustrate what we believe are ten essential features of an XML query language. We illustrate our examples using XML-QL, YAT$_L$, Lorel, and XQL. Whenever possible, the language providing the most natural or simple formulation will be used first.

We use the following running example. The XML input is in the document `www.bn.com/bib.xml`, containing bibliography entries described by the following DTD.

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ),
              publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

This DTD specifies that a book element contains one title, one or more author elements or one or more editor elements, one publisher element and one price element; it also has a year attribute. An author element contains a last and a first name. An editor element also contains an affiliation. A title, last name, first name, publisher, or price is text.

2.1 Selection and extraction

Our first example selects all titles of books published by Addison-Wesley after 1991. To give a query evaluator maximum flexibility, no order is specified for the output. In Section ??, we will show how to sort the titles by document or alphabetical order.

In XML-QL, YAT$_{\mathrm{L}}$, and Lorel, a query consists of three parts: a *pattern* clause, which matches nested elements in the input document and binds variables; a *filter* clause, which tests the bound variables; and a *constructor* clause, which specifies the result in terms of the bound variables. Nested queries may appear in a constructor. XQL supports patterns and filters, but not constructors. XQL can apply filters to elements and attributes, as well as processing instructions, comments, and entity references. We note that XML-QL, YAT$_{\mathrm{L}}$, and Lorel all provide syntactic shorthands for common idioms in queries, but for clarity, we write queries in their most general form.

We assume that queries specify a fixed data source (via one or more URLs) and return a well-formed XML tree. Of course, queries might act on other representations of XML trees, such as the DOM. For instance, XML-QL has a graph interface, and some implementations of XQL interface with the DOM.

XML-QL

```
CONSTRUCT <bib> {
  WHERE
    <bib>
      <book year=$y>
        <title>$t</title>
        <publisher><name>Addison-Wesley</name></publisher>
      </book>
    </bib> IN "www.bn.com/bib.xml",
    $y > 1991
  CONSTRUCT <book year=$y><title>$t</title></book>
} </bib>
```

In an XML-QL query, patterns and filters appear in the `WHERE` clause, and the constructor appears in the `CONSTRUCT` clause. The result of the inner `WHERE` clause is a relation, that maps variables to tuples of values that satisfy the clause. In this case, the result contains all pairs of year and title values bound to `($y, $t)` that satisfy the clause. The result of the complete query is one `<bib>` element, constructed by the outer `CONSTRUCT` clause. It contains one `<book>` element for each book that satisfies the `WHERE` clause of the inner query, i.e., one for each pair `($y, $t)`.

YAT$_{\mathrm{L}}$

```
make
```

```
  bib [ *book [ @year [ $y ],
                title [ $t ] ] ]
match "www.bn.com/bib.xml" with
  bib [ *book [ @year [ $y ],
                title [ $t ] ],
                publisher [ name [ $n ] ] ]
where
  $n = "Addison-Wesley" and $y > 1991
```

In a YAT$_L$ query, the constructor appears in the make clause, patterns appear in the match clause, and filters appear in the where clause. An * precedes any repeated element. Thus, the pattern expresses that a bib element may have many book elements, but that each book element has one year attribute, one publisher element, and one title element. Here, no nested query is necessary, because the constructor indicates there is one bib element with multiple book elements, i.e., one for each pair ($y, $t) in the result. As in XML-QL, the meaning of the match and where clauses is a relation that maps variables to tuples of values that satisfy the clauses.

Lorel

```
select xml(bib:{
  (select xml(book:{@year:y, title:t})
   from bib.book b, b.title t, b.year y
   where b.publisher = "Addison-Wesley" and y > 1991)})
```

In a Lorel query, the constructor appears in the select clause, patterns appear in the from clause, and both patterns and filters appear in the where clause. In this query, bib is used as the entry point for the data in the XML document. The from clause binds variables to the element ids of elements denoted by the given pattern, and the where clause selects those elements that satisfy the given filters. As in XML-QL and YAT$_L$, the meaning of the from and where clauses is a relation that maps variables to tuples of values that satisfy the clauses. The select clause constructs a new XML book element with a year attribute and a title element.

XQL

```
document("http://www.bn.com")/bib {
  book[publisher/name="Addison-Wesley" and @year>1991] {
        @year | title
  }
}
```

In this XQL query, the pattern `document("http://www.bn.com")/bib` selects all top-level `bib` elements from the input document and evaluates the nested expression for each such element. The nested pattern `book` selects the book elements that are children of a `bib` element and that satisfy the filter clause in brackets. XQL does not have a constructor clause; instead the pattern expressions determine the result of the query. In this case, the result is one `bib` element that contains the selected `book` elements; the inner-most expression projects only the book's year attribute and title element.

2.2 Flattening

Our next query no longer filters on publisher or year, and returns a collection of all title-author pairs. The query flattens the nested structure, each book contributing one pair for each author. Recall that in XML-QL, YATL, and Lorel, the meaning of the patterns and filters is a relation, which is the Cartesian product of all variable bindings that satisfy the patterns and filters. This results in a flattening effect which, as we will see in the next sections, provides the basis for further restructuring of the document.

XML-QL

```
CONSTRUCT <results> {
  WHERE
    <bib>
      <book>
        <title>$t</title>
        <author>$a</author>
      </book>
    </bib> IN "www.bn.com/bib.xml"
  CONSTRUCT
    <result>
      <title>$t</title>
      <author>$a</author>
    </result>
} </results>
```

The `WHERE` clause produces one tuple for each binding of `$t` and `$a` that satisfies the pattern and filter. Each book has one title and possibly multiple authors, therefore there is one tuple for each author of each book. The CONSTRUCT clause produces one result element for each pair of values bound to ($a, $t), i.e., the constructor's free variables.

YATL

```
make
  results [ *result [ title  [ $t ],
                      author [ $a ] ] ]
match "www.bn.com/bib.xml" with
  bib [ *book [ title [ $t ],
                *author [ $a ] ] ]
```

When all pairs of titles and authors are unique, YATL produces the same result elements as the XML-QL query. If the same title and author occurs in different books, however, YATL would preserve these duplicates whereas XML-QL would eliminate them. This is because YATL has a *bag* semantics, which permits duplicates in the intermediate relation, but XML-QL has a *set* semantics, which eliminates duplicates.

Lorel

```
select xml(results:
  (select xml(result:{title: t,
                      author: a})
    from bib.book b, b.title t, b.author a))
```

In this Lorel query, the from clause binds variable b to each book in the input document. All title, author pairs for each book are bound to variables t and a. We create a new element for each pair with the tag result using the xml construct. This resulting element has two subelements, one for the title and one for the author.

XQL

Flattening does not exist in XQL, because the results of patterns and filters are not modeled by an intermediate relation. The result of an XQL query must maintain the original nesting of the nodes in the input document.

2.3 Preserving structure

The previous example returns one result for each possible title-author pair. The next one preserves the grouping of results by title.

XQL

In XQL, grouping is preserved automatically, because the result of a query is always a projection of the original document.

```
document("http://www.bn.com")/bib->results {
    book->result {
        title | author
  }
}
```

For each `book` element, the above query creates a `result` element using the renaming operator `->`. The children of this `result` element are all title and author elements contained in the `book` element, with document order preserved.

YATL

In YATL, this query is written as:

```
make
  results [ *result [ title  [ $t ],
                      $as ] ]
match "www.bn.com/bib.xml" with
  bib [ *book [ title [ $t ],
                *($as) author ] ]
```

This query uses two variables, `$t` and `$as$`, to extract the title and the authors. Because of the `*($as) author` construct, `$as` is bound to the *list* of all `author` elements in a book rather than successively to each author.

Lorel

This query can be expressed in Lorel as:

```
select xml(results:
  select xml(result:{b.title, b.author})
  from bib.book b)
```

The pattern expression `b.author` denotes the *set* of values for author; similarly, `b.title` denotes the set of titles, if there are more than one. Each binding for `b` produces a new `book` element with, as its sub-elements, the set of titles and authors for that book.

XML-QL

In XML-QL, one way to preserve the original document structure is with a nested query:

```
CONSTRUCT <results> {
  WHERE
    <bib>
      <book>
        <title>$t</title>
      </book> CONTENT_AS $b
    </bib> IN "www.bn.com/bib.xml"
  CONSTRUCT
    <result>
      <title>$t</title>
      {  WHERE <author>$a</author> IN $b
         CONSTRUCT <author>$a</>
      }
    </result>
} </results>
```

Here, `CONTENT_AS` binds the variable `b` to the `book` element's content, a collection of elements. The inner `WHERE` clause selects the `author` elements from `$b`. It is also possible to preserve structure in XML-QL without nested queries by using an explicit grouping construct (see Section **??**).

2.4 Changing structure by nesting

Sometimes the result of a query needs to have a structure different than the original XML document. The next query illustrates restructuring by grouping each author with the titles he or she has written. This requires joining elements on their `author` values; the example query treats two authors as equal when they have the same last and first names. We will see more examples of joins in Section **??**.

XML-QL

```
CONSTRUCT <results> {
  WHERE
    <bib>
      <book>
        <author><last>$l</last><first>$f</first></author>
      </book>
    </bib> IN "www.bn.com/bib.xml"
  CONSTRUCT
    <result>
      <author><last>$l</last><first>$f</first></author>
      {
        WHERE
         <bib>
           <book>
             <title>$t</title>  // join on $l and $f
             <author><last>$l</last><first>$f</first></author>
           </book>
         </bib> IN "www.bn.com/bib.xml"
        CONSTRUCT <title>$t</title>
      }
    </result>
} </results>
```

In this XML-QL query, the occurences of `$l` and `$f` in the outer `WHERE` clause causes them to be bound, while their occurrence in the inner `WHERE` clause tests for equality. One `result` element is constructed for each last name, first name pair and contains one `author` element and one or more `title` elements, which are constructed by the nested query.

YATL

```
make
  results [
    *result [
      author [ last [ $l ], first [ $f ] ],
      ( make
          *title [ $t ]
        match "www.bn.com/bib.xml" with
          bib [ *book [ *author [ last [ $l ], first [ $f ] ],
                        title [ $t ] ] ] ) ] ]
match "www.bn.com/bib.xml" with
  bib [ *book [ *author [ last [ $l ], first [ $f ] ] ] ]
```

Like XML-QL, YAT$_L$ uses a nested query to join `author` elements on their first and last names.

Lorel

```
select xml(results:
  (select xml(result:{author: a,
                      (select xml(title: t)
                       from bib.book b, b.title t
                       where b.author.first = a.first and
                             b.author.last = a.last)})
   from bib.book.author a))
```

Like XML-QL and YAT$_L$ Lorel uses a nested query to join `author` elements on their first and last names.

XQL

Even though XQL can express joins (see Section **??**), it cannot express this query, which requires flattening multiple instances of an author's name to produce a single element for each author.

This query demonstrates the importance of the cross-product semantics chosen by the other three languages. To restructure data completely, one must first flatten the data and then reconstruct it in a new way.

2.5 Changing structure by explicit grouping

In addition to grouping by nesting, XML-QL, YAT$_L$, and Lorel provide other constructs to support grouping, which may sometimes be easier to use. YAT$_L$ has a grouping operator, while XML-QL and Lorel both provide Skolem functions for this purpose. We show how to rewrite the query of the previous section using these operators. As before, the query groups each author with the titles he or she has written.

YATL

```
make
  results [ *($l,$f) result [ author [ last [ $l ],
                                       first [ $f ] ],
                              *title [ $t ] ] ]
match "www.bn.com/bib.xml" with
  bib [ *book [ title [ $t ],
              *author [ last [ $l ],
                        first [ $f ] ] ] ]
```

As before, the `match` clause produces one tuple for each binding of `$t`, `$l`, and `$f` that satisfies the pattern. Previously, this led to a flattening effect. Here, the results are nested again by grouping over the last and first name, as indicated by writing `($l,$f)` between `*` and `result` in the `make` clause. This expresses more compactly the same result as the previous nested query.

XML-QL

```
CONSTRUCT <results> {
  WHERE
    <bib>
      <book>
        <title>$t</title>
        <author><last>$l</last><first>$f</first></author>
      </book>
    </bib> IN "www.bn.com/bib.xml"
  CONSTRUCT
    <result ID=author($l,$f)>
      <title>$t</title>
      <author><last>$l</last><first>$f</first></author>
    </result>
} </results>
```

In XML, attributes with type `ID` uniquely identify the elements which bear them: only one element in the document can have an `ID` attribute with the given value. In XML-QL, the distinguished attribute `ID` is taken to have type `ID` and is used to control grouping. Here the value of the attribute is `author($l,$f)`, which denotes some unique function of the last and first name, called a *Skolem function*. This causes all of the separate `result` elements with the same last and first names to be grouped together, the result being a single element with one author and multiple titles.

Lorel

```
select Root()->result->Author(l,f),
       Author(l,f)->author->a,
       Author(l,f)->title->t
from bib.book b, b.author a, a.first f, a.last l, b.title t
```

The syntax for Skolem functions in Lorel reflects its underlying data model and is somewhat different from that presented in earlier queries. This query uses two Skolem functions to create the desired structure. The `Root` Skolem function, which accepts no parameters, creates a single element, with multiple `result` sub-elements. One result sub-element is created by the `Author` Skolem function for each distinct pair of bindings of `l` and `f`. The elements created by `Author` have sub-elements for the authors and titles of their books.

XQL

XQL does not support an explicit grouping operator.

2.6 Combining data sources

Now, we will see how to combine information collected from different portions of documents, which is necessary to merge information from multiple documents. For the next query, assume that we have a second data source at `www.amazon.com/reviews.xml` that contains book reviews and prices, with the following DTD:

```
<!ELEMENT reviews (entry*)>
<!ELEMENT entry   (title, price, review)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT price   (#PCDATA)>
<!ELEMENT review  (#PCDATA)>
```

The example query lists all books with their prices from both sources.

XML-QL

```
CONSTRUCT <books-with-prices> {
  WHERE
    <bib>
      <book>
        <title>$t</title>
        <price>$pb</price>
      </book>
    </bib> IN "www.bn.com/bib.xml",
    <reviews>
      <entry>
        <title>$t</title>
        <price>$pa</price>
      </entry>
    </reviews> IN "www.amazon.com/reviews.xml"
  CONSTRUCT
    <book-with-prices>
      <title>$t</title>
      <price-amazon>$pa</price-amazon>
      <price-bn>$pb</price-bn>
    </book-with-prices>
} </books-with-prices>
```

Note that the use of the same variable `$t` for both titles causes a join between the two data sources.

Even though this join operation may look expensive, numerous techniques have been developed to evaluate joins efficiently [?].

YATL

```
make
  books-with-prices
     *book-with-prices [ title [ $t ],
                         price-amazon [ $pa ],
                         price-bn [ $pb ] ]
match "www.bn.com/bib.xml" with
  bib [ *book [ title [ $t ],
                price [ $pb ] ] ],
      "www.amazon.com/reviews.xml" with
  reviews [ *entry [ title [ $t ],
                      price [ $pa ] ] ]
```

Again, this query is almost identical to the one in XML-QL.

Lorel

For the corresponding Lorel query, we will use the entry point `reviews` to access the data in the second XML document.

```
select xml(books-with-prices:
  (select xml(book-with-prices: { title: t,
                                  price-amazon: pa,
                                  price-bn: pb }
   from bib.book b, b.title tb, b.price pb,
      reviews.entry e, e.title ta, e.price pa
   where tb = ta)))
```

Note that Lorel uses an explicit equality predicate on book titles to perform the join.

XQL

The XQL query is:

```
document("www.bn.com/bib.xml")/bib -> books-with-prices {
  book->book-with-prices[$t:=title] {
    title | price -> price-bn |
    document("www.amazon.com/reviews.xml")/reviews
                                          /entry[title=$t] {
       price -> price-amazon
    }
  }
}
```

In XQL, a join is performed by assigning and using variables. First, the variable assignment `$t:=title` binds the variable `$t` to book titles, then the predicate `title=$t` selects the corresponding titles from the reviews. This explicit assignment followed by its use in a selection suggests an evaluation strategy.

Indeed, the more symmetric syntax of the other languages preserves the tradition of well-known query languages, such as SQL and OQL [?], which separate a query's semantics from the mechanics of its evaluation.

2.7 Indexing

Elements in an XML document are ordered. In some cases, it might be important to refer to the order in which elements appear, to preserve the order in the output, or to impose a new order. Our next query returns each book with its title and the first two authors, and an `<et-al/>` element if there are more than two authors.

XQL

```
document("www.bn.com/bib.xml")/bib/book {
        title | author[1 to 2] | author[3]->et-al { }
}
```

XQL uses subscripts to indicate indexes. A subscript can contain single numbers, ranges, or any combination of these. For instance, the expression `author[1 to 2]` selects the first two authors. The third `author` element is renamed to an empty `et-al` element.

Lorel

```
select xml(bib:
  (select xml(book:{ title: t,
                     (select b.author[1-2]),
                     (select xml(et-al {})
                     where exists b.author[3]) })
   from bib.book b, b.title t))
```

Lorel uses two nested queries to construct the result. The first selects authors with index one or two. The second produces an `<et-al/>` element if there exists an author of index three.

XML-QL

```
CONSTRUCT <bib> {
  WHERE
    <bib>
      <book>
        <title>$t</title>
        <author[$i]>$a</author>
      </book>
    </bib> IN "www.bn.com/bib.xml",
  CONSTRUCT
    <book ID=title($t)>
      <title>$t</title>
      { WHERE $i <= 2 CONSTRUCT <author>$a</author> }
      { WHERE $i = 3 CONSTRUCT <et-al/> }
    </book>
} </bib>
```

For this query, XML-QL uses *index variables*. The pattern binds three variables: the title `$t` and author `$a` are as before, and the index `$i` is bound to the position of the author element in the list of all its siblings. Indexing starts from zero, and there is always a title element before the authors, so the first author gets index one, the second gets index two, and so on. The constructor contains two nested queries. The first selects authors with index one or two. The second produces an `<et-al/>` element if and only if there exists an author of index three.

YATL

```
make
  bib *book [ title [ $t ],
              ( make *author [ $a ]
                match $as with *($$i) author [ $a ]
                where $$i <= 2 ),
              ( make [ et-al ]
                match $as with *($$i) author
                where $$i = 3 ) ]
match "www.bn.com/bib.xml" with
  bib [ *book [ title [ $t ],
                *($as) author ] ]
```

In YATL, the index variable is denoted by `$$`. We start by retrieving the title and the list of authors in variables `$t` and `$as` respectively. The make clause contains two nested queries: the first one returns the first two authors by selecting those whose index in `$$i` is less than 2, the second one creates an element `et-al` whenever a third author exists.

2.8 Sorting

Our first example selected all titles of books published by Addison-Wesley after 1991. In that example, output order was not specified. Here we go back and show how to modify the query so that the titles are listed alphabetically.

XML-QL

```
CONSTRUCT <bib> {
  WHERE
    <bib>
      <book year=$y>
        <title>$t</title>
        <publisher><name>Addison-Wesley</name></publisher>
      </book>
    </bib> IN "www.bn.com/bib.xml",
    $y > 1991
  ORDER-BY $t
  CONSTRUCT
    <book year=$y>
      <title>$t</title>
    </book>
} </bib>
```

This query is identical to the one in Section ??, except for the new ORDER-BY clause, which specifies that the resulting elements should be sorted by their titles (as opposed to, say, their years).

YAT$_L$

```
make
  bib [ *o($t) book [ @year [ $y ],
                      title [ $t ] ] ]
match "www.bn.com/bib.xml" with
  bib [ *book [ @year [ $y ],
                title [ $t ] ],
                publisher [ name [ $n ] ] ]
where
  $n = "Addison-Wesley" and $y > 1991
```

This is identical to the YAT$_L$ query in Section ??, except for the new phrase o($t), which specifies that the resulting elements should be sorted by their titles.

Lorel

```
select xml(bib:
  (select xml(book:{@year:y, title:t})
   from bib.book b, b.title t, b.year y
   where b.publisher = "Addison-Wesley" and y > 1991
   order by t))
```

The `order by` clause sorts the elements satisfying the `from` and `where` clause by book titles before creating the output document in the `select` clause.

XQL

XQL does not currently have a sorting construct. Several proposals are being considered.

Combining sorting with the indexes of the previous section can be used to list the titles in the same order as the input document. Unlike XQL, which always preserves document order, ordering in XML-QL, YATL and Lorel is explicit. This is a deliberate choice, because ordering is usually expensive.

2.9 Tag variables

Because an XML document does not always come with a DTD, we need some means to query documents without a priori knowledge of its structure or the tags of its elements.

The next query selects books in which some element tag matches the regular expression `'*or'` (e.g. author, editor, author) and whose value is "Suciu". The result of the query preserves the original tag.

XML-QL

```
CONSTRUCT <bib> {
  WHERE
    <bib>
      <book>
        <title>$t</title>
        <$a>Suciu</>
      </book>
    </bib> IN "www.bn.com/bib.xml",
    $a LIKE '*or'
  CONSTRUCT
    <book>
      <title>$t</title>
      <$a>Suciu</>
    </book>
} </bib>
```

In XML-QL, tag variables are used to query document structure. Here, the variable `$a` is bound to the tag of each sub-element of `book`. The tag must match the regular expression `’*or’`. The constructor produces elements with the same tag names. Note that the element expressions beginning with tag variables are closed by `</>`, since the opening tag is not known.

YAT$_L$

```
  make
    bib [ * book [ title [ $t ].
                   $$a [ "Suciu" ] ] ]
  match "www.bn.com/bib.xml" with
    bib [ * book [ title [ $t ],
                   *$$a [ $l ] ] ]
  where $l = "Suciu" and
        $$a like "*or"
```

In YAT$_L$, tag variables are denoted by a `$$` symbol.

Lorel

```
select xml(bib:
  (select xml(book: {title: t, xml(LabelOf(a)): l})
   from bib.book b, b.%or@a l , b.title t))
```

This query uses the path variable `a`, which is bound to the paths from `b` to `l` that match the regular expression `%or`. The `LabelOf` function returns the string representation of the `a` path. Path variables are more general than tag variables and can be bound to an arbitrary path in the document.

Tag variables allow manipulation of tags as values. For example, assume that both our data sources contain information about various types of products, e.g., books, cds, etc. In `www.bn.com/bib.xml`, a product type is modeled by an element, but in `www.amazon.com/reviews.xml`, a product type is modeled by the value of a `type` element. Using tag variables, we can generalize the join query from Section **??** over all item types by joining element tags and element values. In the XML-QL version below, `$e` is bound to the tag of product elements in one source and to the value of the `type` element in the other source. The Lorel and YATL formulations are similar.

```
CONSTRUCT <items-with-prices> {
  WHERE
    <bib>
      <$e>
        <title>$t</title>
        <price>$pb</price>
      </>
    </bib> IN "www.bn.com/bib.xml",
    <reviews>
      <entry>
        <title>$t</title>
        <type>$e</type>
        <price>$pa</price>
      </entry>
    </reviews> IN "www.amazon.com/reviews.xml"
  CONSTRUCT
    <$e>
      <title>$t</title>
      <price-amazon>$pa</price-amazon>
      <price-bn>$pb</price-bn>
    </>
} </items-with-prices>
```

This query is particularly powerful, because it can be applied to the sources without knowing all the product types (i.e., element tags or values of the `type` element) a priori. If a new product type is added to either source, this query still works without modification.

XQL

XQL does not support tag variables and therefore cannot express these queries.

2.10 Regular-path expressions

Some queries may be conveniently specified by constraining the path through the tree, via the use of a regular-path expression. For example, the following DTD defines a self-recursive element section.

```
<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, section*)>
<!ELEMENT title (#PCDATA)>
```

A section element may contain other nested section elements to an arbitrary depth. Regular-path expressions are used to match paths of arbitrary depth. The next query retrieves all section or chapter titles containing the word "XML", regardless of the nesting level at which it occurs.

XML-QL

```
CONSTRUCT <results> {
  WHERE
    <chapter.(section)*>
      <title>$t</title>
    </> IN "books.xml",
  $t LIKE '*XML*'
  CONSTRUCT
    <title>$t</title>
} </results>
```

Here, chapter.(section)* is a regular-path expression, and matches a chapter element followed by a sequence of zero or more nested section elements. Regular-path expressions are combined with the alternation (|), concatenation (.), and Kleene-star (*) operators.

Lorel

```
select xml(results:
  (select xml(title:t)
   from chapter(.section)* s, s.title t
   where t like "*XML*"))
```

The path expression component chapter(.section)* s binds the variable s to all elements reachable by following a chapter and a sequence of section elements.

XQL

XQL does not support regular-path expressions but it does support access to immediate children and descendants using the / and // operators respectively. This allows queries of arbitrary depth over unconstrained paths.

The following query selects the title of chapters and sections that that contain the string 'XML', but does not require section elements to be contained within a chapter.

```
document("books.xml")->results {
  chapter[title contains "XML" ] { title } |
  .//section[title contains "XML" ] { title }
}
```

YAT$_L$

YAT$_L$ does not currently provide regular-path expressions.

With regular-path expressions, it is possible to write queries that potentially are expensive to evaluate, e.g., one that returns the entire document. Techniques exist, however, to evaluate certain classes of regular-path expressions efficiently [**?**, **?**].

3 Additional Features

We believe all the examples in the previous section illustrate the essential features of an XML query language. In this section, we present several other useful, but less crucial, features.

3.1 External functions and aggregation

Different application domains often require specialized operations. For instance, decision support applications need aggregate functions and integration applications require approximate comparison between string values [**?**].

For instance, the following XML-QL query accesses a price list collected from multiple bibliographies and returns the minimum price for each book.

XML-QL

```
CONSTRUCT <results> {
  WHERE <book>
          <title>$t</title>
          <price>$p</price>
        </book> IN "books.yahoo.com/prices.xml"
  CONSTRUCT <minprice ID=title($t)>MIN($p)</minprice>
} </results>
```

This query extracts all title and price pairs, then it groups prices for the same book together via a Skolem function. Finally the MIN aggregate function returns the corresponding minimal price.

YAT$_L$

The following YAT$_L$ query computes the average number of authors for all books.

```
  make
    avg([ *count($as) ])
  match "www.bn.com/bib.xml" with
    bib [ * book [ *($as) author ] ]
```

YAT$_L$ uses a functional approach in which aggregate functions are simply functions on collections. This query uses two aggregate functions: count returns the size of its input collection (here the list of authors in each book) and avg for the average.

3.2 Processing of alternatives

XML DTDs provide constructs to describe alternative structure. For instance, the books in our bibliography have either authors or editors. In the next query, we want to extract from each book, either the full content of author elements or the affiliation of the editors.

This can be written in YAT$_L$ as:

YAT$_L$

```
  make
    bib * ( match $b with
               | *($as) author
```

```
                make
                    book [ title [ $t ],
                           $as ]
                | * editor [ affiliation [ $af ] ]
                make
                    reference [ title [ $t ],
                                org [ $af ] ] )
  match "www.bn.com/bib.xml" with
    bib [ * book($b) ]
```

The nested query matches each book from the bibliography with two different patterns separated by the alternative construct `|`. This is similar to pattern-matching in functional programming languages or case statements in imperative languages. If the book matches `*($as) author` then the first make clause is applied, keeping the book title and its authors. If it matches the `* editor [ affiliation [ $af ] ]` pattern, then a reference containing the title and the organization is created.

In XML-QL, alternatives are handled using parallel, nested queries. Each nested query handles one alternative, but they are not mutually exclusive.

XML-QL

```
WHERE <bib>
        <book>
          <title>$t</title>
        </book> CONTENT_AS $b
       </bib> in "www.bn.com/bib.xml"
CONSTRUCT <bib> {
  { WHERE     <author>$a</author> in $b
    CONSTRUCT <book ID=Book($t)><title>$t</title>
                                <author>$a</author></book>
  }
  { WHERE     <editor><affiliation>$af</affiliation>
              </editor> in $b
    CONSTRUCT <reference><title>$t</title>
                         <org>$af</org></reference>
  }
} </bib>
```

3.3 Universal quantification

In some queries, it might be useful to check whether a property holds for *all* elements of a collections. For instance, the next query asks for all couples of books having exactly the same set of authors. This query requires the ability to compare sets of values.

This can be done in Lorel with the following query:

Lorel

```
select xml(original: x, copy: y)
from bib.book x, bib.book y
where for all z in x.author: exists w in y.author: z = w and
      for all t in y.author: exists s in x.author: t = s;
```

The first filter verifies that the authors of `x` are also authors of book `y`, and the second filter checks for the opposite set inclusion. The predicate `for all` is used to universally quantify over all authors.

In YAT$_{\mathrm{L}}$, one can use directly set equality for the same purpose:

YAT$_{\mathrm{L}}$

```
make
  * [ original [ $b1 ],
      copy [ $b2 ] ]
match URL with
  *book($b1) { *($a1) author },
      URL with
  *book($b2) { *($a2) author },
where $a1 = $a2
```

The variables `$a1` and `$a2` contain the sets of authors for each book `$b1` and `$b2`. The filter `$a1 = $a2` tests for the set equality.

XML-QL can express this with a rather complex, nested query, which uses negation and the `isEmpty` predicate. This predicate returns true if its sub-query evaluates to an empty answer. By checking whether there does not exist an author which is in one book and not in the other, XML-QL can provide functionalities similar to universal quantification. XQL cannot express this query.

3.4 Data models and navigation

Now imagine a slight change to our database. For each author, we maintain not only the name, but also an affiliation and an e-mail address. To avoid duplicating this information, we assign a unique `ID` to each author, and change the book elements to refer to the authors by their `ID`. Here is the revised DTD. (We omit the `title`, `editor`, `publisher`,`price`, `first`, `last`, `affiliation`, and `e-mail` elements, which just contain text.)

```
<!ELEMENT bib (book*, person*)>
<!ELEMENT book (title, publisher, price)>
<!ATTLIST book year CDATA>
<!ATTLIST book author IDREFS>
<!ATTLIST book editor IDREFS>
<!ELEMENT person (last, first, affiliation?, e-mail?)>
<!ATTLIST person ID ID>
```

Now, `author` is an attribute of type `IDREFS`, which refers to the corresponding `person` elements. This example illustrates that the same data can be represented in XML in various ways. A complex value can be represented directly by a sub-element or indirectly by a reference. An atomic value can be represented by a sub-element or by an attribute.

Typically, query languages are defined with respect to a *data model*, not the data's physical representation. A data model can be used to unify these different representations. XQL supports a document-based model which distinguishes between representations. XML-QL and YAT$_{\text{L}}$ both support a graph data model, which unifies embedded components and references. Lorel supports both models and can unify attributes and sub-elements.

The advantage of a unifying data model is that queries can be written independently of the underlying representation.

Lorel

For example, with the graph-based model, the following query from Section **??** can be used unchanged:

```
select xml(results:
  (select xml(result:{title: t,
                      author: a})
    from bib.book b, b.title t, b.author a))
```

However, with the document-based model, the queries must be changed completely. The modified query requires an explicit join to access the referenced elements:

```
select xml(results:
  (select xml(result:{title: t, author: p})
    from bib.book b, b.title t, b.author a, bib.person p
    where p.ID = a))
```

The `from` clause binds `a` to `author` attribute and binds `p` to the content of `person` elements. The `where` clause selects the `person` whose `ID` attribute equals `a`, i.e., it is a join on `bib.book.author` and `bib.person.ID`. The output document constructs an `author` element whose contents is the contents of the corresponding `person`.

XML-QL

Because XML-QL uses the graph-based model, the query from Section **??** still applies to the newly reorganized data:

```
CONSTRUCT <results> {
  WHERE
    <bib>
      <book>
        <title>$t</title>
        <author>$a</author>
      </book>
    </bib> IN "www.bn.com/bib.xml"
  CONSTRUCT
    <result>
      <title>$t</title>
      <author>$a</author>
    </result>
} </results>
```

YATL

```
make
  results [ *result [ title  [ $t ]
                      author [ $a ] ] ]
match "www.bn.com/bib.xml" with
```

```
  bib [ *book [ title [ $t ],
                @author [ *$a ] ] ]
```

The YATL query requires a small modification to access the author attribute. The variable `$a` is bound to the referenced element.

XQL

Suppose we wish to modify the following query to make it work on the new document:

```
document("http://www.bn.com")/bib->results {
    book->result {
        title | author
  }
}
```

Even those XQL uses a document-based model, it provides an explicit indirection operation, the `id()` function, which takes a string and returns the element whose id matches the parameter:

```
document("www.bn.com/bib.xml")/bib/book {
        title | author/id(@IDREF)
}
```

4 Conclusion

In this paper, we have presented features from four XML query languages that support the requirements of various applications that will process, integrate, and transform XML data sources.

Based on our experience designing, implementing, and using database query languages, we think these languages provide a good compromise between expressive power, simplicity and performance. We believe an XML query language should take advantage of this experience.

Although this paper emphasizes the expressive power and declarativity of these languages, we are confident that these features can be evaluated efficiently. Many well-known optimization techniques exist for the most expensive features described, such as joins [?], nested queries [?], path expressions [?, ?], and aggregation functions [?].

References

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Deplhi, Greece, 1997. Springer-Verlag.
`http://cosmos.inria.fr:8080/cgi-bin/publisverso?what=abstract&query=103`

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.
`ftp://db.stanford.edu/pub/papers/lorel96.ps`

[3] C. Beeri and T. Milo Schemas for Integration and Translation of Structured and Semi-Structured Data. In *Proceedings of the International Conference on Database Theory*, Jerusalem, Israel, 1999. Springer Verlag.
`ftp://ftp.math.tau.ac.il/pub/milo/icdt99-2.ps.Z`

[4] P. Buneman. Tutorial: Semistructured data. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 117–121, 1997.
`http://www.acm.org/pubs/citations/proceedings/pods/263661/p117-buneman/`

[5] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.
`http://www.research.att.com/~mff/files/icdt97.ps.gz`

[6] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[7] R. G. Cattell The Object Database Standard: ODMG 2.0. Morgan Kaufmann, 1997.

[8] S. Chaudhuri and K. Shim An Overview of Cost-based Optimization of Queries with Aggregates. In *Data Engineering Bulletin 18(3)*, 1995.
`http://www-db-in.research.bell-labs.com/~shim/bulletine95.ps.gz`

[9] V. Christophides, S. Cluet and G. Moerkotte Evaluating Queries with Generalized Path Expressions. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1996.
`http://cosmos.inria.fr:8080/cgi-bin/publisverso?what=abstract&query=080`

[10] C. L. A. Clarke, G. V. Cormack and F. J. Burkowski An algebra for structured text search and a framework for its implementation. In *The Computer Journal, 38(1)*, 1995.

[11] S. Cluet, C. Delobel, J. Siméon and K. Smaga Your Mediators Need Data Conversion! In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 177-188, 1998.
`http://cosmos.inria.fr:8080/cgi-bin/publisverso?what=abstract&query=138`

[12] S. Cluet, S. Jacqmin and J. Siméon The New YATL: Design and Specifications. *Working draft.*

[13] S. Cluet and G. Moerkotte Nested Queries in Object Bases In *Proceedings of International Workshop on Database Programming Languages*, 1993.
`http://cosmos.inria.fr:8080/cgi-bin/publisverso?what=abstract&query=064`

[14] W. W. Cohen: Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

[15] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *International World Wide Web Conference*, 1999.
`http://www.research.att.com/~mff/files/final.html`

[16] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.
`http://www.research.att.com/~mff/files/sigmod98.ps.gz`

[17] M. Fernandez and D. Suciu Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of International Conference on Data Engineering*, 1998.
`http://www.research.att.com/~mff/files/icde98.ps.gz`

[18] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, Philadelphia, Pennsylvania, June 1999.

[19] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of Very Large Data Bases*, pages 436–445, September 1997.

[20] G. Graefe Query Evaluation Techniques for Large Databases. In *Computing Surveys 25(2)*, 1993.

[21] D. Maier. Database Desiderata for an XML Query Language. In *W3C Workshop on Query Languages for XML.*
`http://www.w3.org/TandS/QL/QL98/pp/maier.html`

[22] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of VLDB*, Edinburgh, UK, September 1999.
`http://www-db.stanford.edu/~mchughj/publications/qo_short.ps`

[23] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.
`http://www.research.att.com/~suciu/workshop-papers.html`

[24] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.
`http://www.db.ucsd.edu/publications/icde95.ps`

[25] J. Robie. The design of XQL, 1999.
`http://www.texcel.no/whitepapers/xql-design.html`

[26] J. Robie, editor. XQL '99 Proposal, 1999.
`http://metalab.unc.edu/xql/xql-proposal.html`

[27] D. Suciu. An overview of semistructured data. *SIGACT News*, 29(4):28–38, December 1998.
`http://www.research.att.com/~suciu/strudel/external/files/_F242554565.ps`