Towards Declarative XML Querying

Mengchi Liu
School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6
mengchi@scs.carleton.ca

Tok Wang Ling
School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
lingtw@comp.nus.edu.sg

Abstract

How to extract data from XML documents is an important issue for XML research and development. However, how to view XML documents determines how they can be queried. In this paper, we first describe a natural way to view XML documents as in complex object data models so that we can easily comprehend XML data from database point of view. We then illustrate how to use logical variables to extract data from XML documents. We also describe a rule-based declarative query language for XML. We demonstrate that our rule-based language provides a uniform framework that is advantageous over other XML query languages including XQuery in the following ways. First, it provides a natural way for separating querying and result constructing using the body and the head respectively. Second, several rules can be used for the same query so that complex queries can be expressed in a simple and natural way. Also, its use of logical variables and rules makes many functions and operators in XQuery and XPath unnecessary or definable constructively. Finally, it provides a natural and direct support for recursion as in deductive databases and has logical foundations that have played a significant role in database research in the past.

1 Introduction

How to extract data from XML documents is an important issue for XML research and development. Various XML query languages have been proposed in the past several years, such as Lorel [2], XML-GL [5], XQL [22], XPath [9], XML-QL [11], XSLT [8], YATL [10], XDuce [13], XQuery [7], etc. For a comparative analysis of some of these language, see [3]. Some of them are in the tradition of database query languages, others more closely inspired by XML. The XML Query Working Group has recently published XML Query Requirements for XML query languages [6]. The discussion is going on within the World Wide Web Consortium, within many academic forums and within IT industry, with XQuery been selected as the basis for an official W3C query language for XML.

However, there are several serious problems with existing XML query languages, including XQuery. Firstly, they are based on a low level data model such as XML Query Data Model [12] which use various nodes, such as element, attribute, text, comment, and trees of nodes to represent XML documents and force users to view XML documents from such a low level programming point of view. Furthermore, they forces users to navigate through the trees of XML documents in their queries, which makes the queries hard to comprehend.

Secondly, most of the existing XML query languages are based on SQL and OQL [4]. Unlike queries on traditional relational databases whose results are always flat relations, the results for XML queries are complex. Thus, XML queries have to have two components: querying part and result constructing part. The existing XML query languages have to intermix the querying part and result constructing in a nested way and thus make queries complicated, which is an inherent problem inherited from SQL and OQL. For example, in XML-QL, there are explicit two constructs: where and construct. The where clause is used for querying and the construct clause is for result constructing. However, the construct clause can contain nested where and construct clauses so that querying and result constructing are intermixed. XQuery extends the two constructs in XML-QL into four constructs: for, let, where and return, i.e., FLWR expressions. The for and let clauses are used for variable bindings. The for clause binds one value to a variable at a time, while the let clause binds all values to a variable. The while clause is used as a filter for the values generated by the for and let clause. The return clause is used for result constructing. Again, the return clause can contain nested FLWR expressions.

Although XQuery is based on XPath, XQL, XML-QL, SQL, and OQL, it is an algebraic language that relies on a number of predefined functions and operators, such as document, text, node, attribute, element, child, parent, descendent, position, filter, shadow, etc. Some of these functions and operators are unintuitive, which makes the query language hard to learn and remember.

Thirdly, recursion cannot be expressed using the normal querying and result constructing constructs. Instead, user-defined functions have to be used to handle recursion. However, the purpose of functions is for general and common computation that would be needed for many times rather than a sideway for other purpose. In our view, this is the drawback of the language design anyway.

Finally, existing XML query languages lack logic foundations that have played a significant role in database research in the past.

In this paper, we describe a natural way to view XML documents as in complex object data models [1], so that we can easily comprehend XML data from database point of view. Based on this view, we then illustrate how to use logical variables to extract data from XML documents. We also describe a rule-based declarative query language for XML. We demonstrate that our rule-based language provides uniform framework that is advantageous over other XML query languages including XQuery in the following ways. First, it provides a natural way for separating querying and result constructing using the body and the head respectively. As a result, no matter how complex the query is, the querying part and the result constructing part are strictly separated. Second, several rules can be used for the same query so that complex queries can be expressed in a simple and natural way. Also, the use of logical variables and rules makes many functions and operators in XQuery and XPath are unnecessary or definable constructively in our language. Finally, it provides a natural and direct support for recursion as in deductive databases and has logical foundations [18].

The rest of the paper is organized as follows. Section 2 discusses how to model XML documents as in databases. Section 3 investigates how to query XML documents declaratively. Section 4 shows how to use rule-based language for query result constructing. Section 5 summarizes and points out further research issues.

2 Modeling XML Documents as in Databases

In this section, we show that we do not have to view XML data in a low level as in XML Query Data Model [12]. Instead, we can view it as a complex object in complex object data models. With such a view, query representation becomes also higher level as well. Consider the following simple XML document:

```
<person id="o111">
    <name>
        <first>John</first>
        <last>Smith</last>
    </name>
    <age>25</age>
</person>
```

where *person, name, first, last, age* are element names and *id* is an attribute name. This XML document is represented in XML Query Data Model as a tree shown in Figure 1, in which *person, name, last, first* are element nodes, *id* is an attribute node, and *o111, John, Smith, 25* are text nodes.

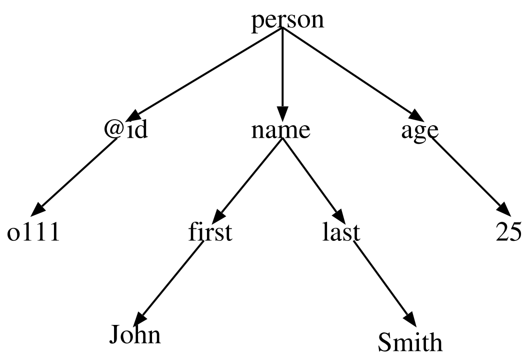

Figure 1. XML Query Data Model Representation

However, this kind data is not new to the database community. We can view it as a complex object in complex object data model as follows:

```
person ⇒[
    @id ⇒o111,
    name ⇒[
        first ⇒John,
        last ⇒Smith],
    age ⇒25]
```

We call this complex object as *element object*, which is a pair of element name and element value. The element value [@id ⇒o111, name ⇒[first ⇒John, last ⇒Smith], age ⇒25] is a *tuple object*, which contains an attribute object @id ⇒o111, nested tuple object [first ⇒John, last ⇒Smith], and an element object age ⇒25. o111, John, Smith, 25 are lexical objects, The symbol @ in front of id denotes id is an attribute, ⇒ is used to separate element name from element value, attribute name from attribute value.

By comparing with XML Query Data Model, here we have introduced several higher-level notions such as *ele-*

```
<people>
    <person id="o123">
        <name>Jane</name>
    </person>
    <person id="o234" mother="o456">
        <name>John</name>
    </person>
    <person id="o456" children ="o123 o234">
        <name>Mary</name>
    </person>
    <person id="o567">
        <name>Joan</name>
    </person>
    <person id="o678" children ="o456 o567">
        <name>Tony</name>
    </person>
    <person/>
</people>
```

Figure 2. Sample XML Document 1

ment object, *attribute object*, *tuple object*, *lexical object* that naturally correspond to XML notations. However, element values are different from element contents in XML as element values contain attributes and element contents that elements have.

Besides the above objects, we also need other kinds of objects. Consider the XML document shown in Figure 2.

It is represented in our data model as an element object as follows:

```
people ⇒ [
    person ⇒[
        @id ⇒o123,
        name ⇒Jane],
    person ⇒[
        @id ⇒o234,
        @mother ⇒o456,
        name ⇒John],
    person ⇒[
        @id ⇒o456,
        @children ⇒{o123,o234},
        name ⇒Mary]
    person ⇒[
        @id ⇒o567,
        name ⇒Joan]
    person ⇒[
        @id ⇒o678,
        @children ⇒{o456,o567},
        name ⇒Tony]
    person ⇒null]
```

Note here the attribute name *children* has list values such as {o123,o234} and {o456,o567}, and there are several elements with the same element name *person* but different values, especially null value.

From querying point of view, if we just want one value of the element *person* at a time, then the above representation is enough. However, if we want all the values of the element *person*, then it is not clear what should be returned from this representation. To solve this problem, we extend the notion *list object* to cover element values and represent the above XML document as follows:

```
people⇒[
    person⇒{
        [@id ⇒o123, name⇒Jane],
        [@id⇒o234, @mother⇒o456, name⇒John],
        [@id⇒o456, @children⇒{o123,o234},
         name⇒Mary]
        [@id ⇒o567, name ⇒Joan]
        [@id ⇒o678, @children ⇒{o456,o567},
         name ⇒Tony]
        null}]
```

In other words, we can treat *person* element as list-valued if we are interested in all of its values. This extension is necessary as demonstrated by the use of *LET* expression in XQuery.

Thus, elements in tuple objects here can have two kinds of representations: single-valued and list-valued. Similarly, attributes in tuple objects can also have two kinds of representations: single-valued and list-valued as the value of an attribute can be a list object. Therefore, tuple objects can be represented in different ways: flat representation in which there is no list objects, integrated representation in which list objects if any must be used, regular representation in which element objects use flat representation and attribute objects use integrated representation.

Example 1 The following examples demonstrate their difference:

```
[@children⇒o123, @children⇒o234,
 title ⇒XML, author⇒John, author ⇒Tony]

[@children⇒{o123,o234},
 title⇒XML, author⇒{John, Tony}]

[@children⇒{o123,o234},
 title⇒XML, author⇒John, author ⇒Tony]
```

where the first representation is flat, the second representation is integrated and the last representation is regular. Regular representation directly corresponds to how data is represented in XML.

XML also allows mixed content documents. Consider the following example:

```
<Address>
    John lives on
    <Street>Main St</Street>
    with house number
    <Number>123</Number>
    in
    <City>Ottawa</City>
    <Country>Canada</Country>
</Address>
```

Our notion of tuple object naturally covers this case as follows:

```
Address⇒ [
    John lives on,
    Street⇒Main St,
    with house number,
    Number⇒123,
    in,
    City⇒Ottawa,
    Country⇒Canada]
```

Note that if ',' and '⇒' occurs in the XML document, they are simply part of the strings that are not interpreted as ',' and '⇒' in our data model.

A well-formed XML document over the web has a URL that specifies its location and contains exactly one root element. The URL consists of the hostname, domain name, directories and the filename. In XQuery, the XML document is obtained through the *document* function. In our framework, we treat the URL and the associated element as first class citizen.

Consider the sample XML document at URL *www.abc.com/people.xml* shown in Figure 3, where *www.abc.com/people.xml* is the URL, *www* is the hostname, *abc.com* is the domain name, *people* is the directory name, and *people.xml* is the file name. It contains one bib element with two book sub-elements and one journal sub-element. Its representation in our data model in regular representation as an XML object is shown in Figure 4.

Note that the data model describe here is just intended for us to conceptualize XML documents based on which we can express queries.

3 Declarative Querying

Our objective is to be able to extract data from XML documents in a declarative way. Thus, we use logical variables for querying. Logical variables are place holders and are different from variables in procedural languages. They are used to bind/match the values at their places. Logical variables can be typed or non-typed. Based on the data model introduced above, we have different kinds of objects in XML document: lexical, list, tuple, attribute with attribute name and value, element with element name and value. Therefore, we should have corresponding variables so that we can match various components: lexical variables, list variables, tuple variable, attribute variables, attribute name variable, attribute value variables, element variables, element name variable, and element value variables. For URL, we may query host name, domain name, directories and file name that contain certain XML documents. Thus we need corresponding variables for them as well.

```
<bib name="IT" year="1998">
    <book year="1995">
        <title>Databases</title>
        <author><last>Date</last><first>C.</first>
        </author>
        <publisher><name>Springer</name>
        </publisher>
    </book>
    <book year="1998">
        <title>XML</title>
        <author><last>Date</last><first>C.</first>
        </author>
        <author><last>Darwen</last><first>D.</first>
        </author>
        <publisher><name>Spinger</name>
        </publisher>
    </book>
    <journal year="1998">
        <title>XML</title>
        <editor><last>Date</last><first>C.</first>
</editor>
    </journal>
</bib>
```

Figure 3. Sample XML document 2

To be flexible and easy to use in practice, we prefer non-typed variables. A variable can be used for any location but the same variable occurs in different places in the same context should mean the same value from logic point of value. Because XML objects have flat representation and integrated representation, we could partition the variables into two kinds: single-valued variables and list-valued variables, but it would be confusing and counterintuitive to have two sets of variables in the language. Therefore, we only allow single-valued variables that start with \$, and \$ itself is an anonymous variable. For a list, we use a grouping variable of the form {\$X}, which is constructed from the single-valued variable \$X that ranges over the elements in the list. Such grouping variables are common in advanced deductive database language such as F-Logic [14], ROL [15], Relationlog [17], and ROL2 [19].

```
(www.abc.com/people.xml)/
bib⇒[@name ⇒ IT,
       @year ⇒  1998,
       book⇒    [@year⇒1995,
                  title⇒Databases,
                  author⇒[last⇒Date, first⇒C.],
                  publisher⇒[name⇒Springer]],
       book⇒    [@year⇒1998,
                  title⇒XML,
                  author⇒[last⇒Date, first⇒C.],
                  author⇒[last⇒Darwen, first⇒D.],
                  publisher⇒[name⇒Springer]]
       journal⇒ [@year⇒1998,
                  title⇒XML,
                  editor ⇒[last⇒Date, first⇒C.]]
```

Figure 4. Regular Representation

3.1 URL-related Querying and Built-in Functions

As we make URL and associated XML element first class citizen in our framework, we can also use various variables for URL and URL components. Consider the following examples:

($URL)/$document

It is expected that the system should find a URL and the corresponding document over the Internet one at a time.

The following query finds two distinct URLs that contain the same document.

($URL1)/$document, ($URL2)/$document,
$URL1 ≠ $URL2

The next query is more specific. It asks for the URL for an XML document that contains a book element with 1998 for attribute *year* and XML for element *title*.

($URL)/bib/book ⇒[@year ⇒1998, title ⇒XML]

Obviously, the above queries are quite expensive. To reduce the search space, we can limit the search to a certain site as follows:

(www.abc.com/$Path)
/bib/book ⇒[@year ⇒1998, title ⇒XML]

This time, the system should just search within the given site and find the path that contains the expected XML document and bind it to the variable $Path.

We can even restrict the search to a specific directory and a specific file type at a site as follows:

(www.abc.com/$File.xml)
/bib/book ⇒[@year ⇒1998, title ⇒XML]

where $File is a variable that binds to the filename satisfying the condition, if any.

Current XPath and XQuery do not support such kind of queries.

To query XML document, we also provide functions to manipulate tuple, list, and string objects. However, unlike XPath and XQuery which use a large number of unintuitive functions for almost anything, here we only use several natural functions that are obvious and easy to understand. We use several examples to demonstrate.

The built-in functions supported in the language are used in a object-oriented fashion.

Example 2 If the variable $name binds to a string, we can check whether the string contains a specific string using the built-in function as follows:

$name.contains(XML)

This expression evaluates to true or false depending on what $name binds to. Only the string that makes it true will be retrieved.

Example 3 If the grouping variable {$Numbers} binds to a list of numbers, we can obtain their count, average, minimal, maximum, and sum using the built-in functions as follows:

{$Numbers}.count() = $countNumber
{$Numbers}.avg() = $avgNumber
{$Numbers}.min() = $minNumber
{$Numbers}.max() = $maxNumber
{$Numbers}.sum() = $sumNumber

In such an expression, if the single-valued variable $Numbers binds to something other than a number in the list, then the expression evaluates to false and nothing will be retrieved.

Example 4 If a variable {$authors} binds a list of authors, we can have the following comparison expressions using the built-in operators whose meanings are self-explanatory. Note that the expressions in the same line are related.

{$authors}.count() > 2
{$authors}.first() = [last: $Last, first: $F]
{$authors}.firstTwo() ∋ $author
{$authors}.third() = $author
{$authors}.last() = $author
{$authors}.last() = $A, {$authors}.precedes($A) = $A'
{$authors}.sort() = {$sortedAuthors}
{$authors}.distinct() = {$uniqueAuthors}

3.2 XML Document Querying

In the following, we demonstrate how to use variables to obtain or match various components in XML documents based on our data model. Our discussion is based on the two sample XML documents in Figures 2 and 3 in the last section.

Example 5 If we want to obtain the whole XML document at www.abc.com/people.xml shown in Figure 3, we can simply use a binding variable $bib to match the XML object as follows:

(www.abc.com/people.xml)/$bib

The expression is equivalent to XPath expression /bib.

Example 6 If we want to obtain the value of the *bib* element that is a tuple of two attribute objects, two *book* element objects, and one *journal* element object, we can use a binding variable $bibValue as follows:

(www.abc.com/people.xml)/bib ⇒$bibValue

In XPath, we have to use two expressions /bib/* and /bib/@* to get the same result.

Example 7 If we want to get the *title* of a book, we can use the variable $title as follows:

(www.abc.com/people.xml)
/bib/book/title ⇒$title

The expression corresponds to XPath expression /bib/title/text(). Note that here the function text() is used in XPath (also in XQuery). In fact, in XPath and XQuery, we have to know exactly what is there and use proper functions such as text(), node(), etc. in order to express the query properly. In our framework, we can simple use variables to match whatever is there so that query expression becomes a lot easier for users.

The attribute values can be queries in the same way as well.

Example 8 If we want the *year* of a book, we can use the variable $year as follows:

(www.abc.com/people.xml)
/bib/book/@year ⇒$year

The expression corresponds to XPath expression /bib/book/@year/text().

Example 9 If we want an attribute in a book element, we can use single-valued attribute variable @$attr as follows:

(www.abc.com/people.xml)/bib/book/@$attr

Example 10 Writing the complete path is not convenient for users. Thus, we also allow users to use path abbreviation as in XPath. The following are two examples:

(www.abc.com/people.xml)//last ⇒$lastname

(www.abc.com/people.xml)/bib//last ⇒$lastname

These expressions correspond to XPath expression //last/text() and /bib//last/text() respectively.

Example 11 If we want to obtain child element object in *bib*, we can use a variable $element as follows:

(www.abc.com/people.xml)/bib/$element

where $element binds one of the *book* and *journal* element objects in the *bib* element object at a time. The expression is equivalent to XPath expression /bib/*.

Example 12 If we want to obtain attribute objects with name *year* in the *bib* element object, we can use one of the following expressions that contain an attribute variable together with a selection condition:

(www.abc.com/people.xml)/bib/@$attr(@year ⇒$)

(www.abc.com/people.xml)/bib/@$attr(@year)

(www.abc.com/people.xml)/bib/@$attr(@$A),
@$A ≠ name

The attribute variable @$attr binds an attribute object and the selection condition deals with the attribute name and value. When the value is not needed, we can use an anonymous variable, as in the first one or simply omit it as in the second and third. These expressions correspond to XPath expression /bib/book/* in terms of the result.

Note that in the third expression, we do not specify the attribute name in the selection condition. This query cannot be expressed in XPath and XQuery.

Example 13 If we want to obtain the attribute object with value *IT* in the *bib* element object, we can use the following expression:

(www.abc.com/people.xml)/bib/@$attr($ ⇒IT)

Note here we do not specify the name of the attribute in the selection condition. Instead, we simply use an anonymous variable. This query also cannot be expressed in XPath and XQuery.

Example 14 If we want all attributes in the *bib* element object, we can use the grouping variable {@$attr} as follows:

(www.abc.com/people.xml)/bib/book/{@$attr}

The expression corresponds to XPath expression /bib/@*.

Element object queries are similar. The following are several exmaples.

Example 15 If we are only interested in the *book* elements, not the *journal* one, we can use one of the following expressions that contain a variable \$book together with a selection condition:

(www.abc.com/people.xml)/bib/\$book(book $\Rightarrow$\$)

(www.abc.com/people.xml)/bib/\$book(book)

(www.abc.com/people.xml)/bib/\$book(\$name),
\$name $\neq$ journal

Example 16 If we want to obtain all book element object in the *bib* element, we can use a grouping variable as follows:

(www.abc.com/people.xml)/bib/{\$books(book)}

XPath does not support such kind of matching. In XQuery, we need to use the following let construct to handle it:

Let \$doc := document("www.abc.com/people.xml")
Let \$book := \$doc/bib/book

Example 17 From all the book element objects obtained above, if we just want the second book element object, we can select it into a variable \$secondbook with the built-in location operator from the list using one of the following expressions:

(\$URL)/bib/{\$books(book)},
{\$books(book)}.second() = \$sBook (1)

(\$URL)/bib/{\$books(book)}.second() = \$sBook (2)

The first one has two separate expressions. The second combines the two expressions into a single one. These expressions correspond to XPath expression /bib/book(2)/*.

Example 18 Continue with the above result, if we want the second author of the second book, we can use two variables as follows where the grouping variable {\$authors} matches all author element objects in the second book element object and the single-valued variable \$secondauthor matches the second author element object:

(\$URL)/bib/{\$books}.second()/{\$authors}.second() =
\$secondAuthor

The expression corresponds to XPath expression /bib/book(2)/author(2)/*.

Sometimes, we want to search through XML documents and find objects that satisfy more than one conditions.

Example 19 The following example find the *book* element that have value 1998 for attribute *year* and XML for element *title*.

(www.abc.com/people.xml)
/bib/\$bookElement(@year$\Rightarrow$1998, title $\Rightarrow$XML)

The variable \$bookElement matches the second *book* element object. The expression corresponds to XPath expression /bib/book(@year ="1998" and title = "XML")

Example 20 Continue with the above example, if we are not interested in the book element, rather we just want author names, we don't need to use the variable \$bookElement. We can simply use the following expression instead:

(www.abc.com/people.xml)
/bib/book $\Rightarrow$[@year$\Rightarrow$1998, title$\Rightarrow$XML, author$\Rightarrow$\$A]

As shown above, the main difference between XPath and our querying framework is the use of binding variables which make a logical foundation for XML query language possible.

4 Rule-based Query Result Construction

As discussed in Section 2, an XML document can be viewed as a complex object as in a complex object data model. Although we can obtain information from XML documents using logical variables as discussed in the previous section, we cannot generate well-formed XML document, as we cannot construct results into XML documents. As we use logical variables for querying, it is natural to use logic-based language, especially rule-based language, for result constructing as well.

Indeed, rule-based languages provide a natural way for separating querying and result constructing using the body and the head respectively, as demonstrated in advanced deductive database languages Relationlog [17], ROL [16], and ROL2 [19], and rule-based HTML document query language [20]. Also, rule-based languages allow complex queries to be expressed using several rules.

In XQuery, the FLWR construct is not powerful enough to support recursion so that recursion has to be dealt with using user-defined functions. Rule-based languages support recursion in a natural and direct way.

In this section, we first introduce result constructing expression. The result constructing expression is similar to an XML object with an URL part and element part. The URL part specifies the URL for the file where the result will be held. When the file is the standard output, we can simply omit the URL part to simplify the expression. The element part is used to construct the result element.

Consider the following five expressions:

/results/\$b (1)

(file:/home/users/xml/result.xml)/results/\$b (2)

(file:/home/users/xml/result.xml)/results/result/\$b (3)

(file:/home/users/xml/result.xml)
/results/result $\Rightarrow$[@\$year, \$title] (4)

(file:/home/users/xml/result.xml)
/results/book ⇒[title ⇒$T, authors ⇒{$A}] (5)

The first expression tells the system to use the standard output file, i.e., the screen, for the result. The second expression gives the URL of the file. Obviously, the user should have write permission on this file. For these two expressions, the resulting element contains the object that variable $b holds. If $b matches several objects (one at a time), then the results will not be a well-formed XML document as it will have several root elements. The third expression does not have this problem as there will be only one root element *results* and each element object that $b matches will be inside a child *result* element object. The next expression constructs a child *result* element object that has an attribute denoted by variable @$year and an element denoted by variable $title. The last expression contains a grouping variable {$A} so that each author that $A binds to is grouped into a list.

A rule has two parts: querying part and result constructing part with the following form:

querying $Exp_1, ..., Exp_n$ **constructing** Exp

where $Exp_1, ..., Exp_n$ are querying expressions discussed in the previous section and Exp is a result constructing expression.

A query can be expressed using one or more rules. Let us look at several queries based on the book XML document in Figure 3.

Example 21 List book elements for books published by Springer after 1991.

querying
(http://www.abc.com/bib.xml)/bib/book⇒$b
(publisher/name ⇒Springer, year ⇒¿ 1991, title ⇒$t)
constructing
/results/book ⇒$b

The result is a list of book elements under the root element results:

```
<results>
    <book year="1995">
        <title>Databases</title>
        <author><last>Date</><first>C.</></>
        <publisher><name>Spinger</name></>
    </book>
    <book year="1998">
        <title>XML</title>
        <author><last>Date</><first>C.</></>
        <author><last>Darwen</><first>D.</></>
        <publisher><name>Spinger</name></>
    </book>
</results>
```

Example 22 Create a flat list of all the title-author pairs, with each pair enclosed in a result element.

querying
(http://www.abc.com/bib.xml)
/bib/book⇒[title ⇒$t, author ⇒$a]
constructing
/results/result ⇒[title: $t, author: $a]

Note here the variable $a in the querying part matches one author at a time and the result is

```
<results>
    <result>
        <title>Databases</title>
        <author><last>Date</><first>C.</></>
    </result>
    <result>
        <title>XML</title>
        <author><last>Date </><first>C.</></>
    </result>
    <result>
        <title>XML</title>
        <author><last>Darwen</><first>D.</></>
    </result>
</results>
```

Example 23 For each author in the bibliography, list the author's name and the titles of all books by that author, grouped inside a result element.

querying
(http://www.abc.com/bib.xml)
/bib/book ⇒[title ⇒$t, author ⇒$a]
constructing
/results/result ⇒[author ⇒$a, title ⇒{$t}]

The grouping variable {$t} in the constructing part is used to group the titles of all books by the author $a. The result is as follows:

```
<results>
    <result>
        <title>Databases </title>
        <author><last>Date </><first>C.</></>
    </result>
    <result>
        <title>XML </title>
        <author><last>Date</><first>C.</></>
        <author><last>Darwen</><first>D.</></>
    </result>
</results>
```

Example 24 For each book that has at least two authors, list the title and first two authors.

querying
(http://www.abc.com/bib.xml)
/bib/book ⇒[title ⇒\$t, author ⇒{\$a}],
{a}.count()≥2, {a}.firstTwo() ∋ \$b
constructing
/results/result ⇒[title ⇒\$t, author ⇒{\$b}]

In deductive database language Datalog, a query can be expressed using several rules with different head or temporary relations. Thus, Datalog does not have universal and existential quantifiers as they can be transformed into equivalent rules without them [21]. For XML queries, it is better to use less rules and thus we include universal quantifier **foreach** and existential quantifier **exists** in the construct of the form **foreach ... exists ... such that**.

Example 25 List the people who are authors of every book.

querying
(www.abc.com/people.xml)//author ⇒\$a
foreach \$b (www.abc.com/people.xml)//book ⇒\$b
exists \$a' \$b/author ⇒\$a'
such that (\$a = \$a')
constructing
/results/result ⇒\$a

Let us now see how to handle recursion using our query language.

Example 26 Consider the sample XML document in Figure 2. The following query lists the ID of a person and his/her ancestors IDs in an ancestors attribute with two rules:

querying
(http://www.abc.com/people.xml)
/person ⇒[@id ⇒\$p, @children ⇒\$c]
constructing
/results/result ⇒[@id ⇒\$c, @ancestors ⇒{\$p}]

querying
(http://www.abc.com/people.xml)
/person ⇒[@id ⇒\$p, @children ⇒\$c],
/results/result ⇒[@id ⇒\$p, @ancestors ⇒\$a]
constructing
/results/result ⇒[@id ⇒\$c, ancestors ⇒{\$p}]

The first rule says for each person identified by \$p, if \$c is a child, then \$p is an ancestor of \$c. The second rule says if \$c is a child of \$p and \$a is an ancestor of \$p, then \$p is also an ancestor of \$c. Note here the second rule is recursively defined. The result of this query is as follows:

```
<results>
    <result @id="o123" @ancestors="o456 o678"/>
    <result @id="o234" @ancestors="o456 o678"/>
    <result @id="o456" @ancestors="o678"/>
    <result @id="o567" @ancestors="o678"/>
</results>
```

These two rules can be combined into one using **or** as in logic programming as follows:

querying
(http://www.abc.com/people.xml)
/person ⇒[@id⇒\$p, @children⇒\$c]
or
(http://www.abc.com/people.xml)
/person ⇒[@id⇒\$p, @children⇒\$c],
/results/result ⇒[@id⇒\$p, @ancestors⇒\$a]
constructing
/results/result ⇒[@id⇒\$c, @ancestors⇒{\$p}]

5 Conclusion

In this paper, we have described a natural way to model XML documents as in complex object data models. As a result, we can easily comprehend XML data from database point of view. Based on this view, we have also illustrated how logical variables can be used to extract data from XML documents and how rules can be used to construct query results. We have also demonstrated the benefits that rule-based query language can bring to us. In addition, our language supports not only data extraction from XML documents but also URL-related searches. Thus, it supports the functionalities of search engines. None of the existing XML query languages supports this feature.

Unlike other XML query languages, the language described here has a well-defined logical foundation [18]. The language been implemented in Java. We will make it available from the web site at *http://www.scs.carleton.ca/∼mengchi/XML/* after further testing and debugging.

We would like to investigate how the data model described here can be used as a basis for other XML query languages such as XQuery. Also, we would like to extend the language into a full-fledged one by adding other useful features for XML querying and transformation. We plan to develop a natural language interface to our XML query language as well so that users can directly use a simplified English to query XML documents. A direct XML storage manager that supports the rule-based language is also underway.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Intl. Journal of Digital Libraries*, 1(1):68–88, 1997.

[3] A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *SIGMOD Record*, 29(1):68–79, 2000.

[4] R. G. G. Cattell and D. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, Los Altos, CA, 1997.

[5] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a Graphical Language for Querying and Restructuring WWW data. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada, 1999.

[6] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML Query Requirements. *http://www.w3.org/TR/2001/WD-xmlquery-req-20010215*, February 2001.

[7] D. Chamberlin, D. Florescu, J. Robie, J. Simon, and M. Stefanescu. XQuery: A Query Languge for XML. *http://www.w3.org/TR/2001/WD-xquery-20010215*, February 2001.

[8] J. Clark. XSL Transformations (XSLT) Version 1.0. *http://www.w3.org/TR/xslt*, November 1999.

[9] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. *http://www.w3.org/TR/1999/REC-xpath-19991116*, November 2001.

[10] S. Cluet and J. Simeon. YATL: a Functional and Declarative Language for XML. *http://db.bell-labs.com/user/simeon/icfp.ps*, 1999.

[11] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. *http://www.w3.org/TR/1998/Note-xml-ql-19980819*, August 1998.

[12] M. Fernandez and J. Robie. XML Query Data Model. *http://www.w3.org/TR/2001/WD-Query-datamodel-20010215*, February 2001.

[13] H. Hosoya and B. Pierce. XDuce: A Typed XML Processing Language (Preliminary Report. In *Proceedings of WebDB Workship*, 2000.

[14] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of ACM*, 42(4):741–843, 1995.

[15] M. Liu. ROL: A Deductive Object Base Language. *Information Systems*, 21(5):431 – 457, 1996.

[16] M. Liu. The ROL Deductive Object Base Language (Extended Abstract). In *Proceedings of the 7th International Workshop on Database and Expert Systems Applications (DEXA Workshop '96)*, pages 122–131, Zurich, Switzerland, September 9-10 1996. IEEE CS Press.

[17] M. Liu. Relationlog: A Typed Extension to Datalog with Sets and Tuples. *Journal of Logic Programming*, 36(3):271–299, 1998.

[18] M. Liu. A Logical Foundation for XML. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE '02)*, pages 568–583, Toronto, Canadan, May 27-31 2002. Springer-Verlag LNCS 2348.

[19] M. Liu and M. Guo. ROL2: A Real Deductive Object-Oriented Database Language. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER '98)*, pages 302–315, Singapore, Nov. 16-19 1998. Springer-Verlag LNCS 1507.

[20] M. Liu and T. W. Ling. A Conceptual Model and Rule-based Query Language for HTML. *World Wide Web Journal*, 4:49–77, 2001.

[21] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2 edition, 1987.

[22] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). *http://www.w3.org/TandS/QL/QL98/pp/xql.html*, 1998.