View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

XTree and XTreeQuery for Declarative XML Querying

CHEN ZHUO
(B.Comp (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2003

Acknowledgments

First and foremost, I would like to thank my supervisor, Professor Ling Tok Wang, for his invaluable guidance and advice throughout my whole research work. Throughout the project, he has been guiding me very carefully not only on how to solve problems, but also on how to solve them efficiently and completely. I would like to thank him also for his kindness, patience, and his ingenuity in solving the problems. His priceless remarks, suggestions and supports always encouraged me to strive for good work.

Also I would sincerely appreciate my lab fellows, Chen Yabin, Ni Wei, He Qi and Li Changqing for their generous suggestions and help in my research, and for the pleasant and friendly environment of the database research lab.

Table of contents

Acknowledgments ………………………………………………………………….. i

Table of contents ………………………………………………………………… ii

List of figures ……………………………………………………………………. v

List of tables …………………………………………………………………….... vi

Summary ………………………………………………………………………… vii

Chapter 1. Introduction ………………………………………………………… 1

Chapter 2. Preliminaries ……………………………………………………….... 4

2.1 XPath …………………………………………………………………….... 5

2.2 XQuery …………………………………………………………………… 8

2.3 Limitations of XPath and XQuery ……………………………………….. 12

2.3.1 Limitations of XPath ………………………………………………….. 12

2.3.2 Limitations of XQuery ………………………………………………... 15

2.4 Other declarative XML query languages ………………………………… 24

2.5 Modeling XML documents as in databases ……………………………… 33

2.5.1 XML query data model ……………………………………………….. 34

2.5.2 Complex object data model …………………………………………… 35

2.5 Summary ………………………………………………………………….. 38

Chapter 3. XTree ………………………………………………………………... 40

3.1 XTree syntax ………………………………………………………………... 41

3.2 XTree for querying ………………………………………………………….. 42

3.2.1 Binding variables on URLs …………………………………………… 43

3.2.2 Binding variables on XML data ………………………………………. 44

3.2.3 List-valued variables and OO functions ………………………………. 47

3.2.4 Conditions …………………………………………………………….. 51

3.3 XTree for result construction ……………………………………………….. 52

3.4 Summary ……………………………………………………………………. 54

Chapter 4XTreeQuery ……………………………………………………….. 56

4.1 Basic syntax of XTreeQuery ………………………………………………... 57

4.2 Join …………………………………………………………………………….. 58

4.3 Negation ……………………………………………………………………….. 60

4.4 Group by ………………………………………………………………………. 62

4.5 Recursion ……………………………………………………………………… 65

4.6 Quantification ………………………………………………………………….. 67

4.7 Special queries …………………………………………………………………. 68

4.7.1 URL-related querying ………………………………………………… 68

4.7.2 Structure level querying ………………………………………………. 69

4.7.3 Sample querying ……………………………………………………… 70

4.7.4 Top-k querying ………………………………………………………... 70

4.8 Updates ……………………………………………………………............ 71

4.9 Comparison of related works ……………………………………………… 73

4.10 Summary ……………………………………………………….......... 76

Chapter 5. Algorithms to transform XTreeQuery to XQuery ………………….. 78

5.1 Transformation algorithm for querying part ………………………………... 81

5.2 Transformation algorithm for result construction part ……………………… 85

5.3 An example of our algorithm ……………………………………………….. 89

5.4 Summary …………………………………………………………………… 92

Chapter 6. Conclusion and future works ………………………………………… 94

6.1 Conclusion ……………………………………………………………….... 94

6.2 Future works …………………………………………………….......... 96

Publication list ……………………………………………………………………... 97

References ……………………………………………………………………….. 98

Appendix I. Sample XML document of bibliography data ……………………... 102

Appendix II. Sample DTD for three XML documents ………………………….. 103

Appendix IIIA. Sample XML document of employee list ……………………… 104

Appendix IIIB. Result of recursive query: employee tree ……………………… 104

Appendix IV. Sample XML document of people ………………………………... 105

Appendix V. Formal description of XTree/XTreeQuery syntax ……………….. 106

List of figures

Figure 1a. Wrong XQuery script of sample query ………………………………………. 13

Figure 1b. Correct XQuery script of sample query ……………………………………… 13

Figure 2. DTD for employees.xml ………………………………………………………… 17

Figure 3. XML Query Data Model Representation ……………………………………... 35

Figure 4. Algorithm to transform an XTree expression in querying part ……………. 82

Figure 5. Algorithm to transform an XTree expression in result construction part …. 85

Figure 6. Function *translate($expr)* ……………………………………………………... 87

Figure 7a. Sample XTree graph for querying part ……………………………………... 90

Figure 7b. Sample XTree graph for result construction part …………………………... 90

Figure 8. Result XQuery of Example 5.4 ………………………………………………. 92

List of tables

Table 1. Sample XPath expressions 5

Table 2. Sample XPath expressions with conditions 6

Table 3. Variable binding expressions in querying part 44

Table 4. Comparison between XML query languages 75

Summary

XML is becoming prevalent in data representation and data exchange on the Internet. How to query XML documents to extract and restructure information is an important issue in XML research. Currently, XQuery based on XPath is the most promising standard from the W3C. However, XPath and XQuery do have some limitations: XPath is only a linear path, which is not like the XML's tree structure; in XQuery, we can only assign one variable for each XPath expression, which is inefficient to use, and difficult to reveal the relationship among correlated XPaths; XQuery handles group-by operation and recursive querying in some non-normative ways, by nested querying and defining recursive functions respectively, which are inefficient in practice. In addition, Current version of XQuery can only query XML documents, but cannot do update operation on XML documents.

In this thesis, we will propose a new set of syntax rules called XTree, which is a generalization of XPath. XTree uses a complex object data model that models XML documents as in databases. It has a tree structure, which is similar to the structure of XML documents, so that a user can bind multiple variables in one XTree expression. XTree explicitly identifies list-valued variables, uniquely determines their values, and defines some natural built-in functions to manipulate them in an object-oriented way. XTree

expressions can be used not only in querying part, but also in result construction part, to define result format.

We also develop a query language called XTreeQuery, which is based on XTree expressions. By using XTree expressions on both querying part and result construction part, XTreeQuery can avoid nested querying structure, and make the whole query easy to read and comprehend. XTreeQuery can effectively handle join operation, group-by operation and recursive querying in a direct way, and can express some special kinds of queries, such as URL-related queries, structure level queries, sample queries and top-k queries; all these queries cannot be expressed by XQuery, or cannot be expressed efficiently by XQuery. With these features, we believe that XTreeQuery is much more compact and efficient, and is more convenient to write and understand than XQuery. In addition, we design XTreeQuery to be the analog of SQL in XML data. XTreeQuery is not only a data query language, but also a data management language: it can specify updates on XML documents. These features dramatically widen the usage of XTreeQuery.

To be compatible with current XQuery parsers, we have also designed algorithms that convert some simple XTreeQuery queries based on XTree expressions to standard XQuery queries. There are two algorithms, the first one is to transform an XTree expression in querying part to a set of XPath expressions, and the second one is to transform an XTree expression in result construction part to some nested XQuery expressions.

Chapter 1

Introduction

XML is fast emerging as the dominant standard for data representation and exchange in the web. How to query XML documents is an important issue in XML research and development. Various query languages have been proposed in the past few years, such as XPath[32], XQuery[33], Lorel[1], XML-GL[4], XQL[29], XML-QL[14], XSLT[39], YATL[8], XDuce[17], a rule-based semantic query language[7], a declarative XML querying in [23]. Bonifati and Ceri [2] gave a comparative analysis of some of these query languages. Some of these query languages 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[36], and XQuery has been selected as the basis for an official W3C query language for XML.

Most of the existing XML query languages are based on SQL and OQL[3]. However, unlike querying on relational databases whose results are always flat relations, the result of queries on XML documents are complex, and need to be formatted explicitly. Thus, XML queries must 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 constructing respectively. However, the *construct* clause can contain nested *where-construct* clauses so that querying and result-constructing are intermixed. For XQuery, it uses five constructs: *for*, *let*, *where*, *order by* and *return*, i.e., FLWOR expressions. 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.

In this thesis, we will analyze some limitations of XPath, and propose a new set of syntax rules called XTree, which is a generalization of XPath, and show how it can efficiently replace the notations of XPath. XTree has a tree structure, which is similar to the structure of an XML document, so that a user can bind multiple variables in one XTree expression. It explicitly identifies list-valued variables, uniquely determines their values, and defines some natural built-in functions to manipulate them. It supports the binding of variables on the URL or part of the URL so that it can also be queried. It can also be used for the result constructing part of a query, to make that part easy to read and comprehend. We will also define a querying language called XTreeQuery based on XTree expressions, which is more compact in the query structure, and has more expressive power than current XQuery. To be compatible with current XQuery parsers, we also give algorithms to convert XTreeQuery queries to standard XQuery queries.

The rest of this thesis is organized as follows. Chapter 2 gives some literature surveys on existing declarative XML query languages, with emphasis on XPath and XQuery, which are the most promising standard of W3C. It also discusses the limitations of XPath and XQuery, and briefly introduces a complex object data model for XML data. Chapter 3 introduces the XTree syntax by giving some examples, and shows its advantages over XPath. Chapter 4 introduces the XTreeQuery which is based on XTree with some

examples, and shows that it can express many kinds of queries easily and efficiently. A comparison is made among our XTreeQuery and some other declarative XML query languages. Chapter 5 presents two algorithms to transform simple XTreeQuery queries to standard XQuery queries. Finally, Chapter 6 summarizes this thesis and points out future research directions.

Chapter 2

Preliminaries

During the development of XML, researchers have proposed many declarative query languages to extract data from XML documents. W3C has selected XQuery based on XPath as basis for an official standard of XML query language. In this chapter, we will give a background introduction of XPath and XQuery in Section 2.1 and Section 2.2 respectively, and discuss their limitations in Section 2.3. A literature survey on some other declarative XML querying languages will be given in Section 2.4, and a complex object data model for modeling XML documents will be introduced in Section 2.5. Finally, in Section 2.6 we summarize the existing declarative XML query languages.

Basically there are two classes of XML query languages: graphical query languages and declarative query languages. The former includes XML-GL[4], Equix[9], GLASS[28], BBQ[27] etc, and the latter includes Lorel[1], XQL[29], XML-QL[14], XQuery[33], Quilt[6], YATL[8], a rule-based semantic query language[7], a declarative XML querying language [23], etc.

The graphical query languages have been researched ever since the first application of QBE[13] in 1970s. They are thought to be intuitive and user-friendly compared to

traditional textual languages. Also, for querying of XML document, the tree-like structure of XML data can be naturally represented as a graph.

Comparing to declarative query languages, these graphical query languages are more natural and easier to understand. However, it is very difficult to express complex queries with many nesting levels in graphical query language. The query graph cannot be drawn too big and too complex; otherwise users will get confused about the meaning of the query graph. Also, sometimes it is much simpler for use to declare what he/she wants to query, than to express this request in the graph representation. Thus, for more complex queries, declarative query languages will be advantageous over graphical query languages.

This thesis will focus on declarative XML querying languages.

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, Table 1 gives some examples of XPath expressions, according to the XML document in Appendix I.

Table 1. Sample XPath expressions

XPath expression	Description
/bib/book	get each “book” element of the root element “bib”
/bib/book/@year	get attribute “*year*” of each book.
/bib/book/author	get element “*author*” of each book.
//author	get all elements named “*author*”, even if they have different absolute paths. Here “//” means any number of levels down.
*/bib/book/**	get all sub-elements of each book.
*/bib/book/@**	get all attributes of each book.
/bib/book[1]	get the first “book” element.
/bib/book[last()]	get the last “book” element.

XPath uses a pattern expression to identify nodes in an XML document. An XPath pattern is a slash-separated list of child element (or attribute at the last position) names that describe a path through the 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*. 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 (*) can select all elements located by proceeding path. An index number enclosed in a square bracket in an XPath expression can further specify an element: A number in the brackets gives the position of the element in the selected set; the function last() selects the last element in the selection. Attributes are specified by @ prefix.

In an XPath expression, brackets can also be used to specify selection conditions (some people view this feature as the abbreviated format of XQuery). Table 2 gives some examples of XPath expressions with conditions, according to the XML document in Appendix I.

Table 2. Sample XPath expressions with conditions

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

Since the brackets can be used to enclose both an index and a selection condition, in the XPath expressions where these two usages are mixed, they will have same preference, and they will be executed in a left-to-right order. For example, the path expression

/bib/book[2][starts-with(title, "Data")]

means to select the second book and its title must start with "Data" (if the title of the second book is not "Data", then it will return an empty result), and

/bib/book[starts-with(title, "Data")][2]

means to select the second book whose title starts with "Data".

XPath also supports some axes with the syntax *axisname::nodetest[predicate]*, where an axis defines a node-set relative to the current node, and the node test is used to identify a node within an axis. Such axes including *ancestor*, *ancestor-or-self*, *child*, *descendent*, *descendent-or-self*, *following*, *following-sibling*, *parent*, *preceding*, *preceding-sibling, self*, etc. For example,

*/bib/book/title/following-sibling::**

returns the sub-elements "author", "publisher" and "price" of each book;

/bib/book/author/ancestor::[@year]*

returns all ancestors of some "author" element, which have an attribute "year" (actually just all the book elements and journal elements).

However, these axes notations either can be abbreviated to normal XPath expression (*child* can be abbreviated to "/", *self* can be abbreviated to ".", *parent* can be abbreviated to "..", *descendent-or-self* can be abbreviated to "//", etc), or they only concern the structure of the document, not the semantics (such as *following, following-sibling, preceding, preceding-sibling*, etc), so these axes are rarely used in real queries (they never appear in the query examples in [37]).

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[14], YATL[8], Lorel[1].

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, which generalize the "Select-From-Where-Order by" statements in SQL. *For* clause and *let* clause bind values to variables: *for* clause (syntax: "*For $var in xpath-expression*") iterates the variable over the result of the XPath expression, whereas *let* clause (syntax: "*Let $var := xpath-expression*") binds the variables to the whole result of the XPath expression as a list. *Where* clause filters these bindings by some conditions, the *order-by* clause orders the surviving bindings based on some item, and the *return* clause defines the result format, and constructs the result based on the evaluation of the variable bindings. XQuery provides facilities to build output XML fragments with arbitrarily complex structures. There are two types of constructors: direct (using an XML notation) and computed (using a notation with braces { } to indicate enclosed expressions, which force the inner code to be evaluated and replaced by their value, instead of being treated as literal text), for element and attribute construction.

Example 2.1. List the title and year of all books.

```
<bib>
{
  for $book in /bib/book
  return <book>
           { $book/@year, $book/title }
         </book>
}
</bib>
```

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

Example 2.2. Get the title, year and number of authors of all books published before 2000, and sort in publishing year.

```
<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 built-in aggregate function *count* returns the number of items in the list.

In *return* clauses of XQuery, we can use conditional expression "*if ... then ... else ...*" to define conditional treatments for different cases.

Example 2.3. List the year, title and price of each book. If the price is less than 50, just put an empty element <lowprice/> instead of its concrete price value.

```
<bib>
{
   for $book in /bib/book
   return <book>
             { $book/@year, $book/title,
               if ($book/price < 50)
               then <lowprice/>
               else $book/price
             }
            </book>
}
</bib>
```

The result of the above query is as follows:

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix Environment</title>
    <price>59.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <lowprice/>
  </book>
</bib>
```

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

Example 2.4. Find the books which 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 2.5. Find 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
```

XQuery is a functional language, besides the extensive collection of built-in functions (such as *doc* for specifying target XML file, *concat, compare, contains, starts-with, ends-with, string-length, substring, normalize-space* for string management; *matches, replace, tokenize* for pattern matching; *ceiling, floor, round* for number operation; *count, avg, max, min, sum* for aggregate operation; *name, local-name, string* to get element/attribute name and value; *position, last* to get the position/last position of the context node. The detailed XQuery built-in functions can be found at [35]), XQuery also allows a user to create reusable function expressions in their queries, in the following syntax:

```
declare function function_name(parameters) as return_type
```

Example 2.6. Get the year, title and number of authors of all the books.

```
declare function summarize($book as element()) as element()
{
   let $authors := $book/author
   return <book>
            { $book/@year, $book/title }
            <authors> { count($authors) } </authors>
            </book>
}

<bib>
{
   for $book in /bib/book
   return summarize($book)
}
</bib>
```

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

Example 2.7. For each book that has at least one author, list its title and first two authors, 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>
```

By executing the above query, the third book (which has 3 authors) will be outputted as follows:

```
<book>
   <title>Data on the Web</title>
   <author><last>Abiteboul</last><first>Serge</first></author>
   <author><last>Buneman</last><first>Peter</first></author>
   <et-al/>
</book>
```

2.3 Limitations of XPath and XQuery

From the above examples, we can see that XPath can clearly define a unique path in the XML tree; and XQuery can effectively express queries on XML documents, based on XPath expressions. However, they have some limitations. In this section we will discuss some limitations of XPath and XQuery.

2.3.1 Limitations of XPath

In this sub-section, we will introduce the limitations of XPath. XPath has four main limitations as follows:

(1) One path one variable

Firstly, in an XPath expression, although the condition can be a branch, there is still only one linear path. Thus, we can only assign one variable for each XPath expression, which is inefficient. If a query needs several variables from some paths, a user must assign a variable to each path, or use one path when referring to another path (relative path).

Example 2.8. If a user is interested in title, authors and publisher (but not year and price, etc) of each book in the bibliography data in Appendix I, we have to write the following statement:

```
For $b in /bib/book
Let $t := $b/title
    $a := $b/author
    $p := $b/publisher
```

Note that each variable holds a set of nodes defined by an XPath expression, and *$t*, *$a* and *$p* are defined relatively from *$b*.

(2) Unclear relationship among XPaths

Secondly, it is difficult to reveal the relationship among correlated XPaths. For example, suppose there are two XPath expressions: */bib/book/title* and */bib/book/author*, in fact these two paths are correlated, however, the XPath expressions do not show this correlation explicitly, and this may result in some mistakes if the user does not pay attention when writing a query.

Example 2.9. Create a flat list of all the title-author pairs, with each pair enclosed in a “result” element.

The XQuery script in Figure 1a will output a wrong answer, because it does not pay attention to the correlation of XPath expressions. It will produce a Cartesian product of all authors and titles, regardless of whether they are of the same book. Figure 1b gives the correct version of this query.

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

Figure 1a. Wrong query

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

Figure 1b. Correct query

(3) Inefficient for distant conditions

Thirdly, it is inefficient to read a query which returns elements at path *A* while the condition is in a distant path *B*, such query will be very lousy and difficult to comprehend when written in XPath.

Example 2.10. Suppose we want to find the value of publisher id of a book which has an author with last name as "*Stevens*" and first name as "*W.*". The XPath expression is as follows:

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

We can see that it is difficult to distinguish the condition branch from target branch, especially for multiple conditions and nested conditions.

(4) Difficult to split name-value pair structure

Finally, 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. Thus it is difficult to query XML documents with unknown structures, or to rename the elements or attributes in the query result construction.

Example 2.11. Suppose that for each book, we want to list all it 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. Each *$elem* instance is a name-value pair of some sub-element of a *$book* instance, we use *local-name()* function to get the tag name of current *$elem* instance. The above nested query is difficult to read and understand.

In addition, if we want to get the node value only, without the associated node name, then for element node bound to variable *$var*, we have to use *$var/**, *$var/@** and *$var/text()* to get all its possible values; for attribute node bound to variable *$var*, we have to call the function *string($var)* to get its value.

2.3.2 Limitations of XQuery

In this sub-section, we will introduce the limitations of XQuery. XQuery has eight main limitations as follows:

(1) Join operation as sub-query

Firstly, we know that *join* operation are widely used to combine data from multiple sources into one single result; it is a very important type of query. However, XQuery supports join in the following way: it binds a variable on the join field of the first source; then, for each of the other sources, it uses sub-queries with join field in the conditions to get the instances that have the current join field value. This way for join is very unnatural, which is quite different from SQL or QBE (Query By Example, [13]), and the query is difficult to read and comprehend. We illustrate this point by the following example:

Example 2.12. Appendix II is the DTD for three XML files: sailors.xml, boats.xml and reservations.xml, whereas sailors.xml and boats.xml record information of sailors and boats respectively, and reservations.xml records all the reservations of certain boats by

certain sailors, which is a relationship between sailors and boats. Suppose we want to get the sailor name and boat name 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>
```

Note that function *text()* is used to get the text value of an element. In the above XQuery, it is not so easy for a reader to find out what join operations are performed. This "join as sub-queries" style is indirect and unnatural.

(2) Grouping as sub-query

Secondly, many queries involve forming data into groups and applying aggregate functions to each group. However, XQuery does not support grouping operations explicitly, as the *groupby* operator in SQL does. In XQuery, grouping is done by sub-querying structure, which is difficult to read and understand; and is very inefficient, as it will scan the entire document once for each value of the grouping field. Moreover, such kind of sub-querying may even get error result when there are two or more grouping fields, due to some invalid empty groups generated.

Example 2.13. For the bibliography document in Appendix I, list 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 XQuery may scan the bibliography document many times, each time for a specific *year* value. This will be very expensive, when there are many different values in the *year* field. However, it is possible that some query optimizer can execute the above query efficiently, and scan the whole document only once, by remembering the books of each "year" group, instead of executing the query as the way it is written.

Example 2.14. Figure 2 shows the DTD for the document employees.xml. Find the average salary of employees, grouping by department and jobtitle.

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

Figure 2. DTD for employees.xml

The XQuery query for this grouping operation is as follows:

```
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. We can add extra code to eliminate groups with no employees. For example, we can use *if…then…else* conditional statement in *return* clause, however, the query will be more lengthy and difficult to read:

```
return {
  if (count($salary) > 0)
  then {
    <type dept={ $dept } job={ $jobtitle }>
      <avgsalary> { avg($salary) } </avgsalary>
    </type>
  }
  else ()
}
```

The essential reason of getting invalid empty groups is due to that 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 XPaths as a tuple. In fact, even if the function *distinct-value()* can accept multiple XPaths as its arguments and remove the duplicate tuples, we will still have problems when assigning the output of this function to a variable, because in XQuery there is no such type of variable that can bind to tuple values.

(3) Recursion by user-defined recursive function

Thirdly, sometimes it is necessary to scan over a hierarchy of elements recursively, applying some transformation at each level of the hierarchy. XQuery does not support *recursive querying* explicitly; instead, it handles recursions by user-defined recursive functions. This indirect way of expressing recursion will make the query difficult to read. Also we think that the purpose of functions is for general and common computation that would be needed for many times rather than a sideway for other purpose. In our view, this is the drawback of the language design anyway.

Example 2.15. Consider the list of employees in Appendix IIIA. Each employee element has an attribute *name* indicating his/her name, an attribute *id* indicating his/her unique employee number, and an optional attribute *manager* indicating the ID number of his/her manager. 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. Following is the XQuery solution:

```
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 in Appendix IIIB.

(4) Nested querying structure

Fourthly, in practice, XQuery usually has nested querying structure, which is difficult to read and comprehend. The nesting is mainly due to the result formatting purpose in the *return* clause (unlike SQL for relational databases, whose result is always a flat structure, in XQuery the result format must be defined explicitly in the query. For example, if the result of an XQuery query is to be stored as an XML document, then it must have one and only one root element, as required by XML syntax.) Thus the *return* clause is often quite long and lousy, mixing up the plain XML segments, enclosed expressions and even sub-queries, which makes it difficult to read and understand.

Example 2.16. For the documents in Appendix II, list all the sailors in alphabetic order by name, and for each sailor, list the boats that the sailor has a reservation, in alphabetic order by the boat name.

```
<result>
{
  for $s in doc("sailors.xml")/sailors/sailor
  order by $s/sname
  return
    <sailor>
    { $s/@sid, $s/sname }
    {
      for $bid in distinct-values(doc("reservations.xml")/reservations
                                            /reservation[@sid = $s/@sid]/@bid)
      let $bname := doc("boats.xml")/boats/boat[@bid = $bid]/bname/text()
      order by $bname
      return <boatname> { $bname } </boatname>
    }
    </sailor>
}
</result>
```

Example 2.17. Get the titles and publishers of the most expensive books, according to the bibliography document in Appendix I.

```
let $prices := /bib/book/price
let $maxprice : = max($price)
return
  <result>
  {
    for $book in /bib/book[price = $maxprice]
    return
      <expensive_book>
        { $book/title , $book/publisher }
      </expensive_book>
  }
</result>
```

From the above example, we can see that the nested and mixed *return* clause is difficult to read, thus we want to write the *return* clause in a more compact and clearer way.

(5) Built-in functions

Fifthly, in XQuery most built-in functions are used in functional manner, rather than in object-oriented manner (but functions *position()* and *last()* are used in an object-orient

way). Thus we often need to refer to the context node again in the argument of the function, which is not so intuitive. We think it will be more natural to use the functions in an object-oriented fashion, whose meanings are obvious and easy to understand.

Example 2.18. Consider the bibliography in Appendix I, find the books which has more than two authors, and the title contains the word "Data".

```
for $book in /bib/book
let $author : = $book/author
    $title := $book/title/text()
where count($author) > 2 and contains($title, "Data")
return $book
```

Example 2.19. Consider the bibliography in Appendix I. Find 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.

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

In the above example, in the *let* clause, the two "**.**" in the condition refer to *$book/**, but this is difficult to read, since they are textually far away from their substituting context node.

(6) Special queries

Sixthly, XQuery does not support some special kind of queries, or does not support them efficiently. 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 some documents, thus a user cannot write an XQuery to query some unknown URL or URL components.

As the last limitation of XPath we discussed before, since variables are bound to the name-value pairs of some XML nodes, and special built-in functions are needed to split them, thus XQuery is very inefficient to handle queries over XML documents with unknown structures, or to rename elements/attributes in the result construction without knowing their inner structure.

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

Example 2.20. Consider the bibliography in Appendix I. Suppose we do not know the sub-structure of book elements, now we want to restructure books in this way: keep text nodes and sub-elements unchanged, but convert attributes to be sub-elements in the format of <attribute name="*attributeName*", value="*attributeValue*"/>. We can write XQuery as follows:

```
for $book in /bib/book
let $attrib := $book/@*
return
   <book>
      { $book/text(), $book/* }
      <attribute name={ local-name($attrib) } value={ string($attrib) }/>
   </book>
```

Note that the function *string()* is used to get the content value of some XML attribute. Without knowing sub-structures of *book* element, we have to consider all its possible children: */text()* for text nodes, */** for all its sub-elements, and */@** for all its attributes.

(7) Negation on paths only

Seventhly, although XQuery supports universal and existential quantification (*every ... in ... satisfies ...*and *some ... in ... satisfies ...*), it 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, but not a sub-tree structure of the XML data. Thus, it is difficult to express complex negation (such as a negative sub-tree) or nested negation.

Example 2.21. Consider the bibliography in Appendix I. Suppose we want to get the books which do not have a sub-element named "comments", and do not have an author with last name "Stevens".

```
for $book in /bib/book
where not(exists($book/comments)) and not($book/author/last="Stevens")
return $book
```

Note that the function *exists* will return false if the path indicated by its parameter does not exist. The unary function *not* will reverse the truth value of its parameter, which is normally a condition evaluation on a path.

(8) No update operations

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

2.4 Other declarative XML query languages

Since the introduction of XML as standard for data representation and exchange, comprehensive research has been done in the database community on the development of XML query languages. The closest research topics to the work presented in this thesis are the various declarative query languages proposed for XML documents. Besides XPath and XQuery that we have discussed before, here we will also briefly introduce some other declarative XML query languages, such as Lorel[1], XQL[29], XML-QL[14], Quilt[6], XDuce[17], a rule-based semantic query language[7], a declarative XML querying language[23]. Note that XQuery is developed from XQL, XML-QL and Quilt.

Lorel

Lorel[1] was part of *Lore* project (for *Lightweight Object Repository*) in Stanford University, which aims to provide convenient and efficient storage, querying and updating for semi-structured data. Lorel was initially designed to query semi-structured data, and was later migrated to support XML. It uses *OEM* (for *Object Exchange Model*) data model and has the SQL/OQL style, which is a *select-from-where* format.

The most important features of Lorel includes the extensive use of coercion to relieve the user from the strict typing of OQL, and the powerful path expressions that permit a flexible form of declarative navigational access. In addition, Lorel also supports declarative update on the database. We will give several examples to illustrate Lorel language.

Example 2.22. Suppose some restaurants information has been stored in Lore system. Find the names and zipcodes of all "cheap" restaurants. Here the zipcode may be a part of

the address, or may instead be a direct subobject of the restaurant. Also, we do not know if the string “cheap” will be part of a category, price, description, or other subobject of restaurant. We are still able to ask the query in Lorel as follows:

```
select Guide.restaurant.name,
       Guide.restaurant(.address)?.zipcode
where Guide.restaurant.% grep "cheap"
```

In the above query, the *?* after *.address* means that the address is optional in the path expression; the wildcard *%* will match any subobject of the restaurant; and the comparison operator *grep* will return true if the string “cheap” appears anywhere in that subobject’s value.

Example 2.23. Find the names of restaurants whose category is “gourmet”.

```
select N
from Guide.restaurant R, R.name N
where R.category = "gourmet"
```

The above query is very similar with SQL/OQL queries. Object variables *R* and *N* will bind to restaurants and their names respectively.

Lorel can also express the update of XML data, by *update-from-where* statement.

Example 2.24. Add the restaurant’s city as a direct subobject of the restaurant object whenever the city is Palo Alto or Menlo Park.

```
update R.city += C
from Guide.restaurant R, R.address.city C
where C = "Palo Alto" or C = "Menlo Park"
```

Although Lorel was one of the earliest proposed query languages, it is very powerful and easy to use, even comparing to latest query languages. Its flexible and powerful path expression and type coercion can express complicated queries; and the SQL/OQL syntax style is very clear and simple to use. However, to make queries on XML documents using

Lorel, the documents must be mapped and stored in Lore database management system, which limits the usage of Lorel query language. In addition, the returned result is not in XML format, but some flat structure as in SQL/OQL.

XQL

XQL (XML Query Language) [29] is a notation for addressing and filtering the elements and text of XML documents. XQL is an extension to the XSL pattern syntax; it provides a concise, compact and understandable notation for pointing to specific elements and for searching for nodes with particular characteristics.

XQL is designed specifically for XML documents. It is a general purpose query language; it builds upon the XSL capabilities of identifying classes of nodes, by adding Boolean logic, filters, indexing into collections of nodes, etc.

The basic of XQL querying syntax mimics the directory navigation syntax, and the navigation is through the elements in the XML tree. This is quite like XPath.

Example 2.25. Suppose we want to find books which do not have any author with last name “Bob”. We can write XQL query as follows:

/bib/book[all author/last != “Bob”]

Note that brackets *[ ]* are used to enclose condition expressions, and keyword *all* means universal quantification.

For some semi-joins, XQL also supports nested queries.

Example 2.26. Find all books whose year information is the same as the year of the book “Data on the Web”.

/bib/book[@year = /bib/book[title=“Data on the Web”]/@year]

XQL is very simple and compact; it has similar path expression as XPath. However, XQL does not indicate the format of the output, but rather the logical returns. The result of an XQL query could be a node, a list of nodes, an XML document, an array, or some other structure. XQL can only query one XML document; it does not support joins over several XML documents. It even cannot define variable bindings for the use in complex queries. Thus, the expressive power of XQL is quite limited.

XML-QL

XML-QL [14] was designed at AT&T Labs in 1998. It can express queries which extract pieces of data from XML documents, as well as transformations, which, for example, can map XML data between DTDs and can integrate XML data from different sources. The data model for XML-QL is a variation of the semi-structured data model.

XML-QL has a *where-construct* construct similar to *select-where* in SQL, making it easy for users familiar with SQL. It uses XML like tags to specify constraints or conditions. XML-QL supports nested queries: a *construct* clause can contain other *where-construct* clauses.

Example 2.27. Get the titles of books published by “Addison-Wesley”. We can write XML-QL query as follows:

```
where <bib><book>
            <title> $t </>
            <publisher>Addison-Wesley</>
      </></> in “bib.xml”
construct <result>
              <title> $t </>
            </>
```

In the above query, for convenience, we use </> to denote the abbreviation of the corresponding end tags.

Example 2.28. For each book published by "Addison-Wesley", get its title and authors.

```
where <bib><book> $b </></> in "bib.xml",
        <title> $t </>, <publisher>Addison-Wesley</> in $b
construct <result>
              <title> $t </>
              where <author> $a </> in $b
              construct <author> $a </>
            </>
```

XML-QL has some other important features: it can express joins by using same variable name in different places in the querying part, which is similar to QBE (Query by Example) style; it supports querying of element tags using tag variables; it supports regular expression on element tags, to traverse arbitrary paths through XML elements; it can also use object identifies and Skolem functions to control how result is produced and grouped in the data integration.

However, the patterns that are used in XML-QL for binding variables tend to be unnecessarily verbose. More important, the result of the *where* clause is a relation composed of scalar values, which is not sufficient to preserve all the information about the hierarchic and sequential relationships among the elements of the original document. As a result, XML-QL is weak at expressing queries based on hierarchy and sequence, such as "Find the second author of the third book."

Quilt

Quilt[6] is the integration of some previous XML query languages such as XML-QL, XQL, XPath and XSQL. It adapts features from those languages and assembles them to form a new powerful XML querying language.

Although Quilt borrows from many languages (e.g., it borrows path expressions from XQL and XPath, and notions of variable bindings from XML-QL), its conceptual integrity comes from a deep reliance on the structure of XML, which is based on hierarchy, sequence, and reference. Quilt is able to express queries based on document structure and to produce query results that either preserve the original structure or generate a new structure. Quilt can operate on a broad range of data sources, ranging from documents to relational databases.

Quit has very similar syntax as XQuery (actually XQuery is derived from Quilt), it has *for*, *where* and *return* constructs. *For* clauses bind variables on some XPath expressions, *where* clauses specify filters and *return* clauses define output format using the values of the bound variables. The Quilt queries can be nested; a *return* clause can contain another *for-where-return* statement.

Example 2.29. For each book after year 1993, get its title and authors information, and order the authors alphabetically. We can write the Quilt query as follows:

```
for $b in /bib/book
where $b/@year > 1993
return <book>
          <title> $b/title </title>
          {
            for $a in $b/author
            return $a sortby(.)
          }
         </book>
```

Note that the above query is nested, and *sortby(.)* indicates that the generated elements (authors) are to be ordered by their contents.

XDuce

XDuce[17] is a statically typed programming language that is specifically designed for processing XML data. It defines regular expression types, and designs a corresponding mechanism for regular expression patter matching. Regular expression types are a natural generalization of DTDs, describing the structures in XML documents using regular expression operators. Regular expression pattern matching is similar to ML pattern matching except that regular expression types can be embedded in patters. XDuce also provides a powerful notion of subtyping, which not only gives substantial flexibility in programming, but also is useful for schema evolution or integration.

The main part of an XDuce program is a series of function definitions, which define the input patterns (for matching) as well as the output formats.

Example 2.30. Suppose we have an address book which contains a list of triplets (name, address and optional telephone information), we want to convert it into a telephone list, by deleting the address information. We can write the XDuce program as follows:

```
type Addrbook = addrbook[(Name, Addr, Tel?)*]
type Name = name[String]
type Addr = addr[String]
type Tel = tel[String]
fun mkTelList : (Name, Addr, Tel?)* → (Name, Tel)* =
        name[n:String], addr[a:String], tel[t:String], rest:(Name, Addr, Tel?)*
            → name[n], tel[t], mkTelList(rest)
      | name[n:String], addr[a:String], rest:(Name, Addr, Tel?)*
            → mkTelList(rest)
      | ( )
            → ( )
```

In above program, the function *mkTelList* takes a value of type *(Name, Addr, Tel?)** and returns a value of type *(Name, Tel)**. The body is a pattern match consiting of three cases: the first case matches when the first tuple of input sequence has a *tel* label; the

second case matches when the first tuple of input sequence does not have a *tel* label; and the third case matches the empty input sequence, which means the termination of this function call.

XDuce is more oriented to a programming language, instead of a querying language. Its static typing mechanism can ensure that the programs never yield run-time type errors and the result always conform to specified types; and it supports very complicated transformation, it can even apply dynamic programming algorithms in its program.

However, we need to define all the types before we can use XDuce (like *Name*, *Addr*, *Tel* in the above example), this may be not convenient if a user just want to do a simple query. Also, some pattern matching may be exhaustive and ambiguous, which requires some special treatments to handle. In addition, XDuce can only bind variables on the fixed nodes; it cannot bind variables on the unknown nodes, since it does not know their types.

A rule-based semantic query language

Chippimolchai *et. al* [7] proposed a rule-based semantic query language for XML databases. It employs XML Declarative Description (XDD) to model the databases, and can be used to retrieve the semantics of XML data based on semantic information as well as syntactic information.

Example 2.31. Find employees whose salary is more than 2000, assuming that a faculty is also an employee. We can write two querying rules as follows:

```
<Employee xsi:type="TFaculty" $P:attrs>
  $E:subElements
</Employee>    ←    <Faculty $P:attrs>
                       <$E:subElements>
                    </Faculty>
```

```
<answer>
   <Employee $P:attrs>
      $E:otherElements1
      <Salary>$S:salary</Salary>
      $E:otherElements2
   </Employee>
</answer> ← <Employee $P:attrs>
               $E:otherElements1
               <Salary>$S:salary</Salary>
               $E:otherElements2
            </Employee>
            GreaterThan(<Value>$S:salary</Value>,<Value>2000</Value>)
```

In the above query, the first rule specifies that a faculty is also an employee. The second rule queries employees whose salaries are above 2000. Thus if a faculty has a salary more than 2000, he/she will be included in the answer. Note that variables are typed: a variable prefixed with *P:* (e.g., *$P:*attrs in the example) means a sequence of attribute-value pairs; a variable prefixed with *E:* (e.g., *$E:subElements*) means a sequence of XML expressions; and a variable prefix with *S:* (e.g., *$S:salary*) means some strings.

The semantic query language has some advantages: it is efficient for dynamically changed documents and frequently used queries, since changes on the schema do not require re-writing of the whole queries. However, the variables of this query language are *typed*, so it is not flexible to assign variables on various XML structures; and in the query rule, the header and body (corresponding to result construction part and querying part respectively) are written in XML format directly, which makes the rule long and difficult to read. In addition, it is not clear how to form nested querying structure for complicated queries.

A declarative XML query language

Liu and Ling [23] proposed a rule-based declarative XML querying language, based on the data model that views XML structure as complex objects. It can bind multiple variables in one query statement, and naturally separates querying and result construction parts, to make the queries compact and easy to read.

Example 2.32. List the books published by Addison-Wesley after 1993.

querying /bib/book Þ $b (publisher Þ $p, @year Þ $y)
$p = "Addison-Wesley", $y > 1993
constructing /results/book Þ $b

Note that in the above query, variables *$b*, *$p* and *$y* are bound to *book* element, *publisher* element and *year* attribute respectively, in one expression.

In this query language, several rules can be used for the same query so that complex queries can be expressed in a simple and natural way; also it provides a natural and direct support for recursion as in deductive databases. In addition, it explicitly supports list-valued variables. However, the semantics of list-valued variables are not formally defined, and the syntax for binding list-valued variables is in an unnatural way. To bind *{$books}* on all the *book* elements under *bib*, it writes */bib/{$books(book)}*, which is not so easy to read and comprehend.

2.5 Modeling XML documents as in databases

In this section, we will introduce the complex object data model proposed by Liu and Ling[23], which provides a natural way to model XML document as complex objects. As a result, a user can easily comprehend XML data from database point of view. In this data model, XML data are modeled as complex objects with nested structure. There are five

types of objects: element objects, attribute objects, tuple objects, lexical objects and list objects. We will use this complex object data model for our XTree and XTreeQuery.

2.5.1 XML query data model

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. A language is *closed* with respect to a data model if the value of every expression in the language is guaranteed to be in the data model.

XPath and XQuery model XML data by XML Query Data Model[34], which is a low level data model. The XML Query Data Model is more oriented on document structure, instead of database structure, so it is not very suitable for database operations such as querying and data management. For our XTree and XTreeQuery, we want to view XML data as a complex object by the data model proposed in [23]. With such a view, querying presentation also becomes higher level.

Consider the following simple XML document:

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

This XML document is represented in XML Query Data Model as a tree structure shown in Figure 3, in which *person*, *name*, *first*, *last*, *gender*, *age* are element nodes, *id* is an attribute node, and *p123*, *John*, *Smith*, *Male*, *25* are text nodes.

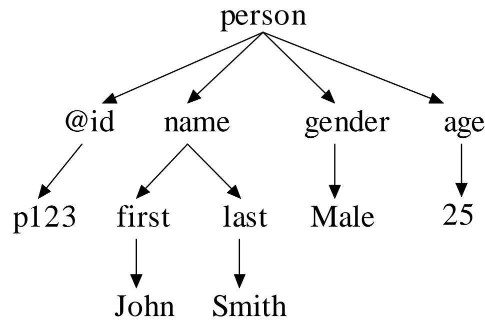

Figure 3. XML Query Data Model Representation

From the above tree graph, we only know the hierarchical structure of the source data. The elements, attributes and text are all represented as nodes in the tree. It is easy to identify attributes, since they are prefixed by symbol "@", however, it is difficult to differentiate empty elements from literal text.

2.5.2 Complex object data model

For the XML data in the above example, we can view it as a nested complex object in complex object data model 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 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. (Note that XPath uses square brackets to denote the conditions in paths. However, later in Section 3.2.4 we will show that our XTree does not need to specially use square brackets to denote conditions, instead, a user can write those conditions in a direct way in XTree notations).

By comparing with XML Query Data Model, we have introduced several higher level notations in our complex object data model, such as *element object*, *attribute object*, *tuple object* and *lexical object* that naturally correspond to XML notations.

Besides the above objects, we also need other kinds of objects. Consider the XML document in Appendix IV, it can be represented in our data model as an element object as follows:

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

Note that the attribute *children* (actually IDREFS type in the DTD) has list values such as {o123, o234} and {o456, o567}. Such list values are called *list objects* in our data model, which are enclosed by a pair of braces { } with commas as separators. Also there are several elements with the same element name *person* and similar sub-structures, but different values, especially the *null* value for an empty *person* element. From querying point of view, if we just want one instance of element *person* at a time, then the above representation is enough. However, if we want all the instances of the element *person*, then it is not clear what should be returned from this representation. To solve this problem, the data model also extends the notion *list object* to cover element values and represent the above XML document as follows:

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

Thus, we can treat *person* element as list-valued if we are interested in all of its values. This extension is necessary as it corresponds to the *let* clause in XQuery (In XQuery, a *let* clause will bind a variable to the whole result of the XPath expression as a list, whereas a *for* clause will iterate a variable over the result of the XPath expression).

An XML document over the web has a URL that specifies its location and contains exactly one root element. The URL consists of the protocol name, domain name, directories and the file name. In XQuery, the URL of some XML document is indicated by *doc()* function. In our framework, we treat the URL and the associated element as the first class citizen.

Example 2.33. Consider the bibliography XML document in Appendix I. Suppose it is located at *http://www.abc.com/people/people.xml*, where *http* is the protocol name, *www.abc.com* is the domain name, *people* is the directory name, and *people.xml* is the file name. It contains one *bib* element consisting of three *book* sub-elements and one *journal* sub-element. Its representation in complex data model as an XML object is as follows:

```
(http://www.abc.com/people/people.xml)/
bib → [@name → IT,
        book → [@id → b001, @year → 1994,
                title → ICP/IP illustrated,
                author → [last → Stevens, first → W.],
                publisher → [@pid → p01, name → Addison-Wesley],
                price → 65.95],
        book → [@id → b002, @year → 1992,
                title → Advanced Programming in the Unix environment,
                author → [last → Stevens, first → W.],
                publisher → [@pid → p01, name → Addison-Wesley],
                price → 59.95],
        book → [@id → b003, @year → 2000,
                title → Data on the Web,
                author → [last → Abiteboul, first → Serge],
                author → [last → Buneman, first → Peter],
                author → [last → Suciu, first → Dan],
                publisher → [@pid → p02, name → Morgan Kaufmann],
                price → 39.95],
        journal → [@id → j001, @year → 1998,
                 title → XML,
                 editor → [last → Date, first → C.],
                 editor → [last → Gerbarg, first → M.],
                 publisher → [@pid → p02, Morgan Kaufmann]]]
```

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

2.6 Summary

In this chapter, we gave a brief introduction of XPath and XQuery through examples, and discussed their limitations. Basically, XPath is only a linear path leading to a specific set

of nodes in XML documents, which can only be bound to one variable; and it is difficult to reveal the relationship among correlated XPaths. XQuery often has a nested structure (in the *return* clause) in order to express complex queries, or just for result formatting purpose. In addition, XQuery is quite inefficient to express join, grouping, recursion, these are done by nested sub-queries and user-defined recursive functions. Due to lack of proper functions, XQuery can only output a whole set of nodes which satisfy the query requirement, but cannot just pick up some samples randomly or some top *k* elements; also it is very inefficient to express queries on unknown structures, or to re-format the structure in the result. XQuery also cannot do update on XML documents.

We have also surveyed some other declarative XML querying languages, such as Lorel[1], XQL[29], XML-QL[14], Quilt[6], XDuce[17], a rule-based semantic query language[7], and a declarative XML query language[23]. They are effective in querying data from XML documents, however, except the last one, their querying parts are all based on XPaths or XML-segment patterns, and the result construction parts are usually nested, which are not efficient. None of these declarative query languages have solved the limitations discussed in Section 2.3. Thus, we want to propose a new set of syntax rules which generalizes XPath, and based on it, a new declarative query language XTreeQuery that solves the abovementioned limitations, to make the queries compact and efficient.

The XML Query Data Model (used in XPath and XQuery) is a low data model, which is more oriented on the document structure, instead of database structure. Our XTree and XTreeQuery will adopt the complex object data model proposed by Liu and Ling [23], which models XML data as complex objects with nested structure, thus a user can easily comprehend XML data from database point of view.

Chapter 3

XTree

After discussing the limitations of XPath and XQuery, in this chapter, we will introduce a new set of syntax rules called **XTree**, which is a generalization of XPath and is advantageous over XPath. It has a tree structure which is similar to the structure of XML document, and multiple variables can be bound in one XTree expression in the querying part of a query. XTree can also be used in the result construction part of a query to define the result format efficiently. Besides single-valued variables, XTree supports list-valued variables explicitly, and defines their semantics concisely. Section 3.1 will introduce the basic syntax of XTree. In Section 3.2, we will describe how to use XTree expressions for selection and variable bindings in the querying part of a query, and discuss the explicit indication of list-valued variables in XTree expressions and how to determine their values uniquely. In Section 3.3, we will describe how to use XTree expressions to define result format in the result construction part of a query, to avoid unnecessary nesting. Finally, we summarize our contributions in Section 3.4.

3.1 Basic syntax of XTree

XTree is a new set of syntax rules, which generalizes XPath. It has a tree structure like the structure of XML documents. As in XPath, child node follows parent node via a slash /, and a double-slash // means no matter how many levels down. However, in the XTree expressions, sibling tree nodes are enclosed by a pair of square brackets [ ] and are separated by commas, and [ ] can be nested. For an XML document, only interested sub-trees are written in XTree, not the entire XML tree structure.

XTree treats the URL of an XML document and the associated element as first class citizens, we use parentheses ( ) to enclose the URL at the beginning of XTree expression, in order to indicate the source of the XML document. 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 missing, then the document is from default input. In XPath the URL is specified by the function *doc*, and its parameter must be a known URL.

In XTree expressions, we use logical variables as place holders to bind/match the name-value pairs of XML nodes at their places. Because we need to bind various parts of XML documents to logical variables, in order to be flexible and easy to use in practice, the variables in XTree are *non-typed*. A variable can be used for any location, but when the same variable occurs in different places in the query, it has the same value. There are two kinds of variables: single-valued variables and list-valued variables. *Single-valued variables* start with *$*, such as *$X, $abc*. *List-valued variables* are of the form *{$X}* which is constructed from the single-valued variable *$X* that ranges over all *$X* instances in a list.

Such list-valued variables are common in advanced deductive database languages such as F-Logic[19], ROL[20], RelationLog[21], and ROL2[22].

There are two types of XTree expressions: one used in the querying part of a query, to indicate the interested paths and bind variables on them; and the other used in the result construction part of a query, to define the format of the returned result. Symbol → is used to assign values to variables, it is used in the querying part of a query only; symbol ← is used to get values from variables, it is used in the result construction part of a query only. The use of these two symbols is as follows:

sourcePath → *$var* to assign the value of *sourcePath* to variable *$var*

targetPath ← *$var* to place the value of variable *$var* at the path *targetPath*

Because we can write the result construction part of a query in an XTree expression, we do not need to use curly braces { } to indicate the enclosed expressions or nested query blocks as in XQuery; also we can effectively reduce the nesting level in the query, comparing to XQuery, as most of the nestings are due to result formatting.

Appendix V gives the formal description of the XTree syntax.

3.2 XTree for querying

In the querying part of a query, we can write XTree expressions to bind variables on the URL or part of the URL, to get the URL information of the XML document, or to bind variables on some specific set of nodes. An important advantage of XTree is that a user can bind multiple variables in one XTree expression, whereas only one variable can be bound in one XPath expression.

3.2.1 Binding variables on URLs

As we make URL and its associated XML element first class citizen in our framework, we can also use various variables for URL and URL components. Such variable bindings will not involve symbol →. Consider the following examples:

($URL)/$document

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

Example 3.1. If we want to search the site *http://www.abc.com* to find some XML documents in the directory /dir that contain a book element with 1994 for attribute *year* and "TCP/IP Illustrated" for sub-element *title*, we can write 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, and *.xml* extension restricts the file type to XML documents. Currently XPath and XQuery do not support such kinds of queries.

Example 3.2. Find two distinct URLs that contain the same document.

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

The feature of binding variables on URLs or URL components enable a user to make a query on the location of some document, as what a search engine does. However, such query is not efficient, as we need to search over all the files under some websites, or even in the Internet.

3.2.2 Binding variables on XML data

To bind variables on specific set of nodes, we can use symbol → to assign the value of nodes on the left side to the variable on the right side. If the right side is a single-valued variable, it means to iterate the nodes one by one, as in *for* clause of XQuery; if the right side is a list-valued variable, it means to keep all nodes in the list, as in *let* clause of XQuery. The left side of symbol → can also be a variable, which means to bind the name of the nodes to the variable at the left side. In one XTree expression a user can bind multiple variables, which is easy and efficient to write. Table 3 gives the meaning of the variable bindings in the querying part of a query.

Table 3. Variable binding expressions in querying part

Variable binding	**Description**
elem_name→$var_value	bind the value of elements with name “elem_name” to the variable $var_value.
@attr_name→$var_value	bind the value of attribute with name “attr_name” to the variable $var_value.
**→$var_value*	bind the value of all elements to the variable $var_value.
@→$var_value*	bind the value of all attributes to the variable $var_value.
text()→$var_value	bind the lexical text to the variable $var_value.
$var_name→$var_value	bind the name of every element to the variable $var_name, and the corresponding value to the variable $var_value.
@$var_name→$var_value	bind the name of every attribute to the variable $var_name, and the corresponding value to the variable $var_value.

Here we just use single-valued variables as an illustration, which means to bind each value of the target nodes iteratively. In fact XTree also supports another type of variables, list-valued variables, which bind all the values of the left side target nodes as a list. We will discuss list-valued variables and their semantics in details at Section 3.2.3.

Note that in XPath expressions, a variable is bound to the whole node (element or attribute) structure, as a name-value pair. It is not so easy to split this pair to be name and value separately, especially when we do not know the internal sub-structure of this node

(as in Example 2.11). However, in XTree expressions, the bindings of names and values are clearly separated. Variables at the left side of symbol → will bind to the node names, and variables at the right side of symbol → will bind to the node values.

Example 3.3. Suppose the XML document in Appendix I is located at the URL *www.abc.com/bib*.xml. If we want to get each of the books, we can use the variable *$b* in the following XTree expression:

(http://www.abc.com/bib.xml)/bib/book→$b

The above expression corresponds to the XPath expression */bib/book/(*|@*|text())*. Note that the variable *$b* is bound to the content of *book* elements. In XQuery, we have to know exactly what is there and use proper functions such as *local-name()*, *text()*, *string()*, in order to express query accurately. However, in our framework, we simply use variables to match whatever is there so that query expression becomes much easier for users.

Example 3.4. If we want to get the year of each book, we can use the variable $year in the following XTree expression:

(http://www.abc.com/bib.xml)/bib/book/@year→$year

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

A user can instantiate many variables in one XTree expression, whereas each XPath expression can only instantiate one variable. Thus XTree expressions are very easy and efficient to write, read and comprehend. Consider the following example:

Example 3.5. If we are interested in the *year* and *title* of each book, and its authors' first names and last names, we can use the 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 corresponds to the following six XPath expressions:

let $doc := doc("http://www.abc.com/bib.xml")
for $book in $doc/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 3.6. Suppose we want to get the first name and last name elements at whatever depth in the document, we can write XTree expression as follows:

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

Note that the pair of square brackets [ ] *e*nclosing *last* and *first* specifies that the two elements *last* and *first* are sibling nodes, they share a common parent, which is */bib/book/author* or */bib/journal/editor* for the sample XML document in Appendix I.

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

Example 3.7. If we want to obtain child elements of *bib*, which is not *title*, we can use variables $elem_name and $elem_value in the following XTree expression:

(www.abc.com/bib.xml)/bib/$elem_name→$elem_value
$elem_name ≠ "title"

XPath cannot bind variables on XML node names directly; it can only bind variables on the nodes and use function *local-name()* to get the node names, as in Example 2.11. Thus XTree expression is more convenient to use in this case.

Example 3.8. If we want to obtain some attribute name with value "1992" in some book element, and bind variable $b to this book, we can write the following XTree expression:

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

According to the sample XML 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)

3.2.3 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 our XTree expressions, a list-valued variable is explicitly indicated by a pair of curly braces { } (e.g., *{$books}* for a list of book elements). Also, since we model the XML documents as complex objects, thus unlike XPath and XQuery which use many unintuitive functions (as in functional languages), we define some natural functions that are obvious and easy to understand, and they are used in an object-oriented fashion.

Example 3.9. Suppose we want to obtain all the books in the bibliography document in Appendix I, and bind them to a list-valued variable *{$b}*. We can write the following XTree expression:

/bib/book→{$b}

The value of *{$b}* is a list of three books in the bibliography document. The above XTree expression is equivalent to the XQuery statement

let $b := /bib/book

However, in the above *let* clause, the list-valued variable *$b* looks no different with other single-valued variables defined in *for* clause. In our XTree expressions, we use a pair of curly braces *{ }* to explicitly distinguish list-valued variables.

We have defined some built-in object-oriented functions for list-valued variables:

Example 3.10. Suppose a list-valued variable {$number} binds to a list of numbers, then we can obtain their aggregate values as follows:

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

Example 3.11. Suppose {$authors} bind to a list of authors element, then we have the following built-in operators:

{$authors}[1-3, 6]	gives a sub-list that contains first to third items, and the sixth item
{$authors}.last()	gives 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
$author Î {$authors}	checks whether $author is in the list
{$authors'}Í {$authors}	checks if the first list is a sub-list of the second list
{$authors}.iterate()	returns an item in the list each time, one by one
{$authors}.some()	some item in the list satisfies a certain condition (like existential quantifier)
{$authors}.every()	all items in the list satisfy a certain condition (like universal quantifier)
{$authors}.none()	no item in the 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")* in XQuery, we write the object-oriented version as *$title.contains("XML")*. It is very easy to convert the functions from functional fashion to object-oriented fashion.

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 on the schema of the XML document, from root of the document to the nodes represented by *$a* (or *{$a}*). The *associated path* is a schema path, not an object path; it leads to a set of objects conform to that path, instead of a single XML node.

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

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

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

Example 3.12. 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 next book.

Note that in the above example, for the first expression, the value of *{$a}* is the concatenation of all the authors of all the books (the order of values in *{$a}* is the same as that in the document), which may include duplicated authors (an author may write several books). We can use the function *{$a}.distinct()* to remove duplicates. For the second expression, *{$a}* has an ancestor variable *$b*, thus the value of each *{$a}* instance is related to the corresponding *$b* instance.

List-valued variables can still have children variables. If the child variable is single-valued, that means to iteratively get the child based on each value in the list; if the child variable is list-valued, that means to create a list of children based on all the values in the list.

Example 3.13. For the bibliography document in Appendix I, consider the XTree expressions and their corresponding XQuery statements in the following table:

XTree expression	XQuery statements
/bib/book→$b/author→$a	*for $b in /bib/book, $a in $b/author*
/bib/book→$b/author→{$a}	*for $b in /bib/book* *let $a := $b/author*
/bib/book→{$b}/author→$a	*let $b := /bib/book* *for $a in $b/author*
/bib/book→{$b}/author→{$a}	*let $b := /bib/book* *let $a := $b/author*

The pair of curly braces { } can also be used as a grouping operator. If the enclosed expression of a pair of curly braces is not a list-valued variable defined before, then it will evaluate the enclosed expression, and group all the result instances to a list. This gives an easy and flexible way to generate a list of instances for immediate use, without assigning it to a list-valued variable for later usage. For example, *{bib/book}* will generate a list of all book instances. If we have defined *{$b}* by the XTree expression: */bib/book→{$b}*, then *{$b/author}* will generate a list of authors of book items in the list *$b*.

This grouping operator also supports tuple values in its enclosed expression. If we have defined *$b* as */bib/book→{$b}*, then ***{$b/[title, name]}*** means to generate a list of tuples, and each tuple consists of title and name of a book instance in the list *$b*.

3.2.4 Conditions

Unlike in XPath that 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 to indicate that its enclosed expression is not a part of the target path but the conditions along the path. Otherwise, if we write the condition directly without this special mark, it will be difficult to interpret the XPath expression. However, for our 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 square brackets for indication. Instead, in XTree expressions, square brackets are used to indicate a tuple of sibling nodes, thus to ensure the clear hierarchy of the tree structure. If there is only one node to be considered, the enclosed square brackets can be omitted.

Example 3.14. For the bibliography document in Appendix I, suppose we want to get 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 3.15. If we want to obtain 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, if we want to obtain all the authors who have a book published on 1992, we can write an XTree expression as follows:

/bib/book/[@year="1992", author→{$a}]

3.3 XTree for result construction

XTree expressions not only can be used to bind variables in the querying part, but also can be used to define the result format. We can use symbol ← to get values of variables from right side and assign them to the expressions on the left side. The variables on the right side of symbol ← must be previously defined in the querying part; and the left side of symbol ← cannot be a variable, it must be a concrete element name or attribute name. If the right side is a single-valued variable, we just put its value of current iteration to the left side expression; if the right side is a list-valued variable, we will put all the values in the list to the left side expression. Unlike in XQuery, which often mixes XML plain text and variable values and even sub-queries in the *return* clause, here the result constructing part is just an XTree expression, which is 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 and well-defined, which does not allow any condition checking or uncertainty in the structure.

Example 3.16. Suppose we have bound the variables by the following XTree expression:

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

If we only want to retain 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 to the result (www.xyz.com/books.xml, which 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. Suppose that *$y* and *$t* are defined to hold year and title information of a book in the *for* clauses, the above expression is equivalent to the following XQuery *return* clause:

```
<result>
{
   return <book> { $y, $t }
}
</result>
```

Comparing the XTree expressions between XQuery *return* clauses, we can see that XTree expression for result construction is also more compact, and is easier to read and write. In addition, XQuery *return* clause cannot specify the location to store the result.

Example 3.17. Suppose we have bound the variables by the following XTree expression:

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

If we want to count the number of authors for each book, and only show the title and the first author of each book, we can write the following XTree expression:

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

Note that without specifying a URL at beginning, the result is directed to the standard output. *{$a}[1]* gives the first item in the *{$a}* list, and *{$a}.count()* counts the number of items in the *{$a}* list.

The right side of symbol ← does not have to be some pre-defined variables or invocations of functions on variables. Instead, it can also be some literal text, or even be omitted, which means an empty value. In such case, we just put the literal text directly into the structure of the left side.

Example 3.18. Suppose we want to return a book whose title is "Computer Architecture", and which does not have a specified author. We can write the following XTree expression:

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

The above XTree expression will output the following XML segment:

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

3.4 Summary

In many existing declarative XML querying languages, XPath expressions are used to select querying patterns. Although XPath expression can clearly define a unique path in the XML tree, it has some limitations, such as one path one variable, unclear relationship among paths, inefficient for distant conditions, difficult to split name-value pair structure (as discussed in Section 2.3.1).

In this thesis, we proposed a new set of syntax rules called XTree, which generalizes XPath and is more advantageous and efficient than XPath. XTree has a tree structure,

which is similar to the structure of XML data. For the queries based on XTree expressions, in the querying part, multiple variables can be defined in one XTree expression; in the result construction part, a user can just write one XTree expression to define the result format. The separation of querying part and result construction part effectively avoids nested structure in the query, and makes the whole query easy to read and understand. 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.

Writing XML queries based on XTree expressions is better than on XPaths, this is mainly because of the limited expressive power of XPath: it is only a linear path to target XML nodes, and can only bind one variable for each XPath. Essentially, if we write a query based on XPath expressions, we have to split the whole query tree into several XPaths for representation; each XPath only represents a piece of information of that query. However, to execute the query, the query parser may need to reassemble these XPaths to rebuild the global view of the whole query. Thus, the query based on XPaths is not only lengthy and difficult to read, but also may cause inefficiency in execution. On the other hand, if we write a query based on XTree expressions, each query tree corresponds to an XTree expression. This makes the query script very compact, and the query parser can directly get the global view of the query tree instead of reassembling pieces of information.

Chapter 4

XTreeQuery

After introducing XTree notations, in this chapter we will propose a new query language called **XTreeQuery**, which makes use of XTree expressions. XTreeQuery is more powerful than XQuery, by solving some limitations of XQuery listed in Section 2.3.2. It uses the same complex object data model as XTree, and explicitly supports list-valued variables. XTreeQuery can express join, negation, grouping, recursion and quantification directly and efficiently, can write some special queries such as URL-related querying, structure level querying, sample querying, top-k querying; and supports updates on XML documents. These features are not well supported by current XQuery. Section 4.1 will introduce the basic syntax of XTreeQuery. From Section 4.2 to Section 4.8, we will show how to express join operation, negation, grouping operation, recursion, quantification, some special queries and update operation in XTreeQuery, respectively, through some examples. In Section 4.9, we will make a briefly comparison among our XTreeQuery and some other existing declarative XML query languages. Finally, we will summarize our contributions in Section 4.10.

4.1 Basic syntax of XTreeQuery

The syntax of XTreeQuery is similar to that of XQuery. Unlike XQuery's FLWOR (From-Let-Where-Order by-Return) statements, XTreeQuery has the **QWOC** (**Q**uery-**W**here-**O**rder by-**C**onstruct) statements for querying. *Query* clause contains one or more XTree querying expressions for selection and variables binding, which is similar to the *for* clauses (for the binding of single-valued variables) and *let* clauses (for the bindings of list-valued variables) in XQuery. *Where* clause and *order-by* clause are optional, they are used for specifying constraints and ordering respectively, which are the same as *where* and *order-by* in XQuery. *Construct* clause contains exactly one XTree result construction expression to define the output format; it does not need a nested structure as what often happens in the *return* clause in XQuery, thus makes the result construction part more concise and easier to understand.

Appendix V gives the formal description of the XTreeQuery syntax.

Example 4.1. For the bibliography document in Appendix I, if we want to list the title and authors (but not publisher information) of books published after 2000, we can write the query in XTreeQuery as follows:

> ***query*** */bib/book/[@year→$y, title→$t, author→{$a}]*
> ***where*** *$y ³ 2000*
> ***construct*** */results/newbook/[title←$t, author←{$a}]*

We can see that the querying part is an XTree expression which binds necessary variables, and the result constructing part is another XTree expression which defines the structure of the returned result: under the root *results*, each *newbook* element will store the title and authors information of that book. *Where* clause specifies the condition of the query.

Example 4.2. For each book that has equal or more than three authors, list its title and the first two authors, and use 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*** *{$a}.count() ³ 3*
> ***order by*** *$t*
> ***construct*** */result/book/[title←$t, authors/[author←{$a}[1-2],* ***et-al****]]*

The output of the above query is as follows:

```
<result>
   <book>
      <title>Data on the Web</title>
      <authors>
         <author><last>Abiteboul</last><first>Serge</first></author>
         <author><last>Buneman</last><first>Peter</first></author>
         <et-al/>
   </book>
</result>
```

Note that in the *construct* clause of the above query, *{$a}[1-2]* will return a sub-list of the first two items in *{$a}* (i.e., first two author elements); and 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 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".

4.2 Join

Join operation is one of the most important and widely used operations in database queries. It combines data from multiple database sources into one single result. XQuery

does not support join operations explicitly, instead, it actualizes join operations by nested sub-queries, which is difficult to read and comprehend, as illustrated in Example 2.12.

In our XTreeQuery, we implement join in a more natural way, which is quite similar to SQL and QBE[13]. In the XTree expressions of the *query* clauses, variables of the same name mean a join operation, and the instances of them must have the same value. Now we can re-write Example 2.12 in our XTreeQuery format:

Example 4.3. According to the three XML documents in Appendix II, get the sailor name and boat name for each reservation. We can write the following XTreeQuery.

```
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, two occurrences of variable $sid means a join operation over sailors and reservations, and two occurrences of variable $bid means a join operation over boats and reservations. This natural way to express join is easier for a user to read and write queries over multiple data sources. The XQuery version of this query is as follows:

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

Note that in the above query, the second and third *for* clauses (which defines *$sname* and *$bname* respectively) are actually sub-queries, through the equity selection at attribute *@sid* and *@bid* respectively.

4.3 Negation

Negation is an important feature in database query, especially for deductive databases[20, 22]. However, XQuery can only express negations in the condition clause (*where* clauses), but not the query clause (*for* clause and *let* clause). In addition, because XQuery is based on XPath expressions, thus its negative condition can only be set on some XPaths, but not a sub-tree structure. Thus, in XQuery it is difficult to express complex negation or nested negation.

However, in our XTreeQuery, we define a unary negation operator *not()* in the query clause for negative conditions. The enclosed expression of *not()* is a sub-tree structure, which also conform to the XTree syntax. The negation operator *not()* forces the conditions on the enclosed XTree expression to be evaluated as false, or the enclosed XTree structure does not exist in the document.

As the negation operator in other languages, in XTreeQuery, the expression enclosed by *not()* is by default existentially quantified, and the negation operator *not()* can be nested.

Example 4.4. According to the bibliography document in Appendix I, find the books which do not have an author named “Stevens W.”. We can write the XTreeQuery 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), instead of just a path. It has the existential quantification and forces the conditions on the enclosed sub-tree to be evaluated as false, i.e., for a book *$b*, there is no such an author named “Stevens W.”. It is equivalent to the following XQuery:

```
<results>
  for $b in /bib/book
  where every $a in $b/author satisfies ($a/last≠"Stevens" and $a/first≠"W.")
  return $b
</results>
```

We can see that in the XQuery, for negation, we can only set negative conditions on XPaths (*$a/last* and *$a/first*), and we have to explicitly define the quantification (*where every ... satisfies ...*).

Example 4.5. Find the books which do not have a sub-element named "reference". We can write the XTreeQuery 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 will force that the inner XTree structure does not exist, i.e., the book *$b* does not have a sub-element with the name "reference". It is equivalent to the following XQuery:

```
<results>
  for $b in /bib/book
  where not($b/reference)
  return $b
</results>
```

Example 4.6. Select all the books except those whose title contain word "TCP" and price more than 50. We can write the XTreeQuery as follows:

```
query /bib/book→$b/not(title.contains("TCP") and price>50)
construct /results/book←$b
```

Note that *title* and *price* are sibling nodes under *book*, keyword "and" means both condition must be satisfied. This will select the second and the third books in the bibliography document in Appendix I.

Example 4.7. Find the books whose title does not contain word "TCP", and whose price is not more than 50. We can write the XTreeQuery as follows:

> *query /bib/book→$b/**not**(title.contains("TCP") **or** price>50)*
> *construct /results/book←$b*

Note that *title* and *price* are sibling nodes under *book*, keyword "or" means either condition can be satisfied. This will select only the third book in the bibliography document in Appendix I.

Example 4.8. For the bibliography document in Appendix I, 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 which do not have any author who does not have an address in New York, i.e., find the books with all the authors having 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. It is equivalent to the following XQuery:

```
<results>
  for $b in /bib/book
  where not (some $a in $b/author satisfies
             not (some $addr in $a/address satisfies $addr/city="New York"))
  return $b
</results>
```

4.4 Group by

Grouping operation is often used in database queries to form data into groups and apply aggregate functions to each group. Currently XQuery does not support grouping directly, as the *groupby* operator used in SQL. Instead, XQuery implements grouping by sub-

queries, which is not only difficult to read and comprehend, but also inefficient to execute, as illustrated by Example 2.13. Moreover, such kind of sub-querying may even get error results when grouping by multiple fields, due to invalid empty groups generated by nested *for* clauses, as illustrated by Example 2.14.

However, our XTreeQuery supports grouping in an explicit way. It has the keyword *groupby* for grouping operations. *Groupby* will classify the XML nodes into groups, based on the grouping fields, and assign the result to a two-dimensional list-valued variable. Unlike in SQL, where all data are in flat format so the grouping result is clear and easy to use, XML data has more complicated structures, and thus we have to put the grouping result into some variables for future use. The variable that holds the grouping result has a two-dimensional structure: it is a list of list, each inner list is the items of a particular group. The syntax of *groupby* is as follows:

{{$result}}* ← *{$a} ***groupby*** *$b*

The above statement means to group items in *{$a}* based on the values of *$b*, and assign the result to the two-dimensional list *{{$result}}*. *{{$result}}* has a two dimensional structure: it is a list of list. 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 field, then the function *key()* will return the tuple value of the grouping fields for the current group, with the first item *key()[1]* be the value of the first grouping field, the second item *key()[2]* be the value of the second grouping field, and so on. By default, duplicate values will be eliminated in the grouping field; otherwise we cannot distinguish two groups with same value in the grouping field.

In this way, the grouping operation is explicit and easy to read. Also, the entire document will only be scanned once for the grouping, it is much more efficient than the nested-querying of XQuery, which may scan the document once for each value of the grouping field (maybe a good query optimizer can avoid multiple scans). Finally, our grouping will not generate invalid empty groups when there are multiple grouping fields, as 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. XQuery may generate such empty groups because it uses nested *for* clauses which are the Cartesian products of the grouping fields, instead of the tuples.

Here we re-write the Example 2.13 and Example 2.14 in our XTreeQuery format:

Example 4.9. For the bibliography document in Appendix I, list the book titles published in each year.

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

Note that in the above query, by the grouping definition, each *{$yeargroup}* is a list of books with the same publishing year, *{$yeargroup}.key()* returns the value of publishing year (which is the grouping field) of the books in the list *{$yeargroup}*, and *{$yeargroup/title}* generates the titles of the books in the list *{$yeargroup}*.

Example 4.10. For the document employees.xml with DTD shown as in Figure 2, find the average salary of employees, grouping by department and jobtitle.

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

Note that in the above grouping, the grouping fields are the tuple of department and jobtitle (indicated by *$e/[department, jobtitle]*). If some combination of department and jobtitle does not occur in the document (i.e., no employee has that combination of department and jotitle), it will also not appear in our grouping result.

4.5 Recursion

Sometimes we need to scan over a hierarchical structure of elements, applying a transformation at each level of the hierarchy. XQuery does not support recursion directly; instead, it uses user-defined recursive functions. This indirect way of expressing recursion makes the query difficult to read, and such usage of function is not natural, as illustrated by Example 2.15. The main reason that XQuery cannot write recursion directly is because that XQuery defines the return format in nested *return* clause, which is difficult to be used as the source for another query.

In contrast, our XTreeQuery supports recursion directly in an easy and natural way. The *construct* clause of XTreeQuery 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 returned by 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.

Here we re-write the Example 2.15 in our XTreeQuery format:

Example 4.11. Consider the list of employees in Appendix IIIA. Convert the list to a tree structure of employees, in which a parent node is the manager, and the children nodes are the direct subordinates.

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

*query /employeetree//employee→$e/[@id→**$x**, not(employee)],*
*/employeetlist/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 (specified by *not(employee)*, meaning that this employee does not have a child element named "employee") in the tree, we search the employee list to find those employees whose manager is this employee (by the use of common variable *$x*), and put them one level below this employee in the tree (as the subordinates of the employee). After 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 *return* clause. It will run until no more new results are produced.

Also, in the second query, the specified return format in *construct* clause does not start from an XML root node (/), but refers to the position of XML node *$e*, which is some employee element in the employee tree (similar to a relative XPath), i.e., the XTree expression in the *construct* clause can be used in another *construct* clause again. This will make all the subordinate employees be placed correctly under their supervisor employee. The result of the queries is shown in Appendix IIIB.

4.6 Quantification

Sometimes in a query, we want some or all of the items in a list satisfy some conditions, thus we need to enforce the quantification check. In XQuery, we use *every…in…satisfies* in *where* clauses for universal quantification, and *some…in…satisfies* in *where* clauses for existential quantification. In the query the default state is existential quantification.

Our XTreeQuery expresses quantification in a more direct way. It can apply the built-in functions *some(), every(), none()* to a list-valued variable for existential, universal and non-existential quantification on that list respectively.

Example 4.12. Consider the bibliography document in Appendix I, select those books whose authors all have a first name called "Peter". In XQuery we can write as follows:

```
for $b in /bib/book
where every $a in $b/author satisfies $a/first = "Peter"
return $b
```

However in our XTreeQuery we can write the quantification functions directly on the list-valued variables, as following:

```
query /bib/book→$b/author→{$a}
where {$a}.every()/first="Peter"
return /results/book←$b
```

Note that function *every()* will check each item in the list *{$a}*, i.e., each author of a book, to see whether its first name is "Peter".

Example 4.13. According to the three XML documents in Appendix II, find the sailors, if any, who have reserved every boat. In XQuery we can write as follows:

```
for $s in doc("sailors.xml")/sailors/sailor
where
   every $b in doc("boats.xml")/boats/boat satisfies
      some $r in doc("reservations.xml")/reservations/reservation satisfies
         ($b/@bid = $r/@bid and $s/@sid = $b/@sid)
return $s
```

In our XTreeQuery we can write the query as follows:

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

Note that *{$r}* will be a list of reservations of a certain sailor *$s*, since the common variable name *$sid* in the first and third XTree expressions in the *query* clause indicates a join operation, as we described in Section 4.2.

4.7 Special queries

Because our XTreeQuery is rich in syntax and semantics, it can express some special queries that are not supported by XQuery, or not supported efficiently.

4.7.1 URL-related querying

As we described before, XTree treats the URL of an XML document and the associated element as first class citizens, and variables can be bound on the URL or part of the 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 4.14. Suppose we want to find the bibliography documents located in the website http://www.abc.com and directory /docs/bib, which has the structure as in Appendix I and contains a book with 1994 for the attribute *year* and "TCP/IP Illustrated" for the sub-element *title*. We can write XTreeQuery query as follows:

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

4.7.2 Structure level querying

Structure level queries are the queries on XML documents with unknown structure, or the queries that rename the elements/attributes in the result without knowing their inner structure. However, in XQuery, variables are bound to the name-value pairs of some XML nodes, and special built-in functions are needed to split them, thus it is very inefficient to handle structure level queries. This is illustrated by Example 2.11.

However, in our XTreeQuery, names and values of XML nodes are explicitly split in the variable bindings: variables in the left side of symbol → will bind to the node names, and variables in 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 to split them for internal information. This will make the queries on unknown structure easier to write and understand.

Here we re-write the Example 2.11 in our XTreeQuery format:

Example 4.15. Consider the bibliography in Appendix I, suppose we do not know the substructure of *book* elements, now we want to restructure books in this way: keep text nodes and sub-elements unchanged, but convert attributes to be 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 4.16. Find books which have some attribute with the value "1994", and return the titles of the books and the name of that attribute.

query /bib/book/[title→$t, @$attribName="1997"]
construct /result/[title←$t, attribute←$attribName]

Example 4.17. 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

4.7.3 Sample querying

Sample queries are the queries that just pick up several items randomly after selecting and filtering, instead of getting the whole result set. XQuery does not have functions to get a certain subset of a list, thus it is difficult to express sample queries. However, our XTreeQuery can easily handle such queries because it supports list-valued variables explicitly, and defines various functions to manage a list.

Example 4.18. In the bibliography document in Appendix I, just pick up any two books.

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

Note that in the XTreeQuery, function *random(x)* will pick up *x* items in the list randomly to form a sub-list.

Example 4.19. Get any three books with price more than 50.

query /bib/book→{$b}/price>50
construct /result/book←{$b}.random(3)

4.7.4 Top-k querying

Top-k queries are the queries that just pick up the first several items according to some order, instead of getting the whole ordered result set. XQuery does not have functions to get a certain subset of a list, thus it is difficult to express sample queries. However, our XTreeQuery can easily handle such queries because it supports list-valued variables explicitly, and defines various functions to manage a list.

Example 4.20. Get the prices of the two cheapest books.

> *query /bib/book/price→{$p}*
> *construct /result/cheap_price←{$p}.sort()[1-2]*

Note that the above query may return two prices with the same value, since there may be more than one cheapest book (with same price). Function *sort()* will sort the items in the list in ascending order (increasing order for list of numbers, and alphabetic order for list of strings); it will not remove duplicates in the list. To remove the duplicates, we can call function *distinct()*, by writing *{$p}.sort().distinct()[1-2]*.

Example 4.21. Get all the content of the two cheapest books.

> *query /bib/book→{$b}*
> *order by $b/price*
> *construct /result/cheap_book←{$b}[1-2]*

Note that *order by* clause will sort the XML nodes in ascending order of the specified fields. For descending order, we can write the statement as *order by ... descending*. This is same as the *order by* clause in XQuery.

Example 4.22. Get all the content of the two most expensive books.

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

4.8 Updates

Current XQuery language can only query XML documents, but cannot update them. However, in order to fully evolve XML into a universal data representation and sharing format, we must allow users to make updates on XML documents. Researchers have proposed some methods to specify updates and have developed techniques to process them efficiently [24, 30]. Being not only a data querying language, but also a data management language, our XTreeQuery supports update on XML documents.

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. There are five kinds of update expressions in XTreeQuery:

- **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 built-in 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 hold by *var*. Deleting an element node will remove all its contents, i.e., its sub-elements and attributes.

- **replace** var **with** content

  This statement is used to replace the information hold by *var* by the information in *content*, with the position unchanged.

Example 4.23. For the bibliography document in Appendix I, 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 4.24. Add the above new book as the last book in the document.

```
query /bib/book[last()]→$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"]
after $b
```

Example 4.25. Add a sub-element named "comments" with value "among best sellers 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 4.26. Delete all the books whose title contains word "violence".

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

Example 4.27. Delete the two earliest published books.

```
query /bib/book→{$b}
order by $b/@year
delete {$b}[1-2]
```

Example 4.28. 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

In this section, we will compare our XTreeQuery with other existing XML querying languages, such as Lorel[1], XQL[29], XML-QL[14], Quilt[6], XDuce[17], a rule-based semantic query language[7], a declarative XML query language[23], which are briefly introduced in Section 2.4, and XQuery[33] which is the current trend of W3C.

The comparison of these query languages emphasizes on their expressive power, and whether they can make types of queries efficiently. We will use the following criteria:

1. Data model: how to model the XML data. It specifies what information in the XML document is accessible for querying.
2. Expressions: how to specify interested paths in the query.
3. Join: whether the query language supports joins over different data sources, and how efficient are the join operations.
4. Negation: whether the query language can express negative queries.
5. Grouping: whether the query language can divide the data into groups according to some group fields, and apply aggregate functions over each group.
6. Recursion: whether the query language can recursively query a 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 results set.
11. Ordering: whether the query language can order the element instances according to the ascending or descending values of some data of the result.

12. Nesting: whether the query language supports nested querying structure (queries containing nested sub-queries), in order to express complicated queries.
13. Updates: whether the query language can specify update operations on XML documents.

Table 4 compares the expressive power of our XTreeQuery with Lorel, XQL, XML-QL, a rule-based semantic query language, a declarative XML query language, and XQuery.

Table 4. Comparison between XML query languages

	XTreeQuery	**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

Note that for the join, negation, grouping, recursion and quantification operations, our XTreeQuery can express them in a more direct and efficient way. For the join operations, XTreeQuery uses a QBE-like solution, instead of by nested sub-queries; for the negation, XTreeQuery can express a negative sub-tree in the querying part; for the grouping operations, XTreeQuery uses a two dimensional list to hold the values of all groups, to

avoid multiple scans over the document; for the recursion, XTreeQuery can directly query the output XTree expression in the *construct* clause, instead of by defining recursive functions; and for the quantification, XTreeQuery can invoke built-in functions on list-valued variables directly, to express the corresponding quantification on that list of items. In addition, because XTreeQuery is based on XTree expressions, which can bind multiple variables in one expression, thus the queries are more compact and efficient.

Thus we can see that our XTreeQuery has rich expressive power, and outperforms other query languages. It supports most of database operations efficiently.

4.10 Summary

Currently, XQuery is the most promising standard from W3C. However, it has some limitations, such as join operation as sub-query, grouping as sub-query, recursion by user-defined recursive function, nested querying structure, no update operations (as we discussed in Section 2.3.2). In this chapter, we introduced our XML query language named XTreeQuery, which is based on XTree expressions. Through some examples, we showed that our XTreeQuery based on XTree expressions is simpler yet more expressive than XQuery.

For XTreeQuery, 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 just write one XTree expression to define the result format, the values of variables will be substituted at appropriate nodes in the XTree expression. The separation of querying part from result construction part effectively avoids nested structure in the query, and makes the whole query easy to read and comprehend.

XTreeQuery can express join, negation, grouping, recursion and quantification directly and efficiently. It also supports some special queries (URL-related querying, structure level querying, sample querying, top-k querying) and update operations. All these operations are not supported by XQuery, or not supported efficiently. By a comparison between our XTreeQuery and some other existing XML query languages, we proved that XTreeQuery has richer expressive power, and is more advantageous than other query languages.

Chapter 5

Algorithms to transform XTreeQuery to XQuery

We have seen that XTreeQuery has more expressive power than current XQuery, and it can express queries in a much compact and efficient version. However, we would like to transform standard XTreeQuery queries to XQuery queries, to make them executable by existing XQuery parsers. In this chapter, we present two algorithms for the translation. In Section 5.1, we describe an algorithm that transforms an XTree expression in querying part of a query to a set of XPath expressions. In Section 5.2, we describe an algorithm that transform an XTree expression in the result construction part of a query to some nested XQuery expressions. In Section 5.3, an example is given to better illustrate the algorithms. Finally, we summarize our contributions in Section 5.4.

The main idea of the XTreeQuery-to-XQuery translation is to convert the XTree expressions in *query* clause and *construct* clause of XTreeQuery to some XQuery expressions (since the *where* clause and *order by* clause of XTreeQuery have the same syntax as the *where* clause and *order by* clause of XQuery).

Currently our translation algorithms are a basic version; it can only translate basic XTreeQuery queries, which have no variables in the URL, no join, negation, grouping, recursion and quantification operations, and no updates on XML document. We will

extend the algorithms to support join, negation, grouping, recursion and quantification. However, for the XTreeQuery queries with variables in URL and update operations, we cannot translate them to XQuery, since these features are not supported by XQuery.

Before introducing the algorithms, we will make the following definitions:

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

For example, for 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 on the schema of the XML document, that starts from the endpoint of $path_2$ and ends 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 $path_1$. The *relative path* is also a schema path.

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

Relative path will be used to express an XPath expression whose location path is not an absolute path starting from the root, but a relative path starting from some pre-defined XPath (such as *$book/title*, where *$book* is defined before as */bib/book*).

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 *$b* are defined between *path($a)* and *path($b)*.

For example, in XTree expression */bib/book→$b/[title→$t, author/last→$last]*, *$b* is the nearest ancestor variable of *$t* and *$last*.

Nearest ancestor variable will be used to serve as the start point of the relative path expression. For the above example, *$b* has no nearest ancestor variable, so its XPath expression will be an absolute path from the root, as */bib/book*; however, since *$b* is the nearest ancestor variable of *$t*, the XPath expression of *$t* will be a relative path starting from *$b*, as *$b/title*.

In order to construct the result format correctly, we have to figure out the correspondence between the structure of XTree expression in the querying part and the structure of XTree expression in the result construction part.

Definition 8. We say that node B (in the result construction part of a query) is *derivable* from node A (in the querying part of a query) if the content of B can be derived from the content of A. Initiatively, node B is *derivable* from node A means they 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.

Lemma 1. Suppose node A is from querying part, and node B is from result construction part, then node B is *derivable* from node A if and only if one of the following holds:

(1) If node B is a leaf node and it does not contain any variable (e.g., the expression of B is: *node*, *node←"abc"*), then node B is *derivable* with 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 invocation of a function on *$x* (e.g., *$x.substring(1,5)*, *$x.string-length()*, *$x.normalize-space()*), 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 invocation of a list-valued function on *{$x}* (e.g., *{x}[1-3]*, *{x}.distinct()*, *{$x}.sort()*), then node B is *derivable* from node A.

(4) If the expression of node A is: *A→{$x}*, the expression of node B is: *B*← invocation of an aggregate function on *{$x}* (e.g., *{x}.count()*, *{x}.avg()*), then node B is *derivable* from node A.

(5) If node A have variables bound to its children $A_1 \ldots A_m$, node B have variable substitutions on its children $B_1 \ldots B_n$. If every B_i is *derivable* from a A_j, then node B is *derivable* from node A.

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

5.1 Transformation algorithm for querying part

Transforming an XTree expression in the querying part to a set of XPath expressions is not 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, and define them correctly.

It is very easy to get these common ancestors, by just analyzing the textual XTree expression itself. The paths just before every pair of square braces for branching (not for list-valued variables) are the common ancestors we want, because the pair of square braces implies that all the enclosed sibling branches are interested by the user, no matter the sibling branches will head to some variable bindings or some constraints.

Figure 4 gives the pseudo code of the algorithm that transforms an XTree expression in the querying part to a set of XPath expressions.

```
Initialize an empty stack
Process the textual XTree expression from left to right
for each node traversed {
   if it is bound to a single-valued variable $svar {
      try to pop the stack until it is empty, or its top item $ancvar is ancestor of $svar
      // if initially the top item is ancestor of $svar, then no need to pop
      if the stack is empty, output XPath: For $svar in path($svar)
      else    // now the top item of the stack $ancvar is ancestor of $svar
         output XPath: For $svar in $ancvar/relaPath(path($svar), path($ancvar))
      push $svar and path($svar) into stack
   }
   else if it is bound to a list-valued variable {$lvar} {
      try to pop the stack until it is empty, or its top item $ancvar is ancestor of {$lvar}
      if the stack is empty, output XPath: Let $lvar := path({$lvar})
      else output XPath: Let $lvar := $ancvar/relaPath(path({$lvar}), path($ancvar))
      push $lvar and path({$lvar}) into stack
   }
   else if it is followed by a pair of square braces [ ] for branching, and it is not the root {
      assign a new single-valued variable $new_svar to this node
      try to pop the stack until it is empty, or its top item $ancvar is ancestor of $new_svar
      if the stack is empty, output XPath: For $new_svar in path($new_svar)
      else output XPath:
         For $new_svar in $ancvar/relaPath(path($new_svar), path($ancvar))
      push $new_svar and path($new_svar) into stack
   }
}
empty the stack
```

Figure 4. Algorithm to transform an XTree expression in querying part

The main idea of this algorithm is that we find all the common ancestors of variables, except the root (since an XML document only has one root node), and assign single-valued variables on them if they are not bound to variables originally. Then we traverse the XTree expression from left to right (which corresponds to the depth-first order of the tree), translate each single-valued variable to be an XPath expression in a *for* clause, and translate each list-valued variable to be an XPath expression in a *let* clause. We will use a stack to store the ancestor variables during the translation; each item in the stack is a pair of variable name (that may be an ancestor for the nodes processed later) and its path. For each variable, we will find its nearest ancestor variable by popping the stack until the

stack is empty, or the top item of the stack is ancestor of it. If the stack is popped empty (no such ancestor found), we will output the XPath expression of this variable to be the absolute path from the root of the document; otherwise we will output the XPath expression of this variable to be the relative path from the top item in the stack (its nearest ancestor variable). Then we will push this variable together with its path into the stack, since it may be the ancestor of some variables processed later. When we have processed all the nodes in the XTree expression, we can empty the stack.

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

Example 5.1. Translate the following XTree expression to a set of XPath expressions:

query /bib/book→$b/author/last→$last

By the algorithm, we process the XTree expression from left to right, when we reach node *book*, which binds to variable *$b*, we will output an XPath expression from root, since now the stack is empty:

for $b in /bib/book

and push a pair (*$b*, /bib/book) into the stack. Later when we reach node *last*, we check that the top item of the stack (which is *$b* and its path) is an ancestor of *$last*, thus we will output XPath expression of *$last* to be the relatively path from *$b*:

for $last in $b/author/last

Example 5.2. Translate the following XTree expression to a set of XPath expressions:

query /bib/book→$b/author/[last→$last, first→$first]

By the algorithm, we process the XTree expression from left to right, when we reach node *book*, we output XPath expression:

for $b in /bib/book

and push a pair (*$b*, /bib/book) into the stack. Later when we reach node *author*, since it is followed by a branch (i.e., it is a branching node), we assign a new single-valued variable *$a* to it. We check that the top item of the stack (which is *$b* and its path) is an ancestor of *$a*, thus we output XPath expression of *$a* to be the relative path from *$b*:

for $a in $b/author

and push a pair (*$a*, /bib/book/author) into the stack. Later when we reach node *last*, we check that the top item of the stack (which is *$a* and its path) is an ancestor of *$last*, thus we output XPath expression of *$last* to be the relative path from *$a*:

for $last in $a/last

and push a pair (*$last*, /bib/book/author/last) into the stack. Later when we reach node *first*, we check that the top item of the stack (which is *$last* and its path) is not an ancestor of *$first*, thus we pop the stack, and check again. Now the top item of the stack (which is *$a* and its path) is an ancestor of *$first*, thus we output XPath expression of *$first* to be the relative path from *$a*:

for $first in $a/first

Thus the final output XPath expressions are as follows:

for $b in /bib/book
for $a in $b/author
for $last in $b/last
for $first in $a/first

5.2 Transformation algorithm for result construction part

Transforming an XTree expression in the result construction part of a query to some XQuery expressions is more complicated, since we often encounter nested sub-queries in XQuery. Also, if the node name to get the variable value is different from the node name where the variable was bound in the querying part (i.e., the user wants to restructure the result), it will be difficult to handle, since we have to explicitly split the name-value pair of variables in XQuery. Figure 5 gives the pseudo code of the algorithm that transforms an XTree expression in the result construction part to some XQuery statements.

```
1.  String $begin := "", String $end := ""
2.  Process the XTree expression from left to right
3.  for each node node_i traversed {
4.     if node_i is trivially derivable {
       //suppose the expression of current node is $expr := node_i or node_i←string
5.        $begin := $begin + "return " + translate($expr)
       }
6.     else if node_i has no "←" symbol {
7.        if it is derivable from a non-leaf node (bound to variable $anc) in last algorithm
10.          $begin := $begin + "{" + the XPath expression of $anc from last algorithm
11.          $begin := $begin + "return <node_i>"
12.          $end := "</node_i>" + $end
13.      else      //it is the first several levels in result structure, e.g., root of result
14.          $begin := $begin + "<node_i>", $end = "</node_i>" + $end
       }
16.    else {
          //suppose the expression of current node is: $expr_i := node_i←$var_i or node_i←{$var_i}
18.       $begin := $begin + "{" + the XPath expressions of variable $var_i or {$var_i}
19.       $begin := $begin + "return " + translate($expr_i) + "}"
20.       if next node is descendent of current node, or this is the last node (no next node)
21.          $end := "</node_i> }" + $end
22.       else if next node is sibling of current node
23.          $begin := $begin + "</node_i> }"
24.       else //next node is sibling of some node_j processed before
             //let string $end_j be the substring of $end from beginning till "</node_j> }"
26.          $begin := $begin + "</node_i> }"
27.          $begin := $begin + $end_j,, $end := $end - $end_j
       }
    }
30. output $begin + $end
```

Figure 5. Algorithm to transform an XTree expression in result construction part

The main idea of this algorithm is that we process the XTree expression in result construction part step by step. We will find the corresponding XPath expressions of each variable in the output of last algorithm, and use curly braces *{ }* to form sub-query blocks according to the structure of the XTree expression in *construct* clause. If a step is a leaf node with only plain text (not variable value substitution) (line 4), then this node will be written in the *return* clause of XQuery directly as an XML tag (line 7). If the node does not get value from a variable but is compatible with a non-leaf node in the querying part (line 9), we will get the XPath expression of its compatible node (line 10), and write its node name directly in the *return* clause (line 11-12) (the algorithm will process its children nodes later in sub-query blocks); otherwise if it is not compatible with any node in the querying part (this usually occurs as the first several levels of result construction, such as */results/result/...*) (line 13), we will just write node tags directly (line 14). For the last case, if the node gets values from a variable (line 16), we will find the corresponding XPath expression of this variable (line 18), and translate the variable substitution statements depending on whether the node name for value substitution is the same as node name where the variable was bound in the querying part, as a sub-query block (line 19). Then we will check whether the next node to be processed is the descendent of current node (line 20), the sibling of current node (line 22), or the sibling of a node processed before (line 24), and adjust the sub-query blocks accordingly (line 21, 23, 26-27). At last we output the whole translated XQuery queries (line 30).

In the above algorithm, function *translate($expr)* will translate a plain node segment or a node with value substitution expression to a *return* clause in XQuery. It is defined as in Figure 6.

```
define function translate($expr) {
   case 1. $expr is of format: element     //no ← symbol
      return <element/>
   case 2. $expr is of format: element ← string
      return <element> string </element>
   case 3. $expr is of format: attribute ← string
      return attribute = "string"
   case 4. $expr is of format: element ← $var {
      if element is the same name as the node where $var was bound in querying part
         return $var
      else    //name changed
         return <element> {$var/*} {$var/@*} {$var/text()} </element>
   }
   case 5. $expr is of format: element ← {$var} {
      if element is the same name as the node where {$var} was bound in querying part
         return $var
      else
         return { for $x in $var
                  return <element> {$x/*} {$x/@*} {$x/text()} </element> }
   }
   case 6. $expr is of format: @attribute ← $var {
      if attribute is the same name as the node where $var was bound in querying part
         return $var
      else           //suppose the parent element of $var is element
         return <element attribute = {string($var)}>
   }
}
```

Figure 6. Function *translate($expr)*

For case 1 to case 3, the node only has plain text, and there is no variable value substitution involved, we just write the XML segment directly. For case 4 to case 6, the node has a variable value substitution expression, we translate the expression depending on whether the node name for value substitution is the same as the node name where the variable was bound in querying part. If the node name remains the same, we just put the value of the variable at the place. Otherwise if the node name is changed, we will get all possible inner structure (sub-elements, attributes, text fields, etc) of the node and put their values enclosed by the new node name. Case 4 is for expression with element value substitution from a single-valued variable; case 5 is for expression with element value substation from a list-valued variable; and case 6 is for expression with attribute value

substitution from a single-valued variable. Note that list-valued variables cannot be used for attributes, since in XML an element cannot have multiple attributes with the same name.

Example 5.3. Suppose for each book, we want to keep its title and author information, and rename the "author" element to be "writer". Following is the XTreeQuery for this query, translate it to XQuery version:

```
query /bib/book/[title→$t, author→$a]
construct /result/book/[title←$t, writer←$a]
```

By the first algorithm, the XTree expression in the querying part will be translated as the following XPath expressions:

```
for $b in /bib/book
for $t in $b/title
for $a in $b/author
```

Now we process the XTree expression in the result construction part. The first node *result* does not have value assignment symbol "←", and is not derivable from any node in the XTree of the querying part, thus by line 14 of the algorithm, its tags <result> and </result> will be put at the beginning and end of the query directly.

For the node *book*, it is derivable from the node *book* in the XTree of the querying part (which bounds to variable *$b*), thus by lines 10-12 of the algorithm, we will form a querying block:

```
{
  for $b  in /bib/book
  return <book>
            … (to be filled by later processed node translations)
           </book>
}
```

For the node *title*, by lines 18-19, and line 23 (since the next node *author* is a sibling of *title*), we will form a sub-querying block:

```
{
  for $t in $b/title
  return $t
}
```

Lastly, for the node *writer*, by lines 18-19, and line 21 (since it is the last node to be processed), we will form a sub-querying block:

```
{
  for $a in $b/author
  return <writer> {$a/*} {$a/@*} {$a/text()} </writer>
}
```

Thus, combine the above sub-querying blocks together, we have the whole translated XQuery as follows:

```
<result> {
  for $b  in /bib/book
  return <book> {
    for $t in $b/title
    return $t
  }
  {
    for $a in $b/author
    return <writer> {$a/*} {$a/@*} {$a/text()} </writer>
  }
  </book>
}
</result>
```

5.3 An example of our algorithm

To illustrate how our algorithms transform an XTreeQuery query based on XTree expressions into standard XQuery query, we use our algorithms to process the following example:

Example 5.4. Consider the following XTreeQuery query. We want to list each book and each journal. For each book, we rename the *title* to *name*, add an element *authors* that consists of all the authors of this book, add an attribute *count* in *authors* which counts the

number of authors of the book, and rename *author* element to *au*. For each editor of a journal, we put his/her first name before last name. The XTreeQuery query is as follows:

query /bib/[book/[title→$t, author→{$a}],
journal/[title→$jt, editor/[last→$last, first→$first]]]
construct /results/[book/[name←$t, authors/[@count←{$a}.count( ), au←{$a}],
journal/[title←$jt, editor/[first←$first, last←$last]]]

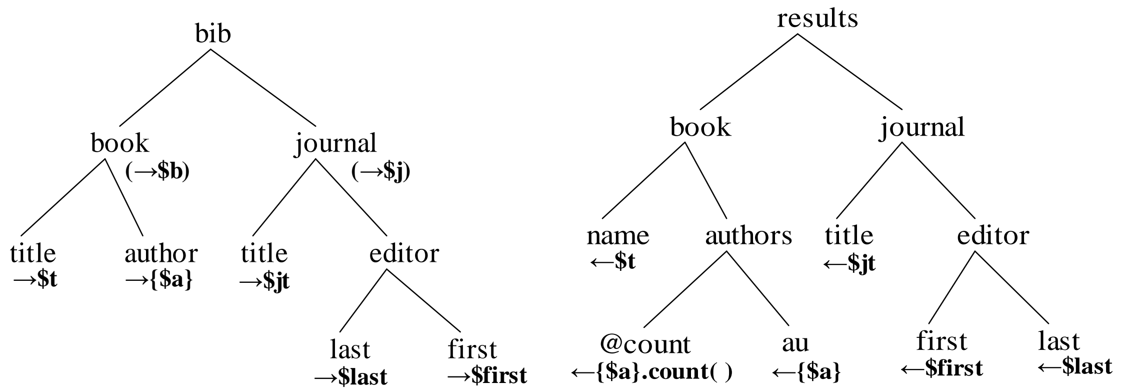

Figure 7a. XTree graph for querying   Figure 7b. XTree graph for result construction

Figure 7a shows the XTree graph for querying part, and Figure 7b shows the XTree graph for result construction part.

For the XTree expression in the querying part, by applying the first algorithm, we process it from left to right. The algorithm will output an XPath expression for each node bound to a variable and each branch node.

First we reach the node *book*, which is followed by a branch. It is not bound to any variable originally, and is not the root, thus we assign a new single-valued variable *$b* to it, output a *for* clause: *for $b in /bib/book*, and push the pair (*$b*, /bib/book) into the stack.

Next we reach the node *title*, which is bound to a single-valued variable *$t*, we check that the top item of the stack (which is *$b* and its path) is an ancestor of *$t*, thus we output the XPath expression of *$t* to be a relative path from *$b*: *for $t in $b/title*, and push the pair (*$t*, /bib/book/title) into the stack.

Next, for node *author*, which is bound to a list-valued variable *{$a}*, we check that the top item of the stack (which is *$t* and its path) is not an ancestor of *{$a}*, thus we pop the stack, and check again. Now the top item of the stack (which is *$b* and its path) is an ancestor of *{$a}*, thus we output the XPath expression of *$a* to be a relative path from *$b*: *let $a := $b/author*, and push the pair (*$a*, /bib/book/author) into the stack.

Then, for node *journal*, which is followed by a branch. It is not bound to any variable originally, and is not the root, thus we assign a new single-valued variable *$j* to it. By checking whether the top item of the stack is an ancestor of *$j*, we will pop the stack to be empty, thus we output the XPath expression as: *for $j in /bib/journal*, and push the pair (*$j*, /bib/journal) into the stack.

By continuing such procedure, finally we will have the following output:

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

From the above translated XPath expressions. We can see that for this example, one XTree expression in the querying part corresponds to 8 XPath expressions here (7 *for* clauses and 1 *let* clause). Among the 8 variables, 5 variables (*$t*, *$a*, *$jt*, *$last* and *$first*) are originally defined in the XTree expression, and the rest 3 variables (*$b*, *$j* and *$e*) are the generated by the algorithm, they are common ancestors used as starting points of relative paths.

For the result construction part, by applying the second algorithm, we will get the following XQuery query (detailed execution omitted) as in Figure 8.

```
<result> {
   for $b in /bib/book
   return <book> {
      for $t in $b/title
      return <name> {$t/*} {$t/@*} {$t/text( )} </name>
   }
   {
      let $a := $b/author
      return <authors count={count($a)}> {
         for $x in $a
         return <au> {$x/*} {$x/@*} {$x/text()} </au>
      }
      </authors>
   }
   </book>
}
{
   for $j in /bib/journal
   return <journal> {
      for $jt in $j/title
      return {$jt}
   }
   {
      for $e in $j/editor
      return <editor> {
         for $first in $e/first
         return {$first}
      }
      {
         for $last in $e/last
         return {$last}
      }
      </editor>
   }
   </journal>
}
</result>
```

Figure 8. Result XQuery of Example 5.4

5.4 Summary

In this chapter, we have developed algorithms to transform a simple XTreeQuery query to an XQuery query. There are two algorithms for the transformation: the first algorithm transforms an XTree expression in the querying part of a query to a set of XPath expressions, and the second algorithm transforms an XTree expression in the result construction part of a query to some nested XQuery expressions. The pseudo code of the

two algorithms is presented, and some examples are given to illustrate how the algorithms work.

After transformation, the output XQuery query can be executed by current XQuery parsers.

Chapter 6

Conclusion and future works

In this chapter, we summarize our research work in Section 6.1. In addition, some future research directions are recommended in Section 6.2 to further enhance the expressive power of XTree and XTreeQuery.

6.1 Conclusion

In this thesis, we have surveyed some existing XML query languages, discussed the limitations of XPath and XQuery, and proposed our XTree and XTreeQuery based on the complex object data model[23].

XTree is a generalization of XPath; it is more compact and convenient to use than XPath. XTree can be used in both querying part and result construction part of a query. It has a tree structure that is similar to XML document, thus in the querying part, a user can instantiate multiple paths (each path is bound to a variable) at the same time in one XTree expression. It explicitly identifies list-valued variables, uniquely determines their values and defines some natural built-in functions to manipulate them in an object-oriented manner. XTree also supports URL-related searches by binding variables on the URL or

part of the URL so that they can also be queried, thus it supports the functionality of search engines. To prevent the mixing of XML plain text and variable values and even sub-queries in the result construction, we can also use an XTree expression in the result construction part, to define the result format in a clearer and more compact way.

XTreeQuery is a query language based on XTree expressions. It is more powerful and efficient than XQuery. In the querying part, multiple variables can be defined in one XTree expression; in the result construction part, the result format can be defined in just one XTree expression, which prevents nested structure in the query. The separation of querying part and result construction part effectively avoids nested structure in the query, and makes the whole query easy to read and understand. Thus the XTreeQuery queries are much shorter in length and clearer to understand, compared to XQuery. Our XTreeQuery also effectively supports join, negation, grouping, recursion, quantification, updates and some special queries which are not supported by XQuery, or not supported efficiently. By a comparison between our XTreeQuery and some other existing XML query languages, we can see that our XTreeQuery has richer expressive power, and supports most of the database operations efficiently.

To utilize the current XQuery parsers, we have also designed two algorithms that convert an XTreeQuery query to a XQuery query. The first algorithm transforms an XTree expression in the querying part into a set of XPath expressions, and the second algorithm transforms an XTree expression in the result construction part into some nested XQuery statements.

6.2 Future works

For the future research, we would like to implement the XTreeQuery system that supports XTreeQuery query and update language. An XTreeQuery query parser will execute queries directly, instead of translating it into XQuery queries. The querying evaluation will be more efficient on this approach, since we will have a global view of the entire query tree.

Currently, our algorithms for translating XTreeQuery to XQuery is a basic version, which only support queries without join, negation, grouping, recursion and quantification. We will extend the algorithms by adding translation rules for join, negation, grouping, recursion and quantification operations.

Also, the output XQuery queries of our transformation algorithms look very lengthy, because the algorithms assume nothing is known in the structure of the XML document, thus all the possible contents of an element/attribute are considered. If we also know the schema of the document (e.g., in the bibliography data in Appendix I, *title* is a text element, without sub-elements and attributes, each *author* element has only one last name and one first name, etc) , we may optimize the queries to be more compact. We will think about the formal optimization algorithms which can utilize the schema of the XML documents, and make the output XQuery queries simpler.

In addition, we will also observe the progressive development of XQuery to continuously enhance the expressive power of our XTreeQuery.

Publication list

1. Zhuo Chen, Tok Wang Ling, Mengchi Liu, Gillian Dobbie. "XTree for Declarative XML Querying", In *proceedings of the 9th Int'l Conference on Database Systems for Advanced Applications*, Korea, 2004.

References

[1] S.Abiteboul, D.Quass, J.McHugh, J.Widom, and J.L. Wiener. The Lorel Query Language for Semistructured Data. *Intl. Journal of Digital Libraries,* 1(1):68-99, 1997.

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

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

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

[5] S.Ceri, S.Comai, E.Damiani, P.Fraternali, and L.Tanca. Complex Queries in XML-GL. *SAC(2)* 2000:888-893.

[6] D.Chamberlin, J.Robie, and D.Florescu. Quilt: An XML query language for heterogeneous data sources, In *Proceedings of International Workshop on the Web and Databases*, 2000.

[7] P.Chippimolchai, V.Wuwongse and C.Anutariya. Semantic Query Formulation and Evaluation for XML Databases. In *Proceedings of WISE 2002*, 205-214, Singapore, 2002.

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

[9] S.Cohen, Y.Kanza, Y.Kogan, W.Nutt, Y.Sagiv and A.Serebrenik. Equix – Easy Querying in XML Databases. In *proceedings of Webdb'98 – The Web and Database Workshop*, 1998.

[10] S.Comai, E.Damiani, P.Fraternali. Computing Graphical Queries over XML Data. *ACM Transactions on Information Systems*, Vol. 19, No. 4, October 2001, Pages 371-430.

[11] S.Comai, E.Damiani, L.Tanca. The WG-Log System: Data Model and Semantics. *INTERDATA technical report*, T2-R06, July 1998.

[12] I.F.Cruz, A.O.Mendelzon, and P.T.Wood. A Graphical Query Language Supporting Recursion. Proc. *ACM SIGMOD Conf. on Management of Data*, 1987, pp. 323—333.

[13] C.J.Date. An Introduction to Database Systems. 3rd Edition, *Addison-Wesley Publishing Company*, 1981.

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

[15] G.Dobbie, X.Y.Wu, T.W.Ling, M.L.Lee. ORA-SS: An Object-Relationship-Attribute Model for Semistructured Data. TR21/00, *Technical Report*, Department of Computer Science, National University of Singapore, December 2000.

[16] P.W.Eklund, J.Leane, and C.Nowak. GrIT: An implementation of a graphical user interface for conceptual structures. *Technical Report TR94-03*, Computer Science Department, The University of Adelaide, February 1994.

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

[18] M.Kifer, G.Lausen: F-Logic: A Higher-Order language for Reasoning about Objects, Inheritance, and Scheme. *SIGMOD Conference 1989*: 134-146

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

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

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

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

[23] M.Liu and T.W.Ling. Towards Declarative XML Querying. In *Proceedings of WISE 2002*, 127-138, Singapore, 2002.

[24] M.Liu, L.Lu and G.R.Wang. A Declarative XML-RL Update Language. In *Proceedings of ER2003*, 506-519, Chicago, USA, 2003

[25] L.Mark, *et. al*. XMLApe. College of Computing, Georgia Institute of Technology. *http://www.cc.gatech.edu/projects/XMLApe/*

[26] Y.Y.Mo, T.W.Ling. Storing and Maintaining Semistructured Data Efficiently in an Object-Relational Database. *Research Report*. School of Computing, NUS.

[27] K. D. Munroe and Y. Papakonstantinou. BBQ: A visual interface for integrated browsing and querying of XML. *In Proceedings of Visual Database Systems*, May, 2000.

[28] W. Ni, T. W. Ling. GLASS: A Graphical Query Language for Semi-Structured Data. *DASFAA 2003*.

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

[30] I.Tatarinov, Z.G.Ives, A.Y.Halevy and D.S.Weld. Updating XML. In *Proceedings of SIGMOD 2001*, pages 413-424, 2001.

[31] SQLX Working Draft. *http://www.sqlx.org/*

[32] XML Path Language (XPath) 2.0. W3C Working Draft, August 2003. *http://www.w3.org/TR/xpath20/*

[33] XQuery 1.0: An XML Query Language. W3C Working Draft, August 2003. *http://www.w3.org/TR/xquery/*

[34] XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, August 2003. *http://www.w3.org/TR/xquery-semantics/*

[35] XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft, May *2003*. *http://www.w3.org/TR/xpath-functions/*

[36] XML Query Requirements. W3C Working Draft, June 2003. *http://www.w3.org/TR/xquery-requirements/*

[37] XML Query Use Cases. W3C Working Draft, August 2003. *http://www.w3.org/TR/xquery-use-cases/*

[38] XML Schema. W3C Recommendation, May 2001. *http://www.w3.org/XML/Schema*

[39] XSL Transformations (XSLT) Version 1.0. W3C Recommendation, November 1999. *http://www.w3.org/TR/xslt*

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 named sailors.xml, boats.xml and reservations.xml use the following Document Type Definition (*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's a line <!DOCTYPE sailors SYSTEM "sample.dtd"> at the beginning; in boats.xml, there's a line <DOCTYPE boats SYSTEM "sample.dtd"> at the beginning; and in reservations.xml, there's a line <DOCTYPE reservations SYSTEM "sample.dtd"> at the beginning.

Appendix IIIA. Sample XML document of employee list

```
<?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” manager =“Ivan”/>
   <employee id=“11” manager =“10” name=“Gerald”/>
   <employee id=“12” manager =“10” name=“Albert”/>
   <employee id=“20” name =“Michael”/>
</employeelist>
```

Appendix IIIB. Result of recursive query: employee tree

```
<?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>
```

Appendix IV. Sample XML document of people

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE people [
   <!ELEMENT people (person*)>
   <!ELEMENT person (name)>
   <!ATTLIST person id ID #IMPLIED
                    mother IDREF #IMPLIED
                    children IDREFS #IMPLIED>
   <!ELEMENT name (#PCDATA)>
]>

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

Appendix V. Formal description of XTree/XTreeQuery syntax

```
XTreeExpr        ::= ["(" URLExpr ")"] ("/" | "//") TreeExpr
URLExpr          ::= [ProtoExpr] SVarName
                   | [ProtoExpr] SVarName "/" FileExpr
                   | [ProtoExpr] DomainExpr ("/" DirExpr)* "/" FileExpr
ProtoExpr        ::= <String> "://"
DomainExpr       ::= (<Stirng> ("." <String>)+) | SVarName
DirExpr          ::= <String> | SVarName
FileExpr         ::= ((<String> | SVarName) "." <String>) | SVarName
TreeExpr         ::= NodeExpr
                   | NodeExpr ("/" | "//") TreeExpr
                   | "{" TreeExpr ("," TreeExpr)+ "}"
NodeExpr         ::= NodeName
                   | CondExpr
                   | VarExpr
NodeName         ::= ["@"] <String>
CondExpr         ::= (NodeName | VarValue) CompOp (VarValue | <Constant>)
VarValue         ::= VarName ["." FuncExpr]
VarName          ::= SVarName | LVarName
SVarName         ::= "$" <String> [("/" NodeName)*]
LVarName         ::= "{$" <String> [("/" NodeName)*] "}"
FuncExpr         ::= FuncName "(" ParaListExpr ")"
FuncName         ::= <String>
ParaListExpr     ::= ParaExpr [("," ParaExpr)*]
ParaExpr         ::= VarValue | <Constant>
CompOp           ::= "=" | "<" | "<=" | ">" | ">=" | " !=" | …
VarExpr          ::= VarBindExpr | VarAssignExpr
VarBindExpr      ::= LeftSideExpr "→" VarName
VarAssignExpr    ::= LeftSideExpr "←" (VarName | <Constant>)
LeftSideExpr     ::= NodeName | (["@"] SVarName)

XTreeQueryExpr   ::= Query_clause Where_clause? Orderby_clause? Return_clause
Query_clause     ::= "Query" XTreeExpr ("," XTreeExpr)*
Where_clause     ::= "Where" CondExpr ("," CondExpr)*
Orderby_clause   ::= "Order by" VarName ("," VarName)*
Return_clause    ::= "Return" XTreeExpr
```