Design and Implementation of Indexing Strategies for XML Documents

指導教授：張 玉 盈 教授
研究生：林 茂 桐 撰

中華民國九十一 年 六月
DESIGN AND IMPLEMENTATION OF INDEXING STRATEGIES FOR XML DOCUMENTS

A Thesis
Submitted to the Faculty

of
National Sun Yat-sen University

by
Mao-Tong Lin

In Partial Fulfillment of the Requirements for the Degree

of
Master of Science

June 2002
ABSTRACT

In recent years, many people use the World Wide Web and Internet to find information that they want. HTML is a document markup language for publishing hypertext on the WWW. HTML has been the target format for content developers around the world. Basically, HTML tags serve the primary purpose of describing how to display a data item. Therefore, HTML documents are difficult to find some useful information. That is because, HTML documents are mixed content with display tags. On the other hand, XML is the another data format for data exchange inter-enterprise applications on the Internet. In order to facilitate data exchange, industry groups define public Document Type Definitions (DTD) that specify the format of the XML documents to be exchanged between their applications. Moreover, WWW/EDI or Electric Commerce is very popular and a lot of business data uses XML to exchange information on the World Wide Web. Basically, XML tags describe the data itself. The contents (meaning) of the XML documents and the display format is separated. It could be easily to find meaningful information of the XML documents and analyze the information. Moreover, when a large volume of business data (XML documents) exists, one way to support the management of the XML documents is to apply the relational databases. For such an approach, we must transform the XML documents to the relational databases. In this thesis, we design and implement the indexing strategies to efficiently access XML documents. XML document is fundamentally different from relational data. XML is a hierarchical and nested document, it is very similar to the semistructured data model. The characteristic of semistructured data is that it may not have a fixed schema and it may be irregular or incomplete. Though, the semistructured data model is flexible in data modeling, it requires a large search space in query processing since there is no schema fixed in advance. Indexing is the way of how to improve query performance efficiently. However, due to the special properties of semistructured data, there are up to five types of queries: (1) complete single path, (2) specified leaf only, (3) specified intrapath, (4) specified attribute/element(value), and (5) multiple paths with the same level. In this thesis, we classify all possible queries into those five query types. Next, we create different indexes for different query types. Moreover, we design and implement the query transformation from XML query statements to SQL statements. Also, we create a user-friendly interface for users to input XML query statements. The whole system is implemented in JAVA and SQL Server 2000. From our experiences, we show that our indexing strategies can improve the XML query processing performance very well.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>i</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 XML</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Query Languages</td>
<td>5</td>
</tr>
<tr>
<td>1.3 XML-QL</td>
<td>5</td>
</tr>
<tr>
<td>1.4 XSL Patterns</td>
<td>11</td>
</tr>
<tr>
<td>1.5 XQuery</td>
<td>14</td>
</tr>
<tr>
<td>1.6 Motivations</td>
<td>16</td>
</tr>
<tr>
<td>1.7 Organization of the Thesis</td>
<td>17</td>
</tr>
<tr>
<td>2. A Survey</td>
<td>18</td>
</tr>
<tr>
<td>2.1 The Object Exchange Model</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Extracting Indexing Information from XML DTDs</td>
<td>19</td>
</tr>
<tr>
<td>2.2.1 Key Ideas</td>
<td>19</td>
</tr>
<tr>
<td>2.2.2 Classification of DTD Elements</td>
<td>20</td>
</tr>
<tr>
<td>2.2.3 DTD Automata</td>
<td>21</td>
</tr>
<tr>
<td>2.3 DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases</td>
<td>22</td>
</tr>
<tr>
<td>2.3.1 Existence of Multiple DataGuides</td>
<td>24</td>
</tr>
<tr>
<td>2.4 Four Different Types of Index Structures</td>
<td>26</td>
</tr>
<tr>
<td>2.4.1 Value Index (Vindex)</td>
<td>26</td>
</tr>
<tr>
<td>2.4.2 Text Index (Tindex)</td>
<td>26</td>
</tr>
<tr>
<td>2.4.3 Link Index (Lindex)</td>
<td>27</td>
</tr>
</tbody>
</table>
3. Query Types .................................................. 29

4. Query Transformation ........................................... 40
   4.1 System Architecture ......................................... 40
   4.2 System Flowchart ............................................ 41
   4.3 DTD of Movie.xml ........................................... 42
   4.4 Constructing an XML Query .................................. 46
   4.5 Constructing a SQL Query .................................... 50
   4.6 Examples ..................................................... 56

5. The Implementation of Index Strategies ......................... 66
   5.1 Index Constructing ........................................... 66
       5.1.1 Constructing the Value Index ......................... 67
       5.1.2 Constructing the Text Index ......................... 67
       5.1.3 Constructing the Link Index ......................... 70
       5.1.4 Constructing the Path Index ......................... 73
   5.2 Query Processing by Indexes ................................. 73
   5.3 Performance Study ........................................... 82
       5.3.1 The Performance Model ................................. 82
       5.3.2 Simulation Results .................................... 83

6. Conclusion ..................................................... 86
   6.1 Summary ..................................................... 86
   6.2 Further Research Work ...................................... 87

BIBLIOGRAPHY ..................................................... 88
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 An XML document: Catalog</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Parts of XML document</td>
<td>4</td>
</tr>
<tr>
<td>1.3 The DTD</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Limitations of well-formed XML documents</td>
<td>6</td>
</tr>
<tr>
<td>1.5 An example of a valid XML</td>
<td>7</td>
</tr>
<tr>
<td>1.6 bookDTD</td>
<td>8</td>
</tr>
<tr>
<td>1.7 Query 1: Matching data using patterns</td>
<td>8</td>
</tr>
<tr>
<td>1.8 Query 2: Constructing XML data</td>
<td>9</td>
</tr>
<tr>
<td>1.9 An XML document</td>
<td>10</td>
</tr>
<tr>
<td>1.10 Query result for Query 2</td>
<td>10</td>
</tr>
<tr>
<td>1.11 Query 3: Grouping with nested queries</td>
<td>10</td>
</tr>
<tr>
<td>1.12 Grouping result for Query 3</td>
<td>11</td>
</tr>
<tr>
<td>2.1 An OEM Database</td>
<td>19</td>
</tr>
<tr>
<td>2.2 An XML document</td>
<td>20</td>
</tr>
<tr>
<td>2.3 A DTD automaton</td>
<td>22</td>
</tr>
<tr>
<td>2.4 A classification tree and a classification table</td>
<td>23</td>
</tr>
<tr>
<td>2.5 A sample OEM database</td>
<td>24</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>2.6 A DataGuide for Figure 2.5</td>
<td>25</td>
</tr>
<tr>
<td>2.7 A source and two DataGuides</td>
<td>25</td>
</tr>
<tr>
<td>3.1 Movie.xml</td>
<td>30</td>
</tr>
<tr>
<td>3.2 The relationships between tables</td>
<td>31</td>
</tr>
<tr>
<td>3.3 The relationships between tables (containing data)</td>
<td>32</td>
</tr>
<tr>
<td>3.4 The value tables</td>
<td>33</td>
</tr>
<tr>
<td>3.5 The value tables (continued)</td>
<td>34</td>
</tr>
<tr>
<td>3.6 This index supporting the complete path query</td>
<td>35</td>
</tr>
<tr>
<td>3.7 This index supporting the specified leaf only query</td>
<td>36</td>
</tr>
<tr>
<td>3.8 This index supporting the Specified intrapath query</td>
<td>37</td>
</tr>
<tr>
<td>3.9 This index supporting the Specified attribute/element (value)</td>
<td>38</td>
</tr>
<tr>
<td>3.10 This index supporting the multiple paths with the same level</td>
<td>39</td>
</tr>
<tr>
<td>4.1 System architecture</td>
<td>41</td>
</tr>
<tr>
<td>4.2 System flowchart</td>
<td>42</td>
</tr>
<tr>
<td>4.3 DTD of Movie.xml</td>
<td>43</td>
</tr>
<tr>
<td>4.4 The DTD tree</td>
<td>44</td>
</tr>
<tr>
<td>4.5 System interface</td>
<td>45</td>
</tr>
<tr>
<td>4.6 An example</td>
<td>46</td>
</tr>
<tr>
<td>4.7 Procedure <em>Construct_XQL</em></td>
<td>47</td>
</tr>
<tr>
<td>4.8 Flowchart of the procedure to construct an XML query</td>
<td>49</td>
</tr>
<tr>
<td>4.9 Procedure <em>XQLFindNextNode</em></td>
<td>50</td>
</tr>
<tr>
<td>4.10 Flowchart of procedure <em>XQLFindNextNode</em></td>
<td>51</td>
</tr>
<tr>
<td>4.11 Procedure <em>Construct_SQL</em></td>
<td>52</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>4.12 Flowchart of the procedure to construct a SQL query</td>
<td>53</td>
</tr>
<tr>
<td>4.13 Procedure SQLFindNextNode</td>
<td>55</td>
</tr>
<tr>
<td>4.14 Flowchart of procedure SQLFindNextNode</td>
<td>56</td>
</tr>
<tr>
<td>4.15 Blocks A, B, C, D, E and F</td>
<td>57</td>
</tr>
<tr>
<td>4.16 Procedures ConstructSQLA, ConstructSQLB, ConstructSQLC, ConstructSQLD, ConstructSQUE and ConstructSQLF</td>
<td>58</td>
</tr>
<tr>
<td>4.17 Example 1</td>
<td>59</td>
</tr>
<tr>
<td>4.18 Example 2</td>
<td>60</td>
</tr>
<tr>
<td>4.19 Example 3</td>
<td>62</td>
</tr>
<tr>
<td>4.20 Example 4</td>
<td>63</td>
</tr>
<tr>
<td>4.21 Example 5</td>
<td>65</td>
</tr>
<tr>
<td>5.1 Movie.xml</td>
<td>68</td>
</tr>
<tr>
<td>5.2 The relationships between tables (containing data)</td>
<td>69</td>
</tr>
<tr>
<td>5.3 The hash table for the value index</td>
<td>69</td>
</tr>
<tr>
<td>5.4 The hash table for the text index</td>
<td>70</td>
</tr>
<tr>
<td>5.5 The hash table of the link index</td>
<td>71</td>
</tr>
<tr>
<td>5.6 The hash table for the link index (continued)</td>
<td>72</td>
</tr>
<tr>
<td>5.7 The hash table for the path index</td>
<td>74</td>
</tr>
<tr>
<td>5.8 The hash table for the path index (continued)</td>
<td>75</td>
</tr>
<tr>
<td>5.9 Flowchart of processing a XML query supported by indexes</td>
<td>76</td>
</tr>
<tr>
<td>5.10 Flowchart of processing an XML query supported by the path index</td>
<td>77</td>
</tr>
<tr>
<td>5.11 Flowchart of query processing supported by the text index and the link index</td>
<td>80</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.12</td>
<td>Flowchart of query processing supported by the value index and the link index</td>
</tr>
<tr>
<td>5.13</td>
<td>Parameters used in the experience</td>
</tr>
<tr>
<td>5.14</td>
<td>The parameters and their settings</td>
</tr>
<tr>
<td>5.15</td>
<td>The database used in the simulation</td>
</tr>
<tr>
<td>5.16</td>
<td>The mean processing time of each query type without supported by indexing strategies</td>
</tr>
<tr>
<td>5.17</td>
<td>The mean processing time of each query type supported by index strategies</td>
</tr>
<tr>
<td>5.18</td>
<td>A comparison of the mean processing time between two strategies</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table               Page
CHAPTER 1
Introduction

XML is a new standard that supports data exchange on the World-Wide Web [21]. It is likely to become as important and widely used as HTML. There are many applications of XML. One important application is interchange of electronic data (EDI) between two or more data sources on the web. This problem is increasingly important as more businesses choose to provide access to their databases and to exchange data with related business and organizations. One of XML’s benefits is its simplicity. An XML document is a sequence of elements, each consisting of free text and/or other elements. The only restriction is that element tags must match, e.g., each <ADDRESS> must have a matching < /ADDRESS>, and must nest properly. An XML document that has matching and properly nested tags is called well-formed. Given its flexibility, it’s likely that XML will facilitate exchange of huge amount data on the Web, just as HTML enabled vast numbers of documents. Dozens of applications of XML already exist, including a Chemical Markup Language for exchanging data about molecules and the Open Financial Exchange for exchanging financial data between banks or banks and customers. However, the availability of huge amounts of XML data poses several technical questions that the XML standard does not address. In particular,

1. How will data be extracted from large XML documents?
2. How will XML data be exchanged, e.g., by shipping XML documents or by shipping queries?

3. How will XML data be exchanged between user communities using different but related ontologies (or DTD’s)?

4. How will XML data from multiple XML sources be integrated?

Data extraction, transformation, and integration are all well-understood database problems. Their solutions often rely on a query language, either relational (SQL) or object-oriented (OOL). These query languages do not apply immediately to XML, because the XML data differs from traditional relational or object-oriented data. XML data, however, is very similar to a data model recently studied in the research community: the semistructured data model. Several research query languages have been designed and implemented for semistructured data.

1.1 XML

XML (Extensible Markup Language) is a new standard approved by W3C for data representation and exchange on the Internet. Although XML succeeds HTML in time, its design is based on SGML, which predates HTML and the web altogether. XML is a hierarchical data format for information exchange in the World Wide Web. XML tags describe the data itself. That is, XML data is self-describing; therefore, it is possible for programs to interpret the data. An XML document consists of nested element structures, starting with a root element. Each element has a tag associated with it. The Element data can be in the form of attributes or sub-elements. Figure 1.9 shows an XML document that contains information about the plant catalog. In this example, there is a CATALOG element which has one sub-element, PLANT. The PLANT element has a BESTSELLER attribute with “no” and is further nested to provide some information. Alternatively, an XML document contains two parts of documents, a prolog and document instance as shown in Figure 1.2. The DTD document could be declared in the prolog of XML document or not. Moreover,
Figure 1.1 An XML document: Catalog

an XML document can have an associated DTD (Document Type Definition) which describes the structure of XML documents and are like a schema for XML documents. This kind of the XML document that satisfies the constraint of a DTD is said to be valid with respect to that DTD. The DTD is optional, but a powerful feature of XML that provides a formal set of rules to define a document structure. The DTD establishes formal document structure rules. It defines the elements and the attributes of elements that may be used, and dictates where the may be applied in relation to each other. It therefore specifies the document hierarchy.

A DTD specifies the structure of an XML element by specifying the names of its sub-elements and attributes. Sub-element structure is specified using the operators, + (set with one or more elements), * (set with zero or more element), ? (optional), | (or). All values are assumed to be string values, unless the type is ANY in which case the can be an arbitrary XML fragment. The id attribute uniquely identifies an element within a document and can be referenced through an IDREF field in another element. IDREFs are untyped. Finally, there is no concept of a root of a DTD - an XML document conforming to a DTD can be rooted at any element specified in the DTD. Figure 1.3 shows the DTDs. However, DTDs have some disadvantages.
DTDs have different syntax to the rest of the document. It is syntax different of XML document’s syntax and DTD has no data type. But DTD is a very important document.

However, an XML document without DTD can still be parsed. It is called well-formed XML. In summary, there are two kinds of XML documents:

- Well-formed: Data is not included of DTD as shown in Figure 1.9. The Well-formed XML documents have some limitations as shown in Figure 1.4
- Valid XML: Data is included of DTD as shown in Figure 1.5. A valid XML document must be a well-formed XML document.

The basic ideas underlying XML are very simple and flexible: tags on data elements identity the meaning of the data. Designed to meet the challenges of large-scale electronic publishing, XML will also play an increasingly important role in the exchange of a wide variety of data on the web.
1.2 Query Languages

XML is an extremely versatile markup language [8], capable of labeling the information content of diverse data sources including structured and semi-structured documents, relational databases, and object repositories. A query language that uses the structure of XML intelligently can express queries across all these kinds of data, whether physically stored in XML or view as XML via middleware. As increasing amounts of information are stored, exchanged, and presented using XML, the ability to intelligently query data sources becomes increasingly important. One of the great strengths of XML is its flexibility in representing many different kinds of information from diverse sources. To exploit this flexibility, an XML query language must provide features for retrieving and interpreting information from these diverse sources.

1.3 XML-QL

XML-QL [21] can express queries, which extract pieces of data from XML documents, as well as transformations, which, for example, can map XML data between
(1) The first row of an XML document must be the XML’s declaration.
   - <?xml version="1.0"?>
(2) A document must begin with a "root".
   - There is exactly one element, called the root
(3) The start tag and the end tag must be matched in pairs.
   - <book>Database System</book>
(4) The empty tag can occur.
   - <title></title> or <title/>
(5) Nest elements are allowed.
   - <i>Database System Lab.,<b>F5023</b></i> (O)
   - <i>Database System Lab.,<b>F5023</b></i> (X)
(6) English characters in uppercase are different from those in lowercase.
   - <Data>XML document</Data> (X)
(7) Attributes can have values.
   - <Book money="US">20</Book> (O)
   - <Book money='US'>20</Book> (O)
   - <Book money=US>20</Book> (X)
(8) Specific characters must be encoded carefully.
   - & --> &amp;
   - > --> &gt;
   - " --> &quot;
   - < --> &lt; 
   - &lt;Leech@leech@cse.nsysu.edu.tw&gt;</Email> (X)
   - <Email>Leech&amp;lt;leech@cse.nsysu.edu.tw&amp;gt;</Email> (O)

Figure 1.4 Limitations of well-formed XML documents
<?xml version="1.0"  encoding="big5" ?>
<!DOCTYPE computer [ 
<!ELEMENT computer(maker,brand,mouse,price+)>
<!ELEMENT maker (#PCDATA)> 
<!ELEMENT brand (#PCDATA)> 
<!ELEMENT mouse (#PCDATA)> 
<!ELEMENT price (#PCDATA)> 
<!ATTLIST brand logo CDATA #IMPLIED > 
<!ATTLIST price type CDATA #REQUIRED unit ( US | TW | AU | JP ) "JP"> ]> 
<computer>
    <maker>IBM</maker>
    <brand logo="aptiva.gif">Aptiva</brand>
    <mouse>Hello Kitty</mouse>
    <price type="dealer" unit="US">1000</price>
    <price type="list" unit="TW">10000</price>
</computer>

Figure 1.5 An example of a valid XML

DTDs and can integrate XML data from different sources. We introduce XML-QL through example queries. The examples show how XML-QL can extract data from XML documents and how it can construct new XML data. We then introduce XML-QL’s underlying data model. All query languages have an underlying data model that abstracts away from the physical representation of the data; for example, relational query languages operate on relations and object-oriented query languages on objects. Therefore, it is necessary to define precisely a data model for XML. For XML, a variant of the the semistructured data model is appropriate. Given this model, we then describe more advanced features of XML-QL, such as transforming and integrating XML data.

Examples in XML-QL. We introduce XML-QL through examples. This simplest XML-QL queries extract data from an XML document. Our example XML input is in www.a.b.c/bib.xml, and we assume that it contains bibliography entries that conform to the bookDTD as shown in Figure 1.6.

This bookDTD shown in Figure 1.6 describes a book element which contains one or more author elements, one title, and one publisher element and has a year attribute.
An article is similar, but its year element is optional, it omits the publisher, and it contains one shortversion or longversion element. An article also contains a type attribute. A publisher contains name and address elements, and an author contains an optional firstname and one required lastname, we assume that name, address, firstname, and lastname are all CDATA, i.e., string values.

Matching Data using Patterns. XML-QL uses element patterns to match data in an XML document. This example shown in Figure 1.7 produces all authors of books whose publisher is Addison-Wesley in the XML document at www.a.b.c/bib.xml. Any URI (uniform resource identifier) that represents an XML-data source may apper on the right-hand side of IN.

Informally, this query matches every <book> element in the XML document www.a.b.c/bib.xml that has at least one <book> element, one <author> element, and one <publisher> element whose <name> element is equal to "Addison – Wesley". For each such match, it binds the variables t and a to every title and author pair. The
WHERE <book>
   <publisher><name>Addison-Wesley</name></publisher>
   <title>$t</title>
   <author>$a</author>
</book> IN "www.a.b.c/bib.xml"
CONSTRUCT <result>
   <author>$a</author>
   <title>$t</title>
   <result>

Figure 1.8 Query 2: Constructing XML data

result is the list of authors bound to \( a \). Note that variable names are preceded by \( $ \) to distinguish them from string literals in the XML document (like Addison – Wesley).

**Constructing XML Data.** The query above returns a list of all authors \(< firstname > \< lastname >\) pairs from the input document. Often it is useful to construct new XML data in the result (i.e., data that did not exist in the input document). The query shown in Figure 1.8 returns both \(< author >\) and \(< title >\), and groups them in a new \(< result >\) element. For example, consider the XML data shown in Figure 1.9, when applied to the example data, our example query would produce the result shown in Figure 1.10.

**Grouping with Nested Queries.** The query shown in Figure 1.8 ungroups the authors of a book, i.e., different authors of the same book appear in different \(< result >\) elements. To group results by book title, we use a nested query, which produces one result for each title and contains a list of all authors. For the example data shown in Figure 1.9, the query shown in Figure 1.11 would produce the result shown in Figure 1.12.
Figure 1.9  An XML document

```
<result>
  <author><lastname>Date</lastname></author>
  <title>An introduction to Database Systems</title>
</result>

<result>
  <author><lastname>Date</lastname></author>
  <title>Foundation for Object/Relational Databases</title>
</result>

<result>
  <author><lastname>Darwen</lastname></author>
  <title>Foundation for Object/Relational Databases</title>
</result>
```

Figure 1.10 Query result for Query 2

```
  <title> $t</title>
  <publisher><name>Addison-Wesley</name></publisher> IN $p

CONSTRUCT <result>
  <title> $t</title>
  WHERE <author> $a</author> IN $p
  CONSTRUCT <author> $a</author>
</result>
```

Figure 1.11 Query 3: Grouping with nested queries
1.4 XSL Patterns

XSL Patterns is a version of Extensible Query Language (XQL) [1]. XSL Patterns describes how XQL is implemented in Microsoft Internet Explorer 5. Because XML is about data, and data is useful only if it is available, XSL Patterns was created to make XML document data more accessible. XSL Patterns is a general purpose language that is suited to many different applications. It was created to solve several types of problems. Some of these include:

- Performing queries within an XML document.
- Performing queries across a collection of documents.
- Addressing segments of data within a document or across documents.
- Querying within XSL style sheets.

Just like XSL, XSL Patterns is a declarative, not a procedural, language. That is, XSL Patterns queries specify what should be found in an XML document, not how to find it. This provides the application with much more flexibility to determine the most efficient method to use to find a piece of data.

The declarative nature of XSL Patterns follows the same philosophy of XML itself. XML is a declarative language, in which it indicates what the data is but does not
specify how it should look in a document. XSL Patterns, like XML, is a general language that can be used in many different ways, depending on the application.

XSL Patterns matches data in an XML document to a specified pattern, and then it returns the result of the pattern match through the XSL Patterns object model. XSL Patterns Language Syntax is described as follows:

**Context.** For an XSL Patterns query to work, you must give it the context in which it will operate. Context is the node or range of nodes on which the query will be focused. Remember that an XML document is structured in the form of a tree, and XSL Patterns can operate at the root level of that tree or at any branch level. Obviously, the context in which a query operates can vastly change the results. A query prefixed with '/' (forward slash) uses the root context. A query may optionally explicitly state that it is using the current context by using the './/' (dot, forward slash) prefix. Both of these notations are analogous to the notations used to navigate directories in a file system.

The '/' prefix is only required in one situation. A query may use the '//' operator to indicate recursive descent. When this operator appears at the beginning of the query, the initial '/' causes the recursive decent to perform relative to the root of the document or repository. The prefix '///' allows a query to perform a recursive descent relative to the current context. Let's see some examples.

- Find all *author* elements within the current context. Since the period is really not used alone, this example forward-references other features: .//author. Note that this is equivalent to: *author*.

- Find the *root* element (*bookstore*) of this document: /bookstore.

- Find all *author* elements anywhere within the current document: //author.

- Find all books where the value of the *style* attribute on the *book* is equal to the value of the *specialty* attribute of the *bookstore* element at the root of the document: *book[ /bookstore/@specialty = @style ].*
Results. The collection returned by an XQL expression preserves document order, hierarchy, and identity, to the extent that these are defined. That is, a collection of elements will always be returned in the document order without repeats, but there is no implicit order since attributes are by definition unordered.

Collections. The collection of all elements with a certain tag name is expressed using the tag name itself. This can be qualified by showing that the elements are selected from the current context './', but the current context is assumed and often need not be noted explicitly. Let’s see some examples.

- Find all first-name elements. These examples are equivalent: ./first-name and first-name.


Selecting children and descendants. The collection of elements of a certain type can be determined using the path operators ('/' or '///'). These operators take as their arguments a collection (left side) from which to query elements, and a collection indicating which elements to select (right side). The child operator ('/') selects from immediate children of the left-side collection, while the descendant operator ('///') select from arbitrary descendants of the left-side collection. In effect, the '///' can be thought of as a substitute for one or more levels of hierarchy. Note that the path operators change the context as the query is performed. By stringing them together, users can ‘drill down’ into the document. Let’s see some examples.

- Find all first-name elements within an author element. Note that the author children of the current context are found, and then first-name children are found relative to the context of the author elements: author/first-name.

- Find all title elements, one or more levels deep in the bookstore (arbitrary descendants): bookstore/title. Note that this is different from the following query, which finds all title elements that are grandchildren of bookstore elements: bookstore/*/title.
• Find *emph* elements anywhere inside book excerpts, anywhere inside the bookstore: *bookstore/book/excerpt/emph*.

• Find all *titles*, one or more levels deep in the current context. Note that this situation is essentially the only one where the period notation is required: *//title*.

**Collecting element children.** An element can be referenced without using its name by substituting the '*' collection. The '*' collection returns all elements that are children of the current context, regardless of their tag name. Let’s see some examples.

• Find all element children of author elements: *author/*.

• Find all *last-name* that are grandchildren of *books*: *book/*/last-name.

• Find the grandchildren elements of the current context: */*/.

• Find all elements with specialty attributes. Note that this example uses sub-queries, which are covered in filters, and attributes, which are discussed in finding an attribute: */[@specialty].

**Finding an attribute.** Attribute names are preceded by the '@' symbol. XQL is designed to treat attributes and sub-elements impartially, and capabilities are equivalent the two types wherever possible. Let’s see some examples.

• Find the *style* attribute of the current element context: @style.

• Find the *exchange* attribute on *price* elements within the current context: *price/@exchange*.

• The following example is not valid: *price/@exchange/total*.

### 1.5 XQuery

Because query languages have traditionally been designed for specific kinds of data, most existing proposals for XML query languages are robust for particular
types of data sources but weak for other types. A new query language called XQuery [8], which is designed to be broadly applicable across all types of XML data sources.

XQuery is derived from an XML query language called Quilt [9], which in turn borrowed features from several other languages. From XPath [10] and XQL [11], it took a path expression syntax suitable for hierarchical documents. From XML-QL [21], it took the notion of binding variables and then using the bound variables to create new structures. From SQL [12], it took the idea of a series of clauses based on keywords that provide a pattern for restructuring data (the SELECT-FROM-WHERE pattern in SQL). From OQL [13], it took the notion of a functional language composed of several different kinds of expressions that can be nested with full generality. Quilt was also influenced by other XML query languages such as Lorel [14] and YATL [15].

Like OQL, XQuery is a functional language in which a query is represented as an expression. XQuery supports several kinds of expressions, and the structure and appearance of a query may differ significantly depending on which kinds of expressions are used. The various forms of XQuery expressions can be nested with full generality.

The principle forms of XQuery expressions are as follows:

1. Path expressions.
2. Element constructors.
3. FLWR expressions.
4. Expressions involving operators and functions.
5. Conditional expression.
6. Quantified expression.
7. Expressions that test or modify datatypes.
1.6 Motivations

XML is rapidly becoming a popular data format. It can be expected that soon large volumes of XML document will exist. An XML document is either produced manually (like HTML document today), or it is generated by a new generation of software tools for WWW and/or EDI (Electric data interchange). So, we need to use the database technology to organize large amounts of XML documents. In general, numerous different options to store XML documents exist. In addition to relational database, XML can be stored in a file system, or an object-oriented database. A file system could be used with very little effort to store XML document, but a file system would not provide any support for querying the XML document. Object-oriented database systems would allow to cluster XML elements and sub-elements; this feature might be useful for certain applications, but the current generation of object-oriented database systems is not mature enough to process complex queries on large databases. Therefore, it is indeed required to use a standard commercial relational database system to evaluate powerful queries over XML documents.

Data extraction, transformation, and integration are all well-understood database problems. Their solutions often rely on a query language, either relational (SQL) or object-oriented (OOL). These query languages can not be applied immediately to XML data, because that the XML data differs from traditional relational or object-oriented data. XML data is a hierarchical and nested document; however, it is very similar to the semistructured data model. The characteristic of semistructured data is that it may not have a fixed schema and it may be irregular or incomplete.

As data in XML is an instance of a semistructured data model based on labeled-edge graph, it can be mapped to a semistructured data model and queries can be processed against it. Though the semistructured data model is flexible in data modeling, it requires a large search space in query processing since there is no schema fixed in advance. Indexing is the way of how to improve query performance efficiently; it allows fast access to data by essentially replicating portions of the database in special-purpose structures. However, due to the special properties of semistructured
data, there are up to five types of queries: (1) complete single path, (2) specified leaf only, (3) specified intrapath, (4) specified attribute/element(value), and (5) multiple paths with the same level. In this thesis, we classify all possible queries into those five query types. Next, we create different indexes for different query types. Moreover, we design and implement the query transformation from XML query statements to SQL statements. Also, we create a user-friendly interface for users to input XML query statements. The whole system is implemented in JAVA and SQL Server 2000. From our experiences, we show that our indexing strategies can improve the XML query processing performance very well.

1.7 Organization of the Thesis

This thesis is concerned with the design and implementation of indexing strategies for XML documents. The rest of the thesis is organized as follows. In Chapter 2, we give a survey of some previous proposed indexing techniques for XML documents. In Chapter 3, we classify all possible queries into several types, and create different indexes for each type of queries. In Chapter 4, we present the design and implementation of the query transformation from XML query statements to SQL statements. In Chapter 5, we present the design and implementation of the index strategies for different query types. Finally, Chapter 6 gives the conclusion and future work.
CHAPTER 2

A Survey

In this chapter, we give a survey of several strategies for the indexing between XML documents and relational databases.

2.1 The Object Exchange Model

The Object Exchange Model (OEM) is designed for semistructured data [2]. Data in this model can be thought of as a labeled directed graph. For example, the OEM graph shown in Figure 1 depicts a tiny portion of a database containing information about movies. (Although the example database is mostly tree-structured, OEM permits arbitrary graph-structured databases.) The vertices in the graph are objects; each object has a unique object identifier (OID), such as $5$. Atomic objects have no outgoing edges and contain a value from one of the basic atomic types such as integer, real, string, gif, java, audio, etc. All other objects may have outgoing edges and are called complex objects. Object $4$ is complex and its subobjects are $13$, $14$, $15$, and $16$. Object $5$ is atomic and has a value "Blade Runner". Names are special labels that serve as aliases for objects and as entry points into the database.
2.2 Extracting Indexing Information from XML DTDs

2.2.1 Key Ideas

From the flexibility of XML data, we can classify each element using DTDs, and give a hint to a query processor in run time [18]. For example, let's assume that a DTD declaration for the `person` element shown in Figure 2.2 is as follows:

$$
\text{<!ELEMENT person (name,e-mail*, (school | company))>}
$$

From the DTD, we can classify the `person` element into four groups:
Figure 2.2 An XML document

1. those who have one or more e-mail address and work for companies;

2. those who have no e-mail address and work for companies;

3. those who have one or more e-mail addresses and are students;

4. those who have no e-mail address and are students.

When each element is classified in this way, the search space can be reduced. For example, when the query that is related to students who have e-mails is processed, the nodes denoting persons who have no e-mail and work for companies need not be traversed.

### 2.2.2 Classification of DTD Elements

DTDs provide structural information about elements by regular expressions. So, they can classify DTD elements from DTDs. First, they make some assumptions about DTDs as in [17], i.e., XML documents always have DTDs, and do not have attribute other than the ID attribute, and so on. Let $N$ be a set of element names,
they abstract a DTD as a set of \((n : r)\) pairs, where \(n \in N\), \(r\) is either a regular expression over \(N\) or PCDATA which denotes a character string.

2.2.3 DTD Automata

They construct DTD automata from each regular expression \(r\) for the corresponding element \(n\) to classify elements in DTDs. When a regular expression has the form of \(r^+\), this means that a particular attribute or a composition of attributes exists more than once. This kind of information is not necessary in reducing the search space, because they should process all the attribute values when an attribute exists more than once. So the relaxed regular expression can extract only the necessary information during the query processing.

In [18], a relaxed regular expression is constructed from a given regular expression.

**Example 1.** The DTD declaration in formula (1) is abstracted to

\[
(person:(name, e-mail^*, (school | company))),
\]

and the corresponding relaxed regular expression is

\[
(person: (name, (e-mail | \perp), (school | company))).
\]

**Example 2.** Figure 2.3 shows an automaton constructed by algorithm for the person element in Example 2.2.3, and the algorithm’s name is the construction of DTD automata in [18].

**Example 3.** Figure 2.4 shows a classification tree and corresponding classification table constructed from the DTD automaton of the person element in Figure 2.3, using the construction of classification trees from DTD automata in [18].
2.3 DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases

The DataGuide is intended to be *concise*, *accurate*, and *convenient* summary of the structure of a database. Hereafter, we refer to a database that we summarize as the *source database*, or simply the *source*. We assume that a given source database is identified by its root object. To achieve *conciseness*, we specify that a DataGuide describes every unique label path of a source exactly once, regardless of the number of times it appears in that source. To ensure *accuracy*, we specify that the DataGuide encodes no label path that does not appear in the source. Finally, for *convenience*, we require that a DataGuide itself be an OEM object so we can store and access it using the same techniques available for processing OEM databases. The formal definition is as follows.

**Definition.** A *DataGuide* for a OEM source $s$ is an OEM object $d$ such that every label path of $s$ has exactly one data path instance in $d$, and every label path of $d$ is a label path of $s$.
Figure 2.4 A classification tree and a classification table

Figure 2.5 is the source OEM database, and Figure 2.6 shows a DataGuide for Figure 2.5. Using a DataGuide, we can check whether a given label path of length $n$ exists in the original database by considering at most $n$ objects in the DataGuide. For Example, in Figure 2.6, we need only examine the outgoing edges of objects 12 and 13 to verify that the path Restaurant.Owner exists in the database. Similarly, if we traverse the single instance of a label path $l$ in the DataGuide and reach some object 0, then the labels on the outgoing edges of 0 represent all possible labels that could ever follow $l$ in the source database. In Figure 2.6, the five different labeled outgoing edges of object 13 represent all possible labels that ever follow Restaurant in the source. Notice that the DataGuide contains no atomic values. Since a DataGuide is intended to reflect the structure of a database, atomic values are unnecessary.

A considerable theoretical foundation behind DataGuides can be found in [19]. That paper proved that creating a DataGuide over a source database is equivalent to conversion of a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA).
2.3.1 Existence of Multiple DataGuides

From automata theory, we know that a single NFA may have many equivalent DFAs. Similarly, as shown in Figure 2.7, one OEM source database may have multiple DataGuides. Figure 2.7 (b) and Figure 2.7 (c) are both DataGuides of the source in Figure 2.7 (a). Each label path in the source appears exactly once in each DataGuide, and do not exist in the source. Figure 2.7 (c) is in fact minimal: the smallest possible DataGuide [20]. Given the existence of multiple DataGuides for a source, it is important to decide what kind of DataGuide should be built and maintained in a semistructured database system. Intuitively, a minimal DataGuide might seem desirable, furthering our goal of having as concise a summary as possible; [19] also suggests building a minimal DataGuide. A minimal DataGuide is not always best, because incremental maintenance of a minimal DataGuide can be very difficult.
Figure 2.6 A DataGuide for Figure 2.5

Figure 2.7 A source and two DataGuides
2.4 Four Different Types of Index Structures

The first two identify objects that have specific values; the next two are used to efficiently traverse the database graph [3].

2.4.1 Value Index (Vindex)

A Vindex in a database is built over all atomic objects of base type integer or real, that have an incoming edge with a given label \( l \). This Vindex allows the query engine to quickly locate all objects reachable by an edge and matching a comparison predicate. While we could have chosen to support a single label-independent Vindex, a specific desired incoming usually is known at query processing time, so it is useful to partition the Vindex by labels. This approach also allows the administrator to build Vindexes selectively for frequently used labels.

Example 2.4.1. Suppose we create a Vindex for incoming label \( \text{price} \) over the database in Figure 2.1. If we perform a lookup for values > 15.00 with incoming edge \( \text{price} \), the result is \{&11,&15\}.

2.4.2 Text Index (Tindex)

A Vindex is useful for finding values that satisfy basic comparisons such as =, <, etc. However, for string values, an information-retrieval style keyword search can be very useful, especially for strings containing a significant amount of text. In these situations, the Vindex is not powerful enough and a different indexing structure, the Tindex is used.

Text indexes are implemented using inverted lists, which map a given word \( w \) and label \( l \) to a list of atomic values with incoming edge \( l \) that contain word \( w \). Like the Vindex, Tindexes are created by the administrator for a given label, for the reasons outlined earlier, but the label can always be omitted for a full search. A Tindex
lookup returns a list of postings, where each posting is of the form \( \langle o, n \rangle \). A posting indicates that \( w \) appears in object \( o \) as the \( n \)th word in the value, and \( o \) has an incoming edge labeled \( l \). The inverted lists are stored in hash tables on disk keyed on \( w \).

**Example 2.4.2.** Consider a \( Tindex \) lookup for all objects with an atomic string value containing the word "Ford" and an incoming edge \( Name \), performed over the database in Figure 2.1. The result is \( \langle &21, 2 \rangle \).

### 2.4.3 Link Index (Lindex)

The \( Lindex \) provides a mechanism for retrieving the parents of an object via a given label. A \( Lindex \) lookup takes a "child" object \( c \) and a label \( l \), and returns all parents \( p \) such that there is an \( l \)-labeled edge from \( p \) to \( c \). The \( Lindex \) also supports lookups with no label, in which case all parents and their labels are returned.

**Example 2.4.3.** Suppose we had located all atomic objects containing the word "Ford" via the \( Tindex \) lookup in Example 2.4.2, and we now wanted to traverse up to parent subobjects connected via the label \( Name \). The \( Lindex \) lookup for object \&21 returns parent object \&13.

### 2.4.4 Path Index (Pindex)

Find all objects reachable by a given labeled path through the database is an important part of query processing. A \( Pindex \) lookup for a path \( p \) returns the set of objects \( O \) reachable via \( p \).

**Example 2.4.4.** Suppose the query is simply "select DB.Movie.Title" applied over the database in Figure 2.1. Instead of exploring the graph, we can use the \( Pindex \) to directly locate all objects reachable via DB.Movie.Title. The \( Pindex \) lookup operation returns \{\&5,\&9,\&14\}.
While the $Pindex$ may appear very attractive, we cannot use it for all queries and path expressions: in addition to the limitations above. In some queries, we also must obtain the objects along a given path in addition to those at the end of the path. See [7] for details.
CHAPTER 3

Query Types

Based on the query languages as mentioned before, we can classify all possible queries into several query types as follows.

1. Complete single path.
2. Specified leaf only.
4. Specified attribute/element(value).
5. Multiple paths with the same level.

We use the Movie.xml shown in Figure 5.1 as our example data to illustrate those query types. Moreover, to show how to design an efficient index for each of those query types, we show the related relational tables for Movie.xml based on the mapping strategy in [4]. Figures 3.2 and 5.2 show the relationships between the primary key and the foreign key of each table and the corresponding relational tables are shown in Figures 3.4 and 3.5.

Example 1.
First, the following query 1 is an example of a complete single path query. For this query, the result is as follows. An index shown in Figure 3.6 can be used to support
<?xml version="1.0"?>
<DB>
  <Movie id="m1">
    <Title>The Rock</Title>
    <Actor>
      <Name>Nicolas Cage</Name>
      <Character>Standly</Character>
    </Actor>
    <Director>
      <Name>Michael Bay</Name>
    </Director>
    <Year>1996</Year>
    <Price>
      <Amount>20</Amount>
      <Currency>US</Currency>
    </Price>
  </Movie>
  <Movie id="m2">
    <Title>Spiderman</Title>
    <Actor>
      <Name>Tobey Maguire</Name>
      <Character>Peter Park</Character>
    </Actor>
    <Director>
      <Name>Sam Raimi</Name>
    </Director>
    <Year>2002</Year>
    <Price>
      <Amount>200</Amount>
      <Currency>TW</Currency>
    </Price>
  </Movie>
  <Movie id="m3">
    <Title>Star Wars</Title>
    <Actor>
      <Name>Mark Hamill</Name>
      <Character>Luke</Character>
    </Actor>
    <Director>
      <Name>George Lukas</Name>
    </Director>
    <Year>2001</Year>
    <Price>
      <Amount>25</Amount>
      <Currency>US</Currency>
    </Price>
  </Movie>
</DB>

Figure 3.1 Movie.xml
A is a table’s name

A’s primary key is to be B’s foreign key

Figure 3.2 The relationships between tables

this query, and this query is supported by Path index.

Query 1:

/DB/Movie/Title

Result:

Title ”The Rock”
Title ”Spiderman”
Title ”Star Wars”

Example 2.
Second, the following Query 2 is an example of a Specified leaf only query. For this query, the result is as follows. An index shown in Figure 3.7 can be used to support
this query, and this query is supported by Path index.

Query 2:

//Name

Result:

Name "Nicolas Cage"
Name "Tobey Maquire"
Name "Mark Hamill"
Name "Michael Bay"
Name "Sam Raimi"
Name "George Lukas"

Example 3.
<table>
<thead>
<tr>
<th>DB_ID</th>
<th>Movie_ID</th>
<th>id</th>
<th>DB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>m1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>m2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>m3</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title_ID</th>
<th>Movie_ID</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>The Rock</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>Spiderman</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
<td>Star Wars</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actor_ID</th>
<th>Movie_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>26</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name_ID</th>
<th>Actor_ID</th>
<th>Director_ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>NULL</td>
<td>Nicolas Cage</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>NULL</td>
<td>Tobey Maquire</td>
</tr>
<tr>
<td>27</td>
<td>26</td>
<td>NULL</td>
<td>Mark Hamill</td>
</tr>
<tr>
<td>8</td>
<td>NULL</td>
<td>7</td>
<td>Michael Bay</td>
</tr>
<tr>
<td>19</td>
<td>NULL</td>
<td>18</td>
<td>Sam Raimi</td>
</tr>
<tr>
<td>30</td>
<td>NULL</td>
<td>29</td>
<td>George Lukas</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Director_ID</th>
<th>Movie_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td>27</td>
<td>24</td>
</tr>
</tbody>
</table>

**Figure 3.4** The value tables
### Character

<table>
<thead>
<tr>
<th>Character_ID</th>
<th>Actor_ID</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
<td>Standly</td>
</tr>
<tr>
<td>17</td>
<td>15</td>
<td>Peter Park</td>
</tr>
<tr>
<td>28</td>
<td>26</td>
<td>Luke</td>
</tr>
</tbody>
</table>

### Year

<table>
<thead>
<tr>
<th>Year_ID</th>
<th>Movie_ID</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2</td>
<td>1996</td>
</tr>
<tr>
<td>20</td>
<td>13</td>
<td>2002</td>
</tr>
<tr>
<td>31</td>
<td>24</td>
<td>2001</td>
</tr>
</tbody>
</table>

### Price

<table>
<thead>
<tr>
<th>Price_ID</th>
<th>Movie_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>13</td>
</tr>
<tr>
<td>32</td>
<td>24</td>
</tr>
</tbody>
</table>

### Amount

<table>
<thead>
<tr>
<th>Amount_ID</th>
<th>Price_ID</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
<td>200</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>25</td>
</tr>
</tbody>
</table>

### Currency

<table>
<thead>
<tr>
<th>Currency_ID</th>
<th>Price_ID</th>
<th>Currency</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>US</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>TW</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>US</td>
</tr>
</tbody>
</table>

Figure 3.5 The value tables (continued)
Third, the following Query 3 is an example of a Specified intrapath query. For this query, the result is as follows. An index shown in Figure 3.8 can be used to support this query, and this query is supported by Path index.

**Query 3:**

\[/DB/Movie//Amount\]

**Result:**

- Amount "20"
- Amount "200"
- Amount "25"

**Example 4.**
Figure 3.7 This index supporting the specified leaf only query
Fourth, the following Query 4 is an example of a *Specified attribute element (value)* query. For this query, the result is as follows. An index shown in Figure 3.9 can be used to support this query, and this query is supported by Text index and Link index.

**Query 4:**

\[ DB/\text{Movie[@id='m1']}/\text{Year} \]

**Result:**

Year 1996

**Example 5.**

Fifth, the following Query 5 is an example of a *Multiple paths with the same level query*. For this query, the result is as follows. An index shown in Figure 3.10 can be used to support this query, and this query is supported by Text index and Link index.
Figure 3.9 This index supporting the *Specified attribute/element (value)*

**Query 5:**

\[ DB/\text{Movie}[\text{Title}='\text{The Rock}']/\text{Actor}/[\text{Year}] \]

**Result:**

Name "Nicolas Cage"
Character "Standly"
Year "1996"
Figure 3.10 This index supporting the *multiple paths with the same level*
CHAPTER 4

Query Transformation

In this Chapter, we present the design and implementation of the query transformation from XML query statements to SQL statements. The whole system is implemented in Java.

4.1 System Architecture

Figure 4.1 is our system architecture, it can be divided to 5 components, and they are described as follows:

1. Client: our client part is a desktop PC, users can input their XQL query statements through our interface, and those query statements will be transferred to the server part via the World Wide Web.

2. XML Query: we choose the XQL as our XML query language.

3. Server: our server part listens to the client’s requests all the time. If, this XQL statement is not supported by index, the server part will transform those XQL statements to SQL statements and transfer those SQL statements to the relational database. If this XQL statement is supported by index, the server part will transfer the XQL statement to XML index database.

4. RDBMS: we choose Microsoft SQL Server 2000 to be our relational database.
Figure 4.1  System architecture

5. XML Index Database: we also choose Microsoft SQL Server 2000 to be our XML index database.

4.2 System Flowchart

Figure 4.2 is the system flowchart of the proposed approach. When the user inputs XML queries through the interface, our system will construct XML-Query statements first. Next, if those queries are supported by the index, our system will catch the index from XML Index Database and get the query results. On the other hand, if those XML queries are not supported by the index, they will be transformed to SQL statements, and then, these SQL statements will be transferred to the relational database and get query results.
4.3 DTD of Movie.xml

Figure 4.3 is the DTD of Movie.xml. The DTD tree which we build is shown in Figure 4.4, where each element or attribute of DTD of movie.xml appears only once in the DTD tree. We record the relationship between elements and attributes in the DTD tree. For example, element DB is the root; element Movie is DB’s child; element Movie has 6 children: one is id, which is an attribute, the others are Title, Actor, Director, Year and Price, which are elements.

Figure 4.5 is the system interface. It has four parts. Its left part is a tree; the user can input the XML query by selecting some nodes on the tree. Its right part is used to show the query results which are returned from the relational database and the XML Index database. The top part will show the XQL statement of the user chooses some icon from the XML tree. The bottom part will show the transformed
SQL statement for the given XQL statement; The advantage of this interface is that users do not need to know the XML query language syntax, the only thing which they need to do is to select those elements and attributes which they are interested in the left part of the interface. Then, they will get the query results in the right part of the interface.

When the user chooses the XQL button on the interface, our system will show the corresponding XML Query Statement at the XQL field. On the other hand, when the user chooses the SQL button on the interface, our system will show the corresponding SQL statement at the SQL statement area. If the user chooses the Query button on the interface, our system will show the query results at the right part of our interface. If the user chooses the Index button on the interface, our system will show the XML query result supported by the index at the right part of our interface. The Reset button is to reset our system to initial status, and the user can input the other XML query.

For example, if the user wants to know all actors’ names in the movie database, first, he (she) needs to select the following elements on the tree in turn - DB, Movie, Actor, and Name. Then, the all actors’ names in the movie database, Nicolas Cage,
Figure 4.4 The DTD tree
Figure 4.5 System interface
Figure 4.6 An example

*Tobey Maguire*, and *Mark Hamill* are shown at the right part of our interface. This example is shown in Figure 4.6.

### 4.4 Constructing an XML Query

When the user inputs a query by selecting some nodes on the interface, first, we store these nodes into an array `Nodes` (the size of array of `Nodes` is `N`, `N` is the number of nodes) in sequence of root to leaf. Second, we call procedure `Construct_XQL` shown in Figure 4.7 to construct XQL query statements.
Procedure `Construct_XQL`;

```
var
  Integer N;
  array Nodes[N] ;
  // user selects N nodes through the interface
  String XQL = "";
  //XQL is the constructed XML Query statement
  Integer i = 0;
begin
  Load the DTD tree;
  if (Nodes[N] is not empty) then
    begin
      if (Nodes[i] is the root) then
        XQL = XQL+"/"+Nodes[i]
      else
        XQL=XQL+"/"+Nodes[i];
        XQLFindNextNode(Nodes[i]);
    end;
  end;
```
In procedure `Construct_XQL`, it has 4 steps. In step 1, it loads the *DTD tree*. Because the *DTD tree* contains the relationship between elements and attributes, which can help us to analyze the user’s queries. If array `Nodes` is empty, which means that the user does not select any node on our interface, then the whole procedure is terminated; otherwise, `Nodes[0]` is the first node which is selected by the user. Then, if `Nodes[0]` is the root element in the *DTD tree*, the process moves to step 2; otherwise, the process moves to step 3. In step 2, because `Nodes[0]` is the root of the *DTD tree*, it means that, the user queries the XML document starting from the root. So, we let the `XQL = XQL + "//" + Nodes[i]`. In step 3, because `Nodes[0]` is not the root of the *DTD tree*, it means that, the user queries the XML document starting from the middle level of the XML document. So, we let `XQL = XQL + "/" + Nodes[i]`. After the `Nodes[0]` has been analyzed over, the process moves to step 4. In step 4, we need to deal with the rest of nodes selected by the user. So, we call procedure `XQLFindNextNode` shown in Figure 4.9 to analyze the next node in array `Nodes`. Finally, when all of nodes in array `Nodes` are analyzed, the string, `XQL`, is the user’s XML query through our interface. The flowchart of the procedure to construct an XML query is shown in Figure 4.8.

In procedure `XQLFindNextNode`, it has 5 steps. In step 1, if `Nodes[i]` is the last node in array `Nodes`, the whole procedure is terminated; otherwise, the process moves to step 2. In step 2, we analyze the next node in array `Nodes`. If this node is not the former’s child and not the former’s sibling in the *DTD tree*, the process moves to step 3; otherwise, if this node is not the former’s child in the *DTD tree*, but it is the former’s sibling in the *DTD tree*, then the process moves to step 4. If this node is the former’s child in the *DTD tree*, and it is an attribute, then the process moves to step 4, too. If this node is the former’s child in the *DTD tree*, but it is not an attribute, then the process moves to step 5. In step 3, we let `XQL = XQL + "/" + Nodes[i]`. In step 4, we let `XQL = XQL + "][" + Nodes[i] + "]"`. In step 5, we let `XQL = XQL + "/" + Nodes[i]`. The flowchart of procedure `XQLFindNextNode` is shown in Figure 4.10.
Figure 4.8 Flowchart of the procedure to construct an XML query
Procedure XQLFindNextNode(Nodes[i]);
begin
while (i < N-1) do
begin
  i = i+1;
  if (Nodes[i] is Nodes[i-1]'s child) then
    if (Nodes[i] is an Attribute) then
      XQL=XQL+"["+Nodes[i]+"]"
    else
      XQL=XQL+"/"+Nodes[i];
  else
    if (Nodes[i] is Nodes[i-1]'s sibling) then
      XQL=XQL+"["+Nodes[i]+"]"
    else
      XQL=XQL+"//"+Nodes[i];
end;
end;

Figure 4.9 Procedure XQLFindNextNode

4.5 Constructing a SQL Query

Because our XML data is stored in the relational database already [4], we need to transform user's XML query on the interface to SQL query statements. We call procedure construct_SQL shown in Figure 4.11 to construct SQL query statements, and its flowchart is shown in Figure 4.12.

In procedure construct_SQL, it has 9 steps. In step 1, first, we load the DTD tree, and copy all the nodes in array Nodes to array TAB, because we do not want to alter array Nodes. If array TAB is empty, then the whole procedure is terminated; otherwise, we analyze every node in array TAB in turn and begin with the fist node. If this node is an element and it is the only node in array TAB, then the process moves to step 2. In step 2, we call procedure ConstructSQLA, it lets $SEL = SEL + TAB[i]$ and $FRO = FRO + TAB[i]$. Otherwise, if this node is an element and it is the last node in array TAB, the process moves to step 3. In step 3, we call procedure ConstructSQLB, it lets $SEL = SEL + TAB[i], FRO = FRO + "," + TAB[i]$, and
Figure 4.10  Flowchart of procedure $XQLFindNextNode$
Procedure Construct_SQL:
var
String: Query = "";
String: SEL = "SELECT";
String: FRO = "FROM";
String: WHE = "WHERE";
Integer i = 0;

begin
Load the DTD tree;
TAB[0..N-1] = Nodes[0..N-1]; //TAB[0..N] is equal to Nodes[0..N]
if (TAB[N] is not empty) then
begin
    while (TAB[i] is an element) do
begin
        if (i = N-1) then //if TAB[i] is the last node
        begin
            if (N = 1) then //TAB[N] has only noe node
                ConstructSQLA(SEL,FRO)
            else
                ConstructSQLB(SEL,FRO,WHE);
            if (TAB[i] is not a leaf node in the DTD tree) then
                TreeTravel(TAB[i])
            else
                Query = SEL+FRO+WHE;
        end
    elseif (i = 0) then //TAB[i] is the first node
        ConstructSQLC(FRO)
    elseif //TAB[i] is the middle node
        ConstructSQLD(FRO,WHE);
        SQLFindNextNode(TAB[i]);
    end;
end; (* while (TAB[i] is an element) *)
while (TAB[i] is an attribute) do
begin
    ConSQLF(SEL,FRO);
    if (i = N-1) then // if TAB[i] is the last node
    begin
        if (TAB[i] is not a leaf node in the DTD tree) then
            TreeTravel(TAB[i])
        else
            Query = SEL+FRO+WHE
    end;
else
begin
    TAB[i]=TAB[i].parent;
    //TAB[i] becomes TAB[i]'s parent in the DTD tree
    SQLFindNextNode(TAB[i]);
end;
end; (* while (TAB[i] is an attribute) *)
end; (* if (TAB[N] is not empty) *)
end;

Figure 4.11 Procedure Construct_SQL
Figure 4.12 Flowchart of the procedure to construct a SQL query
WHE = WHE + "and" + TAB[i-1].TAB[i-1].ID = TAB[i].TAB[i].ID. Otherwise, if this node is an element and it is the first node in array TAB, then the process moves to step 4. In step 4, we call procedure ConstructC, it lets FRO = FRO + TAB[i]. Otherwise, if this node is an element and it is a middle node in array TAB, then the process moves to step 5. In step 5, we call procedure ConstructD, it lets FRO = FRO + "," + TAB[i] and WHE = WHE + "and" + TAB[i-1].TAB[i-1].ID = TAB[i].TAB[i].ID. Otherwise, if this node is an attribute, then the process moves to step 6. In step 6, we call procedure ConstructF, it lets SEL = SEL + TAB[i] and FRO = FRO + "," + TAB[i-1]. After step 2 and 3 are finished, if this node is not a leaf node in the DTD tree, the process moves to step 9. In step 9, we call procedure TreeTravel, it lets this node to be the root and travels all its children. After step 4 and 5 are finished, the process moves to step 7. In step 7, we call procedure SQLFindNextNode shown in Figure 4.13. After step 6 is finished, if this node is the last node in array TAB, the process moves to step 8. In step 8, it lets TAB[i] = TAB[i]’s parent in the DTD tree, because we need to know which table in the database this attribute belongs to. After step 8 is finished, the process moves to step 7. Otherwise, if this node is the last node in array TAB and it is not a leaf node in the DTD tree, the process moves to step 9; otherwise, if this node is the last node in array TAB and it is a leaf node in the DTD tree, the process is terminated.

Procedure SQLFindNextNode is shown in Figure 4.13 and its flowchart is shown in Figure 4.14. It has 3 steps, In step 1, because this node is checked before, we check the next node in array TAB. If this node is not the former’s child in the DTD tree, the process moves to step 2; otherwise, the whole procedure is terminated. In step 2, we call procedure ConstructSQL, it lets FRO = FRO + "," + TAB[i] and WHE = WHE + "and" + TAB[i].TAB[i].parent.ID = TAB[i].parent.TAB[i-1].parent.ID. Then the process moves to step 3. In step 3, we let TAB[i] = TAB[i]’s parent in the DTD tree.
Procedure $SQLFindNextNode(TAB[i])$

begin

\[i = i + 1;\]

while \((TAB[i] \text{ is not } \text{TAB[i-1]’s child in the DTD Tree})\) do

begin

\[\text{ConSQL}(FRO, WHE);\]
\[\text{TAB}[i] = \text{TAB}[i].\text{parent};\]
\(\text{//TAB[i] becomes TAB[i]’s parent in the DTD Tree}\)

end;

end;

Figure 4.13 Procedure $SQLFindNextNode$

Blocks $A$, $B$, $C$, $D$, $E$ and $F$ in the flowchart of the procedure to construct a SQL query and in the flowchart of procedure $SQLFindNextNode$ are shown in Figure 4.15. Procedures $ConstructSQLA$, $ConstructSQLB$, $ConstructSQLC$, $ConstructSQLD$, $ConstructSQLE$, $ConstructSQLF$ are shown in Figure 4.16.
4.6 Examples

We give some query examples to describe the query transformation.

Example 1.
Query 1 is an example of a complete single path query. We choose the DB, Movie, and Title at the left part of our interface. They are from the root node to the leaf node in sequence. This query means that we want to know the Title of Movie in DB database. The query result is shown in Figure 4.17.

Query 1:

/DB/Movie/Title

SQL representation:

```
SELECT Title
FROM DB, Movie, Title
WHERE DB.db.ID = Movie.db.ID
AND Movie.movie_ID = Title.movie_ID;
```

Result:

Title ”The Rock”
Def:
String: SEL = "SELECT";
String: FRO = "FROM";
String: WHE = "WHERE";

A:
SEL = SEL+TAB[i];
FRO = FRO+TAB[i];

B:
SEL = SEL+TAB[i];
FRO = FRO+"","+TAB[i];
WHE = WHE+"and"+TAB[i-1].TAB[i-1]_ID = TAB[i].TAB[i]_ID;

C:
FRO = FRO+TAB[i];

D:
FRO = FRO+"","+TAB[i];
WHE = WHE+"and"+TAB[i-1].TAB[i-1]_ID = TAB[i].TAB[i-1]_ID;

E:
FRO = FRO+"","+TAB[i];
WHE = WHE+"and"+TAB[i].TAB[i].parent_ID = TAB[i].parent.TAB[i].parent_ID;

F:
SEL = SEL+TAB[i];
FRO = FRO+"","+TAB[i-1];

Figure 4.15 Blocks A, B, C, D, E and F
Procedure $ConstructSQLA(SEL,FRO)$;
begin
  SEL=SEL+TAB[i];
  FRO=FRO+TAB[i];
end;

Procedure $ConstructSQLB(SEL,FRO,WHE)$;
begin
  SEL=SEL+TAB[i];
  FRO=FRO+","+TAB[i];
  WHE=WHE+"and"+TAB[i-1].TAB[i-1].ID=TAB[i].TAB[i].ID;
end;

Procedure $ConstructSQLC(FRO)$;
begin
  FRO=FRO+TAB[i];
end;

Procedure $ConstructSQLD(FRO,WHE)$;
begin
  FRO=FRO+","+TAB[i];
  WHE=WHE+"and"+TAB[i-1].TAB[i-1].ID=TAB[i].TAB[i].ID;
end;

Procedure $ConstructSQLE(FRO,WHE)$;
begin
  FRO=FRO+","+TAB[i];
  WHE=WHE+"and"+TAB[i].TAB[i].parent_ID=TAB[i].parent.TAB[i].parent_ID;
end;

Procedure $ConstructSQLF(SEL,FRO)$;
begin
  SEL=SEL+TAB[i];
  FRO=FRO+","+TAB[i-1];
end;

Figure 4.16 Procedures $ConstructSQLA$, $ConstructSQLB$, $ConstructSQLC$, $ConstructSQLD$, $ConstructSQLE$ and $ConstructSQLF$
Figure 4.17 Example 1

Title "Spiderman"

Title "Star Wars"

Example 2.

Query 2 is an example of a specified leaf only query. We only choose the leaf node Name on the left part. This query means that we want to know all Names in DB database. The query result is shown in Figure 4.18.

Query 2:

//Name

SQL representation:

SELECT Name
FROM Name;

Result:
Name "Nicolas Cage"
Name "Tobey Maquire"
Name "Mark Hamill"

Example 3.
Query 3 is an example of a Specified intrapath query. We choose the three nodes DB, Movie, and Amount, and Amount is not the child of Movie. This query means that we want to know Amount of Movie in DB database. The query result is shown in
Query 3:

/DB/Movie//Amount

SQL representation:

SELECT Amount
FROM DB, Movie, Price, Amount
WHERE DB.db_ID = Movie.db_ID
AND Movie.movie_ID = Price.movie_ID
AND Price.price_ID = Amount.price_ID;

Result:

Amount 20
Amount 200
Amount 25

Example 4.

Query 4 is an example of a Specified attribute / element(value) query. This query means that we want to know the Director of Movie.id equals to ’m1’ in DB database. The query result is shown in Figure 4.20.

Query 4:

/DB/Movie[@id='m1']/Director

SQL representation:

SELECT Director
FROM DB, Movie, Director
WHERE DB.db_ID = Movie.db_ID
AND Movie.movie_ID = Director.movie_ID
AND Movie.id = ’m1’;

Result:

Director ”Michael Bay”

Example 5.

Query 5 is an example of a Multiple paths with the same level query. This query means that we want to know Actor and Year of Movie, where Title equals to ”The
Figure 4.19  Example 3
Figure 4.20  Example 4
Rock’ in DB database. The query result is shown in Figure 4.21.

Query 5:

\[ DB/Movie[Title='The Rock'][Actor][Year] \]

SQL representation:

SELECT Name, Character, Year
FROM DB, Movie, Title, Actor, Year, Name, Character
WHERE DB.db_ID = Movie.db_ID
    AND Movie.movie_ID = Title.movie_ID
    AND Title = 'The Rock'
    AND Movie.movie_ID = Actor.movie_ID
    AND Actor.actor_ID = Name.actor_ID
    AND Movie.movie_ID = Year.movie_ID
    AND Actor.actor_ID = Character.actor_ID;

Result:

Name "Nicolas Cage"
Character "Standly"
Year 1996
Figure 4.21  Example 5
CHAPTER 5

The Implementation of Index Strategies

In this Chapter, we describe the types of indexes that are built in our system and how they support our five different query types. We build four different types of index structures: the first two identify objects that have specific values; the next two are used to efficiently traverse the database.

1. A value index, or $Vindex$, locates atomic objects with certain values.

2. A text index, or $Tindex$, locates string type atomic values containing specific words.

3. A link index, or $Lindex$, locates parents of a specific object.

4. A path index, or $Pindex$, provides fast access to all objects reachable via a given labeled path.

5.1 Index Constructing

Figure 5.1 is Movie.xml, and our database is established from Movie.xml by the mapping strategy in [4]. Our database is shown in Figure 5.2. In our database, the name of each table is the name of the element in Movie.xml. For example, the table $Title$ in our database is the element $Title$ in Movie.xml. The primary key of each table is the table name concatenated with $ID$. For example, the primary key of the table

66
Title is Title.ID. The primary key of each table is unique in the whole database, so the primary key is like the element identifier in Movie.xml. Each table has a foreign key except for the table which is the root element in Movie.xml. The foreign key is the primary key of the other table which is the parent element in Movie.xml. For example, the table Title has the foreign key Movie.ID. If a table has an attribute name which is the same as the table name, it means that the table is a leaf element in Movie.xml.

5.1.1 Constructing the Value Index

A value index in our database is built over all elements consisting of the base type integer or real. The value index allows the query engine to quickly locate all values by the table name and matching a comparison predicate. We use the hash table structure to build the value index. We use the table name to be the key value, and the values of primary keys to be the hash values. In our database, tables Year and Amount are used to construct the value index, because they have an attribute name the same as its table name, and the data type of this attribute is integer. The hash table for the value index is shown in Figure 5.3.

5.1.2 Constructing the Text Index

A value index is useful for finding values that satisfy basic comparison such as <, =, etc. However, for string values, an information-retrieval style keyword search can be very useful. A text index in our database is built over all elements consisting of the string type. We use the hash table structure to build the text index, and use the table name to be the key value. In our database, tables Title, Name, Character, and Currency are used to construct the text index, because they have an attribute name the same as its table name, and the data type of this attribute is string. The hash table for the text index is shown in Figure 5.4.
<?xml version="1.0"?>
<DB>
  <Movie id="m1">
    <Title>The Rock</Title>
    <Actor>
      <Name>Nicolas Cage</Name>
      <Character>Standy</Character>
    </Actor>
    <Director><Name>Michael Bay</Name></Director>
    <Year>1996</Year>
    <Price>
      <Amount>20</Amount>
      <Currency>US</Currency>
    </Price>
  </Movie>
  <Movie id="m2">
    <Title>Spiderman</Title>
    <Actor>
      <Name>Tobey Maguire</Name>
      <Character>Peter Park</Character>
    </Actor>
    <Director><Name>Sam Raimi</Name></Director>
    <Year>2002</Year>
    <Price>
      <Amount>200</Amount>
      <Currency>TW</Currency>
    </Price>
  </Movie>
  <Movie id="m3">
    <Title>Star Wars</Title>
    <Actor>
      <Name>Mark Hamill</Name>
      <Character>Luke</Character>
    </Actor>
    <Director><Name>George Lukas</Name></Director>
    <Year>2001</Year>
    <Price>
      <Amount>25</Amount>
      <Currency>US</Currency>
    </Price>
  </Movie>
</DB>

Figure 5.1 Movie.xml
Figure 5.2 The relationships between tables (containing data)

Figure 5.3 The hash table for the value index
### 5.1.3 Constructing the Link Index

After we use the value index or the text index to get an element identifier, we need to find its parent element identifier. We construct the link index to provide a mechanism for retrieving the parent element identifier via a given element identifier and an element name. (Note that for a given attribute as the key value, then we record its own element identifier as the parent element identifier.) The link index lookup takes a "child" element $c$ and its element name $n$, and returns its parent $p$ such that there is an $n$-labeled edge from $p$ to $c$. In our system, the link index is implemented using hash table. The hash table for the link index is shown in Figures 5.5 and 5.6.
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DB</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>Movie</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>Movie</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>Movie</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Title</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>Title</td>
<td>13</td>
</tr>
<tr>
<td>25</td>
<td>Title</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>Actor</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>Actor</td>
<td>13</td>
</tr>
<tr>
<td>26</td>
<td>Actor</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>Name</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>Name</td>
<td>15</td>
</tr>
<tr>
<td>27</td>
<td>Name</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Name</td>
<td>7</td>
</tr>
<tr>
<td>19</td>
<td>Name</td>
<td>18</td>
</tr>
<tr>
<td>30</td>
<td>Name</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Character</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>Character</td>
<td>15</td>
</tr>
<tr>
<td>28</td>
<td>Character</td>
<td>26</td>
</tr>
</tbody>
</table>

A is the primary key
B is the table name or the attribute name
C is the foreign key

Figure 5.5 The hash table of the link index
A is the primary key
B is the table name or attribute name
C is the foreign key

Figure 5.6 The hash table for the link index (continued)
5.1.4 Constructing the Path Index

Finding all objects reachable by a given labeled path through the database is an important part of querying processing. The path index lookups for a path $p$ returns the set of objects $O$ reachable via $p$. In our system, the path index is implemented by using the hash table structure. We use the path to be the key. In our database, only the leaf elements contain values, so we record the paths from all internal elements to leaf elements. For example, "/DB/Movie/Title" is a path; it starts in element $DB$, through element $Movie$, and ends in leaf element $Title$. "/DB///Title" is the other path; it starts in element $DB$, and ends in leaf element $Title$. Figures 5.7 and 5.8 are the hash tables for the path index.

5.2 Query Processing by Indexes

When the user asks the XML queries supported by indexes, he (she) will choose the Index button on our interface. Those XML queries will be analyzed, if those XML queries contain [ ] clauses, they will be supported by the value index, text index and link index. Otherwise, they will be supported by the path index. The flowchart of the XML query supported by indexes is shown in Figure 5.9.

We give some query examples to describe the query supported by indexes. Examples 1, 2 and 3 are supported by the path index; Examples 4 and 5 are supported by the value index, the text index and the link index.

Example 1.
Query 1 is an example of a complete single path query. We choose the $DB$, $Movie$, and $Year$ at the left part of our interface. They are from the root node to the leaf node in sequence. This query means that we want to know the $Year$ of $Movie$ in $DB$ database. This query is supported by the path index. We use this query statement /DB/Movie/Year to be the key, and the results of IDs are 8, 18, and 28. Finally, we use these IDs to get the query results from our database. The flowchart of query
A is the path of XML
B, C, D are primary keys

Figure 5.7  The hash table for the path index
<table>
<thead>
<tr>
<th>Path</th>
<th>Node IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>/DB/Movie/Director/Name</td>
<td>8</td>
</tr>
<tr>
<td>/DB//Director/Name</td>
<td>19</td>
</tr>
<tr>
<td>//Movie/Director/Name</td>
<td>30</td>
</tr>
<tr>
<td>/DB//Movie/Price/Amount</td>
<td>11</td>
</tr>
<tr>
<td>/DB//Movie//Amount</td>
<td>22</td>
</tr>
<tr>
<td>/DB//Price/Amount</td>
<td>33</td>
</tr>
<tr>
<td>/DB//Amount</td>
<td></td>
</tr>
<tr>
<td>/Movie/Price/Amount</td>
<td>12</td>
</tr>
<tr>
<td>/Movie//Amount</td>
<td>23</td>
</tr>
<tr>
<td>/Price/Amount</td>
<td>34</td>
</tr>
<tr>
<td>/Amount</td>
<td></td>
</tr>
<tr>
<td>/DB/Movie/Price/Currency</td>
<td>9</td>
</tr>
<tr>
<td>/DB/Movie//Currency</td>
<td>20</td>
</tr>
<tr>
<td>/DB//Price/Currency</td>
<td>31</td>
</tr>
<tr>
<td>/DB//Currency</td>
<td></td>
</tr>
<tr>
<td>/Movie/Price/Currency</td>
<td></td>
</tr>
<tr>
<td>/Movie//Currency</td>
<td></td>
</tr>
<tr>
<td>/Price/Currency</td>
<td></td>
</tr>
<tr>
<td>/Currency</td>
<td></td>
</tr>
<tr>
<td>/DB/Movie/Year</td>
<td></td>
</tr>
<tr>
<td>/DB//Year</td>
<td></td>
</tr>
<tr>
<td>//Movie/Year</td>
<td></td>
</tr>
<tr>
<td>//Year</td>
<td></td>
</tr>
</tbody>
</table>

A is the path of XML
B, C, D are object IDs of leaf node in path A

Figure 5.8  The hash table for the path index (continued)
processing supported by the path index is shown in Figure 5.10.

**Query 1:**

```
/DB/Movie/Year
```

**Result:**

- Year 1996
- Year 2002
- Year 2001

**Example 2.**

Query 2 is an example of a *Specified leaf only query*. We only choose the leaf node *Director* on the left part. This query means that we want to know all *Director* in *DB* database. This query is still supported by path index. We use this query statement `/Director` to be the key, the IDs which the specified key is mapped in the path
Figure 5.10  Flowchart of processing an XML query supported by the path index

index are 7, 17, and 27. Finally, we use these IDs to get the query results from our database. The flowchart of query supported by the path index is shown in Figure 5.10.

**Query 2:**

`//Director`

**Result:**

Director "Michael Bay"
Director "Sam Raimi"
Director "George Lukas"

**Example 3.**

Query 3 is an example of a *Specified intrapath query*. We choose the three nodes *DB*,

- **DB**
- **C4**
- **C5**
Movie, and Amount, and Amount is not the child of Movie. This query means that we want to know Amount of Movie in DB database. This query is supported by the path index, too. We use this query statement to be the key, the results of IDs are 10, 20, and 30. Finally, we use these IDs to get the query results from our database. The flowchart of query processing supported by the path index is shown in Figure 5.10.

Query 3:
/DB/Movie/*Amount

Result:
Amount 20
Amount 200
Amount 25

Example 4.
Query 4 is an example of a Specified attribute/element(value) query. This query means that we want to know the Director of Movie.id equals to 'm1' in DB database. It is supported by the text index and the link index. In step 1, we analyze the clause [@id = 'm1']. Since the type of 'm1' is string, we use the text index to find its element ID. In step 2, we use @id to be the key, and the results of nodes are ('m1', 2), ('m2', 12), and ('m3', 22); we can find the value of the first field in the first node is 'm1', so the value of the second field is the element ID 2. In step 3, we use the link index to find its parent ID. We use the element ID 2 and element name @id as the key, and the query result of ID is 2. (Note that since @id is an attribute, so the "parent" ID is its own element ID). Because element ID 2 is not equal to -1, it means that @id is not the root element. We use the link index to find its ancestor ID. We use the parent ID 2 and element name Movie to be the key, and the query result of ID is 1. Since 1 is not still equal to -1, We use the link index again. We use the ancestor ID 1 and element name DB to be the key, and the query result of ID is -1. In step 4, we use the parent ID 2 and element name Director to construct a SQL statement: "SELECT Director FROM Director WHERE Movie_ID = 2;". The flowchart of query processing supported by the text index and the link index is shown in Figure 5.11.
Query 4:

/DB/Movie[@id='m1']/Director

Result:

Director "Michael Bay"

Example 5.

Query 5 is an example of a Multiple paths with the same level query. This query means that we want to know Title and Director of Movie, where Year equals to 2002 in DB database. This query is supported by the value index and the link index. In step 1, we analyze [Year = 2002]. Since the type of 2002 is integer, we use the value index to find its parent element ID. In step 2, we use Year to be the key, and the results of nodes are (1996, 8), (2001, 28), and (2002, 18). We can find the value of the first field in the third node is 2002, so the value of the second field is the element ID 18. In step 3, we use the link index to find the parent ID, the element ID 18 and element name Year is the key, and the result of ID is 12. Because element ID 12 is not equal to -1, it means that Year is not the root element. We use the link index to find the ancestor ID. We use the element ID 12 and element name Movie as the key, and the result of ID is 1. Since 1 is not still equal to -1, we use the link index again. This time, we use the ancestor ID 1 and element name DB to be the key, and the result of ID is -1. In step 4, we construct a SQL statement: "SELECT Title, Director FROM Title, Director WHERE Title.Movie_ID= 2 and Director.Movie_ID= 2;". The flowchart of query processing supported by the value index and the link index is shown in Figure 5.12.

Query 5:

DB/Movie[Year=2002][Title][Director]

Result:

Title "Spiderman"

Director "Sam Raimi"
Figure 5.11 Flowchart of query processing supported by the text index and the link index
Figure 5.12 Flowchart of query processing supported by the value index and the link index
### 5.3 Performance Study

In this section, we study the performance of XML query supported by our indexing strategies and make a comparison with query processing by transformation from XML query statements to SQL statements which is presented in Chapter 4 by simulation.

#### 5.3.1 The Performance Model

The parameters in the model are shown in Table 5.13. $N$ is the number of tables in our database; $R$ is the number of records in each table; $T$ is the query times of each query type; $H$ is the height of the XML hierarchy.

The parameters and their settings are shown in Table 5.14. The value of $N$ is 13, it means that there are 13 tables in our database as shown in Figure 5.15. The value of $R$ is 10,000, it means that there are 10,000 records in each table. The value of $T$ is 20, it means that we execute 20 times of each query type. The value of $H$ is 10, it means that the height of XML hierarchy is 10.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>13</td>
</tr>
<tr>
<td>$R$</td>
<td>10,000</td>
</tr>
<tr>
<td>$T$</td>
<td>20</td>
</tr>
<tr>
<td>$H$</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 5.14 The parameters and their settings

### 5.3.2 Simulation Results

The mean processing time of each query type without supported by indexing strategies is shown in Table 5.16. The mean processing time of each query type supported by indexing strategies is shown in Table 5.17. A comparison of the mean processing time between two strategies is shown in Figure 5.18. From our experiences, we show that our indexing strategies can improve the XML query processing performance very well.
Figure 5.15 The database used in the simulation

<table>
<thead>
<tr>
<th>Query Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean ProcessingTime (millisecond)</td>
<td>630</td>
<td>35</td>
<td>635</td>
<td>106</td>
<td>130</td>
</tr>
</tbody>
</table>

Figure 5.16 The mean processing time of each query type without supported by indexing strategies
Figure 5.17 The mean processing time of each query type supported by index strategies

<table>
<thead>
<tr>
<th>Mean Processing Time (millisecond)</th>
<th>Query Type 1</th>
<th>Query Type 2</th>
<th>Query Type 3</th>
<th>Query Type 4</th>
<th>Query Type 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>36</td>
<td>35</td>
<td>36</td>
<td>6</td>
<td>90</td>
</tr>
</tbody>
</table>

Figure 5.18 A comparison of the mean processing time between two strategies
CHAPTER 6

Conclusion

In this thesis, we have designed and implemented the indexing strategies for XML documents. In this Chapter, we give a summary of the thesis and point out some future research work.

6.1 Summary

XML is a popular standard for exchanging data on the World Wide Web. It can be expected that soon large volumes of XML document will exist. So we need to use the database technology to organized large amounts of XML documents. In any DBMS, the tradeoff between efficient query performance vs. space and update cost must be considered. Indexing allows fast access to data by essentially replicating portions of the database in special-purpose structures. However, these structures must be kept up to date incrementally: each change to the base data must be reflected in all applicable indexes. Despite of the cost of index maintenance, the added storage, and the added complexity in the query processing, indexes have shown themselves to be a useful and integral part of all database system.

XML is a hierarchical and nested document, it is very similar to the semistructured data model. As a result of these reasons, we classify many different query types. Moreover, the way of how to design an efficient index for each of those query types is not easy. There are a lot of techniques to process query and indexing for XML
documents. In this thesis, we have designed and implemented of Indexing strategies for XML documents.

In Chapter 2, we have given a survey of some previous proposed indexing technique for XML documents. In Chapter 3, we have classified all possible queries into several query types. Moreover, we have created different indexes for different query types. In Chapter 4, first, we have presented the design and implementation of the query transformation from XML query statements to SQL statements. Next, we have created a user-friendly interface to specify these different queries. In Chapter 5, we have presented the design and implementation of the indexes strategies for different query types.

6.2 Further Research Work

XML is a popular standard for exchanging data on the World Wide Web. It can be expected that soon large volumes of XML document will exist. When XML data has been stored in a relational database, the way of how to use the hash functions to retrieval XML data efficiently may be another good way to support it. Moreover, a signature file can act a search filter to prune most of the unsatisfactory data. Therefore, how to apply the hash functions and signatures to improve the performance is possible future work.
BIBLIOGRAPHY
BIBLIOGRAPHY


