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, XUL, 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 [15], YATL [11, 12], and Lorle [2, 18]. 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 [10]. From this community, one language that has emerged for processing XML data is XQL [26, 25].

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, YATL, 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, YATL, 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, YATL, 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 [21]. We also refer the reader to a substantial body of research, including the motivation for and typical applications of semistructured data, [1, 4, 27], data models for semistructured data [24],
query-language design [2, 6, 16], query processing and optimization [22],
schema languages [3, 5, 19], and schema extraction [23].

The next section presents what we consider to be ten essential queries.
Section 3 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, YAtI, 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 doc-
ument www.bn.com/bib.xml, containing bibliography entries described by
the following DTD.

```xml
<!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 spec-
ified for the output. In Section 2.8, we will show how to sort the titles by
document or alphabetical order.
In XML-QL, YATL, 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, YATL, 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).

YATL

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 YATL 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 YATL, 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 \texttt{document("http://www.bn.com")/bib}
selects all top-level \texttt{bib} elements from the input document and evaluates the
nested expression for each such element. The nested pattern \texttt{book}
selects the book elements that are children of a \texttt{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 \texttt{bib} element that contains the selected \texttt{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

\begin{verbatim}
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>
\end{verbatim}

The \texttt{WHERE} clause produces one tuple for each binding of \texttt{$t$} and \texttt{$a$}
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 \texttt{CONSTRUCT} clause produces one \texttt{result} element for each pair of values bound to \((\$a, \$t)\), i.e., the constructor's free variables.

\textbf{YATL} \\
make 
results [ *result [ title [ \$t ], 
author [ \$a ] ] ] 
machine "www.bn.com/bib.xml" with 
bib [ *book [ title [ \$t ], 
*author [ \$a ] ] ] 

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

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

\textbf{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 \texttt{b.author} denotes the set of values for author; similarly, \texttt{b.title} denotes the set of titles, if there are more than one. Each binding for \texttt{b} produces a new \texttt{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:

\begin{verbatim}
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$</author>
    }
  </result>
} <//results>
\end{verbatim}

Here, \texttt{CONTENT AS} binds the variable \texttt{b} to the \texttt{book} element's content, a collection of elements. The inner \texttt{WHERE} clause selects the \texttt{author} elements from \texttt{b}. It is also possible to preserve structure in XML-QL without nested queries by using an explicit grouping construct (see Section 2.5).

### 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 \texttt{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 2.6.
In this XML-QL query, the occurrences 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.
Like XML-QL, YATL uses a nested query to join *author* elements on their first and last names.

**Lorel**

```sql
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 YATL, Lorel uses a nested query to join *author* elements on their first and last names.

**XQL**

Even though XQL can express joins (see Section 2.6), 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, YATL, and Lorel provide other constructs to support grouping, which may sometimes be easier to use. YATL 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.
\texttt{YATL}

\begin{verbatim}
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.
\end{verbatim}

\texttt{XML-QI}

\begin{verbatim}
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 \texttt{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-QI, the distinguished attribute \texttt{ID} is taken to have type \texttt{ID} and is used to control grouping. Here the value of the attribute is \texttt{author($l,$f)}, which denotes some unique function of the last and first name, called a \textit{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**

```xml
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:

```xml
<!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 [20].

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.

```xml
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:

```xml
    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 [7], 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**

    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",}
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 \(<\text{et-al}/>\) element if and only if there exists an author of index three.

\[ \text{YATL} \]

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

In \text{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 \text{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>
      <bib>
        <book year=$y>
          <title>$t</title>
        </book>
      </bib>
    WHERE
      <bib>
        <book year=$y>
          <title>$t</title>
        </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 2.1, 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).

**YATL**

```
make
  bib [ *c($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 YATL query in Section 2.1, except for the new phrase `c($t)`, which specifies that the resulting elements should be sorted by their titles.
Lorel

```xml
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**

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

YATL

```yattl
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 YATL, tag variables are denoted by a $$ symbol.

Lorel

```lore
select xml(bib:
    (select xml(book: {title: t, xml(LabelOf(a)): l})
    from bib.book b, b.%cr@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 2.6 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.

```xml
<!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**

```xml
CONSTRUCT <results> {  
  WHERE  
    <chapter.(section)*>  
      <title>$t</title>  
    </chapter> 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**

```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)*` 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")-&gt;results {
chapter[title contains "XML"] { title }
.///section[title contains "XML"] { title }
}

YATL

YATL 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 [9, 17].

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 [14].

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\textsubscript{L}

The following YAT\textsubscript{L} query computes the average number of authors for all books.

\begin{verbatim}
make
    avg([ *count($as) ])
match "www.bn.com/bib.xml" with
    bib [ * book [ *($as) author ] ]
\end{verbatim}

YAT\textsubscript{L} uses a functional approach in which aggregate functions are simply functions on collections. This query uses two aggregate functions: \texttt{count} returns the size of its input collection (here the list of authors in each book) and \texttt{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\textsubscript{L} as:

\begin{verbatim}
YAT\textsubscript{L}
make
    bib * ( match $b with
        | *($as) author
\end{verbatim}
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>
</bib> CONTENT_AS $b
</bib> in "www.bn.com/bib.xml"
CONSTRUCT <bib> { 
    { WHERE <author>$a</author> in $b
        CONSTRUCT <book ID=Book($t)>$t</title>
                      <author>$a</author></book>
    }
    { WHERE <editor><affiliation>$af</affiliation>
       </editor> in $b
        CONSTRUCT <reference>$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 collection. 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 YATL, one can use directly set equality for the same purpose:

YATL

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.)

```xml
<!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 YaTL 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 2.2 can be used unchanged:

```sql
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:

```sql
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 2.2 still applies to the newly reorganized data:

```sql
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**

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

The XML 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 though 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:

  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 [20], nested queries [13], path expressions [9, 17], and aggregation functions [8].
References


ftp://db.stanford.edu/pub/papers/lorel96.ps


http://www.acm.org/pubs/citations/proceedings/pods/263661/pl17-buneman/


http://www-db-in.research.bell-labs.com/~shim/bulletine95.ps.gz

http://cosmos.inria.fr:8080/cgi-bin/publisverse?what=abstract&query=080


