XTree: A Declarative Query Language for XML Documents

ZHUO CHEN, TOK WANG LING
School of Computing, National University of Singapore, Singapore
MENGCHI LIU
School of Computer Science, Carleton University, Ottawa, Canada
GILLIAN DOBBIE
Department of Computer Science, University of Auckland, New Zealand

XML is becoming prevalent in data presentation and data exchange on the internet. One important issue in the XML research community is how to query XML documents to extract and restructure information. Currently, XQuery based on XPath is the most promising standard. In this paper, we discuss limitations of XPath and XQuery, and propose a generalization of XPath called XTree that overcomes these limitations. Using XTree, multiple variable bindings can be instantiated in one expression; and XTree expressions, which represent a tree rather than a path, can be used in both the querying part and the result construction part of a query. Based on XTree, we develop an XTree query language, which is more compact and convenient to use than XQuery, and supports common query operations such as join, negation, grouping, and recursion in a direct way. We describe an algorithm that converts XTree query scripts to XQuery scripts. This algorithm provides not only a means of executing queries written in XTree query language but also highlights differences between the two query languages.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages-*Query Languages*; E.5 [**Data**]: Files-*Sorting/Searching*; D.3.3 [**Programming Languages**]: Language Constructs and Features
General Terms: Languages, Algorithms
Additional Key Words and Phrases: XML, XPath, XQuery

1. INTRODUCTION

XML is fast emerging as the dominant standard for data representation and exchange on the web. How XML documents are queried is an important issue in XML research and development. Various query languages have been proposed, such as XPath [Bergund et al. 2003], XQuery [Boag et al. 2003], Lorel [Abiteboul et al. 1997], XML-GL [Ceri et al. 1999;2000; Comai et al. 1998;2001], Equix [Cohen et al. 1998], XQL [Robie et al. 1998], XML-QL [Deutsch et al. 1998], XSLT[Kay 2003], YATL [Cluet and Simeon, 1999], XDuce [Hosoya and Pierce 2000], BBQ [Munroe and Papakonstantinou 2000], a rule-based semantic query language [Chippimolchai et al. 2000], and a declarative XML query language [Liu and Ling 2002]. A comparative analysis of some of the query languages is presented in [Bonifati and Ceri 2000]. While some of the query languages

Authors' addresses: Zhuo Chen, Tok Wang Ling, National University of Singapore, Lower Kent Ridge Road, Singapore 119260,{chenzhuo,lingtw}@comp.nus.edu.sg, Mengchi Liu, Carleton University, Ottawa, Ontario, Canada K1S 5B6, mengchi@scs.carleton.ca and Gillian Dobbie, University of Auckland, Private Bag 92019, Auckland, New Zealand, gill@cs.auckland.ac.nz

are in the tradition of database query languages; others are more closely inspired by XML. The XML Query Working Group has published XML Query Requirements for XML query languages [Chamberlin et al. 2003], and XQuery has been selected as the basis for an official W3C query language for XML.

Many of the existing XML query languages are based on SQL and OQL [Cattell and Barry 1997]. However, unlike queries on relational databases where the result is a flat relation, the result of queries on XML documents can be complex, and needs to be formatted explicitly. Thus, XML queries have two parts: a querying part and a result construction part. The existing XML query languages intermix these two parts in a nested way, making the queries cumbersome to express and difficult to comprehend. For example, XML-QL has two constructs: *where* and *construct*, for querying and result construction respectively. However, the *construct* clause can contain nested *where-construct* clauses so that querying and the result construction are intermixed.

In XQuery, there are five constructs: *for*, *let*, *where*, *order by* and *return*, i.e., FLWOR expressions, and XPath expressions are embedded within *for* clauses and *let* clauses. As in XML-QL, FLWOR expressions can be nested in the *return* clause to form a nested querying structure. As a result XQuery scripts are often lengthy with unnecessary nesting which is in part due to the fact that XQuery uses XPath expressions. Unlike the tree structure of XML documents, XPath expressions describes a linear path, and cannot express more complex substructures of the XML document tree. Thus we propose a generalization of XPath, called XTree; and a query language, XTree query language, which uses embedded XTree expressions. This paper contains a description of XTree's data model and the XTree query language, with its properties, and a translation from the XTree query language to XQuery.

Specifically, the paper makes the following contributions:

1. It discusses the limitations of XPath and XQuery.
2. It describes a generalization of XPath called XTree, which leads to queries that are compact, and easy to read and comprehend, because:
   - XTree has a tree structure, which is similar to the structure of an XML document, thus a user can bind multiple variables in one XTree expression.
   - XTree expressions explicitly identify list-valued variables, uniquely determine their values, and define some natural built-in functions to manipulate them.

- XTree expressions can be used not only for the querying part, but also for the result construction part of a query.

3. It describes the XTree query language, which has embedded XTree expressions.
   - In the querying part, multiple variables can be defined in one XTree expression, and the list-valued variables are explicitly identified.
   - In the result construction part, a user can write one XTree expression to define the result format to avoid unnecessary nesting.
   - The XTree query language can express join, negation, grouping, recursion and quantification directly and efficiently. It also supports some special queries (such as URL-related querying, structure level querying, sample querying, top-k querying) and update operations.
4. It provides algorithms to transform XTree query scripts to standard XQuery scripts.
   - The XTree query scripts can be run in any existing XQuery parsers.
   - The algorithm highlights some of the differences between the XTree query language and XQuery.

The paper is organized as follows. Section 2 introduces and discusses the limitations of XPath and XQuery. It also describes a complex object data model for XML data, which is adopted by XTree and the XTree query language. Section 3 introduces the XTree syntax with some examples, and demonstrates its advantages over XPath. Section 4 introduces the XTree query language, which has embedded XTree expressions, and shows that it can express many kinds of queries elegantly. This section culminates in a comparison of the XTree query language and other declarative XML query languages. Section 5 presents the algorithms to transform XTree query scripts to standard XQuery scripts. Finally, Section 6 summarizes this paper and highlights future research directions.

2. PRELIMINARIES

Researchers have proposed many declarative query languages to extract data from XML documents, such as Lorel [Abiteboul et al. 1997], XQL [Robie et al. 1998], XML-QL [Deutsch et al. 1998], XQuery [Boag et al. 2003], Quilt [Chamberlin et al. 2000], YATL [Cluet and Simeon, 1999], a rule-based semantic query language [Chippimolchai et al. 2002], a declarative XML query language [Liu and Ling 2002]. The W3C has selected XQuery based on XPath as the basis for an official standard for XML query languages. In

this section, we give an introduction of XPath and XQuery, and discuss their limitations. A complex object data model for modeling XML documents is also described.

2.1 XPath

XPath is a set of syntax rules for defining parts of XML documents. It uses paths to locate nodes (elements and attributes) in XML documents, and the path expressions look very much like computer file system paths. For example, consider the bibliography XML document in Appendix I and the examples of XPath expressions in Table 1.

XPath expression	Description
/bib/book	select all "book" elements of the root element "bib"
/bib/book/@year	select attribute "year" of each book
/bib/book/author	select element "author" of each book
//author	select all elements named "author". The symbol "//" means no matter how many levels down.
*/bib/book/**	select all sub-elements of each book
*/bib/book/@**	select all attributes of each book
/bib/book[1]	select the first "book" element
/bib/book[last()]	select the last "book" element

Table 1. Sample XPath expressions.

XPath uses a pattern expression to identify nodes in an XML document. An XPath pattern is a slash-separated list of child element names (perhaps with an attribute at the last position) that describe a path through an XML document. The pattern "selects" elements that match the path. If the path starts with a slash (/), it represents an *absolute path* to an element, otherwise it represents a *relative path,* e.g. *//author* is a relative path. If the path starts with two slashes (//), then all elements in the document that fulfill the criteria will be selected (even if they are at different levels in the XML tree). Wildcards (such as *) can select all elements selected by the preceding path. An index number enclosed in a pair of square brackets in an XPath expression can further specify an element: the index number specifies the position of the element in the selected collection; the function *last()* selects the last element in the selected collection. Attributes are specified by prefix @.

In an XPath expression, a pair of square brackets can not only contain index numbers but can also be used to specify selection conditions on the child nodes. (This feature is sometimes viewed as an abbreviated format of XQuery.) Table 2 gives examples of XPath expressions with conditions enclosed in square brackets, with reference to the XML document in Appendix I.

XPath expression	Description
/bib/book[@year]	select all "book" elements that have a "year" attribute
/bib/book[@year="1994"]	select all "book" elements that have a "year" attribute with a value of "1994"
/bib/book[@]*	select all "book" elements that have any attribute.
/bib/book[price]	select all "book" elements that have a "price" sub-element
/bib/book[price>50]	select all "book" elements that have a "price" sub-element with a value greater than 50
/bib/book[position()<4]	select the first 3 "book" elements
/bib/book[count(author)>1]	select all "book" elements that have more than one "author" sub-element
/bib/book[starts-with(title, "Data")]	select all "book" elements that have a "title" sub-element with a value starting with "Data"

Table 2. Sample XPath expressions with conditions.

The conditions are used to select a collection of nodes based on some condition that tests its children nodes. The condition may test for the existence of a particular child node, it may test the value of a child node, or it may use a function to test such conditions as the cardinality of a child node.

2.2 XQuery

XQuery, is a powerful way to search XML documents for specific information. It is derived from several previous proposals, such as XML-QL [Deutsch et al. 1998], YATL [Cluet and Simeon 1999], and Lorel [Abiteboul et al. 1997].

XQuery is based on XPath expressions; each query is built from expressions that can be nested to arbitrary depth. XQuery has the FLWOR (For-Let-Where-Order by-Return) statements: the *for* clause and *let* clause bind values to variables, the *for* clause (syntax: "*for $var in xpath-expression*") iterates the variable over the result of the XPath expression, whereas the *let* clause (syntax: "*let $var := xpath-expression*") binds the variables to the whole result of the XPath expression as a list; the *where* clause filters these bindings by some conditions; the *order-by* clause orders the surviving bindings based on some items; and the *return* clause defines the result format, and constructs the result based on the evaluation of the variable bindings.

Example 1. Consider the bibliography document in Appendix I, and the following XQuery script that gets the year, title, and number of authors of all books published before 2000, sorted by year of publication.

```
<bib>
{
   for $book in /bib/book
   let $authors := $book/author
   where $book/@year < 2000
   order by $book/@year
   return <book>
            { $book/@year, $book/title }
            <authors> { count($authors) } </authors>
            </book>
}
</bib>
```

Note that the outer braces *{ }* (after <bib> and before </bib>) defines a query block; and the inner braces *{ }* indicates an enclosed expression. Without the inner braces, the inner code "*$book/@year, $book/title*" and "*count($authors)*" would be treated as literal text, and be placed in the result directly, without being executed.

In *where* clauses, we can use quantified expressions, such as "*some…in…satisfies…*" or "*every…in…satisfies…*" to define existential and universal quantifications, respectively.

Example 2. The following XQuery script gets the books that have an author with last name "Stevens".

```
for $book in /bib/book
where some $author in $book/author satisfies ($author/last = "Stevens")
return $book
```

Example 3. The following XQuery script gets the books which have no author with last name "Stevens".

```
for $book in /bib/book
where every $author in $book/author satisfies ($author/last != "Stevens")
return $book
```

Because XQuery supports complex queries and complex result constructions with nested clauses, very complicated queries can be expressed in XQuery (which may have a deep nesting level).

Example 4. The following XQuery script returns the title and first two authors for each book that has at least one author, and an empty "et-al" element if the book has additional authors.

```
<bib>
{
   for $book in /bib/book
   where count($book/author) > 0
```

```
    return <book>
            {  $book/title }
            {
               for $author in $book/author[position() <= 2]
               return $author
            }
            {
               if (count($book/author) > 2)
               then <et-al/>
               else ()
            }
           </book>
}
</bib>
```

2.3 Limitations of XPath and XQuery

From the above examples, we can see that XPath can clearly define a unique path in an XML tree; and XQuery can effectively express queries on XML documents, based on XPath expressions. However, both XPath and XQuery have limitations.

2.3.1 Limitations of XPath.

The limitations of XPath make it cumbersome to express even some simple conditions. They include the ability to express only a linear path through an XML document, and the inability to bind more than one variable in each XPath expression.

(1) One path, one variable

In an XPath expression, although a condition can be a branch, there is still only one linear path to the target. Thus, we can only assign one variable to each XPath expression, which leads to queries that are difficult to read. If a query uses several paths, a user must use separate XPath expressions to specify each path and assign a variable to each path.

Example 5. If a user is interested in the title, authors and publisher of each book in the bibliography document in Appendix I, we write the following in the querying part of an XQuery script:

```
for $b in /bib/book, $t := $b/title, $p := $b/publisher
let $a := $b/author
```

(2) Only used in querying part

XPath expressions are only used in the querying part of XQuery scripts, and not in the result construction part. For the result construction part of XQuery, the *return* clause mixes literal text, enclosed expressions and even nested sub-queries, making the query difficult to understand.

(3) Unclear relationship among XPaths

It is difficult to reveal the relationship among correlated XPaths, because the expression of the correlations is not explicit in XPath. This may result in some mistakes if the user does not pay attention when writing a query.

Example 6. Consider an XQuery script that creates a flat list of all the title-author pairs for the books, with each pair enclosed in a "result" element.

The query in Fig. 1a, although it appears quite plausible, produces the wrong result because it does not pay attention to the correlation of XPath expressions. This query produces a Cartesian product of all authors and titles, regardless of whether they belong to the same book. The query in Fig. 1b gives the expected result.

```
for $t in /bib/book/title,
      $a in /bib/book/author
return
      <result>
         { $t }
         { $a }
      </result>
```

Fig.1a. Wrong query

```
for $b in /bib/book,
      $t in $b/title,
      $a in $b/author
return
      <result>
         { $t }
         { $a }
      </result>
```

Fig.1b. Expected query

(4) Confusing for distant conditions

Using XPath, it can be confusing to express a query which returns elements at path *A* while the condition is in a distant path *B*, especially when there are multiple conditions and/or nested conditions.

Example 7. Suppose we want to select the publisher id of a book which has an author with last name "*Stevens*" and first name "*W.*". The XPath expression is as follows:

/bib/book[author[last="Stevens" and first="W."]]/publisher/@pubid

There are a couple of important features to note in this XPath expression. The first is to note that both *author* and *publisher* are subelements of *book*. The second is that the selection of *publisher* is dependent on a condition involving *author*. Although this is obvious when the expression is inspected carefully, it is not immediately obvious.

(5) Difficult to split name-value pair structure

In XPath, a variable can only be bound to the whole node (element or attribute) structure, which is a name-value pair. If we want to get some substructure (name or value) of the

node, we have to call some built-in functions to split the name-value pair. Thus it is difficult to query XML documents with unknown structure, or to rename the elements or attributes in the query result construction.

Example 8. Suppose that for each book, we want to get all the sub-elements, except the sub-element "publisher". In XQuery we write the following query:

```
<bib>
{
   for $book in /bib/book
   return <book> {
      for $elem in $book/*
      where local-name($elem) != "publisher"
      return $elem
   }
   </book>
}
</bib>
```

Note that the function *local-name()* is used to get the node name. In addition, if we want to get the node value, then for an element node bound to variable *$var*, we use *$var/**, *$var/@** and *$var/text()* to get all its possible content (namely, elements, attributes and text content respectively); for an attribute node bound to variable *$var*, we call the function ***string($var)*** to get its value.

2.3.2 Limitations of XQuery.

The limitations of XQuery make the language unintuitive, and therefore queries in the language are complex and hard to understand, which can mean queries are difficult to optimize. The limitations include the lack of support for the join operation and an inability to express updates.

(1) Join operation as sub-query

The *Join* operation is widely used to combine data from multiple sources into one single result. However, XQuery supports join in the following way: it binds a variable on the join field of one source; then, for each of the other sources, it uses sub-queries with the join field in the conditions to get the instances that have the same join field value. In XQuery, the join is indirect and unnatural, unlike in SQL or QBE (Query By Example [Date 1981]), making the query difficult to read and comprehend.

Example 9. Consider three XML files which are valid with respect to the DTDs in Appendix II. The XML files are sailors.xml, boats.xml and reservations.xml, where

sailors.xml and boats.xml record information of sailors and boats respectively, and reservations.xml records the reservations of certain boats by certain sailors, describing a relationship between sailors and boats. Suppose we want to get the sailor name and boat name, and the start time and end time for each reservation.

```
for $r in doc("reservations.xml")/reservations/reservation
let $s := doc("sailors.xml")/sailors/sailor[@sid=$r/@sid],
    $b := doc("boats.xml")/boats/boat[@bid=$r/@bid]
return
   <reservation>
   <sailor> { $s/sname/text() } </sailor>
   <boat> { $b/bname/text() } </boat>
   { $r/start-time, $r/end-time }
</reservation>
```

(2) Grouping as sub-query

Many queries involve forming data into groups and applying aggregate functions to each group. However, unlike the *group by* operator in SQL, XQuery does not support grouping operations explicitly. In XQuery, grouping is done using a sub-querying structure, which is difficult to read and understand; and can be very inefficient. It may scan the entire document once for each value of the grouping field. Moreover, such kinds of sub-querying may even get error results when there are two or more grouping fields, due to some invalid empty groups being generated.

Example 10. Consider the bibliography document in Appendix I, and the XQuery script that gets the book titles published in each year.

```
for $year in distinct-values(/bib/book/@year)
return
   <year value = { $year }>
   {  /bib/book[@year=$year]/title }
</year>
```

Note that the above query may scan the bibliography document many times, each time for a specific *year* value.

Example 11. Figure 2 shows the DTD for a document employees.xml.

```
<!ELEMENT employees (employee*)>
<!ELEMENT employee (name, department, jobtitle, salary)>
<!ATTLIST employee id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT department (#PCDATA)>
<!ELEMENT jobtitle (#PCDATA)>
<!ELEMENT salary (#PCDATA)>
```

Fig. 2. DTD for employees.xml

Consider the XQuery script that returns the average salary of employees, grouped by department and job title.

```
for $dept in distinct-values(/employees/employee/department),
    $jobtitle in distinct-values(/employees/employee/jobtitle)
        let $salary := /employees/employee[department=$dept and
        jobtitle=$jobtitle]/salary
return
   <type dept={ $dept } job={ $jobtitle }>
     <avgsalary> { avg($salary) } </avgsalary>
   </type>
```

In this example, the nested *for* clauses will produce a Cartesian product of departments and job titles, but some pairs of department and job title may not have employees, thus the above query will generate some empty groups, which are not expected.

The reason invalid empty groups are generated is because the function *distinct-value()* can only accept one XPath expression (or a list-valued variable that holds an XPath expression) and remove the duplicates. It cannot process two or more XPath expressions together.

(3) Recursion by user-defined recursive function

Sometimes it is necessary to scan a hierarchy of elements recursively, applying some transformation at each level of the hierarchy. XQuery does not support *recursive querying* explicitly; instead, it handles recursion with user-defined recursive functions. This indirect way of expressing recursion means the query is difficult to read. Also the purpose of functions should be for general and common computation that is needed many times rather than as a backdoor for other purposes. In our view, this is a drawback of the language design.

Example 12. Consider the following list of employees where each employee element has an optional attribute *manager* indicating the ID number of his/her manager.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employeelist>
   <employee id="0" name="John"/>
   <employee id="1" manager="0" name="Tom"/>
   <employee id="2" manager ="0" name="Jack"/>
   <employee id="3" manager ="1" name="Ken"/>
   <employee id="4" manager ="2" name="Bush"/>
   <employee id="5" manager ="2" name="Jeremy"/>
   <employee id="10" name="Ivan"/>
   <employee id="11" manager ="10" name="Gerald"/>
   <employee id="12" manager ="10" name="Albert"/>
```

```
    <employee id="20" name ="Michael"/>
</employeelist>
```

Suppose we want to convert the employee list to a tree structure, where the parent node is the manager and the children nodes are the direct subordinates. An XQuery solution follows:

```
declare function one_level_down($e as element()) as element()
{
   <employee id={ $e/@id } name={ $e/@name }>
   {
      for $a in doc("employeeList.xml")//employee
      where $a/@manager = $e/@id
      return one_level_down($a)
   }
   </employee>
}

<employeeTree>
   {
      for $e in doc("employeelist.xml")//employee[empty(@manager)]
      return one_level_down($e)
   }
</employeeTree>
```

The result of the above query is shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employeetree>
   <employee id="0" name="John">
      <employee id="1" name="Tom">
         <employee id="3" name="Ken"/>
      </employee>
      <employee id="2" name="Jack">
         <employee id="4" name="Bush"/>
         <employee id="5" name="Jeremy"/>
      </employee>
   </employee>
   <employee id="10" name="Ivan">
      <employee id="11" name="Gerald"/>
      <employee id="12" name="Albert"/>
   </employee>
   <employee id="20" name="Michael"/>
</employeetree>
```

(4) Nested querying structure

In practice, XQuery usually has a nested querying structure. However, the nesting is mainly in the *return* clause of the query (unlike SQL queries on relational databases where the result is always a flat structure). Thus the nested *return* clause is often quite

long, mixing up the plain XML segments, enclosed expressions and even sub-queries, making it difficult to read and understand.

(5) Built-in functions

In XQuery most built-in functions are used in a functional manner, rather than in an object-oriented manner (except functions *position()* and *last()* that are used in an object-oriented manner). Thus we often need to refer to the context node again in the argument of the function, which is not intuitive. It is more natural to use the functions in an object-oriented fashion, where the meaning is obvious and easy to understand.

Example 13. Consider the bibliography in Appendix I, and a query that finds the books in which the name of an element starts with the string "au" and the same element contains the string "Suciu" somewhere in its content. For each such book, return its title and the qualifying element. We can write the query in XQuery as follows:

```
for $book in /bib/book
let $elem : = $book/*[contains(string(.), "Suciu") and starts-with(local-name(.), "au")]
where exists($elem)
return
   <book>
      { $book/title, $elem }
   </book>
```

Note that in the *let* clause of the above XQuery, the two "**.**" in the condition refer to *$book/**, but this is difficult to understand, since they are textually far away from their substituting context node.

(6) Special queries

XQuery does not support some special queries, or does not support them directly. These special queries include *URL-related querying*, *structure level querying, sample querying* and *top-k querying.*

XQuery does not support variable bindings on the URL or URL components of documents, thus a user cannot write a query in XQuery to find some unknown URL or URL component.

Since variables are bound to the name-value pairs of XML elements, special built-in functions are needed to split the name and value. As a result the mechanism in XQuery to handle queries over XML documents with unknown structures, or to rename elements/attributes in the result construction part of a query without knowing their inner structure is very clumsy.

Since XQuery does not have functions to get a certain subset of a list, it is very inefficient at handling queries that just pick up several items randomly, or just pick up the first several items according to a certain order, instead of getting the entire result set. For example, we cannot easily write a query to browse details of three random books (not all the books), or list the two most expensive books (not all the books sorted by price).

(7) Negation on paths only

XQuery can only express negation in the condition clause (*where* clause), but not the query clause (*for* clause and *let* clause). In addition, due to the fact that XQuery is based on XPath, a user can only set negative condition on some paths, and not on a sub-tree structure in an XML document. Thus, it is difficult to express complex negation (such as a negative sub-tree) or nested negation.

(8) No update operations

Unlike SQL for relational databases, which is both a data query language and a data management language, currently XQuery can only query XML documents, and cannot do updates on XML documents. We believe the problem of expressing updates over XML data will become prominent in the near future, and it will be necessary to extend XQuery to support update operations.

2.4 Modeling XML documents for databases

The data model of a query language serves two purposes. First, it defines precisely the information contained in the input to a query processor. Second, it defines all permissible values of expressions in the query language.

XPath and XQuery model XML data using the XML Query Data Model, which is a low level data model, representing the XML document structure as a tree graph, instead of a database structure. The result of this is that it is not very suitable for database operations such as querying and data management.

For XTree and the XTree query language, we adopt the complex object data model proposed in [Liu and Ling 2002], which provides a natural way to model an XML document as a complex object with a nested structure. As a result, an XML document can be comprehended from a database perspective, and the representation of queries is at a higher level.

In the complex object data model, there are five types of objects: *element objects*, *attribute objects*, *tuple objects*, *lexical objects* and *list objects*.

Example 14. Consider the following part of a simple XML document:

```
<person id="p123">
  <name>
    <first>John</first>
    <last>Smith</last>
  </name>
  <gender>Male</gender>
  <age>25</age>
</person>
```

We can view it as a nested complex object as follows:

```
person → [
  @id → p123,
  name → [
    first → John,
    last → Smith],
  gender → male,
  age → 25]
```

We call the above complex object an *element object*, which is a pair of element name and element value, connected by the symbol "→". The element value "@id → p123, name → [first → John, last → Smith], gender → male, age → 25]" is a *tuple object*, which contains an *attribute object* "@id → p123", nested element object "name → [first → John, last → Smith]" and element objects "gender → male" and "age → 25". The textual values of attributes and simple element objects such as "p123", "John", "Smith", "male", "25" are called *lexical objects*. The symbol "@" is used to denote an attribute, "→" is used to separate element/attribute name from element/attribute value, and square brackets *[ ]* are used to enclose a list of elements and attributes of the same level to form a tuple object.

3. XTREE

We now introduce **XTree**, which is a generalization of XPath without the limitations of XPath. The input to an XTree expression is a complex object and the result of an XTree selection is a complex object.

3.1 Basic syntax of XTree

XTree has a tree structure which is similar to the structure of an XML document. Like in XPath, a parent and a child node are separated by a slash (i.e. /), and an ancestor and a

descendent node are separated by a double-slash (i.e. *//*) in XTree. However, in XTree sibling tree nodes are enclosed by a pair of square brackets (i.e. *[ ]*) and are separated by commas, and these expressions can be nested. Only the sub-trees that are relevant to the query are written in an XTree expression, not the entire XML tree structure.

XTree uses parentheses (i.e. *( )*) to enclose the source URL at the beginning of the XTree expression. An URL is composed of protocol name, domain name, directories and file name, and any part of the URL can be unknown and bound to a variable. The URL is optional, if it is absent, then the document queried is the default input. In XPath, the URL is specified by the function *doc*, and its parameter must be a known specific URL.

In XTree expressions, the position of logical variables is important since logical variables bind to the name-value pairs of XML nodes based on their position. Because various parts of XML documents can bind to logical variables, in order to be flexible and easy to use in practice, the variables in XTree are *not typed*. A variable can be used in any location, however where the same variable occurs in different places in the same query, it has the same value. There are two kinds of variables in XTree: single-valued variables and list-valued variables. *Single-valued variables* start with *$*, such as *$X*. *List-valued variables* are of the form *{$X}* which is constructed from a single-valued variable *$X* that ranges over all instances in a list.

XTree expressions are used in the querying part of a query, to indicate the subtrees of interest and to bind variables to parts of them; and also used in the result construction part of a query, to define the format of the returned result. Symbol → is used when assigning values to variables in the querying part of a query; and is also used when getting values from variables in the result construction part of a query.

Because we can write the result construction part of a query as an XTree expression, we do not need to use curly braces *{ }* to indicate the enclosed expressions or nested query blocks as is done in XQuery, thus we can effectively reduce the nesting level in the query.

3.2 XTree for querying

In the querying part of a query, we can write XTree expressions to bind variables to nodes, a list of nodes, a URL or part of a URL to get the URL information from the XML documents.

3.2.1 Binding variables on XML data.

To bind variables to a specific set of nodes, we use the symbol → to assign the value of nodes on the left side to a variable on the right side. If the right side is a single-valued variable, the node values are assigned to the variable one by one, as in the *for* clause of XQuery expressions; if the right side is a list-valued variable, a list of node values is assigned to the variable, as in the *let* clause of XQuery expressions. If the left side of the symbol → is a variable then the variable binds to the name of the node rather than the value. In one XTree expression a user can bind multiple variables, producing more compact queries than in XPath.

Example 15. Consider the bibliography document in Appendix I that is located at the URL *www.abc.com/bib.*xml. If we want to select the *year* and *title* of each book, and its authors' first names and last names, then we can use variables *$y*, *$t*, *$first*, *$last* respectively in the following XTree expression:

(http://www.abc.com/bib.xml)
/bib/book/[@year→$y, title→$t, author/[last→$last, first→$first]]

The above XTree expression contains four variable bindings. In fact, it corresponds to the following six XPath expressions:

for $book in doc("http://www.abc.com/bib.xml")/bib/book,
$y in $book/@year, $t in $book/title, $author in $book/author,
$last in $author/last, $first in $author/first

XTree supports path abbreviations as in XPath. Consider the following example.

Example 16. Suppose we want to get the values of elements first name and last name at whatever depth in the document, we can write an XTree expression as follows:

(http://www.abc.com/bib.xml)/bib//[last→$last, first→$first]

Here the pair of square brackets *[ ] e*nclosing nodes *last* and *first* specifies that these two sub-elements *last* and *first* are sibling nodes, sharing a common parent, which in this case is an instance of */bib/book/author* or */bib/journal/editor*.

XTree also allows a user to bind variables on the structure of XML document, that is, a user can write variable *$var* on the left side of → symbol, and *$var* will be bound to the names of the corresponding elements or attributes.

Example 17. Consider a query that selects the names of the sub-elements of *book*, except the sub-element *publisher*. This can be done using the variables *$elem_name* and *$elem_value* in the following XTree expression:

(www.abc.com/bib.xml)/bib/book/$elem_name→$elem_value
$elem_name != "publisher"

Example 18. Consider a query that selects some attribute name with value "1992" in some book element, and binds variable $b to this book. This can be done with the following XTree expression:

(www.abc.com/bib.xml)/bib/book→$b/@$attr="1992"

With the sample bibliography document in Appendix I, *$b* will bind to the second book element, and *$attr* will bind to string "year", which is the attribute name. The XQuery version of this query is as follows, which is more complex:

```
for $b in doc("http://www.abc.com/bib.xml")/bib/book,
    $attr in $b/@*
where string($attr) = "1992"
return local-name($attr)
```

3.2.2 List-valued variables and OO functions.

A list-valued variable can bind its value to a list of XML nodes; it is somewhat like the variable in the *let* clauses of XQuery. However, in XQuery list-valued variables look exactly like single-valued variables. In XTree expressions, a list-valued variable is explicitly indicated by a pair of curly braces *{ }*. In addition, unlike XPath and XQuery that use many unintuitive functions (in functional style), XTree defines some natural functions that are obvious and easy to understand, and they are consistently used in an object-oriented fashion. This is possible since XTree models XML documents as complex objects.

We now illustrate some of the functions that can be applied to a list of numbers followed by functions that can be applied to a list of elements. Suppose a list-valued variable *{$number}* binds to a list of numbers, then we can obtain their aggregate values as follows:

{$number}.count()	returns the number of items in the list
{$number}.avg()	returns the average value of items in the list
{$number}.min()	returns the minimum value in the list
{$number}.max()	returns the maximum value in the list
{$number}.sum()	returns the sum of values in the list

For example the expression *{1, 3, 4}.sum()* returns the value *8.*

Suppose a list-valued variable *{$authors}* bind to a list of authors elements, then we have the following built-in operators:

{$authors}.count()	returns the number of items in the list of authors
{$authors}[2-4, 6]	returns a sub-list containing second to fourth items, and sixth item
{$authors}.last()	returns the last author
{$authors}.sort()	sorts the items in the list in ascending order
{$authors}.sort_desc()	sorts the items in the list in descending order
{$authors}.distinct()	returns a list of authors with no duplicates
{$authors}.random(3)	picks out 3 authors randomly to form a sub-list
$author ∈ {$authors}	checks whether $author is in the list
{$authors'}⊆{$authors}	checks if the first list is a sub-list of the second list
{$authors}.iterate()	returns an item in the list each time, one by one
{$authors}.none()	check whether *none* of the items in list satisfies a certain condition

In fact, since XTree uses an object model, other functions are also redefined in the object-oriented manner. For example, to check whether variable *$title* contains the string "XML", instead of using the expression *contains($title, "XML")* as in XQuery, we write the object-oriented version as *$title.contains("XML")*.

Next we will define the semantics of list-valued variables.

Definition 1. The *associated path* of variable *$a* (or *{$a}*) is the absolute path expression from the root of the document to the nodes represented by *$a* (or *{$a}*).

For example, in the XTree expression */bib/book→$b/title→$t*, the *associated path* of *$b* is */bib/book*, and the *associated path* of *$t* is */bib/book/title*.

Definition 2. Variable *$a* is an *ancestor variable* of *$b* if *$a* and *$b* are defined in the same XTree expression, and the associated path of *$a* is a prefix of the associated path of *$b*.

For example, in the XTree expression */bib/book→$b/[title→$t, author→$a]*, *$b* is an *ancestor variable* of *$t* and *$a*, but *$t* is not an ancestor variable of *$a* (they are siblings).

Definition 3. In an XTree expression, when a variable is bound to a value in the query evaluation, the variable is *instantiated.*

For example, in the XTree expression */bib/book/[author→$a/first→$f, title→$t]*, in the evaluation, when we reach */bib/book/author*, *$a* is *instantiated*; and when we reach */bib/book/author/first*, *$a* and *$f* are *instantiated*.

Definition 4. The *value* of list-valued variable *{$a}* is a list of all instances of *$a* with all its ancestor variables instantiated.

Example 19. Compare the following two XTree expressions:

XTree expression	Value of *{$a}*
/bib/book/author→{$a}	all the author elements of all the books.
/bib/book→$b/author→{$a}	all the authors of a certain book *$b*. When *$b* is bound to next book, *{$a}* will also bind to the authors of the next book.

Note that for the first expression, the value of *{$a}* is the concatenation of all the authors of all the books; whereas for the second expression, *{$a}* has an ancestor variable *$b*, thus the value of each *{$a}* instance is related to the corresponding *$b* instance.

3.2.3 Conditions.

Unlike in XPath where condition expressions are enclosed by a pair of square brackets, a user can write conditions directly in XTree expressions.

In XPath, the square brackets *[ ]* are used to express conditions in the path. This is necessary because an XPath expression is only one linear path to the target node set, thus we have to use square brackets as an indication that the enclosed expression is not a part of the target path, but the conditions that define instances of the path. However, in XTree, each XTree expression can have multiple target paths, and the variable bindings are explicitly defined, thus a user can write conditions directly in the XTree expression, without the use of any special symbol for indication.

Example 20. To find the books which have an author named "Stevens W.", we can write the following XTree expression:

/bib/book→$b/author/[last="Stevens", first="W."]

Condition expressions can also cooperate with list-valued variables.

Example 21. Find the authors of each book published in 1992, we can write an XTree expression as follows:

/bib/book→$b/[@year="1992", author→{$a}]

Each *{$a}* will store the list of authors of a certain book *$b* that was published in 1992, since *$b* is an ancestor variable of *{$a}*. However, to find all the authors who have a book published in 1992, we can write an XTree expression as follows:

/bib/book/[@year="1992", author→{$a}]

3.2.4 Variable Bindings on URLs.

In XTree expressions, we can use variables for URL and URL components. Such variable bindings will not involve the symbol →.

Example 22. Consider a query that selects XML documents in the website *http://www.abc.com* directory */dir* that contain a book element with value 1994 for attribute *year* and "TCP/IP Illustrated" for sub-element *title*. This query will require the following XTree expression:

(http://www.abc.com/dir/$file.xml)
/bib/book/[@year="1994", title="TCP/IP Illustrated"]

Here the variable *$file* will bind to the XML document names, where the *.xml* extension restricts the file type to XML documents. Currently XPath and XQuery do not support this kind of query.

3.3 XTree for Result Construction

XTree expressions can not only be used to bind variables in the querying part of queries, but can also be used to define the result format. We can use symbol → to show the name of the element on the left side and the value of the variable on the right side. If the right side is a single-valued variable, we assign the value of the current iteration to the element; if the right side is a list-valued variable, we assign all the values in the list to the element. Unlike the result construction part in XQuery, which often mixes XML plain text, variable values and even sub-queries in the nested *return* clause; only one XTree expression is needed to define the result format, making it very simple to write and read.

Note that unlike the XTree expressions in the querying part, which allows conditions and abbreviations (such as // for any levels down), the XTree expression in the result construction must be concrete, not allowing any condition checking or uncertainty in the structure.

Example 23. Suppose we have instantiated the variables *$y* and *$t* in the following XTree expression:

(www.abc.com/bib.xml)/bib/book/[@year→$y, title→$t]

If we want to get the *title* and *year* information of each book, and store the result in the file at URL *www.xyz.com/books.xml*, we can write the following XTree expression:

(www.xyz.com/books.xml)/result/book/[@year→$y, title→$t]

The above XTree expression defines the URL address (this is not supported by XQuery) and the structure of the result document. Under the root *result*, each *book* element will store the title and year of that book.

Example 24. Suppose we have instantiated the variables *$t* and *{$a}* in the following XTree expression:

(www.abc.com/bib.xml)/bib/book/[title→$t, author→{$a}]

For each book, if we want to store its title, the number of authors, and the first author only, we can write the following XTree expression:

/result/book/[title→$t, numAuthors→{$a}.count(), firstAuthor→{$a}[1]]

The right side of symbol → does not have to be some predefined variables or a function invocation on variables. Instead, it can be some literal text, or even be omitted, giving an empty element. In the case of the literal text, the value of the element is the literal text, forming some XML segment.

Example 25. Suppose we want to return a book whose title is "Computer Architecture", with an empty "no-author" element. We can write the following XTree expression:

/book/[title → "Computer Architecture", no-author]

The above XTree expression will output the following XML segment:

```
<book>
  <title>Computer Architecture</title>
  <no-author/>
</book>
```

4. XTREE QUERY LANGUAGE

In this section we propose a new query language, which makes use of XTree expressions. The XTree query language can express queries compactly; can express join, negation, grouping, recursion and quantification directly; can write some special queries such as URL-related querying, structure level querying, sample querying, top-k querying; and support updates on XML documents. The input to an XTree query script is a complex object and the result of an XTree query script is a complex object.

4.1 Basic syntax of the XTree query language

The syntax of an XTree query is similar to that of XQuery. Unlike XQuery's FLWOR (From-Let-Where-Order by-Return) statements, an XTree query has the **QWCO** (**Q**uery-**W**here -**C**onstruct-**O**rder by or **Q**uery-**W**here -**C**ache-**O**rder by) statements for querying.

The *query* clause contains one or more XTree querying expressions for selection and variable binding, which is similar to *for* clauses (for the binding of single-valued variables) and *let* clauses (for the bindings of list-valued variables) in XQuery.

The *where* clause is optional, and is used for specifying constraints in the same way the *where* clause is used in XQuery.

The *construct* clause contains exactly one XTree result construction expression to define the output format; it does not need a nested structure as often happens in the *return* clause in XQuery, thus making the result construction part more concise and easier to understand.

The *cache* clause is similar to the *construct* clause; it contains one XTree result construction expression. However, the *cache* clause does not output the result; instead, it holds the result temporarily for further use. The *cache* clause is often used to split a complex query into several simpler queries, to make it clear and easy to comprehend; or to extract a common step of several queries, to reduce redundant work.

The *order-by* clause is optional, and is used for specifying ordering on the constructed or cached result in the same way the *order-by* clause in used in XQuery. Note it is placed after the result construction part of the query to emphasize that it is part of the result construction rather than the querying.

Example 26. For each book that has three or more authors, list the title and the first two authors, and an empty tag <*et-al*/> to indicate the rest of authors. Also the result should be ordered by the book title.

> ***query*** */bib/book/[title→$t, author→{$a}]*
> ***where*** *{$a}.count() ≥ 3*
> ***construct*** */result/book/[title→$t, authors/[author→{$a}[1-2],* ***et-al****]]*
> ***order by*** *$t*

Note that it is more natural to put the *order by* clause after the result construction, since it is the result that is ordered. Due to the absence of symbol →, by default the right side of → is empty, and thus "*et-al*" will be interpreted as an empty tag "<et-al/>" under element "authors". If instead we change the *construct* clause to

construct */result/book/[title→$t, authors/[author→{$a}[1-2],* ***author→"et-al"****]]*

then after the first two author elements, there will be an element "<author>et-al</author>" under element "authors".

Example 27. Find the books that have at least one author, who has written at least one book containing the word "programming" in the title.

> ***query*** */bib/book/[title→$t, author→$a]*
> ***where*** *$t.contains("programming")*
> ***cache*** */temp/author→$a*

query */temp/author→{$a}, /bib/book→$b/author→$au*
where *$au ∈ {$a}*
construct */result/book→$b*

Note that the query is split into two simpler queries: the first one finds those authors who have written a book with "programming" in the title, and the second finds those books that have at least one such author.

4.2 Join

Unlike XQuery which implements join operations by nested sub-queries, the XTree query language supports join in a direct and natural way, which is similar to SQL and QBE [Date 1981]. In the XTree expressions of the *query* clauses, variables of the same name indicate a join operation, because both instances of the variable must have the same value.

Example 28. Consider three XML files which are valid with respect to the DTDs in Appendix II, and a query to get the sailor name and boat name for each reservation. We can write the following XTree query:

*query (sailors.xml)/sailors/sailor/[@sid→**$sid**, sname→$sname],*
*(boats.xml)/boats/boat/[@bid→**$bid**, bname→$bname],*
(reservations.xml)/reservations/reservation/
*[@sid→**$sid**, @bid→**$bid**, start-time→$st, end-time→$et]*
construct /reservations/reservation/
[sailor→$sname, boat→$bname, start-time→$st, end-time→$et]

Note that in the querying part of the above query, the two occurrences of variable $sid indicates a join operation over sailors and reservations, and the two occurrences of variable $bid indicates a join operation over boats and reservations. This natural way of expressing join makes it easier for a user to write queries over multiple data sources.

4.3 Negation

XTree query defines a unary negation operator *not()* that can be used in the *query* clause for negative conditions. The enclosed expression of *not()* is a sub-tree structure, which also conforms to the XTree syntax. The negation operator *not()* requires the conditions on the enclosed XTree expression to be evaluated to false, or requires the enclosed XTree structure to not exist in the document.

Like the negation operator in other languages, in the XTree query language, the expression enclosed by *not()* is by default existentially quantified, and *not()* can be nested.

Example 29. Consider the bibliography document in Appendix I, find books that do not have an author named "Stevens W.". We can write the XTree query as follows:

*query /bib/book→$b/**not(**author/[last="Stevens", first="W."]**)***
construct /results/book→$b

Note that the negative condition enclosed by *not()* is actually a sub-tree structure (in XTree format), and it has the existential quantification, i.e., for a book *$b*, there is no such author named "Stevens W.".

Example 30. Find the books that do not have a sub-element named "reference". We can write the XTree query as follows:

*query /bib/book→$b/**not(**reference**)***
construct /results/book→$b

Note that there is no condition defined in the XTree expression enclosed by *not()*, thus the *not()* operator requires that the inner XTree structure does not exist.

Example 31. Suppose each author of a book has one or more "address" elements, which consists of three sub-elements "street", "city" and "country". To find the books that do not have any author who do not have an address in New York, i.e., find the books where all the authors have an address in New York, we can write the following XQuery:

*query /bib/book→$b/**not(**author/**not(**address/city="New York"**))***
construct /results/book→$b

Note that the nested negations both have existential quantification for their enclosed expressions.

4.4 Group by

The XTree query language supports grouping in an explicit way. It has the keyword *groupby* for grouping operations. *Groupby* classifies the XML nodes into groups, based on the grouping fields, and assigns the result to a two-dimensional list-valued variable. Unlike in SQL, where all data are in a flat format making the grouping result clear and easy to use, XML data has more complicated structures. The variable that holds the grouping result has a two-dimensional structure: it is a list of lists; each inner list is the items of a particular group. The syntax of *groupby* is as follows:

{{$result}} → {$a} ***groupby*** *$b*

The above statement will group items in *{$a}* based on the element or attribute *$b*, and assign the result to the two-dimensional list *{{$result}}*. Here *{{$result}}* has a two dimensional structure, each inner list *{$result}* is the *$a* instances with a certain *$b* value. A special function *key()* will return the value of the grouping field for the current group. If the grouping is based on a tuple of multiple grouping fields, then the function *key()* will return the tuple value of the grouping fields of the current group, with the first item *key()[1]* being the value of the first grouping field, the second item *key()[2]* being the value of the second grouping field, and so on. By default, duplicate values will be eliminated in the grouping field.

In this way, the grouping operation is explicit and easy to read, and the entire document will only be scanned once for the grouping. What is more, our grouping will not generate invalid empty groups when there is more than one grouping field, because we regard multiple grouping fields as tuples, thus if some tuple does not occur in the document, it will also not appear in the grouping result.

Example 32. Consider the bibliography document in Appendix I, list the book titles published in each year.

```
query /bib/book→{$b},
      {{$booksByYear}} → {$b} groupby @year
construct /year/[@value→{$booksByYear}.key(),
          title→{$booksByYear}.iterate()/title]
```

The key or *year* of each grouping is assigned to the attribute value, and the title of each book in the grouping is assigned to the element *title*. Notice the function *iterate()* is used to iterate through the items in the groupings. If *iterate()* was not included then the list of titles in each grouping would be assigned to the element *title*.

Example 33. For the document employees.xml with DTD shown in Fig. 2, find the average salary of employees, grouping by department and jobtitle.

```
query /employees/employee→$e,
      {{$empgrp}} := {$e} groupby [department, jobtitle]
construct /type/[@dept→{$empgrp}.key()[1], @job→{$empgrp}.key()[2],
                avgsalary→({$empgrp}/salary).avg()]
```

4.5 Recursion

The XTree query language supports recursion directly in an easy and natural way. The *construct* clause of an XTree query is an XTree expression itself, which can also serve as a query data source (i.e., the querying part of a query can directly refer to an XTree expression defined in the *construct* clause of some query). Thus we can write a query on

the XTree expression of the *construct* clause in another query; or even write a query on the XTree expression of its own *construct* clause, to make the whole query recursively defined.

Example 34. Consider the list of employees, the XQuery script and result in Example 12. The query converts the list to a tree structure of employees, in which a parent node is the manager, and the children nodes are the direct subordinates. We show how this query is written using the XTree query language.

```
query /employeelist/employee→$e/not(@manager)
construct /employeetree/employee/[@id→$e/@id, @name→$e/@name]

query /employeetree//employee→$e/[@id→$x, not(employee)],
      /employeelist/employee→$e'/@manager→$x
construct $e/employee/{@id→$e'/@id, @name→$e'/@name}
```

The first query states that for each employee that does not have a manager, we put it as a first level employee in the tree. By evaluating this query, we will get an employee tree with only first level employees. Then the second query states that for each leaf employee in the tree, we search the employee list to find those employees whose manager is this employee, and put them one level below this employee in the tree. By evaluating this query, we will expand each leaf node in the employee tree, to add its subordinate employees, and continue this expansion with the newly added employee nodes.

Note that the second query is recursively defined, because its *query* clause queries the XTree expression defined in its own *construct* clause.

4.6 Quantification

Unlike XQuery that uses statements "*every…in…satisfies…*" and "*some…in…satisfies…*" in *where* clauses for universal quantification and existential quantification respectively, The XTree query language expresses universal quantification and existential quantification with the expressions *foreach $ele in {$var}* and *thereexists $ele in {$var}* respectively. In both cases, the variable *$ele* binds to elements in the list bound to *{$var}*.

Example 35. Consider the bibliography document in Appendix I, select those books whose authors all have a first name "Peter". We can write the following XTree query:

```
query /bib/book→$b/author→{$a}
where foreach $e in {$a} $e/first="Peter"
construct  /results/book→$b
```

This query binds the variable *{$a}* to the list of authors of a book, and then *$e* to individual authors in that list.

Example 36. Consider instances of the three DTDs in Appendix II, find the sailors, if any, who have reserved every boat. We can write an XTree query as follows:

*query (sailor.xml)/sailors/sailor→$s/@sid→**$sid***
(boat.xml)/boats/boat→{$b}
*(reservations.xml)/reservations/reservation→{$r}/@sid→**$sid***
where foreach $b in {$b}
thereexists $r in {$r}
$b/@bid = $r/@bid
construct /results/sailor→$s

4.7 Special Queries

The XTree query language can express some special queries that are not supported by XQuery, or not supported directly.

4.7.1 URL-related querying.

In an XTree query, variables can be bound to a URL or part of a URL, thus a user can write a query to get the URL information of the XML document. Such queries are not supported by XQuery.

Example 37. Suppose we want to find the bibliography documents located in the website *http://www.abc.com* in directory */docs/bib*, which have a structure similar to that shown in Appendix I and contain a book with value 1994 for the attribute *year* and "TCP/IP Illustrated" for the sub-element *title*. We can write an XTree query as follows:

query (http://www.abc.com/docs/bib/$file)
/bib/book/[@year="1994", title="TCP/IP Illustrated"]
construct /results/filename→$file

4.7.1 Structure level querying.

Structure level queries may be used to discover the structure of XML documents, or to rename elements/attributes in the result without knowing their inner structure. In XQuery, variables are bound to name-value pairs of some XML nodes, and special built-in functions are needed to split the name and the value. However, in the XTree query language, names and values of XML nodes are explicitly split in the variable bindings: variables on the left side of symbol → will bind to the node names, and variables on the right side of symbol → will bind to the node values. These variables can be used directly in the appropriate places, and no special functions are needed.

Example 38. Consider the bibliography in Appendix I, suppose we do not know the substructure of *book* elements, and we want to restructure books in this way: keep text nodes and sub-elements unchanged, but convert attributes to sub-elements in the format of <attribute name="*attribName*", value="*attribValue*"/>.

```
query /bib/book→$b/@$attribName→$attribValue
construct /result/book[$b/*, $b/text(),
                       attribute/[@name→$attribName, @value→$attribValue]]
```

Example 39. Get all the sub-elements of each book, except the sub-element *price*.

```
query /bib/book/$elemName→$elemValue
where $elemName !="price"
construct /result/book/$elemName→$elemValue
```

4.7.2 Sample querying and top-k querying.

Sample queries are the queries that return several items randomly after selecting and filtering, instead of getting the whole result set. Top-k queries are queries that return the first *k* items according to some order, instead of getting the whole ordered result set. Because XQuery does not have functions to get a certain subset of a list, it is difficult to express sample queries and top-k queries. However, the XTree query language can easily handle these queries since it supports list-valued variables explicitly, and defines various functions to manage a list.

Example 40. Consider the bibliography document in Appendix I, and a query to list any two books.

```
query /bib/book→{$b}
construct /result/book→{$b}.random(2)
```

Example 41. Get all the content of the two most expensive books.

```
query /bib/book→{$b}
construct /result/expensive_book→{$b}[1-2]
order by $b/price descending
```

Note that the statement *order by ... descending* will list items in descending order by the ordering fields.

4.8 Updates

The XTree query language is not only a data querying language, but also a data management language. Currently, the XQuery language can only query XML documents, but cannot update them. Researchers have proposed some methods to specify updates and

have developed techniques to process them efficiently [Liu et al. 2003; Tatarinov et al. 2001].

An update operation often follows some querying, in order to target the specific set of nodes that we want to update. After querying, instead of returning results, we can write update expressions to modify the XML document. In the XTree query language, there are five kinds of update statements:

- **insert** *content* **before** *var*
  This statement is used to insert the information in *content* before the position of *var*, where *content* is a segment of XML data expressed in XTree format, and *var* is either a variable or an invocation of some functions.
- **insert** *content* **after** *var*
  This statement is used to insert the information in *content* after the position of *var*.
- **insert** *content* **into** *var*
  This statement is used to insert the information in *content* into the structure of *var*.
- **delete** *var*
  This statement is used to delete information of *var*.
- **replace** *var* **with** *content*
  This statement is used to replace the information of *var* by the information in *content*, with the position unchanged.

Example 42. Consider the bibliography document in Appendix I, and an expression to add a new book as the first book in the document.

```
query /bib/book[1]→$b
insert book/[@id→"010", year→"2000",
            title→"Introduction to Algorithms",
            author/[first→"Thomas H.", last→"Cormen"],
            author/[first→"Charles E.", last→"Leiserson"],
            publisher→"The MIT Press", price→"69.99"]
before $b
```

Example 43. Add a sub-element named "comments" with value "best seller in year 2001" to the book whose id is 010.

```
query /bib/book→$b/@id="010"
insert comments→"among best sellers in year 2001"
into $b
```

Example 44. Delete all the books whose title contains the word "violence".

```
query /bib/book→$b/title→$t
where $t.contains("violence")
delete $b
```

Example 45. Increase the price by 10% for all books published after 1995.

```
query /bib/book[@year>1995, price→$p]
replace $p with 1.1 * $p
```

4.9 Comparison of related works

Here we make some comparisons between the XTree query language and some other existing XML querying languages, such as Lorel [Abiteboul et al. 1997], XQL [Robie et al. 1998], XML-QL[Deutsch et al. 1998], Quilt [Chamberlin et al 2000], XDuce [Hosoya and Pierce 2000], a rule-based semantic query language [Chippimolchai et al. 2002], a declarative XML query language[Liu and Ling 2002] and XQuery[Boag et al. 2003].

The comparison of these query languages emphasizes their expressive power, and whether types of queries can be written easily in the language. We will use the following criteria:

1. *Data model*: how to model XML data. This specifies what information in the XML document is accessible for querying.
2. *Expressions*: how to specify interesting paths in the query.
3. *Join*: whether the query language supports joins over different data sources.
4. *Negation*: whether the query language can express negative conditions in the query.
5. *Grouping*: whether the query language can divide the data into groups according to some grouping fields, and apply aggregate functions over each group.
6. *Recursion*: whether the query language can recursively query hierarchical data, and apply some transformation at each level of the hierarchy.
7. *Quantification*: whether the query language supports existential quantification and universal quantification.
8. *URL-related querying*: whether the query language supports queries on the URL information.
9. *Structure level querying*: whether the query language supports queries on XML documents with unknown structure.
10. *Sample/Top-k querying*: whether the query language supports queries that only pick up several items randomly, or only pick up the first several items according to some order, instead of the whole result set.
11. *Ordering*: whether the query language can list the items in ascending/descending order by the ordering fields.
12. *Nesting*: whether the query language supports nested querying structure (queries containing nested sub-queries), in order to express complex queries.
13. *Updates*: whether the query language can specify update operations on XML documents.

	XTree query language	**Lorel**	**XQL**	**XML-QL**	**A rule-based semantic querying**	**A declarative XML querying**	**XQuery**
Data model	complex object data model	Lore data model	XML implied data model	Unordered /Ordered data model	XDD (XML Declarative Description)	complex object data model	XQuery /XPath data model
Expression	XTree	OQL-like	XPath	regular tag expression	XML-like patterns	XTree-like expression	XPath
Join	YES	YES	Partial	YES	Unsure	YES	YES
Negation	YES	YES	YES	NO	NO	Unsure	YES
Grouping	YES	YES	NO	NO	NO	YES	YES
Recursion	YES	Unsure	NO	NO	NO	YES	YES
Quantification	YES	YES	YES	existential	existential	YES	YES
URL-related querying	YES	NO	NO	NO	NO	YES	NO
Structure level querying	YES	NO	NO	YES	NO	YES	YES
Sample/Top-k querying	YES	NO	NO	NO	NO	NO	NO
Ordering	YES	YES	NO	YES	NO	NO	YES
Nesting	No need	YES	NO	YES	NO	No need	YES
Updates	YES	YES	NO	NO	NO	NO	NO

Table 3. Comparison between XML query languages

Table 3 shows the comparison between these XML query languages. Note that for the join, negation, grouping, recursion and quantification operations, XTree query can express them in a more direct way: for the join operations, XTree query uses a QBE-like solution; for negation, XTree query can express a negative sub-tree in the query part; for grouping operations, XTree query uses a two dimensional list to hold the values of all groups, to avoid multiple scans over the document; for recursion, XTree query can directly query the output XTree expression in the *construct* clause; and for quantification, XTree query can invoke built-in functions on list-valued variables directly. In addition, because XTree query is based on XTree expressions, which can bind multiple variables in one expression, the queries are more compact.

As a conclusion, XTree query can support most database operations efficiently, outperforming other query languages for XML documents.

5. TRANSFORMING AN XTREE QUERY TO XQUERY

5.1 Transformation algorithm for *query* clause

For each XTree expression in the querying clause of an XTree query, we will transform it to a set of XPath expressions. This is not as trivial as just extracting each path associated

with a variable to be an XPath expression, because variables may correlate to each other by some common ancestors, thus we need to use such common ancestors to constrain the descendent variables. In fact, the common ancestors are those branching nodes, which are the nodes just before every pair of square brackets, because the pair of square brackets implies that all the enclosed sibling branches are interesting to the user, whether or not the sibling branches will head to some variable bindings or some constraints.

We first define some terms before introducing the algorithm.

Definition 5. Definition 1 has defined that the *associated path* of variable *$a* (or *{$a}*) is the absolute path expression from the root of the document to the nodes represented by *$a* (or *{$a}*). Function *path($var)* returns the associated path of variable *$var* in an XTree expression, *path({$var})* returns the associated path of variable *{$var}* in an XTree expression.

For example, for the XTree expression */bib/book/[title→$t, author→{$a}]*, *path($t) = /bib/book/title*, *path({$a}) = /bib/book/author*.

Definition 6. The *relative path* of $path_1$ with regard to $path_2$ is the path starting from the endpoint of $path_2$ and ending at the endpoint of $path_1$. Function *relaPath($path_1$, $path_2$)* returns the relative path of $path_1$ with regard to $path_2$. It can be evaluated by a prefix elimination of $path_2$ in $path_1$.

For example, *relaPath(/a/b/c/d, /a/b) = c/d*, *relaPath(/a/b, /a/b) = null* and *relaPath(/a/b, /a/b/c/d) = null*

Definition 7. Variable *$a* is the *nearest ancestor variable* of variable *$b* if *$a* is an *ancestor variable* of *$b*, and no other ancestor variables of *$b* is defined between *path($a)* and *path($b)*.

For example, in the XTree expression */bib/book→$b/[title→$t, author/last→$last]*, *$b* is the nearest ancestor variable of *$t* and *$last*. In the XTree expression */bib/book→$b/ author→$a/[first→$first,last→$last]*, *$b* is not the nearest ancestor variable of *$last.*

The following algorithm describes how to translate the XTree expressions in the query part of an XTree query script into a set of XPath expressions.

Algorithm TRANS_QUERY

Input: a *query* clause of an XTree query.

Output: a set of XPath expressions in *let/for* clauses of XQuery.

If there is a negation in the XTree expression, process the negative subtree with **TRANS_NOT**;

If there are two or more variables with the same name, process the join with **TRANS_JOIN**;

If there is a grouping operator *groupby*, process the grouping with **TRANS_GROUP**;

Otherwise, for each XTree expression, process it from left to right,
for each node traversed,

Case 1: If it is an anonymous branching node, and is not the root, i.e., in the expression *some_xpath_from_root//[child$_1$..., child$_2$..., ...]*

Use *some_xpath_from_root//*/..* to express this node, and assign a new single-valued variable *$var* to it. Translate this expression to an XPath expression in a *for* clause:

for $var in some_xpath_from_root ///..*

Case 2: If it is a named branching node, and is not the root, and is not originally bound to a variable

Assign a new single-valued variable *$var* to this node, and translate it to an XPath expression in a *for* clause:

for $var in xpath_of_$var

Case 3: If the node value is bound to a single-valued variable, i.e., in the expression *elem→$value* (or *@attrib→$value*)

Translate it to an XPath expression in a *for* clause:

for $value in xpath_of_$value

Case 4: If the node value is bound to a list-valued variable, i.e., in the expression *elem→{$value} (or @attrib→{$value})*

Translate it to an XPath expression in a *let* clause:

let $value := xpath_of_{$value}

Case 5: If the node name is bound to a single-valued variable, i.e., in the expression *$name→$value (or @$name→$value)*

Process it as **→$value (or @*→$value)*, and translate it to an XPath expression in a *for* clause, as in case 4:

for $value := xpath_of_$value (note that this XPath will end with * or @*)

In the *where /construct/order by* clause of the query, replace every occurrence of *$name* to be *local-name($value)*

Case 6: If it is a leaf node without any variable binding

Then it is used to assure existence of such structure.

Assign a new single-valued variable *$var* to it, and translate it to an XPath expression in a *for* clause:

for $var in xpath_of_$var

Add a condition *exists($var)* in the *where* clause.

For each of the above 6 cases, when writing XPath expression of a variable x*path_of_$var* (or *xpath_of_{$var}*), always check whether this variable has some ancestor variable

If it does not have any ancestor variable, then write *path($var)* (or *path({$var})*) as its XPath expression.

Otherwise, if its nearest ancestor variable is *$anc*, then write its XPath expression as *$anc/relaPath(path($var), path($anc))* (or *$anc/relaPath(path({$var}), path($anc))*) ■

The main idea of the algorithm is that for an XTree expression, we find all the common ancestors, except the root (since an XML document only has one root node), and assign single-valued variables to them if they are not bound to variables originally. Then from left to right, translate each single-valued variable binding to an XPath expression in a *for* clause, and translate each list-valued variable binding to an XPath expression in a *let* clause.

Note that in the above algorithm, whenever we encounter a list-valued variable *{$var}*, we will just use its inner name *$var* (without curly braces *{ }*) in the output, because in XQuery a variable defined in a *let* clause does not have curly braces in its name. Also, since we process the XTree expression in a left-to-right manner, the output XPaths of an XTree expression will be in depth-first order of the XTree.

Example 46. Translate the following XTree expression to a set of XPath expressions:

query /bib/[book/[@$attrib→$value, title→$t, author→{$a}],
journal//[last,first]]

Using the algorithm, we process the XTree expression from left to right, when we reach node *book*, based on case 2 we output the XPath expression: *for $b in /bib/book*. Later when we reach node *@$attrib→$value*, based on case 5 we get the XPath expression: *for $value in $b/@** (since *$value* has an ancestor variable *$b*), and replace *$attrib* by *local-name($value)* in other clauses of the query. Next we reach node *title→$t*, based on case 3, we output: *for $t in $b/title*. For the next node *author→{$a}*, based on case 4, we get: *let $a := $b/author*. Then we reach node *journal*, using case 1, we get: *for $var in /bib/journal//*/..* For the last two nodes *last* and *first*, using case 6, we output: *for $last in $var/last*, *for $first in $var/first*, and add two conditions *exists($last)* and *exists($first)* in the *where* clause of XQuery.

Thus the final output is the following XPath expressions:

for $b in /bib/book	(case 2)
*for $value in $b/@**	(case 5)
for $t in $b/title	(case 3)
let $a := $b/author	(case 4)
for $var in /bib/journal///..*	(case 1)
for $last in $var/last, $first in $var/first	(case 6)

Wherever *$attrib* occurs later in the query, replace it by *local-name($value)*;
Add *exists($last)* and *exists($first)* in the *where* clause of XQuery.

For negation, join and grouping operations, we handle them in other algorithms.

Algorithm TRANS_NOT
Input: a negative subtree in an XTree expression in the *query* clause of an XTree query.
Output: a set of XPath expressions in *let/for* clauses and associated conditions in a *where* clause of an XQuery script.

Step 1: For the negative subtree enclosed by the *not()* operator, assign a single-valued variable *$var* for the root and each branching node, and generate its XPath expression *path($var)* (or *$anc/relaPath(path($var), path($anc))* if it has the nearest ancestor variable *$anc*), we denote this XPath to be *%xpath_node$_x$%* for the node *node$_x$*.

Step 2: Generate sub-condition expressions for the leaf nodes.

For each leaf node $node_i$, generate its XPath expression *%xpath_node$_i$%* relative to its nearest ancestor variable.

Case 1: If $node_i$ does not set any condition, which means the existence of such structure

%cond_node$_i$% = exists(%xpath_node$_i$%)

Case 2: If $node_i$ carries some condition

%cond_node$_i$% = condition of node$_i$ using %xpath_node$_i$%

Step 3: Output those common variables of the negative subtree with the (possibly nested) format "*some (descendent $var) in (ancestor $var) satisfies (leaf condition)*", retaining their levels. Use the *and* operator to connect these conditions, and output a negative condition in the *where* clause of XQuery, such as:

not (some var_1$ in %xpath_var$_1$% satisfies
(some var_2$ in %xpath_var$_2$% satisfies %cond_node$_1$%) and
(some var_3$ in %xpath_var$_3$% satisfies %cond_node$_2$% and %cond_node$_3$%)) ■

Example 47. Translate the following 3 negative subtrees in the query clauses of XTree queries:

(1) *query /bib/book→$b/**not(**author/[last="Stevens", first="W."]**)***
(2) *query /bib/book→$b/**not(**reference**)***
(3) */bib/book→$b/**not(**author/**not(**address/city="New York"**))***

Using the algorithm to translate (1), in step 1 we assign *$a* to the root of the negative subtree, whose XPath expression is *$b/author*. In step 2, for leaf node *last*, its XPath expression is *$a/last*, and it carries an equality condition: *$a/last = "Stevens"*; similarly, for leaf node *first*, the condition is *$a/first = "W."*. In step 3, we output:

not (some $a in $b/author satisfies $a/last = "Stevens" and $a/first = "W.")

For (2), in step 1 we assign a single-valued variable *$r* for the root of the negative subtree, whose XPath expression is *$b/reference*. In step 2, there is only one leaf node (also the root node), which does not carry any condition, meaning the existence of such a structure. Thus its condition expression is: *exists($b/reference)*. And in step 3, we output: *not(exists($b/reference))*. In fact, in XQuery, this is equivalent to *not($b/reference)*.

Example (3) demonstrates nested negation. For the outer negation, in step 1 we assign *$a* to the root, whose XPath expression is *$b/author*. Then we meet the inner sub-negation immediately, thus step 2 of the outer negation is omitted. Turning to the inner negation, in step 1, we assign *$addr* to its root, whose XPath expression is *$a/address*. In step 2, for the leaf nodes *city*, generate its condition: *$addr/city = "New York"*. In step 3, we get the translated statement for the inner sub-negation:

not (some $addr in $a/address satisfies $addr/city = "New York")

Back to the outer negation, in step 3, we output a condition in a *where* clause:

not (some $a in $b/author satisfies
not (some $addr in $a/address satisfies $addr/city="New York"))

Algorithm TRANS_JOIN

Input: two or more nodes bound to the same variable name in XTree expressions in the *query* clause of an XTree query.

Output: a set of XPath expressions for these nodes in *let/for* clauses of XQuery.

There are two alternatives for the translation of a join operation:

Alternative 1: Rename and Set equal condition

Suppose there are *n* variables with the same name *$var* that occur in the query clause.

Step 1: Rename these variables to different names var_1, var_2, … var_n , and generate their XPath expressions in *for/let* clauses as before.

Step 2: Add equality constraint in the *where* clause of XQuery:

$var_1 = var_2 = … = var_n$

Alternative 2: Nested sub-query on parent variables

This is used when those parents of the join variables are also bound to variables.

Suppose there are *n* variables with the same name *$var*, and their *parents* are bound to variables p_1, p_2, … p_n, i.e., the query is like:

query .../$parent_1$→p_1/$child_1$→$var
.../$parent_2$→p_2/$child_2$→$var
……
.../$parent_n$→p_n/$child_n$→$var

Step 1: Choose any one of *$var* (say, the first one) and generate its XPath expression in a *for* clause: *for $var in p_1/$child_1$*.

Step 2: For other join nodes, use *$var* as condition of sub-query on their parent nodes:

for $p_2 in .../$parent_2$[$child_2$ = $var]
……
for $p_n in .../$parent_n$[$child_n$ = $var] ■

Alternative 1 is universally applicable, it can be used in all the cases of join. However, it introduces many new variable names in order to resolve the naming conflict, and tests all of them in a condition, which could cause an overhead during query evaluation. On the other hand, alternative 2 can only be used when the parents of the join variable are also bound to variables. However, it may reduce the number of variables in the query, and reduce their associated equality conditions in the *where* clause. Thus, if an equality join is performed on many sources, and most of the parents of the join node are also bound to some variables, we will use alternative 2 for translation.

Example 48. Translate the join nodes in the query clause of the following XTree query:

*query (sailors.xml)/sailors/sailor/[@sid→**$sid**, sname→$sname],*
*(boats.xml)/boats/boat/[@bid→**$bid**, bname→$bname],*
(reservations.xml)/reservations/reservation/
*[@sid→**$sid**, @bid→**$bid**, start-time→$st, end-time→$et]*

Using alternative 1, step 1 performs a renaming to resolve the naming conflict:

for $s in doc("sailors.xml")/sailors/sailor, sid_1$ in $s/@sid, ...
for $b in doc("boats.xml")/boats/boat, bid_1$ in $b/@bid, ...
for $r in doc("reservations.xml")/reservations/reservation, sid_2$ in $r/@sid,
bid_2$ in $r/@bid

Step 2 adds equality conditions in the *where* clause:

sid_1$ = sid_2$ and bid_1$ = bid_2$

We can also use alternative 2, since the parents of join nodes are branching nodes, they are also bound to variables. In step 1, we choose one *$sid* and one *$bid* for their normal XPath expressions:

for $s in doc("sailors.xml")/sailors/sailor, $sid in $s/@sid, ...
for $b in doc("boats.xml")/boats/boat, $bid in $b/@bid, ...

Then in step 2, we use *$sid* and *$bid* for sub-query on the parent node "reservation":

for $r in doc("reservations.xml")/reservations/reservation[@sid=$sid,@bid=$bid]

Algorithm TRANS_GROUP
Input: A grouping statement *{{$group}}* → *{$a}* ***groupby*** *[exp$_1$, exp$_2$, ... exp$_n$]* in the *query* clause of an XTree query.
Output: a set of XPath expressions for the grouping fields and grouped field in the *let/for* clauses of XQuery, and some modifications of other clauses of XQuery.

Step 1: Assign a single-valued variable *var_i$* to each grouping field *exp$_i$*, and output its XPath expression in a *for* clause. The XPath expressions are enclosed in *distinct-values()* function, and the *for* clauses will be nested for multiple grouping fields.

Step 2: Assign a variable *$group* for each group (with the same value in the grouping fields), and output its XPath expression in a *let* clause:

let $group := $a[relaPath(exp$_1$, exp_$a)=var_1$ and ... and
relaPath(exp$_n$, exp_$a)=$var$_n$]

Note that function *relaPath(exp$_i$, exp_$a)* returns the XPath expression of *exp$_i$* relative to the XPath expression of *$a*.

Step 3: In the *where /construct/order by* clauses of the XTree query,
For each occurrence of *{$group}.key()[i]*, replace it by *var_i$*;
For each occurrence of *{$group}*, replace it by *$group*;
Change aggregate functions from object oriented style to functional style;
If there are multiple grouping fields, add a conditional check to prune empty groups in the *where* clause (some combination of grouping fields may not produce any result):

if (count($group) > 0)
then { translated code for construct clause }
else () ■

Example 49. Translate the grouping operation in the following XTree query:

query /bib/book→{$b},

{{$booksByYear}} → {$b} groupby $b/@year
construct /year/[@value→{$booksByYear}.key(), title→{$booksByYear}.iterate()/title]

In step 1, we assign a single-valued variable *$y* for the grouping field:

for $y in distinct-values($b/@year)

In step 2, we assign list-valued variable *$booksByYear* for each group:

let $booksByYear := $b[@year=$y]

In step 3, in the *construct* clause, we replace *{$booksByYear}.key()* to be *$y*, and replace *{$booksByYear}.iterate()/title* by *$booksByYear/title*.

Example 50. Translate the grouping operation in the following XTree query:

query /employees/employee→$e,
{{$empgrp}} := {$e} groupby $e/[department, jobtitle]
construct /type/[@dept→{$empgrp}.key()[1], @job→{$empgrp}.key()[2],
avgsalary→({$empgrp}/salary).avg()]

In step 1, we assign single-valued variables *$dpt* and *$jt* for the two grouping fields, and output the XPath expressions in nested *for* clauses:

for $dpt in distinct-values($e/department)
for $jt in distinct-values($e/jobtitle)

In step 2, we assign list-valued variable *$empgrp* for each group:

let $empgrp := $e[department=$dpt and jobtitle=$jt]

In step 3, in the *construct* clause, we replace *{$empgrp}.key()[1]* by *$dpt*, replace *{$empgrp}.key()[2]* by *$jt*, replace *{$empgrp}/salary* by *$empgrp/salary*, change the function *({$empgrp}/salary).avg()* to be *avg($empgrp/salary)*, and add a conditional check to prune empty groups (some combination of *department* and *jobtitle* may not produce any *employee* result) :

if (count($empgrp) > 0)
then { translated code for construct clause }
else ()

5.2 Transformation algorithm for *where* clause

The *Where* clauses in XTree query and in XQuery are very similar, but some translations are required, for the object oriented functions and list operations.

Algorithm TRANS_WHERE
Input: a *where* clause of an XTree query.
Output: some condition expressions in the *where* clause of an XQuery script.
For each condition expression in *where* clause,
Step 1: Change quantification to the style used in XQuery.

Case 1: *{$var})/.../node op expr* translate to:
some $node in $var/.../node satisfies $node op expr
Case 2: foreach $var in *{$var} {$var}/.../node op expr* translate to:
every $node in $var/.../node satisfies $node op expr

Step 2: Change object-oriented function invocations of variables to the functional style in XQuery syntax.

E.g., change *{$value}.avg()* to *avg($value)*,
change *$title.contains("xml")* to *contains($title, "xml")*

Step 3: Translate the list functions and operations to some quantification statements in the *where* clause of XQuery.

Case 1: *{$var}.none()/.../node op expr* translate to:
not(some $node in $var/.../node satisfies $node op expr)
Case 2: *$x ∈ {$y}* translate to:
some $y' in $y satisfies $x=$y'
Case 3: *$x ∉ {$y}* translate to:
not (some $y' in $y satisfies $x=$y')
Case 4: *{$x} ⊆ {$y}* translate to:
every $x' in $x satisfies some $y' in $y satisfies $x'=$y'
Case 5: *{$x} ⊂ {$y}* translate to:
every $x' in $x satisfies some $y' in $y satisfies $x'=$y'
and not(every $y'' in $y satisfies some $x'' in $x satisfies $x''=$y'')
Etc. ■

Step 4: Change any other use of *{$var}* to *$var*, since during translation of *query* clause, we have translated list-valued variable *{$var}* in XTree query to be *$var* in a *let* clause in XQuery.

5.3 Transformation algorithm for *construct* clause

Transforming an XTree expression in the *construct* clause of an XTree query to some XQuery expressions is complicated, since we will often encounter nested sub-queries in XQuery. Also, if the node name to get the variable value in the result construction part is different from the node name where the variable was bound in the querying part (i.e., the user wants to rename the node), it will be difficult to handle, since we have to explicitly split the name-value pair of variables in XQuery.

Thus, in order to construct the result format correctly, we have to determine the correspondence between the structure of XTree expressions in the *query* clause and the structure of XTree expression in the *construct* clause.

Definition 8. Node B (in the *construct* clause of an XTree query) is ***derivable*** from node A (in the *query* clause of an XTree query) if the content of B can be derived from the content of A. Intuitively, "node B is *derivable* from node A" means the nodes are correlated as follows: node A is binding values to variables in the querying part; and node B is getting values from those variables in the result construction part.

Property 1. Suppose node A is in the *query* clause, node B is in the *construct* clause, then node B is *derivable* from node A if and only if one of the following cases holds:

(1) If node B is a leaf node and its expression does not contain any variable (e.g., *node*, *node→"abc"*, etc), then node B is not correlated to any node A. In this case we say node B is *trivially derivable*.

(2) If the expression of node A is: *A→$x*; the expression of node B is: *B→$x*, or an invocation of a function on *$x* (e.g., *$x.substring(1,5)*, *$x.string-length()*, *$x.normalize-space()*, etc), then node B is *derivable* from node A.

(3) If the expression of node A is: *A→{$x}*; the expression of node B is: *B→{$x}*, or an invocation of a list-valued function on *{$x}* (e.g., *{x}[1-3]*, *{x}.distinct()*, *{$x}.sort()*, etc), or an invocation of an aggregate function on *{$x}* (e.g., *{x}.count()*, *{x}.avg()*, etc), then node B is *derivable* from node A.

(4) If node A has variable bindings on its children A_1, A_2, … A_m; node B has variable substitutions on its children B_1, B_2, … B_n. If every node B_i is *derivable* from some node A_j, then node B is *derivable* from node A.

(5) Node B is NOT *derivable* from node A for any other cases.

The transformation algorithm for the *construct* clause is divided into three procedures. The first procedure traverses the XTree expression for result construction, and selects an XPath expression for each derivable node in the output of the algorithm TRANS_QUERY; the second procedure translates node expressions to the XQuery version; and the third procedure adjusts query blocks to ensure correct structure of the final XQuery.

Algorithm TRANS_CONSTRUCT
Input: a *construct* clause of an XTree query
Output: a query in XQuery format.
PROC_A (Get XPath expressions from result of TRANS_QUERY)
Process the XTree expression from left to right, for each node traversed:
Case A1: If it is a branching node without → symbol, and is not derivable from any common ancestor variable node
It is the first of several levels in the result construction
Write this node directly as a pair of XML tags
Case A2: If it has no → symbol, and is derivable from some common ancestor node that is bound to *$anc* (or *{$anc}*)
Form a new query block
Write the XPath expression of *$anc* (or *{$anc}*) in the result of TRANS_QUERY
Write this node in the *return* clause directly as an XML tag
Case A3: If it has the expression $node_i \rightarrow \$var_i$ (or $node_i \rightarrow \{\$var_i\}$), and it is derivable from a node that is bound to $\$var_i$ (or $\{\$var_i\}$)

Form a new query block
Write the XPath expression of $\$var_i$ (or $\{\$var_i\}$) in the result of TRANS_QUERY
Call **PROC_B** to translate this node expression in a *return* clause

Case A4: If it is a trivially derivable leaf node
Write this node directly as an XML segment

Call **PROC_C** to check the next node to adjust query blocks

PROC_B (Translate node expressions)

The translation will be different depending on whether the node name is changed (in result construction versus in querying). If the node name remains the same, we just get the variable value at the place; if the node name is changed, we need to consider all the possible inner structure (sub-elements, attributes, text fields) of the node, and put them under the new node name.

Case B1: If the expression is: *element* → *$var*
Case B1.a: If the node name is unchanged:
Return "*{ $var }*" (here *{ }* means evaluation of the enclosed expression)
Case B1.b: If the node name is changed:
Return "*<element> {$var/*} {$var/@*} {$var/text()} </element>*"

Case B2: If the expression is: *element* → *{$var}*
Case B2.a: If the node name is unchanged:
Return "*{ $var }*"
Case B2.b: If the node name is changed:
Return "*{ for $x in $var*
return <element> {$x/*} {$x/@} {$x/text()} </element>}*"

Case B3: If the expression is: *@attr* → *$var*
Case B3.a: If the node name is unchanged:
Return "*{ $var }*"
Case B3.b: If the node name is changed:
Suppose its parent element name is *elem*
Return "*<elem attr={ string($var) }*"

In the above cases, if the right side of → symbol is a function invocation of the variable, we will adjust the output accordingly: add the function invocation to that variable, changing the object oriented style call to a functional style call.

PROC_C (Adjust query blocks)

Case C1: If the next node to process is a descendent of the current node
Continue current query block
Next step will decide whether to form a sub-query block inside the current one

Case C2: If the next node to process is a sibling of the current node
Close current query block
Next step will decide whether to form a sub-query block that is parallel to the current one

Case C3: If the next node to process is a sibling of an ancestor node $node_i$ of the current node, which was processed before
Close current query block
Propagate up to close query blocks of the ancestor nodes of current node, till $node_i$ (including the query block of $node_i$)

Next step will decide whether to form a sub-query block that is parallel to the query block of $node_i$

Case C4: If the current node is the last node
Close the current query block
Close query blocks of all the ancestor nodes of the current node ■

The main idea of the algorithm is that we process the XTree expression in the *construct* clause from left to right. For each derivable node, we pick up its corresponding XPath expression in the output of algorithm TRANS_QUERY, translate its node expression to some XQuery statements, and use curly braces *{ }* to form nested query blocks according to the structure defined in the XTree expression in the *construct* clause.

5.4 Transformation algorithm for *order* by clause

The *order by* clause in XTree query is the same as that in XQuery. It is a list of ordering fields, which defines the order of output. The new features of XTree query are not applicable to the *order by* clause: we cannot order the result by a tree structure, and list-valued variables are not used as ordering fields. Thus there is no translation needed for *order by* clauses, and we just need to copy all the ordering fields, preserving their sequence.

Algorithm TRANS_ORDERBY
Input: an *order by* clause of an XTree query.
Output: an *order by* clause in an XQuery script.
Copy all the ordering fields, in their original sequence. ■

Example 51. Translate the following XTree query:

```
query /bib/{book/{title→$t, author→{$a}},
            journal/{title→$jt, editor/{last→$last, first→$first}}}
construct /result/{book/{name→$t, authors/{@count→{$a}.count( ), au→{$a}}},
            journal/{title→$jt, editor/{first→$first, last→$last}}
            lastUpdate→"01/09/2004"}
```

Fig. 3a shows the XTree graph for querying part, and Fig. 3b shows the XTree graph for result construction part.

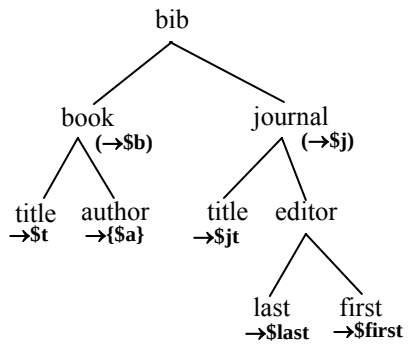

Fig. 3a. XTree graph for querying

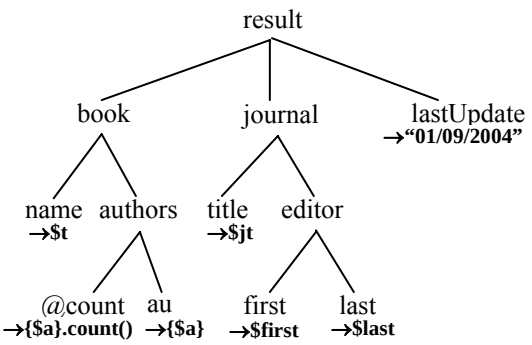

Fig. 3b. XTree graph for result construction

For the XTree expression in the *query* clause, by applying algorithm TRANS_QUERY, we will get the following XPath expressions in *for/let* clauses (which will be selected later in the algorithm PROC_A of TRANS_CONSTRUCT):

```
for $b in /bib/book
for $t in $b/title
let $a := $b/author
for $j in /bib/journal
for $jt in $j/title
for $e in $j/editor
for $last in $e/last
for $first in $e/first
```

This XTree query does not have a *where* clause or an *order by* clause, and there is no new condition added when executing TRANS_QUERY, thus the translated XQuery also has no *where* clause or *order by* clause.

For the *construct* clause, by applying algorithm TRANS_CONSTRUCT, we will get the following XQuery script (the details of the algorithm are omitted):

```
<result> ……..... (case A1, C1)
{
  for $b in /bib/book
  return <book> …… (case A2, C1)
  {
    for $t in $b/title
    return <name> {$t/*} {$t/@*} {$t/text()} </name>
  } …….. (case A3, B1.b, C2)
  {
    let $a := $b/author
    return <authors count={count($a)}> …… (case A2, B3.b, C1)
    {
      for $x in $a
      return <au> {$x/*} {$x/@*} {$x/text()} </au>
    }
```

```
      </authors>
   }
   </book>
   }  ……..        (case A3, B2.b, C3)
for $j in /bib/journal
   return <journal>  ……...        (case A2, C1)
   {
      for $jt in $j/title
      return {$jt}
   }  ……..        (case A3, B1.a, C2)
   {
      for $e in $j/editor
      return <editor>  ……..        (case A2, C1)
      {
         for $first in $e/first
         return {$first}
      }  ……..        (case A3, B1.a, C2)
      {
         for $last in $e/last
         return {$last}
      } </editor>
   } </journal>  ……...        (case A3, B1.a, C3)
   <lastUpdate>01/09/2004</lastUpdate>
} </result>  …….        (case A4, C4)
```

6. CONCLUSION AND FUTURE WORKS

In this paper, we have discussed some limitations of XPath and XQuery, and proposed more general languages XTree and XTree query language based on the complex object data model [Liu and Ling 2002].

XTree has a tree structure, which is more compact and convenient to use than XPath. It can be used in both the querying part and the result construction part of a query: in the querying part, multiple variables can be defined in one XTree expression; in the result construction part, a user can write one XTree expression to define the result format in a clearer and more compact way. In XTree expressions, list-valued variables are explicitly indicated, and their values are uniquely determined. Some natural built-in functions are defined to manipulate list-valued variables in an object-oriented fashion.

The XTree query language effectively avoids nesting in the query, making the query easier to read and comprehend. It also directly supports join, negation, grouping, recursion, quantification, updates and some special queries (such as URL-related querying, structure level querying, sample querying and top-k querying) which are not supported by XQuery, or not supported directly. Compared to XQuery, queries written in XTree query are much shorter in length and easier to understand.

To utilize existing XQuery parsers, we have also designed algorithms to translate an XTree query to a standard XQuery script. The algorithms will translate each clause in an XTree query to XQuery statements, and assemble query blocks to form a nested XQuery script.

We have already started investigating more expressive query languages based on XTree. An example is the query language XDO2 [Zhang et al. 2005], which supports recursive rules and object-oriented features such as methods and inheritance, extending both the expressive and modeling powers of the XTree query language.

For future research, we intend to implement the XTree query system that executes XTree queries directly, instead of translating it to XQuery. The query evaluation will be more efficient, since in XTree query we have a global view of the entire query tree.

We also want to extend the transformation algorithms to support recursive queries. To do this, we need to study how the queries recur in XTree query and define corresponding recursive functions in XQuery.

Usually the XQuery scripts output by the transformation algorithms are very lengthy, because the algorithms assume nothing is known in the structure of the XML document, thus all the possible content of an element/attribute need to be considered. If we do know the schema of the document, we may optimize the queries to be more compact. We will investigate the formal optimization algorithms that can utilize the schema of the XML documents, to make the output XQuery scripts simpler.

In addition, we will observe the progressive development of XQuery to continuously enhance the expressive power of the XTree query language.

REFERENCES

ABITEBOUL S., QUASS D., MCHUGH J., WIDOM J., and WIENER J.L. 1997. The Lorel Query Language for Semistructured Data. *Intl. Journal of Digital Libraries,* 1(1):68-99.

BERGHUND A., BOAG S., CHAMBERLIN D., FERNANDEZ M.F., KAY M., ROBIE J., SIMEON J., 2003. XML Path Language (XPath) 2.0. W3C Working Draft, *http://www.w3.org/TR/xpath20/*

BOAG S., CHAMBERLIN D., FERNANDEZ M.F., FLORESCU D., ROBIE J., SIMEON J., 2003. XQuery 1.0: An XML Query Language. W3C Working Draft, *http://www.w3.org/TR/xquery/*

BONIFATI A., CERI S. 2000. Comparative Analysis of Five XML Query Languages. *SIGMOD Record*, 29(1):68-79.

CATTELL R.G.G., BARRY D. 1997. The Object Database Standard: ODMG 2.0. *Morgan Kaufmann*, Los Altos, CA..

CERI S., COMAI S., DAMIANI E., FRATERNALI P., PARABOSHI S., TANCA L. 1999. XML-GL: a Graphical Language for Querying and Restructuring WWW data. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada.

CERI S., COMAI S., DAMIANI E., FRATERNALI P., TANCA L. 2000. Complex Queries in XML-GL. *SAC(2)* :888-893.

CHAMBERLIN D., FANKHAUSER P., MARCHIORI M., ROBIE J. 2003. XML Query Requirements. W3C Working Draft. *http://www.w3.org/TR/xquery-requirements/*

CHAMBERLIN D., ROBIE J., FLORESCU D. 2000. Quilt: An XML query language for heterogeneous data sources, In *Proceedings of International Workshop on the Web and Databases*.

CHIPPIMOLCHAI P., WUWONGSE V., ANUTARIYA C. 2002. Semantic Query Formulation and Evaluation for XML Databases. In *Proceedings of WISE 2002*, 205-214, Singapore.

CLUET S., SIMEON J. 1999. YATL: a Functional and Declarative Language for XML. *http://db.bell-labs.com/user/simeon/icfp.ps*.

COHEN S., KANZA Y., KOGAN Y., NUTT W., SAGIV Y., SEREBRENIK A. 1998. Equix – Easy Querying in XML Databases. In P*roceedings of Webdb'98 – The Web and Database Workshop*.

COMAI S., DAMIANI E., FRATERNALI P. 2001. Computing Graphical Queries over XML Data. *ACM Transactions on Information Systems*, Vol. 19, No. 4, Pages 371-430.

COMAI S., DAMIANI E., TANCA L. 1998. The WG-Log System: Data Model and Semantics. *INTERDATA technical report*, T2-R06.

DATE C.J. 1981. An Introduction to Database Systems. 3rd Edition, *Addison-Wesley Publishing Company*.

DEUTSCH A., FERNANDEZ M., FLORESCU D., LEVY A., SUCIU D. 1998. XML-QL: A Query Language for XML. *http://www.w3.org/TR/1998/Note-xml-ql-19980819*.

DRAPER D., FANKHAUSER P., FERNANDEZ M.F., MALHOTRA A., ROSE K., RYS M., SIMEON J., WADLER P. 2003. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, *http://www.w3.org/TR/xquery-semantics/*

HOSOYA H., PIERCE B. 2000. XDuce: A Typed XML Processing Language (Preliminary Report). In *Proceedings of WebDB Workshop*.

KAY M. 2003. XSL Transformations (XSLT) Version 2.0, W3C Working Draft 12, *http://www.w3.org/TR/xslt20/.*

LIU M.., LING T.W. 2002. Towards Declarative XML Querying. In *Proceedings of WISE 2002*, 127-138, Singapore.

LIU M., LU L. WANG G.R. 2003. A Declarative XML-RL Update Language. In *Proceedings of ER2003*, 506-519, Chicago, USA.

MUNROE K.D., PAPAKONSTANTINOU Y. 2000. BBQ: A visual interface for integrated browsing and querying of XML. *In Proceedings of Visual Database Systems*.

ROBIE J., LAPPJ., SCHACH D. 1998. XML Query Language (XQL). *http://www.w3.org/TandS/QL/QL98/pp/xql.html*.

TATARINOV I., IVES Z.G., HALEVY A.Y., WELD D.S. 2001. Updating XML. In *Proceedings of SIGMOD 2001*, pages 413-424.

ZHANG W., LING T.W., CHEN Z., DOBBIE G. 2005. XDO2 for XML Deductive Object-Oriented Query Language. To appear *DASFAA'05*.

Appendix I. Sample XML document of bibliography data

```
<?xml version="1.0" encoding="UTF-8"?>
<bib name="IT">
   <book id="b001" year="1994">
      <title>TCP/IP Illustrated</title>
      <author>
         <last>Stevens</last>
         <first>W.</first>
      </author>
      <publisher pid="p01">Addison-Wesley</publisher>
      <price>65.95</price>
   </book>
   <book id ="b002"  year="1992">
      <title>Advanced Programming in the Unix Environment</title>
      <author>
         <last>Stevens</last>
         <first>W.</first>
      </author>
      <publisher pid="p01">Addison-Wesley</publisher>
      <price>59.95</price>
   </book>
   <book id="b003" year="2000">
      <title>Data on the Web</title>
      <author>
         <last>Abiteboul</last>
         <first>Serge</first>
      </author>
      <author>
         <last>Buneman</last>
         <first>Peter</first>
      </author>
      <author>
         <last>Suciu</last>
         <first>Dan</first>
      </author>
      <publisher pid="p02">Morgan Kaufmann</publisher>
      <price>39.95</price>
   </book>
   <journal id="j001" year="1998">
      <title>XML</title>
      <editor><last>Date</last><first>C.</first></editor>
      <editor><last>Gerbarg</last><first>M.</first></editor>
      <publisher pid="p02">Morgan Kaufmann</publisher>
   </journal>
</bib>
```

Appendix II. Sample DTD for three XML documents

Three XML documents *sailors.xml*, *boats.xml* and *reservations.xml* use the following Document Type Definition (file name: *sample.dtd*):

```
<!DOCTYPE sailors [
  <!ELEMENT sailors (sailor*)>
  <!ELEMENT sailor (sname, gender, age)>
  <!ATTLIST sailor sid ID #REQUIRED>
  <!ELEMENT sname (#PCDATA)>
  <!ELEMENT gender (#PCDATA)>
  <!ELEMENT age (#PCDATA)>
]>


<!DOCTYPE boats [
  <!ELEMENT boats (boat*)>
  <!ELEMENT boat (bname, model, year)>
  <!ATTLIST boat bid ID #REQUIRED>
  <!ELEMENT bname (#PCDATA)>
  <!ELEMENT model (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
]>


<!DOCTYPE reservations [
  <!ELEMENT reservations (reservation*)>
  <!ELEMENT reservation (start-time, end-time)>
  <!ATTLIST reservation sid CDATA #REQUIRED>
  <!ATTLIST reservation bid CDATA #REQUIRED>
  <!ELEMENT start-time (#PCDATA)>
  <!ELEMENT end-time (#PCDATA)>
]>
```

Note: In *sailors.xml*, there is a line <!DOCTYPE sailors SYSTEM “sample.dtd”> at the beginning; in *boats.xml*, there is a line <!DOCTYPE boats SYSTEM “sample.dtd”> at the beginning; and in *reservations.xml*, there is a line <!DOCTYPE reservations SYSTEM “sample.dtd”> at the beginning.