A PERFORMANCE EVALUATION OF ALTERNATE NUMBERING BASED XML INDEXING TECHNIQUES

by

CHUL HO AHN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2006

ACKNOWLEDGEMENTS

I would like to express my deep and sincere gratitude to my supervising professor Dr. Ramez Elmasri who has incessantly motivated and guided me throughout my research. He showed me different approaches to problems and gave me priceless advice in a constant manner.

I also would like to say "Thank you indeed" to my committee members, Dr. Leonidas Fegaras and Dr. Gautam Das, for encouraging and giving me invaluable advice.

I am grateful to all of my friends in Arlington who spent time with me in discussing research, old friends in Korea who cheered me up all the time as well as colleagues in CSE Services at the University of Texas at Arlington.

I sincerely appreciate my parents and family who showed me deepest love and parents-in-law who supported and encouraged me in every possible way.

My special thanks go to my wife Mari, whose patient love and encouragement enabled me to complete this work.

April 12, 2006

ABSTRACT

A PERFORMANCE EVALUATION OF ALTERNATE NUMBERING BASED XML INDEXING TECHNIQUES

Publication No. _____

CHUL HO AHN, MS

The University of Texas at Arlington, 2006

Supervising Professor: Ramez Elmasri

Since XML became a standard of representing semi-structured data and exchanging format over the web, the sheer volume of XML data has become larger. While relational database represents data as a structured format, XML represents data in a self-describing way as a hierarchical tree structure. For expediting query processing over XML, many different types of indexing techniques have emerged.

We will focus on numbering based indexing techniques in this thesis. We will present performance comparison according to different XPath queries among three distinct numbering based XML indexing named GENE (Generic numbering based), XISS (Range based numbering), and XACC (Dimension based numbering) over shallow/deep tree structured data generated by ToXgene. By doing experiments, we realized that XACC showed relatively better query response in most of the cases. An analysis goes to three dimensions: varying size of the XML data, distinguished XPath queries having different features, and two different structures of XML data.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The eXtensible Markup Language (XML) is a regular text format language for representing and exchanging a wide variety of data over the web. XML emerged from SGML (ISO 8859) [7] and was designed to enhance the functionality of the internet by furnishing adaptable identification tags [8, 9, 10, 11]. Unlike fixed format of HTML, XML facilitates representing semi-structured data since it separates schema from data. Thus it is called *metalanguage* since we can define customized tags for describing various types of documents. XML data content is fundamentally composed of elements, attributes and character strings. XML data can be represented as a labeled tree or graph whether it has attributes or not. We can also have relationships between the nodes using attributes such as *ID* and *IDref* that are similar to *primary key* and *foreign key* in traditional Relational Database (RDB) systems.

Therefore we can represent any XML document as a labeled hierarchical tree whose *nodes* are elements and pointers showing relationships of parent and child with other nodes. The leaf nodes will contain text. Figure 1.1 shows a traditional XML document and corresponding tree-shaped representation.

To navigate tree structure we use regular path expressions. In many XML query languages, path expressions are the *building blocks* and are often used to traverse irregularly structured XML data where schemas are not populated [11]. To retrieve user-driven data from XML documents, many query languages have been proposed, among them, XQuery [12] and XPath [9] have become W3C standards. The XQuery language is designed to be widely applicable across all types of XML data from documents to databases

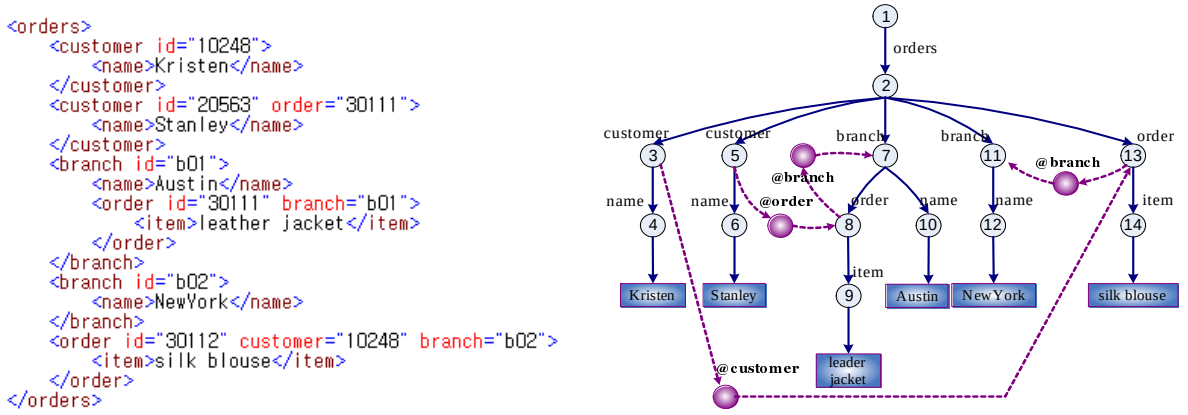


Figure 1.1. (a) An example of B2C XML data (b) A hierarchical tree structure of B2C XML data.

and object repositories. XPath was proposed to quickly locate any information that we need in an XML tree starting from the *context node* which stands for a root node in a subtree. It is obvious that naïve search of all paths in XML documents to obtain the result set of a query is inefficient because it will have massive access requests.

1.1 Background and Purposes

Recently numerous researchers have focused on efficient indexing techniques. The goal is to construct indexes that reduce the search space in order to speed up query processing without over-consuming system resources. Essentially XML indexing is different from relational database indexing because of the tree-structured model of XML. While RDB generate each index based on unique key values, diverse XML indexing techniques create some kinds of numbering schemes or patterns for determining parent-child relationships since XML data is tree-structured as well as semi-structured unlike RDB.

Currently databases storing and managing XML data are divided into two categories: XML-enabled database and native XML database. XML-enabled database stores XML data by converting into a specific format from XML. Native XML database keeps

original XML format. Also XML-enabled database extends the legacy systems of relational databases by supporting interchange between XML documents and relational data. It was designed to store and search data-centric XML documents. Most of the commercial RDBMS put XML data into database as CLOB (Character Large OBject) type to facilitate search. A Native XML database is specifically designed such that it efficiently supports transactions, security, multi-user access, programming API, and query languages. Furthermore, it can store data-centric as well as document-centric XML data in a way that it keeps sequence of documents, comments, CDATA section, and entity, etc. Examples of native XML databases are DOMSafeXML [13], Tamino [14], X-Hive/DB [15], Berkeley DB XML [16], Timber [17] and so forth. We surveyed various types of XML indexing techniques so that we can classify them into similar groups. The four major categories explored in this thesis are sequence based indexes, structure based indexes, numbering based indexes, and keyword based indexes [11]. We will shortly describe those four categories in Chapter 2. Among those categories we will focus on numbering based indexing techniques, specifically dimension based index (XACC), range-based index (XISS), and generic numbering-based index (GENE), in depth. For comparing performance of each indexing technique, we generated simulated XML data having shallow/deep tree structure using ToXgene [18] according to different sizes and distributions. Afterwards, we implemented generic numbering based indexing technique [8] and downloaded XISS/R [3, 2] and modified XPath accelerator [19, 20]. We selected some standard XPath queries for applying to shallow/deep tree structured XML data, and carried out performance evaluation among numbering-based indexing techniques. The performance analysis metrics include loading time of XML data, XPath querying time as well as reconstruction time. Further, we will discuss the reasons why each shows such results and performances.

The contributions of this thesis are as follows:

- Surveyed various types of XML indexing techniques.
- Refined categories of indexing techniques.
- Generated simulated XML data (shallow/deep tree structured) using ToXgene.
- Implemented generic numbering-based XML indexing technique.
- Proposed suitable XPath queries for shallow/deep XML data.
- Carried out performance evaluation among numbering-based indexing techniques and conducted an experimental result analysis.

This thesis is part of a project within "**Research of XML indexing techniques**". Therefore we expect that our contributions will facilitate the ongoing project of work.

1.2 Thesis Organization

The rest of the thesis is organized as follows: In Chapter 2, we describe different types of XML documents and how to query XML documents using XPath and XQuery. In Chapter 3, other related indexing techniques were introduced. How to generate simulated XML data using ToXgene was explained in Chapter 4. Properties of GENE[1], XISS[2], and XACC[3] were sufficiently described in order in Chapters 5, 6, and 7. Experimental results and in-depth analysis were presented in Chapter 8. Chapter 9 presents conclusion and future work.

[1]Generic numbering based indexing technique
[2]Range based numbering technique
[3]Dimension based numbering technique

CHAPTER 2

RELATED WORK

While surveying various XML indexing techniques, we were able to classify similar techniques into categories based on their properties. This chapter provides refined categorization of diverse XML indexing techniques. Further, we briefly summarize representative techniques namely A(k) in structure-based and PRIX in sequence-based.

2.1 Categorizing XML Indexing Techniques

Currently proposed XML indexing techniques are categorized as in Figure 2.1. Basic properties of XML data are hierarchical tree-structured and semi-structured unlike ordinary relational databases. With this in mind, in order to retrieve XML data efficiently we need different types of indexing techniques. We present refined categorization of various XML indexing techniques with four different types: structural-based, numbering-based, sequence-based, and keyword-based.

2.1.1 Structural-based Index

In [21, 22, 23, 24, 25, 26], we have a class of indexing techniques that construct accurate or approximate structural summaries of the semi-structured data. Based on the concept of *bisimilarity*, the nodes in an XML tree are grouped according to the local structure, for example, the incoming path of length of a certain parameter. These structural summaries are either accurate, being the true reflection of the structure of semi-structured databases, or approximate, where only paths shorter than a parameter are of significance. Examples of accurate structural summaries will be strong DataGuides

Structure-based	❍ DataGuide ❍ 1-Index ❍ A(k)-Index ❍ D(k)-Index ❍ M(k)-Index
Numbering-based	❍ XISS and XISS/R ❍ XPath Accelerator (XACC) ❍ Generic (GENE)
Sequence-based	❍ ViST ❍ PRIX
Keyword-based	❍ XRANK ❍ Keyword Proximity ❍ Integrating Keyword Search

Figure 2.1. Refined categories of XML Indexing Techniques.

[21] and 1-index [22]. Examples of approximate structural summaries are Approximate DataGuides [23], A(k)-index [24], D(k)-index [25], and M(k)-index [26]. Structural summaries can greatly speed up the processing of path expressions.

2.1.2 Numbering-based Index

Indexing techniques in this category were used to map the pre-order rank (or together with post-order rank) of a particular node on the XML tree to either a linear interval or coordinates on two-dimension plane. The result is then used to determine the relationships between tree nodes.

XISS [3] proposed a new system for indexing and storing XML data based on an extended pre-order numbering scheme for elements. This numbering scheme quickly determines the ancestor-descendant relationship between elements in the hierarchy of XML data. XPath Accelerator (XACC) [19, 27] maps all element and attribute nodes onto the 2-dimensional plane using its pre-order rank on the x-axis and its post-order rank on the y-axis.

2.1.3 Sequence-based Index

These techniques [28, 29, 30] transform structured XML data into sequences using different methods like Prüfers sequence, depth-first traversal or breadth-first traversal. This is also known as encoding of the XML document. The basic idea is to convert the XML data and the queries into sequences. Then one-to-one correspondence is created between the XML tree and the sequence. To perform the queries, subsequence matching is done. Each XML document is represented by a labeled tree. Each node has its element tag and a number. The number can be any unique number between 1 and the total number of nodes. Pre-order or post-order numbering scheme can be used. In some cases, a node and its position in the tree structure are represented by a pair (X, Y) where X stands for a label of the node and Y stands for its path in the tree.

2.1.4 Keyword-based Index

The work in [31, 32, 33] proposed an indexing that enables keyword search at the granularity of XML elements. Keyword search is also meaningful to query XML data, if the structures of XML data are not known to users. They focus on keyword search in which the users do not have to learn any schema or query language. They used the notion of proximity search to find the most relevant result. To use indexes for speeding up keyword search, the structure of inverted files is also extended to support full-text indexing with additional information of XML documents.

2.2 A(k) Index

The A(k) index is based on the observation that long and complex paths tend to contribute disproportionately to the complexity of an accurate structural summary in [24]. That is why A(k) index is in the category of *structure-based index*. When indexing XML data, we can store it in two ways as follows.

- Store full XML data.
- Store only summary structure that allows to retrieve data.

With the first method, once we store XML data into storage we do not need the original documents any more. In order to retrieve user-driven whole or partial data, reconstruction is required instead. On the other hand, we still need to keep original documents even if we store indexed data with the second method. Structure-based indexing denotes that XML query extracts substructure of the whole data graph. Substructures might be scattered over a graph representing XML data. Therefore using structural index, the search space can be efficiently decreased. Grouping nodes by labels but it still keeps paths and properties. It also associates a set of data nodes with each index node.

2.3 PRIX

PRIX (Indexing and Querying XML using Prüfers Sequences) [29] presents a new way of indexing XML documents and processing *twig patterns* in XML data. Every XML document in the XML database can be transformed into a sequence of labels by Prüfers method that builds a one-to-one correspondence between trees and sequences. During the query processing, a twig pattern is also transformed into its Prüfer sequences. We can find all the occurrences of a twig pattern in the database by executing subsequence matching and a series of refinement phases.

2.4 Summary

In this chapter, we briefly explained various XML indexing techniques so that we can group them into specific categories. Additionally, we briefly described A(k) and PRIX indexing techniques. Next we will present overview of XML and dominant query languages, XPath and XQuery.

CHAPTER 3

OVERVIEW OF XML AND XPATH/XQUERY

In this chapter we present overall concepts of XML and general types of XML documents. Further, we outline basic concepts of XML indexing. To retrieve XML and semi-structured data, several query languages have been proposed.

3.1 Different Types of XML Documents

There are significant structural and content differences among various XML data sets that lead to the classification of two types of XML documents: *data-centric* and *document-centric* [34, 35]. An XML document can also take the form of hybrid content, where parts of it are considered data-centric and other parts document-centric.

3.1.1 Data-Centric XML Documents

Data-centric documents are those where XML is used as a data exchange and transport medium. They are usually highly structured and marked up with XML tags. Such documents include sales orders, patient records, and scientific data. Thus it can be merely stored in a relational database or similar repository. An example shows sales transaction data [11].

3.1.2 Documents-Centric XML Documents

Document-centric XML documents are those in which XML is used for its SGML-like capabilities, such as in user's manuals, static web pages, and marketing flyers or brochures. They are characterized by loose, irregular structure and mixed content thus

```
<Class course="CSE5331">
    <Instructor id="002215">Dr. Elmasri</Instructor>
    <Student sid="000875643">
        <Name>John Doe</Name> is the best student in
        the class. He scored <Grade>40.0</Grade>
        points out of <Grade>40.0</Grade>. His
        research of <Paper>XML Indexing</Paper> is
        quite impressive.
    </Student>
    <Student sid="000875667">
    ...
</Class>
```

Figure 3.1. An example of hybrid-content XML document.

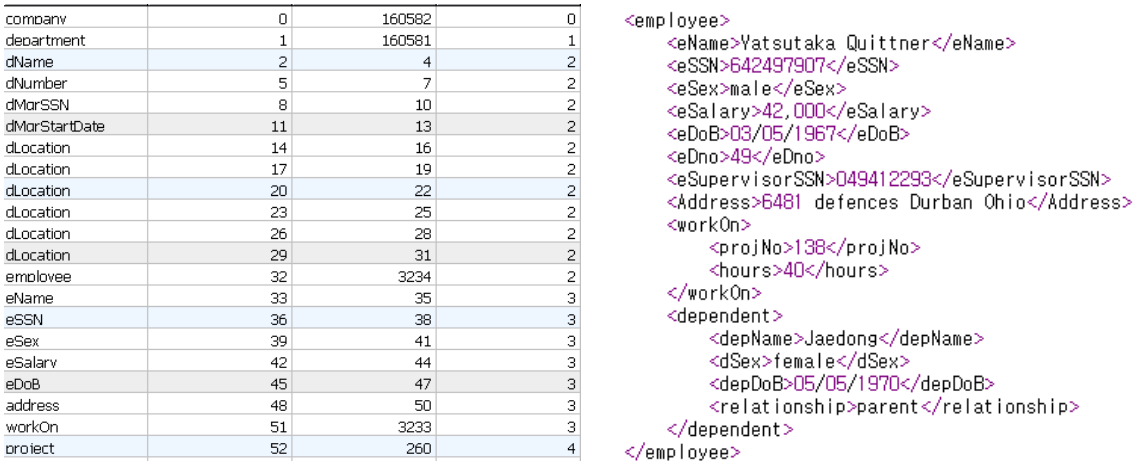
their structural sequence is crucial. Content management systems are typically the tool of choice when considering storing, updating and retrieving various XML documents in a shared repository [35] such as SyCOMAX, Content@ and Frontier, etc. An example would be SIGMOD record periodicals stored as XML documents [11].

3.1.3 Hybrid-Content XML Documents

An example of Hybrid-content XML documents is shown in Figure 3.1. Student elements are data-centric while student information is document-centric. In situations where both data-centric and document-centric models of XML documents will occur, the best data storage choice is usually a native XML database [35] such as Berkeley DB XML [16] and Timber [17].

3.2 Review of XML Indexing

We need completely different scheme for semi-structured data so that we can expedite query processing against XML data. Figure 3.2(a) shows an example of structured data stored into relational database. It has attributes and records. Thus we can formally describe it as a table structure. If it has unique key column (otherwise we still can create one), we can create B^+-tree index for speeding up query processing. However, in Figure 3.2(b), document contents are varying line by line, which is called self-describing. It has begin tag, end tag, and PCDATA. Even if we try to put this into RDB, it will break



company	0	160582	0
department	1	160581	1
dName	2	4	2
dNumber	5	7	2
dMgrSSN	8	10	2
dMgrStartDate	11	13	2
dLocation	14	16	2
dLocation	17	19	2
dLocation	20	22	2
dLocation	23	25	2
dLocation	26	28	2
dLocation	29	31	2
employee	32	3234	2
eName	33	35	3
eSSN	36	38	3
eSex	39	41	3
eSalary	42	44	3
eDoB	45	47	3
address	48	50	3
workOn	51	3233	3
project	52	260	4

```
<employee>
    <eName>Yatsutaka Quittner</eName>
    <eSSN>642497907</eSSN>
    <eSex>male</eSex>
    <eSalary>42,000</eSalary>
    <eDoB>03/05/1967</eDoB>
    <eDno>49</eDno>
    <eSupervisorSSN>049412293</eSupervisorSSN>
    <Address>6481 defences Durban Ohio</Address>
    <workOn>
        <projNo>138</projNo>
        <hours>40</hours>
    </workOn>
    <dependent>
        <depName>Jaedong</depName>
        <dSex>female</dSex>
        <depDoB>05/05/1970</depDoB>
        <relationship>parent</relationship>
    </dependent>
</employee>
```

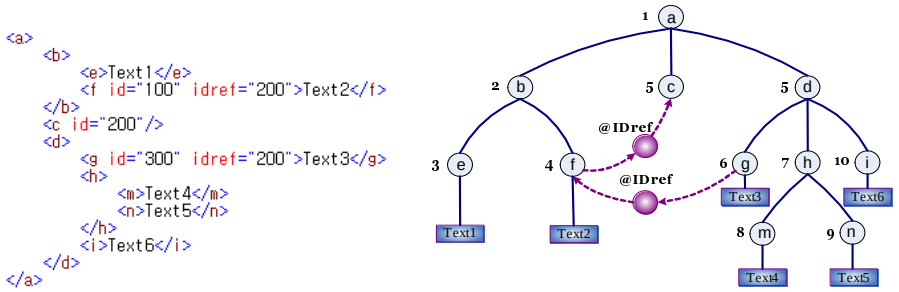
(a) (b)

Figure 3.2. (a) An example of structured data (b) An example of semi-structured data.

normalization. We will have many duplicated fields in each column that significantly cause a waste of disk space. This is why we need a special kind of indexing scheme for semi-structured data. As is mentioned in Chapter 2, numerous researchers are still going on toward how fast we can retrieve user-driven XML data. We are focusing on numbering-based XML indexing techniques among them.

3.3 XPath and XQuery for Querying XML Documents

An XML document is considered as a tree composed of nodes. Some nodes contain other nodes with proper nesting. There is one root node that contains all other nodes. When we deal with such XML data, we need a language for XML data for querying, transforming, integrating, and presenting it. There are three kinds of languages for satisfying these functions currently. Dominating ones are XSLT [36], XPath [9], and XQuery [37]. **XSLT** (eXtensible Stylesheet Language Transformation) essentially is used for extracting from and transforming into an XML document and its grammar follows typical XML structure. **XPath** is a language used for picking nodes and sets of nodes out of this tree. **XQuery** which was influenced by OQL [34] is based on XPath. It has



(a) An typical XML document

(b) A hierarchical tree structure of an example XML document

Figure 3.3. An example of an XML document and corresponding tree structure.

formal semantics based on the XML abstract data model. We will discuss XPath and XQuery rather than XSLT.

3.3.1 XPath

XPath is used for extracting a part of XML data from a single document. It uses *axis navigation* which express a single navigation path in an XML document. Currently many other languages are based on XPath. There are seven XPath node types:

- A root node
- Element nodes
- Attribute nodes
- Namespace nodes
- Processing instruction nodes
- Text nodes
- Comment nodes

An example of the XML document is illustrated in Figure 3.3(a). Generally XML documents are represented in a *hierarchical tree structure*. The tree representation of the above XML document is as illustrated in Figure 3.3(b) . XPath is a node-addressing expression language for XML. It is a set of syntax rules for defining parts of an XML document and uses path expressions to identify nodes in an XML document. It has multiple *directions* of traversal. XPath indicates nodes by absolute position, relative

position, type, content, and several other criteria. A location path is the most widely used in XPath expression. It can be absolute or relative, and results in a node-set. It uses location steps to identify a set of nodes in a document. This set may be empty, contain a single node, or contain several nodes. The location steps are evaluated one at a time from left to right. Location steps consist of two required parts and one optional part:

- An axis (defines a node-set relative to the current node, tells which direction to travel from the context node to next nodes)
- A node test (identifies a node within an axis) and one optional part
- Zero or more predicates (use expressions to modify the set of selected nodes)

The syntax for this is:

$$axisname\text{::nodetest}[predicate]$$

An example of an XPath expression is depicted in Figure 3.4(a).

3.3.2 XQuery

The XQuery is the first public working draft of a query language for XML released from World Wide Web Consortium (W3C) [3, 37]. It is designed to be thoroughly applicable across all types of XML data sources from documents to databases and object repositories. XQuery uses **FLWR** (For-Let-Where-Return) expression in order to retrieve data from XML document(s). It is essentially similar to Select-From-Where of relational SQL. The advantage of using FLWR expressions is that we can utilize nested expressions and apply to multiple XML documents. Since XPath uses single path expression, it cannot be applied to multiple documents. An simple example is depicted in Figure 3.4(b). XQuery was emerged from Quilt [37] and influenced by OQL [38]. A query in XQuery has more than one query expressions. XQuery 1.0 currently support predicates, node elements, FLWR expression, operators, function calls and aggregations. A result of XQuery is an instance of a XML query data model. FLWR expression creates several

```
/orders/customer[name="Stanley"]//phone
```

(a)

```
for $p document("auction.xml")/site/people/person
let $a := for $t in document("auction.xml")/site/cosed_auctions/closed_auction
          where $t/buyer/@person = $p/$@id
          return $t
return <item person="{$p/name/text()}">count($a)</item>
```

(b)

Figure 3.4. (a) An example of XPath expression (b) An example of FLWR expression.

bindings then it leads to a result node set while applying predicates. *For* clause creates bindings in terms of selected nodes whereas *Let* produces single binding. Nested enabling *For* clauses facilitate loop evaluation when a result sequence iterate. There are a few common properties between XQuery and relational SQL.

Table 3.1. Comparing XQuery1.0 versus XPath2.0 [5]

	Advantages	Drawbacks
XQuery 1.0	❍ Can express joins and sort ❍ Can manipulate sequences of values and nodes in arbitrary order ❍ Is easy to write user-defined functions including recursive ones ❍ Allows users to construct temporary XML results in the middle of a query, and then navigate into that ❍ Allows existential and universal quantification	❍ XQuery implementations are less mature than XSLT
XPath 2.0	❍ Provide convenient syntax for addressing parts of an XML document ❍ Can select a node out of an existing XML document or database	❍ Cannot create new XML ❍ Cannot select only part of an XML node ❍ Cannot use variables or *namespace* bindings ❍ Cannot work with date values, calculate the maximum of a set of numbers, or sort a list of strings

- Both provide projection and selection operator (SQL SELECT and XQuery RETURN).
- XQuery can combine multiple XML documents. SQL can combine multiple tables.
- Both allow function calls and user defined functions.
- Both use *WHERE* clause for filtering and *ORDER BY* clause for sorting.

As we have seen so far, XPath is simple enough to retrieve user-driven data from only one XML document while XQuery uses complex nested query expressions that enable users to query from multiple documents.

3.4 Summary

In this chapter, we provided concepts of different types of XML documents, briefly XML indexing, and XPath as well as XQuery. The common features of XPath and XQuery languages are the use of regular path expressions and the ability to extract information regarding the schema from the data. We summarized features of each query language in Table 3.1.

CHAPTER 4

[ToXgene]: SIMULATING REAL-WORLD-LIKE XML DATA

As new types of XML indexing technique were proposed, the need arose to compare them for evaluating performance and functions. In this type of test we are required to use suitably standardized XML data set as well as identical system environment. In this chapter, we describe various XML data generators and compare each of them. Then we show how to generate shallow and deep tree structured XML data set for our performance test.

4.1 Overview of ToXgene

ToXgene [18, 1] is a template-based (*tsl* file) XML generator for large collections of synthetic XML documents at the University of Toronto as a part of the ToX project. It is still under development cooperating with IBM. XML data is generated by a user defined template file named *TSL* (Template Specification Language) [18]. Figure 4.1 shows a general architecture of ToXgene. Niagdatagen [39] generates XML data by simply modifying parameters based on a tree-view from the University of Wisconsin-Madison. XML schema and real-world-like simulated data cannot be used in it. XMLgen [40] and Mbgen [41] were designed for specific performance tests, therefore it has limitations for user driven XML data. XMLgen was used for performance test for XMARK. XML data size is varying by scaling factor. MBXML [42, 41] (Micro-Benchmark for XML) was used by Michigan performance test. Depth and fan-out are two important structural parameters to the size of tree-structured data. Table 4.1 shows how each XML generator

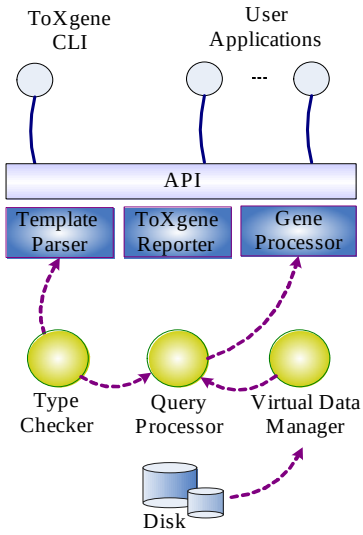


Figure 4.1. Architecture of ToXgene [1].

supports each of the functions. '❍' means completely support its function and '△' means partial support. '×' stands for not supported.

Table 4.1. Comparing functions among XML generators [6]

Functions	ToXgene	Niagdatagen	XMLgen	MBXML
Use of user text data	△	×	×	×
Control structure of XML data	❍	❍	×	×
Control the number of XML data	❍	❍	×	×
Use of XML schema	△	×	×	×
Control size of XML data	❍	×	❍	❍

We consequently chose ToXgene for generating XML data set since it can support all of the functions that we need.

4.2 Generating Template Specification File

ToXgene [18] is an extensible template-based data generator for XML. With template specification language (TSL) we can generate real-world-like simulated XML data sets. Its structure and syntax follow XML schema [43] specifications. We can specify probability distributions and CDATA content descriptors, used for generating both attributes and elements. When we pass TSL file to *Template Parser*, it will check validity and integrity then generate XML data (refer to Figure 4.1). Table 4.2 presents a portion of a TSL file.

As shown in Table 4.2, shallow XML data set has normal distribution for the number of dependents, the number of workers, and the birth dates. Hence, the number of dependents are represented by

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2} \qquad \text{where E(X)=2 and V(X)=2} \tag{4.1}$$

Then equation (4.1) can be rewritten as

$$f_X(x) = \frac{1}{\sqrt{4\pi}} e^{-(x-2)^2/4} \tag{4.2}$$

Other distributions can also be represented in a similar way. Using the TSL files, we generated shallow and deep structured XML data sets for our experiment. The results of the experiment give us significant meaning in that how combined XPath axes affect performance results even if we provide the same sizes of data set. This is the reason why we prepared two kinds of data sets. We will describe the properties of each data set in the following subsections.

Table 4.2. A portion of TSL file for shallow XML data

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<tox-template>
 <!-- enumerate different types of distribution -->
 <!-- generate normal distribution of the number of dependents -->
 <tox-distribution name="n1" type="normal" minInclusive="0" maxInclusive="5" mean="2" variance="2"/>
 <!-- generate exponential distribution of working hours of employees -->
 <tox-distribution name="e1" type="exponential" minInclusive="10" maxInclusive="40" mean="23"/>

 <tox-distribution name="c1" type="constant" minInclusive="1" maxInclusive="1"/>
 <tox-distribution name="e2" type="exponential" minInclusive="1" maxInclusive="10" mean="3"/>
 <tox-list name="department_list" unique="dName" readFrom="input/departments.xml">
  <element name="dName" type="string"/>
 </tox-list>
 <simpleType name="dName_type">
  <restriction base="string">
   <tox-sample path="[department_list/dName]">
    <tox-expr value="[!]"/>
   </tox-sample>
  </restriction>
 </simpleType>
 .........
 <tox-document name="output/company">
  <element name="company" minOccurs="1" maxOccurs="1">
   <complexType>
       <element name="department" minOccurs="10" maxOccurs="53">
        <complexType>
         <tox-scan path="[dept_list/department]" name="d">
          <element name="dName">
           <tox-expr value="[$d/dName]"/>
          </element>
  ........
  </element>
 </tox-document>
</tox-template>
```

4.3 Creating Shallow XML Dataset

A shallow data set contains different sizes of data. The size of data is varied from 500K, 1M, 2M, 3M, 5M, 10M, 20M, 50M, 100M bytes as shown in Table 4.3. Each of of the data sets has maximum level of depth 4 and identical DTD (Document Type Definition) as shown in Table 4.5. This simulated data is called *shallow* XML data because the tree depth is not very large. We increased the size of data in a way that we gradually increase the number of employees and the number of projects as shown

in Table 4.4. Figure 4.2 illustrates the graph representation of hierarchical structure of *shallow* XML data. The full TSL file for generating shallow tree is in Appendix B.

Table 4.3. Summary of shallow XML data set

Dataset Name	Size in Mbytes	# of Nodes	# of distinct labels	Max Depth
Shallow Tree	0.5	23,230	31	4
	1	47,406	31	
	2	92,496	31	
	3	136,711	31	
	5	230,312	31	
	10	467,257	31	
	20	946,860	31	
	50	2,382,685	31	
	100	5,220,799	31	

Table 4.4. Specification of shallow XML data set

Details					
# of department	# of employee	# of dependent	# of worker	# of workOn	# of project
[10..53]	[200..645]	[0..5]	[1..20]	[0..10]	[0..30]
[10..53]	[200..1400]	[0..5]	[1..20]	[0..10]	[0..30]
[10..53]	[200..2800]	[0..5]	[1..20]	[0..10]	[0..30]
[10..53]	[200..4150]	[0..5]	[1..20]	[0..10]	[0..30]
[10..53]	[200..7000]	[0..5]	[1..20]	[0..10]	[0..80]
[10..53]	[200..14,000]	[0..5]	[1..20]	[0..10]	[0..300]
[10..53]	[200..28,000]	[0..5]	[1..20]	[0..10]	[0..1000]
[10..53]	[200..68,000]	[0..5]	[1..20]	[0..10]	[0..5000]
[10..53]	[200..150,000]	[0..5]	[1..20]	[0..10]	[0..10,000]

4.4 Creating Deep XML Data set

A deep data set includes different sizes of data. Each has a size of data 100K, 200K, 400K, 500K, 700K, 1M, 2M bytes as shown in Table 4.6. Each of them has maximum level of depth 8 and identical DTD (Document Type Definition) as shown in

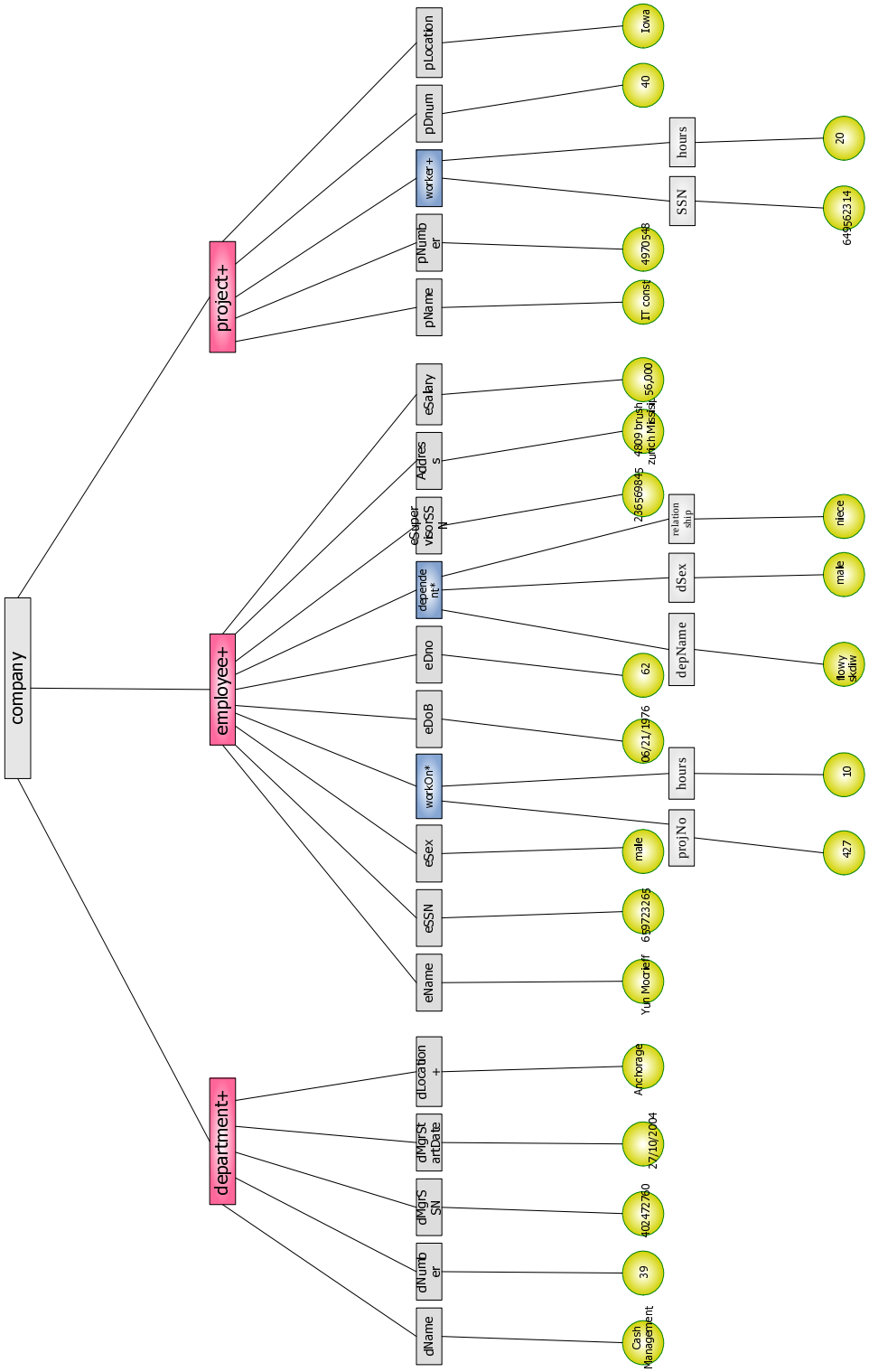


Figure 4.2. A tree structured graph of shallow XML data

Table 4.5. A DTD of shallow XML data

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DTD generated by Chulho Ahn-->
<!ELEMENT company (department+, employee+, project+)>
<!ELEMENT department (dName, dNumber, dMgrSSN, dMgrStartDate, dLocation+)>
<!ELEMENT employee (eName, eSSN, eSex, eSalary, eDoB, eDno, eSupervisorSSN, Address, workOn*, dependent*)>
<!ELEMENT project (pName, pNumber, pLocation, pDnum, worker+)>

<!ELEMENT dName (#PCDATA)>
<!ELEMENT dNumber (#PCDATA)>
<!ELEMENT dMgrSSN (#PCDATA)>
<!ELEMENT dMgrStartDate (#PCDATA)>
<!ELEMENT dLocation (#PCDATA)>

<!ELEMENT eName (#PCDATA)>
<!ELEMENT eSSN (#PCDATA)>
<!ELEMENT eSex (#PCDATA)>
<!ELEMENT eSalary (#PCDATA)>
<!ELEMENT eDoB (#PCDATA)>
<!ELEMENT eDno (#PCDATA)>
<!ELEMENT eSupervisorSSN (#PCDATA)>
<!ELEMENT Address (#PCDATA)>
<!ELEMENT workOn (projNo, hours)>
<!ELEMENT projNo (#PCDATA)>
<!ELEMENT hours (#PCDATA)>
<!ELEMENT dependent (depName, dSex, depDoB, relationship)>
<!ELEMENT depName (#PCDATA)>
<!ELEMENT dSex (#PCDATA)>
<!ELEMENT depDoB (#PCDATA)>
<!ELEMENT relationship (#PCDATA)>

<!ELEMENT pName (#PCDATA)>
<!ELEMENT pNumber (#PCDATA)>
<!ELEMENT pLocation (#PCDATA)>
<!ELEMENT pDnum (#PCDATA)>
<!ELEMENT worker (SSN, hours)>
<!ELEMENT SSN (#PCDATA)>
```

Table 4.8. This simulated data is called *deep* XML data because the corresponding tree is deep. We increased size of data in a way that we gradually increase the number of departments and the number of employees as shown in Table 4.7. Figure 4.3 illustrates graph representation of hierarchical structure of *deep* XML data. The full TSL file for generating deep tree is in Appendix B.

4.5 Summary

In this chapter, we described how we generated real-world-like simulated XML data set using ToXgene. ToXgene utilizes XML schema specifications to describe tem-

Table 4.6. Summary of deep XML data set

Dataset Name	Size in MBytes	# of Nodes	# of distinct labels	Max Depth
Deep Tree	0.1	5,187	26	8
	0.2	10,855	26	
	0.4	20,710	26	
	0.5	25,804	26	
	0.7	35,784	26	
	2	102,708	26	

Table 4.7. Specification of deep XML data set

Details					
# of department	# of employee	# of dLocation	# of worker	# of workOn	# of project
[1]	[4]	[3..10]	[10..20]	[1]	[10..20]
[1]	[7]	[3..10]	[10..20]	[1]	[10..20]
[1]	[15]	[3..10]	[10..20]	[1]	[10..20]
[1]	[18]	[3..10]	[10..20]	[1]	[10..20]
[1]	[25]	[3..10]	[10..20]	[1]	[10..20]
[1]	[35]	[3..10]	[10..20]	[1]	[10..20]
[2]	[36]	[3..10]	[10..20]	[1]	[10..20]

Table 4.8. A DTD of deep XML data

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DTD generated by Chulho Ahn-->
<!ELEMENT company (department+)>
<!ELEMENT department (dName, dNumber, dMgrSSN, dMgrStartDate, dLocation+, employee+)>
<!ELEMENT dName (#PCDATA)>
<!ELEMENT dNumber (#PCDATA)>
<!ELEMENT dMgrSSN (#PCDATA)>
<!ELEMENT dMgrStartDate (#PCDATA)>
<!ELEMENT dLocation (#PCDATA)>
<!ELEMENT employee (eName, eSSN, eSex, eSalary, eDoB, address, workOn)>
<!ELEMENT eName (#PCDATA)>
<!ELEMENT eSSN (#PCDATA)>
<!ELEMENT eSex (#PCDATA)>
<!ELEMENT eSalary (#PCDATA)>
<!ELEMENT eDoB (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT workOn (project+)>
<!ELEMENT project (hours, pName, pNumber, pLocation, pDnum, worker+)>
<!ELEMENT hours (#PCDATA)>
<!ELEMENT pName (#PCDATA)>
<!ELEMENT pNumber (#PCDATA)>
<!ELEMENT pLocation (#PCDATA)>
<!ELEMENT pDnum (#PCDATA)>
<!ELEMENT worker (SSN, hours, wDept)>
<!ELEMENT SSN (#PCDATA)>
<!ELEMENT wDept (wDeptNo, wDeptLoc)>
<!ELEMENT wDeptNo (#PCDATA)>
<!ELEMENT wDeptLoc (#PCDATA)>
```

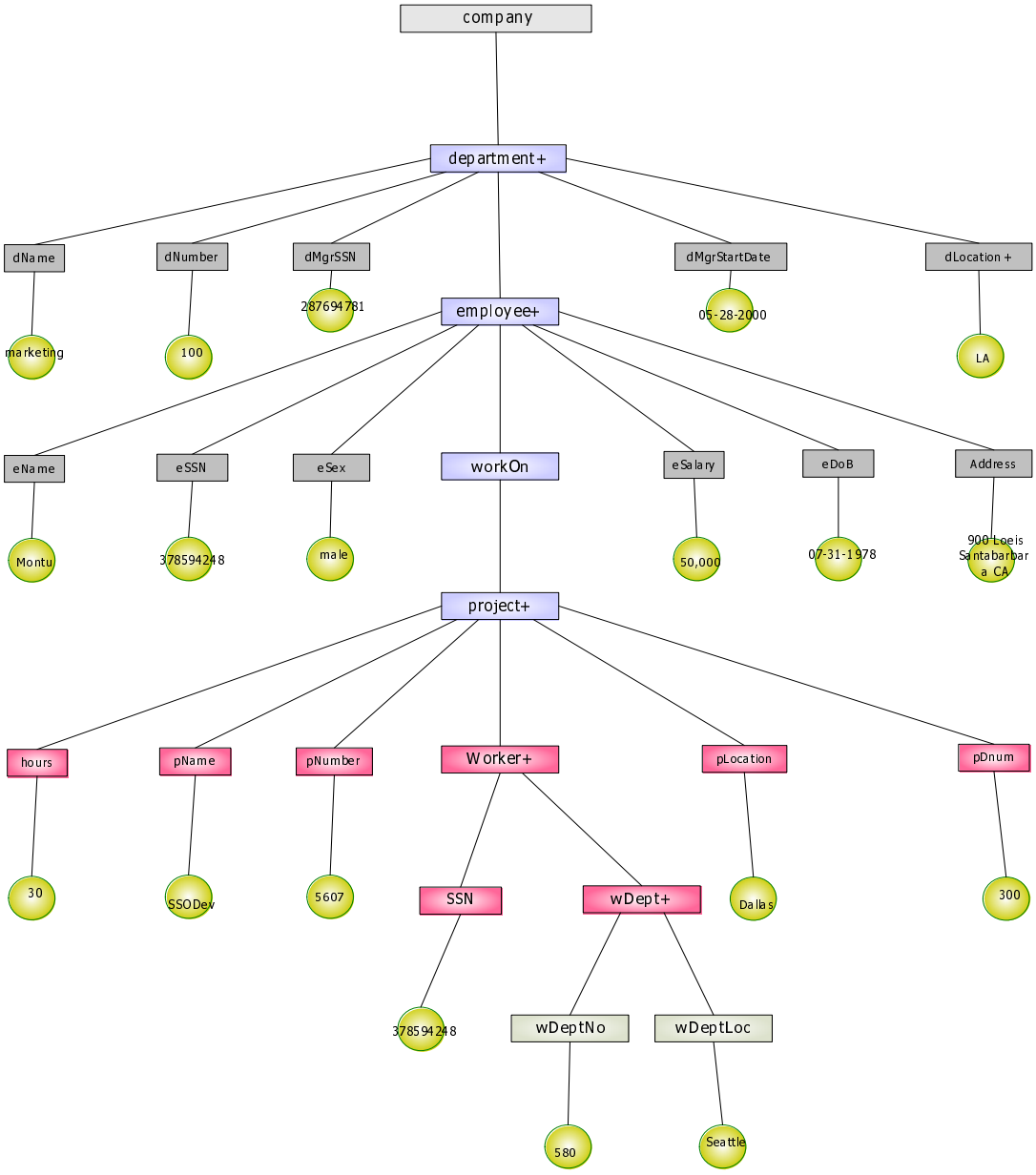


Figure 4.3. A tree structured graph of deep XML data.

plate specification file (TSL), which creates XML data. We generated shallow and deep structured XML data sets for the sake of our experiment. Next we will explain each indexing technique in more detail.

CHAPTER 5

[GENE]: GENERIC NUMBERED INDEX

GENE [44, 45, 8] (Generic Numbered Index) is one of the simplest forms of indexing techniques that would be an improvement over a file scan for searching XML data. It converts hierarchical tree structure of XML to relational schema by applying appropriate XML parser and conversion. While converting, it utilizes inverted index [45] such as search engine for mapping each tag or PCDATA into a sequential number then place encoded numbers into a relational database. In the following subsections, we will describe general concepts then explain the algorithms for encoding XML data subsequently.

5.1 General Concepts

XML provides semi-structured data information using a hierarchical tree structure. XML data is also merely a text file in that we can use any kind of file scan API for searching for wanted information. We can speed up the search and retrieval process if we use general numbering of XML document with underlying relational database. This way enables a user to implement generic mapping regardless of any schema or data composition [8]. In other words, without knowing XML data schema, we can store XML data and indexing information, which is numerically encoded. We store XML data and indexing information into a relational database so that we can benefit from features of relational databases. After indexing XML document, which means storing XML data into relational database, we need to search user-driven data using relational SQL. Thus in the next step, we are required to map XPath query into SQL statement.

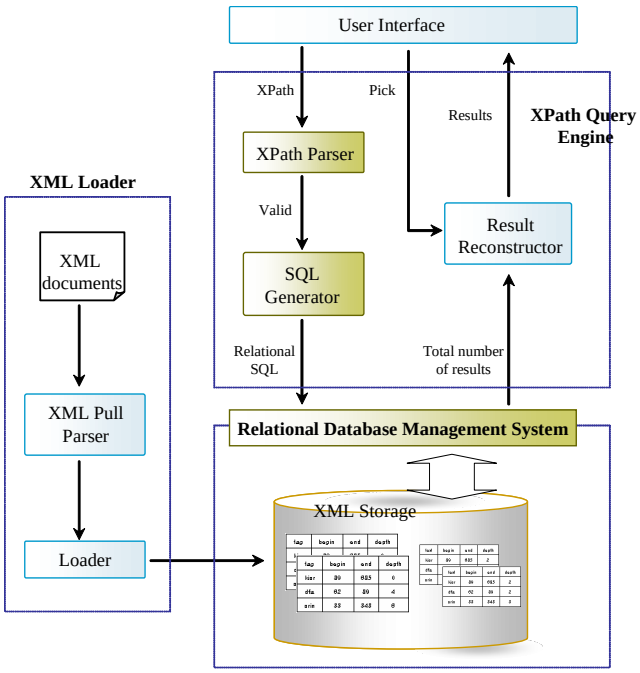


Figure 5.1. Architecture of GENE.

Figure 5.1 illustrates a simplified architecture of GENE. Loader accepts XML document then encodes its data with generic numbering scheme. Encoded data is stored into relational database in a proper schema. When a user sends XPath query, XPath parser checks whether it is in valid format and SQL Generator translates it into SQL statement. Then the results of its query will go back to a user by Result Reconstructor. All of the procedures are controlled by XPath Query Engine.

We will explain how to assign numbers in XML data then show encoded data in relational database. We will also describe the procedure of translating XPath to SQL statement in the subsequent sections.

5.2 Generic Numbering Scheme

There exists various different approaches for implementing generic numbering scheme. We will explain how to assign numbers to XML data in one of those ways. XML data

is composed of start tag, end tag, which surround an element, PCDATA or character strings, and attributes, which function as pointers or as properties of its element. We focus on XML data without attributes in this thesis. As depicted in Figure 5.2, (a) show pretty printed version of company XML data that was generated by ToXgene [18] and (b) shows assigned numbers. We just assign each number to each start tag, end tag, and PCDATA in a straightforward way. This procedure clearly presents encoding process

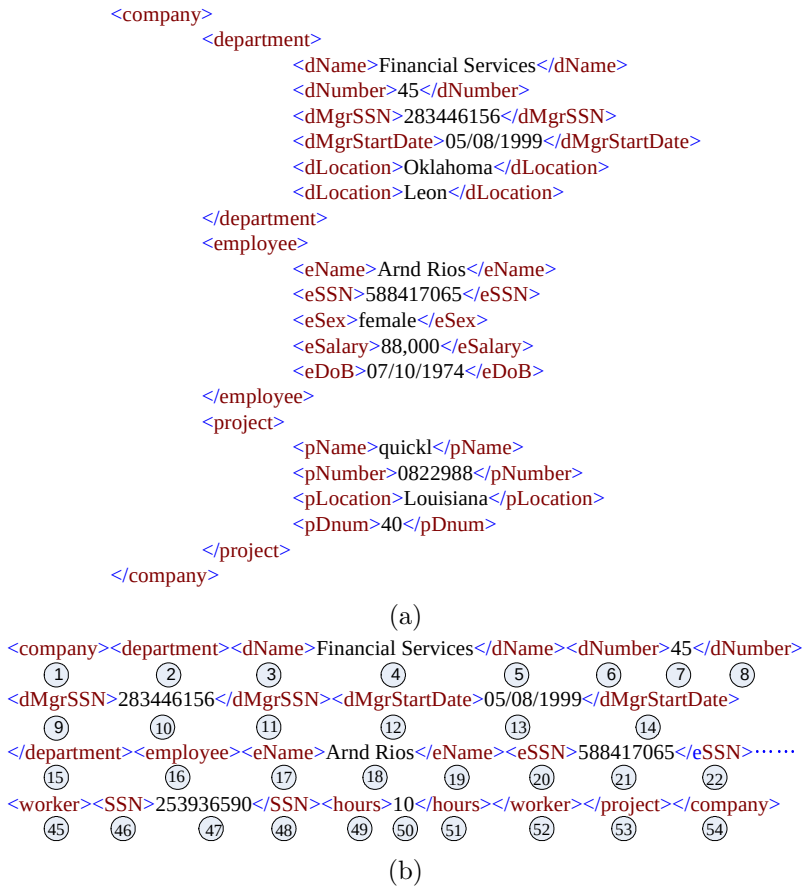


Figure 5.2. (a) An example of pretty print XML data (b) Generically numbered from (a).

of XML document. So far we explained how to assign numbers into XML data. In the

next sections, we will present how to map these numbers into relational schema and how we convert XPath to relational SQL.

5.3 Encoding XML Documents

We are now left with the challenge to put sequentially encoded XML data into relational database. We applied Algorithm 1 to load encoded data. Using *XMLPull parser* [46], starting from beginning of a document we pull each instance one by one. With increasing sequence number, we put numbers, tag name, and text data into designed tables separately. We created two tables *Element* and *Content* for storing encoded data as illustrated in Figure 5.3. Figure 5.4 shows examples of encoded data in each table.

Field	Type
tagname	varchar (30)
begin	int (10) unsigned
end	int (10) unsigned
level	tinyint (3) unsigned

(a)

Field	Type
text	varchar (80)
position	int (10) unsigned
level	int (10) unsigned

(b)

Figure 5.3. (a) A table schema for element data (b) A table schema for text data.

tagname	begin	end	level
company	0	696226	0
department	1	237791	1
dName	2	4	2
dNumber	5	7	2
dMgrSSN	8	10	2
dMgrStartDate	11	13	2
dLocation	14	16	2

(a)

text	position	level
System and Accounting Assurance	3	2
35	6	2
783871286	9	2
02/06/2002	12	2
Nagova	15	2
White	18	2
Pittsburgh	21	2

(b)

Figure 5.4. (a) Element table (b) Content table.

Algorithm 1: Load XML document into RDB in GENE

```
input  : XML document
output: Relational tables
begin
    connect to database through JDBC;
    xpp ← set XMLPull parser;
    Stack S;
    sequenceNo ← 0;
    noOfNodes ← 0;
    depth, level ← 0;
    top ← 0;
    while true do
        type ← xpp.nextToken;
        if type is instance of START_TAG then
            insertElementIndex(tagName, sequenceNo, depth);
            push sequenceNo into stack S;
            sequenceNo ← sequenceNo + 1;
            noOfNodes ← noOfNodes + 1;
            depth ← depth + 1;
        if type is instance of END_TAG then
            top ← pop from stack S;
            updateElementIndex(top, sequenceNo);
            sequenceNo ← sequenceNo + 1;
            depth ← depth - 1;
        if type is instance of TEXT then
            level ← depth - 1;
            insertTextIndex(PCDATA, sequenceNo, level);
            sequenceNo ← sequenceNo + 1;
    end
end
```

5.4 Translating XPath query into SQL statement

We utilize *XPath Query Engine* as shown in Figure 5.1 to retrieve user-requested data. XPath Query Engine consists of XPath Parser, SQL Generator, and Result Constructor. Based on schemas of encoded data in relational database, SQL Generator produces SQL statement employing *n-ary* self-join. Thus if XPath query contains many combined child and descendant axes, the number of self joins will increase. For the XPath expression "/company/project[pLocation=Maryland]/worker", for example, we can ob-

tain the following query: Other combinations of XPath axes can be translated in a

SELECT $e4.*$
FROM *Element* $e1$, *Element* $e2$, *Element* $e3$, *Content* c, *Element* $e4$
WHERE $e1.tagname =$ "*company*"
AND $e1.level = 0$
AND $e2.tagname =$ "*project*"
AND $e2.begin > e1.begin$ **AND** $e2.end > e1.end$ **AND** $e2.level = e1.level + 1$
AND $e3.tagname =$ "*pLocation*"
AND $e3.begin > e2.begin$ **AND** $e3.end > e2.end$ **AND** $e3.level = e2.level + 1$
AND $c.text =$ "*Montana*"
AND $c.position > e3.begin$ **AND** $c.position > e3.end$ **AND** $c.level = e3.level$
AND $e4.tagname =$ "*worker*"
AND $e4.begin > e2.begin$ **AND** $e4.end > e2.end$ **AND** $e4.level = e2.level + 1$;

similar way.

5.5 Summary

In this chapter, we introduced a generic numbered indexing technique called GENE in this thesis. GENE encodes XML document and its subsequent information that is numerically encoded then places them into relational database such that we benefit from features of relational database. Relational schema(s) for storage can be varying for design purposes. To retrieve user-driven data, XPath Query Engine maps XPath query into SQL statement by doing *n-ary* self-join. Next we will describe an encoding scheme and various features of range-based numbering scheme.

CHAPTER 6

[XISS]: RANGE-BASED INDEX

XISS [3] (XML Indexing and Storage System) encodes each of the nodes in XML documents into <order,size> pair. For the sake of efficient future insertion of nodes, *order* should not increase sequentially. This is the reason why we call it *range-based*. Range-based numbering scheme allows determining the ancestor-descendent relationship between the two nodes in O(1) constant time utilizing <order,size>. **XISS/R** [2] is a web version of XISS on top of a relational database. In the experiment we used XISS/R for comparison. In the following subsections we will explain the general concepts of range-based index and proposed numbering scheme. Also we will explicate how XML documents would be encoded and queried to obtain results.

6.1 General Concepts

Conventional approaches for processing regular path expressions can be inefficient due to not only the overhead of traversing for long or unknown path lengths but also an extensive search of XML data tree [3]. Li, Moon, and Harding [3, 2] created primarily a new system called **XISS** for indexing and storing XML data based on a new numbering scheme for elements and attributes, which allows efficient retrieval of all elements or attributes with the same name string. An *extended pre-order numbering scheme* is based on *Dietz's numbering scheme* [2, 4] where pre-order and post-order ranks can be used to determine the ancestor-descendant relationship between any pair of tree nodes within a constant time. To expedite XML query processing by both value and structure searches, how quickly we can determine the ancestor-descendant relationship between elements as

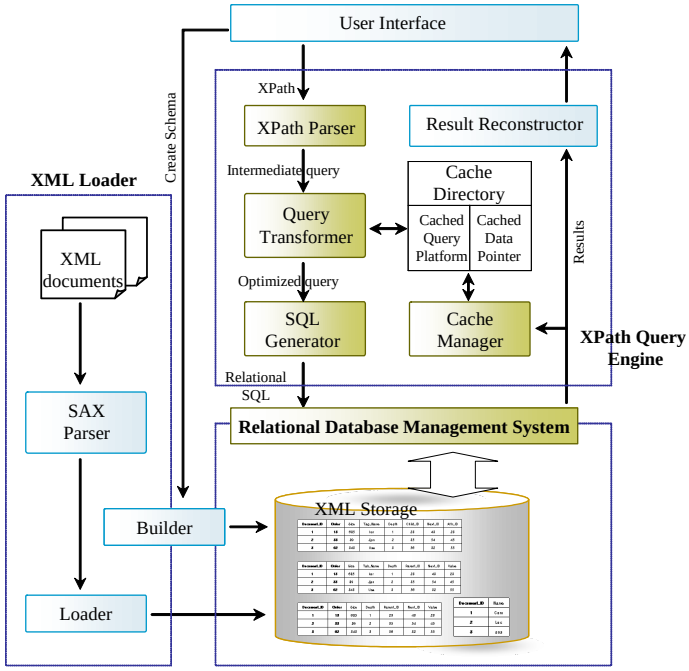


Figure 6.1. Architecture of XISS [2].

well as fast accesses to values is critical. To improve the flexibility for insertion, each node is associated with a numeric pair <order,size>, in which order is the extended pre-order rank of the node, and size is the range of descendants of the node. This is why we call it range-based index. **XISS/R** [2] is a demonstration system of XISS on top of a relational database. XISS/R puts all the elements, attributes and text values into different types of schemas. Upon executing XPath query, the result set will be reconstructed from appropriate join methods. Figure 6.1 illustrates an architecture of XISS/R. The XPath query engine accepts XPath query and generates SQL statement. Then query results from database server will forwarded to user interface. We will explain the procedure of efficient numbering scheme and how to encode XML documents with an extended pre-order numbering scheme and finally how to map XML data into relational database system in the following subsections.

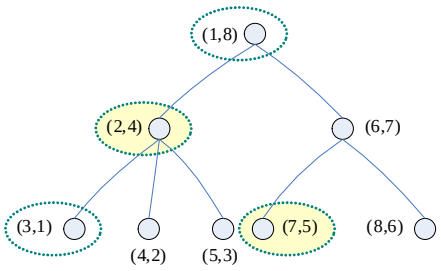


Figure 6.2. Dietz's Numbering Scheme [3, 4].

6.2 Numbering Scheme

XISS employed extended pre-order numbering scheme motivated from pre/post numbering. *Numbering* is to assign numbers or paired numbers in each node. The reasons why we use numbering scheme in XML tree structure are as follows.

- Use structure information for querying.
- Quickly determine *ancestor-descendant* relationship between any pair of nodes in the hierarchy of XML data.
- The join operation can be carried out without traversing XML data tree.

XML data objects are commonly organized by a tree structure where each node represents elements, attributes, and character data. To speed up regular path expression, we need to determine the ancestor-descendant relationship immediately. We will introduce Dietz's numbering scheme for basic pre/post numbering then describe how to overcome its shortcoming.

6.2.1 Dietz's Numbering Scheme

Dietz [4] proposed numbering scheme that utilize tree traversal order to determine the ancestor-descendant relationship in a constant time between any pair of nodes. While assigning (*pre*,*post*) pair in each node in a tree T, it turned out that *for given two nodes x and y of a tree T, x is an ancestor of y if and only if x occurs before y in*

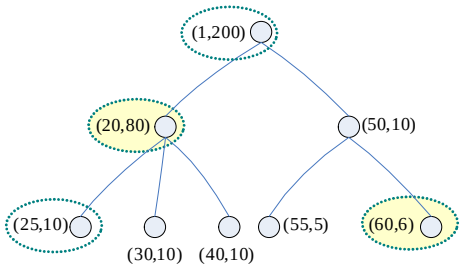


Figure 6.3. Extended pre-order numbering scheme [3].

the pre-order traversal of T and after y in the post-order traversal. In Figure 6.2, for example, x has (1,8) and y has (3,1) then in the pre-order 1 is less than 3 and 8 is greater than 1 in the post-order. It yields x is the ancestor node of y. In another pair, x has (2,4) and y has (7,5) then in the pre-order 2 is less than 7 and 4 is also less than 5 in the post-order. It simply violates ancestor-descendant rule thus pair of nodes is not in the ancestor-descendant relationship. As such, we could determine ancestor-descendant relationship within a constant time. However it has drawback of flexibility, which means if a new node is inserted in T, we need to renumber all pairs of (*pre*,*post*). To get over this limitation, we allocate spare range in the number of descendants.

6.2.2 Extended Pre-order Numbering Scheme

To overcome a flaw of Dietz's numbering scheme [4], an extended pre-order numbering scheme associates each node with a pair of numbers <order,size> having following properties [3].

- For a tree node y and its parent x, $order(x) < order(y)$ and $order(y) + size(y) \leq order(x) + size(x)$. In other words, interval $[order(y), order(y) + size(y)]$ is contained in interval $[order(x), order(x) + size(x)]$.
- For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, $order(x) + size(x) < order(y)$.

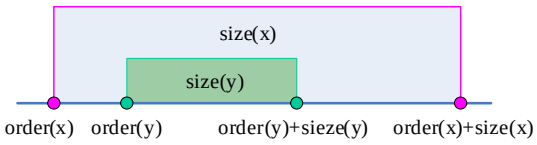


Figure 6.4. Node y and its ancestor x.

For a tree node x, *size(x)* can be an arbitrary integer larger than the total number of the current descendants of x. This allows future insertions to be accommodated gracefully. In Figure 6.3, for example, let (1,200) be x and (25,10) be y then x is an ancestor of y because $order(x) < order(y) \leq order(x)+size(x)$, which is $1 < 25 \leq 1+200$. Therefore the extended pre-order numbering scheme is more flexible since it can deal with anonymous insertions. That is when insertion or deletion happens by chance in the future renumbering would not be occurred until the range (unused order values) is run out of. Figure 6.4 shows how a <order,size> pair works.

6.2.3 Extendible Range-based Numbering Scheme

In favor of running out of spare range, [47] proposed *Extendible range-based* numbering scheme motivated from that extended pre-order numbering scheme still suffers from renumbering. That is *relabeling* is unavoidable when arbitrary insertions are allowed. Although relabeling may be infrequent, very short sequences do exist for which relabeling is unavoidable [48].

6.3 Encoding of XML Documents

From the numeric pair <order,size>, XISS/R creates element index, attribute index, and text index. Both the element index and attribute index are implemented by B^+-tree index using name identifier (*nid*) as primary key. We will briefly describe index structure of XISS/R and explain how to encode such XML documents.

6.3.1 Index Structure

Encoding of XML documents simply means that we put XML data into relational database with separating data and index information using extended pre-order numbering scheme. The index structure of XISS/R is composed of three main components and one sub component. *Element index* allow us to quickly find all elements with the

main components	element index attribute index text index
sub component	document index

same name string. Each element record includes an <order, size> pair and other related information of the element such as depth and parent ID, and the element records are in a sorted order by the order values. *Attribute index* has almost the same structure as the element index, except that the record in attribute index has a *vid* that references value table. Text index encodes text nodes within the system. *Value* stores the actual text. Document index is to separate the document name from element, attribute, and text index.

6.3.2 Loading XML data to RDB

In demonstration version of XISS, XISS/R consists of three components as follows.

- A mapping of XML data to relational schema
- An XPath Query Engine
- A web-based user interface based on Apache, PHP, and MySQL

Loading XML documents to relational database with schema(s) is accomplished by using the extended pre-order numbering scheme. In this thesis we created a schema type A [2] as shown in Figure 6.5. XISS/R requires mainly 5 dimensions of information for each node. Those are the document ID, order and size of a node, depth of a node in a document tree, tag name, and text value of a node. For efficient retrieval, we also

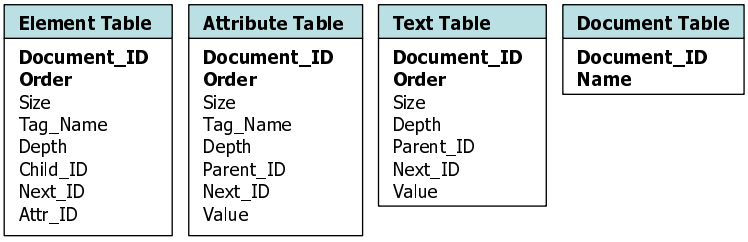


Figure 6.5. Relations in schema A.

need to save subsequent information such as parent node ID, sibling node ID, first child ID, and first attribute ID for each node. Loader uses the *LibXML* library to access XML documents such that their structural information can be encoded with the extended pre-order numbering scheme. Figure 6.6 presents encoded data in schema A.

did	name
1	fixedxml/xmlfile/company200k.xml

(a)

name	did	nid	size	depth	parent_id	prev_id	child_id	attr_id
dName	1	4	1	2	3	-1	-1	-1
dNumber	1	6	1	2	3	4	-1	-1
dMgrSSN	1	8	1	2	3	6	-1	-1
dMgrStartDate	1	10	1	2	3	8	-1	-1
dLocation	1	12	1	2	3	10	-1	-1
dLocation	1	14	1	2	3	12	-1	-1
dLocation	1	16	1	2	3	14	-1	-1
dLocation	1	18	1	2	3	16	-1	-1
dLocation	1	20	1	2	3	18	-1	-1
dLocation	1	22	1	2	3	20	-1	-1
eName	1	25	1	3	24	-1	-1	-1
eSSN	1	27	1	3	24	25	-1	-1
eSex	1	29	1	3	24	27	-1	-1

(b)

did	nid	depth	parent_id	prev_id	value
1	5	3	4	-1	(MEMO)
1	7	3	6	-1	(MEMO)
1	9	3	8	-1	(MEMO)
1	11	3	10	-1	(MEMO)
1	13	3	12	-1	(MEMO)
1	15	3	14	-1	(MEMO)
1	17	3	16	-1	(MEMO)
1	19	3	18	-1	(MEMO)
1	21	3	20	-1	(MEMO)
1	23	3	22	-1	(MEMO)
1	26	4	25	-1	(MEMO)
1	28	4	27	-1	(MEMO)
1	30	4	29	-1	(MEMO)

(c)

Figure 6.6. (a) Encoded data in a Document table (b) Encoded data in a Element table (c) Encoded data in a Text table.

6.4 Mapping XPath to relational SQL

6.4.1 Path Decomposition and EA/EE/KC Join

Through a path decomposition, a complex path expression can be divided into several simple path expressions which each produce an intermediate result that can be

Algorithm 2: Load XML documents into RDB in XISS [49]

```
input  : XML documents
output: Encoded data
// open XML file and begin preorder traversal
my_xml_doc ← xmlNewTextReaderFilename(filename);
if my_xml_doc ≠ null then
    set_docid(filename);
    preorder_traverse(-1,-1);
    xmlFreeTextReader(my_xml_doc);
preorder_traverse(parent_id,sibling_id)
begin
    my_order ← global_order ← global_order+1;
    my_size ← 0;
    my_tag_name ← xmlTextReaderName(my_xml_doc);
    my_depth ← xmlTextReaderDepth(my_xml_doc);
    size_res ← process_attributes(my_order);
    my_size ← my_size+size_res;
    // get next element or text node
    read_next_node();
    // type of next node
    node_type ← xmlTextReaderNodeType(my_xml_doc);
    while node_type ≠ ELEMENT_END do
        if node_type = TEXT_NODE or node_type = CDATA_SEC then
            process_text_node(my_order,prev_child);
            my_size ← my_size+1;
        else if node_type = ELEMENT_START then
            recent_child ← global_order+1;
            size_res=preorder_traverse(my_order,prev_child);
            my_size ← my_size+size_res;
            prev_child=recent_child;
            if my_depth = xmlTextReader(my_xml_doc) and
            xmlTextReaderNodeType(my_xml_doc)=ELEMENT_END then
                break;
        else
            print error;
        read_next_node();
        if my_depth ≥ xmlTextReaderDepth(my_xml_doc) then
            break;
        node_type ← xmlTextReaderNodeType(my_xml_doc);
end
```

joined together to obtain the final result of the given query. In Figure 6.7, XPath

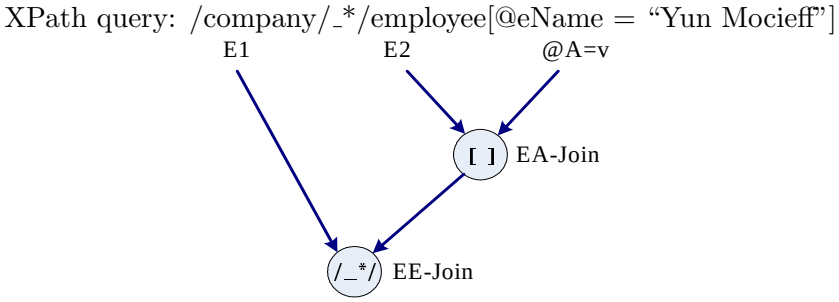


Figure 6.7. XPath query and Path Decomposition.

query is to find an employee whose name is “*Yun Mocieff*” in the company. In this query, *company* and *employee* are elements and *eName* is an attribute. Notations were referred from XQuery working draft [37]. Conventional approaches are to traverse the hierarchy of node objects in either *top-down* or *bottom-up* manner. If a *company* were a root node, we face up a problem that we should traverse the whole XML tree in top-down approach. We can reduce the cost of traversal in bottom-up manner. Even hybrid approach, which traverse in both top-down and bottom-up meeting in the middle of a path expression [3] cannot always guarantee enough effectiveness. We can resolve this problem using XPath decomposition as well as path join algorithms, *i.e.,* **EA-Join**, **EE-Join**, and **KC-Join**. The EA-Join algorithm joins two intermediate results from a list of elements and a list of attributes. The EE-Join algorithm joins two intermediate results, each of which is a list of elements obtained from a sub-expression. The KC-Join algorithm processes a regular path expression that represents zero, one or more occurrences of a sub-expression (*e.g.,* employee* or employee+). In each processing stage, KC-Join algorithm applies EE-Join to the result from the previous stage repeatedly until no more results can be produced. Thus a regular path expression can be decomposed to a combination of subexpressions. XPath query in Figure 6.7 accordingly can be decomposed into “$/E_1/\_*/(E_2$[@A=v]”.

6.4.2 Translating XPath query into SQL statement

Assume that we have loaded XML documents into the relational schema A. Then we are now ready to retrieve user-driven data so that we generate SQL statement. *Query Transformer*, which functions intermediary storage for current query tree pattern in Figure 6.1 passes optimized query to SQL generator. For example, the query "//dependent/depName" would be translated into the following SQL statement when using schema type A:

SELECT $doc.NAME$ as Document_Name, $doc.did$ as Document_ID, $et1.NID$ as Node
FROM $elem\_tab$ $et0$, $elem\_tab$ $et1$, did_tab doc
WHERE $et0.NAME =$ '$dependent$'
AND $et1.NAME =$ '$depName$'
AND $et0.DID = et1.DID$
AND $et0.NID < et1.NID$
AND $et0.NID + et0.SIZE \geq et1.NID$
AND $et1.DID < doc.did$;

Essentially SQL statement is generated in a way that use recursive *n-ary* self join between elements. Above statement has two elements (*dependent*, *depName*) thus it applies two self joins.

6.5 Summary

This chapter presented range-based numbering index technique name XISS and XISS/R. They can efficiently process regular path expressions applying range-based numbering scheme, which consists of <order,size> pair. Thus not only can it gracefully accept future insertion but also it does not suffer from deletion. In the next chapter, we will introduce multi-dimensional index named XACC and describe its encoding scheme as well as properties.

CHAPTER 7

[XACC]: MULTI-DIMENSIONAL INDEX

XPath Accelerator [19, 27] maps every element and attribute node onto the 2-dimensional plane, using its pre-order rank on the x-axis, its post-order rank on the y-axis [19]. This yields XPath Accelerator named to multi-dimensional index. For the sake of naming convention, we named it *XACC*. XACC lives completely inside a relational database system. Thus when implementing XACC , not only can we utilize B-tree, but also we will be benefited considerably if the underlying database supports spatial indexing techniques such as *R-tree*. In the following sections, we will explain general concepts of dimension-based index, document regions in detail, how to encode XML documents, and differences between R-tree and B-tree implementation.

7.1 General Concepts

Multi-dimensional index proposes an index structure that can entirely reside in relational database system as XISS/R and GENE do. A context node (a root node in a sub-tree) and the four major axes of XPath steps, *i.e., ancestor, descendant, preceding, and following* divide the 2-dimensional space into four document regions, each corresponds to one major axis. Given the context node, the process of calculating one of its axes can be simplified as partitioning nodes on the 2-dimensional plane and retrieving the nodes that fall into the region corresponding to the specific axis in query.

One of the key concepts is the recursive nature of the XPath query expression evaluation due to the inherent recursion of XML data, starting with an arbitrary context node and traversing through the queried document using a sequence of location steps

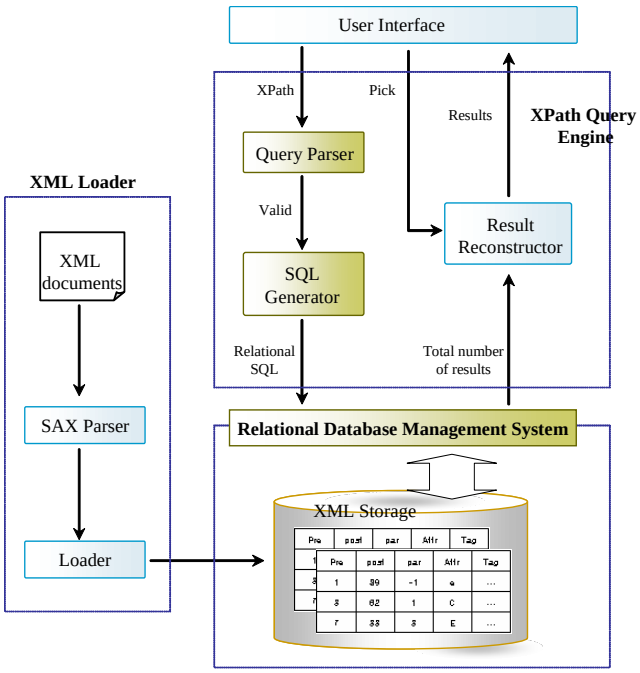


Figure 7.1. Architecture of XACC.

defined by XPath specification. Another key observation made is that the four major axes, namely, ancestor, descendant, preceding, and following partition the XML document into four regions. Together with the context node, it contains each node exactly once. Thus the evaluation of path expressions becomes the repeated partitioning of XML document as the context node changes with each location step [19, 27]. Based on these observations, XPath Accelerator can be implemented with relational database, or better yet, any database engine that supports spatial indexing technique such as R-tree. With the pre-order and post-order ranks calculated from document loading, each node is mapped as a coordinate into the pre/post onto 2-dimensional plane. As illustrated in Figure 7.3, the plane is partitioned into four disjoint regions that correspond to four document regions created by four major axes given a context node. In this thesis, we implemented an index structure using B-tree. Figure 7.1 shows an architecture of an XACC system. XML document loader loads XML data into a relational database. Then, SQL

Generator maps XPath into relational SQL in order to retrieve user-driven data. Result reconstruction is done when a user selects one node out of the result set.

7.2 XPath Axes and XML Documents Regions

XPath expressions indicate a tree traversal via two parameters: context node and document region, which are defined as follows.

- Context node: starting node of tree traversal, not necessarily the root node.
- Document region: a subset of document nodes, given the context node.

When g is the context node as depicted in Figure 7.2, we are interested in basic four axes, *i.e.,* ancestor, descendant, preceding, and following. Those are marked as dotted circles. Numbered pairs on each node represent (pre-order, post-order). Those four axes and a context node can specify a *partitioning*. Hence for any given context node g, the four major axes specify a partitioning of the document containing v, which leads to the following:

$$g/descendant \cup g/ancestor \cup g/following \cup g/preceding \cup \{f\} = \{a \ldots n\}$$

The key point of this work is to find an index structure such that, for any given context node, we can efficiently determine the set of nodes in the four document partitions specified by the major axes [27]. Attribute nodes will be handled same as regular element nodes in loading XML document. We put special mark such as boolean or a character in each node descriptor for them.

7.3 Encoding XML Documents

We could determine document regions of each context node utilizing XPath axes in the previous section. An index structure of XACC uses relational database as a means of XML storage. Basic encoding of XML document starts from creating *query window*.

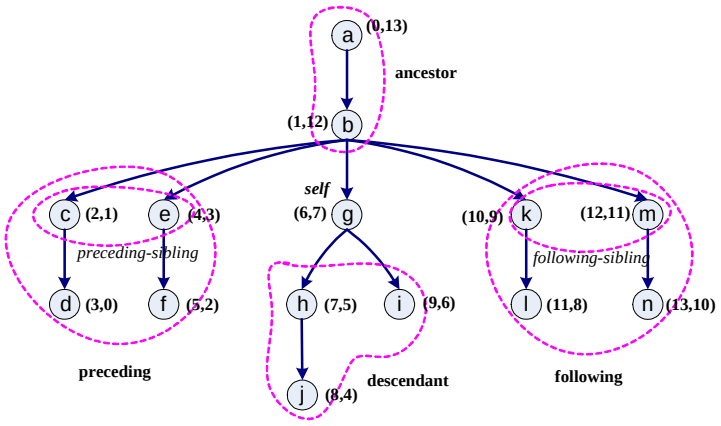


Figure 7.2. Primarily interesting four axes.

When we see the Figure 7.3, we can easily decide relationships between nodes. These can be characterized as follows:

- v' is a descendant of v iff $pre(v) < pre(v') \wedge post(v) > post(v')$. Lower-right partition contains all descendants of node g.
- v' is a ancestor of v iff $pre(v) > pre(v') \wedge post(v) < post(v')$. Upper-left partition contains all ancestors of node g.
- v' is a preceding of v iff $pre(v) > pre(v') \wedge post(v) > post(v')$. Lower-left partition contains all precedings of node g.
- v' is a following of v iff $pre(v) < pre(v') \wedge post(v) < post(v')$. Upper-right partition contains all followings of node g.

Based on the above characteristics we can create a node descriptor ($desc(v)$) for each node while scanning XML document sequentially using SAX parser. Then each node v can be represented by its 5-dimensional *descriptor*:

$$desc(v) = \left| pre(v) \right| post(v) \left| par(v) \right| attr(v) \left| tag(v) \right|$$

which would be mapped into exactly one tuple in the relational database. In each descriptor, $pre(v)$ means pre-order rank of a node v, $post(v)$ means post-order rank of a node v, $par(v)$ means pre-order rank of parent of a node v, $attr(v)$ maintains the boolean

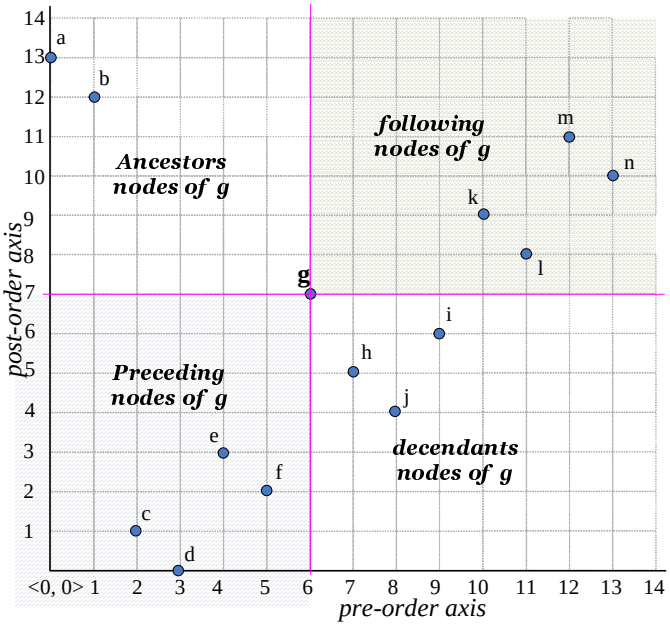


Figure 7.3. Node distribution in the *pre/post* plane and XML document regions as seen from context node g [19].

value for each node v, and finally, *tag(v)* represents element tag or attribute name for a node v. On *X-Y axis*, we can define that *query window* for the name test α::n is *windows* (α, v with its *tag* entry set to n). For a node v' to be a child of context node v, it is sufficient to test the condition $par(v') = pre(v)$.

$$window(child,v) = \left| * \right| * \left| pre(v) \right| false \left| * \right|$$

Notice that since v' is a child of v we know the relationship $pre(v) < pre(v') \wedge post(v) > post(v')$. An encoded XML data for Figure 7.3 is shown in Table 7.1 Once we put XML document into relational database with encoding it, we do not need original XML document any more when reconstructing the result. This is because we already stored all the information that we need. Algorithm 3 shows the loading procedure of XML document. To keep track of elements whose start tag has already been scanned but whose end tag is still to come, we maintain a stack S of partial node descriptors. Apparently, the size of S will be bounded to the maximum depth of XML tree.

Algorithm 3: Load XML document into RDB in XACC [20, 19]

```
input  : XML document
output: accel table
begin
    connect to database through JDBC
    gpre ← 0; gpost ← 0;
    Stack S; S.empty();
    S.push( ⟨ pre=-1, post=⊔, par=⊔, attr=⊔, tagName=⊔ ⟩ );
    sp ← SAXParseFile();
    S.pop();
    while true do
        type ← sp.nextToken
        if type is instance of startElement(t, a, attr) then
            v ← ⟨ pre=gpre, post = ⊔, par=(S.top()).pre, attr=a, tagName=t⟩
            S.push(v);
            gpre ← gpre+1;
            for v′ ∈ attr do
                startElement(v′, true, nil);
                endElement(v′);
        if type is instance of endElement(t) then
            v ← S.pop();
            v.post ← gpost;
            g.post ← gpost+1;
            insert v into table accel;
    end
end
```

7.4 Mapping XPath to relational SQL

We are now left with a challenge to map XPath to relational SQL in order to retrieve user-driven data. Assume that we have already loaded the node descriptors of a XML document into a 5-column table named *accel_filename* whose schema is *pre*|*post*|*par*|*attr*|*tag*. Essentially, XPath uses regular path expression. Thus it is composed of axes such as child (/) and descendant (//). This leads to using recursive XPath evaluation scheme for mapping it to SQL. The mapping scheme generates an SQL query of nesting depth n for a path expression of n steps. To avoid nested query, we transform XPath to n-ary self-join [27]. For the XPath expression /descendant::$n1$/child::$n0$, for example, we can

Table 7.1. Encoded XML data from Figure7.3

pre	post	par	attr	tag
0	13		false	a
1	12	0	false	b
2	1	1	false	c
3	0	2	false	d
4	3	1	false	e
5	2	4	false	f
6	7	1	false	g
7	5	6	false	h
8	4	7	false	j
9	6	6	false	i
10	9	1	false	k
11	8	10	false	l
12	11	1	false	m
13	10	12	false	n

obtain the following query:

SELECT DISTINCT $v0.*$
FROM $accel\_company50m$ $v1$, $accel\_company50m$ $v0$
WHERE $v1.tag$ = "$n1$"
AND $v1.pre = v0.par$
AND $v0.tag$ = "$n0$"
AND $v0.attr$ = "e";

Other combinations of regular path expressions can be translated in a similar way.

7.5 R-Trees and B-Trees

The collection of these node descriptors can be indexed and searched using conventional relational database search techniques, which would yield comparable performance with other proposed approaches. More efficiently, they can also be searched using spatial techniques like R-tree. Algorithms have been provided for initial document loading that

only requires one single pass to calculate the pre-order and post-order for each node. Optimization techniques have been discussed to further reduce the size of search window substantially. It is not surprising that this approach outperforms previous works according to the experiments, especially using R-tree backend. Another key advantage of this approach is that it accelerates all XPath axes, and path expressions do not have to start from the root, rather, they can start from any node that acts the context node.

7.6 Summary

This chapter explained multi-dimensional index technique named XACC. XACC maps all the nodes in an XML document into 2-dimensional plane based on pre-order rank and post-order rank. Furthermore, *pre/post* plane is divided into four document regions by a context node. As such, we can identify ancestor, descendant, preceding, and following nodes. This leads to defining 5-dimensional *descriptor*, which is the table schema as well when we load XML document. Based on recursive translation of XPath, we can retrieve a part of the XML data. If a database supports R-tree index, XACC performs the procedure above better than B-tree. In the next chapter, we present and discuss the results of performance evaluation.

CHAPTER 8

EXPERIMENTAL RESULTS AND DISCUSSION

We compared the query performance over shallow and deep structured XML data applying to three different XML indexing techniques. The indexing technique GENE was developed with reference to [8, 44, 45]. XISS/R [2] (XISS on top of a relational database) was downloaded from [49]. XACC [19, 27] was developed by a previous member of XML Indexing Group. We will describe experimental environments shortly first, and then explain XML data sets were used in our experiments. Then we will present a brief explanation regarding XPath queries used in our experiments. Performance analysis will cover XML data loading time and elapsed querying time for each query as well as result reconstruction time.

8.1 Experimental Setup

We ran all our experiments on Pentium(R)4 CPU 2.4GHz processor with 512 MB RAM running Redhat Linux Enterprise3. A 40GB EIDE disk drive was used to store the data and indexes. All three techniques were built on MySQL database for storing index information. GENE and XACC were compiled using JDK 1.4.2. XISS was running on Apache2 web server with PHP parser 5.0.4 and Zend optimizer. Tuning parameters were set as in the following Table 8.1 and the system environment is illustrated in Figure 8.1.

Table 8.1. Modified parameters for system softwares

Software	Parameter	Value
Apache	Timeout	1800 (5 min)
	KeepAlive	on
	MaxKeepAliveRequests	100
	StartServers	20
	MinSpareServers	5
	MaxSpareServers	10
	MaxClients	150
	MaxRequestsPerChild	0
MySQL	key_buffer	50M
	sort_buffer_size	1024K
	skip-locking	true
PHP	max_execution_time	1800 (secs)

Figure 8.1. An environment for experiments.

8.2 XML Data Sets on Experiments

We used two different data sets retaining different characteristics. Both of them were generated by ToXgene [1] from the University of Toronto. Each data set has following features:

- Shallow data set has a maximum level of depth 4.
- Deep data set has a maximum level of depth 8.
- Both data sets have similar structure representing a company report.

Since we generated two data sets having the above features, we can test if each indexing technique is effective for which data set regarding various XPath queries. With this in mind, we produced appropriate XPath queries.

8.3 XPath Queries on Experiments

We prepared three kinds of XPath query sets since not all kinds of XPath queries can be applied to each indexing technique.

Table 8.2. Supported queries in each indexing technique

Query Type	Description	GENE	XISS	XACC
/A/B	A and B are elements. This selects all B elements that are children of A.	❍	❍	❍
/A//B	A and B are elements. This selects all B elements that are descendants of A.	❍	❍	×
/A[B="C"]	A and B are elements, C is PCDATA. This selects all A elements that have immediate child B elements having value "C".	❍	×	❍
/A[B="C"]/D/E	A, B, D, and E are elements. C is PCDATA. This selects all E elements that are children of D, which are children of B having value "C".	❍	×	❍
A@B	A is an element and B is an attribute. This selects all A elements having attribute B.	×	❍	×
//A/B/C	A, B, and C are elements. This selects all C elements that are children of B, which are children of A that are descendants of a root node.	❍	❍	❍
/A/*/C	A and C are elements. This selects all C elements having grand parent of A.	❍	×	❍

Table 8.2 shows supported XPath queries in each indexing technique. Based on the above features we extracted *Common XPath queries* and *Specific XPath queries* over shallow and deep tree data sets.

Table 8.3 shows extracted 6 common queries, *i.e.,* q1 through q6. Query 1, 2, and 3 will test over a shallow data set. Query 4, 5, and 6 will test over a deep data set. The

Table 8.3. Common queries for all indexing techniques

Dataset	q#	Query Type	Query
Shallow	q1	/A/B	/company/employee
	q2	//A/B	//dependent/depName
	q3	//A/B/C	//employee/workOn/projNo
Deep	q4	//A/B/C/D/E	//employee/workOn/project/worker/wDept
	q5	//A//B/C/D/E	//employee//project/worker/wDept/wDeptLoc
	q6	/A/B/C/D/E/F/G/H	/company/department/employee/workOn/project/worker/wDept/wDeptLoc

way of supporting predicates in XISS is slightly different from XACC and GENE. Table

Table 8.4. Specific queries for GENE vs XACC

Dataset	q#	Query Type	Query	Indexing Technique
Shallow	q7	/A/B[C=D]/E	/company/project[pLocation=Maryland]/worker	XACC
Deep	q8	//A[B=C]/D/E/F	//employee[eSex=male]/workOn/project/worker	
Shallow	q9	/A/B[C=D]/E	/company/project[pLocation=Maryland]/worker	GENE
Deep	q10	//A[B=C]/D/E/F	//employee[eSex="male"]/workOn/project/worker	

8.4 shows specific queries for GENE and XACC. Therefore query 7 through 10 having predicates within XPath query will test over GENE vs XACC. Notice that queries for a deep data have longer path than a shallow data. Table 8.5 summarizes the characteristics of each query.

8.4 Performance Analysis

In this section, we would like to evaluate the performance of each indexing technique regarding a shallow and deep data while executing XPath queries which have different characteristics. We also would like to see if data loading time is linear to the size of data.

Table 8.5. Features of each query

q1	Basic query. Relatively small quantity. It retrieves all employees in a company.
q2	Descendent query. Relatively medium quantity. It retrieves all dependents' names of all employees.
q3	Descendent medium deep query. Relatively large quantity. It retrieves all projects' numbers of all employees.
q4	Descendent deep query.
q5	Duplicate descendant deep query. q5 and q6 retrieve the same number of results.
q6	Extremely deep query. q5 and q6 retrieve the same number of results.
q7, q8, q9, q10	Medium deep query using predicates. q8 and q10 hold relatively more results than q7 and q10.

The result reconstruction time will be analyzed as well. We mainly focus on analyzing query performance for each query with a corresponding graph.

8.4.1 XML Data Loading Time

We define *XML Data Loading Time* as a data loading time onto MySQL database. Each of the indexing techniques uses MySQL database for storing XML data and indexes. This is why we necessarily have to evaluate loading time as another measurement factor. Before we discuss the result of the loading time, we need to understand algorithms for each indexing technique.

While loading XML document into MySQL database, XACC maintains a stack S of yet incomplete node descriptors. Every time we encounter an element's end tag, we are ready to fix up its yet unspecified *post* value and then insert the node into database table [19] (see Section 7.3 for the descriptor having 5 dimensions of stack S). GENE also maintains a stack for end tag number. Differently from XACC, in GENE, when the loader meets begin tag, it inserts its element information (begin, depth, tag name) into database except end tag number and then pushes begin tag number into stack S. By the time the

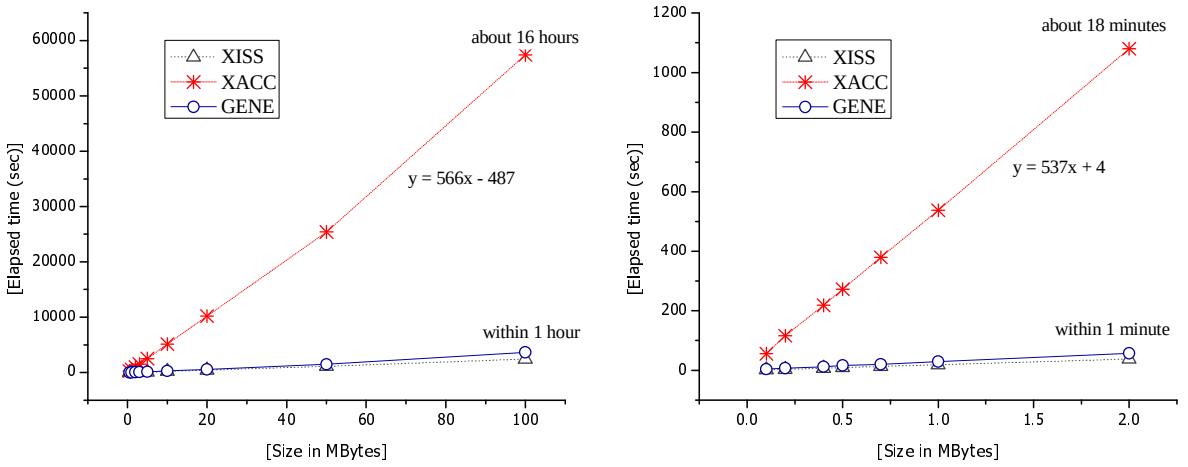


Figure 8.2. (a) Data loading time for shallow data (b) Data loading time for deep data.

loader encounters end tag, it pops begin tag number from stack S and then updates a table. We can speed up this procedure by a primary key on begin column. XISS loads XML documents in a similar way. Figure 8.2(a) and 8.2(b) present loading time for a shallow and deep data. GENE and XISS can load even 100MB data within an hour. However, XACC takes more than 16 hours to load 100MB data. Whenever the loader encounters an end tag, it inserts a fixed descriptor into database. When it happens, the loader always open a connection for the database in XACC because connection pooling was not used. It caused serious bottleneck. By using the DB connection pool, we could resolve this problem. Overall, notice that loading time of all three indexing techniques show a linear increase due to the size of data.

8.4.2 Performance for Shallow Tree

We prepared 500KB to 100MB for a shallow data so that we can see if and how the data size affects elapsed query time. We ran three XPath queries, *i.e.,* q1, q2, and q3 for evaluating query performance of shallow structured XML data set. Each query

has generally a simple and short path (see Section 8.3). We describe performance results using graphs and discuss the reasons why it shows such features. Discussion of the results focuses on the following factors:

- Which indexing technique has a better performance for a small data?
- Which indexing technique has a better performance for a large data?
- Which indexing technique has a better performance for a shallow data?
- General query performance over varying sizes of data.
- Result reconstruction time.

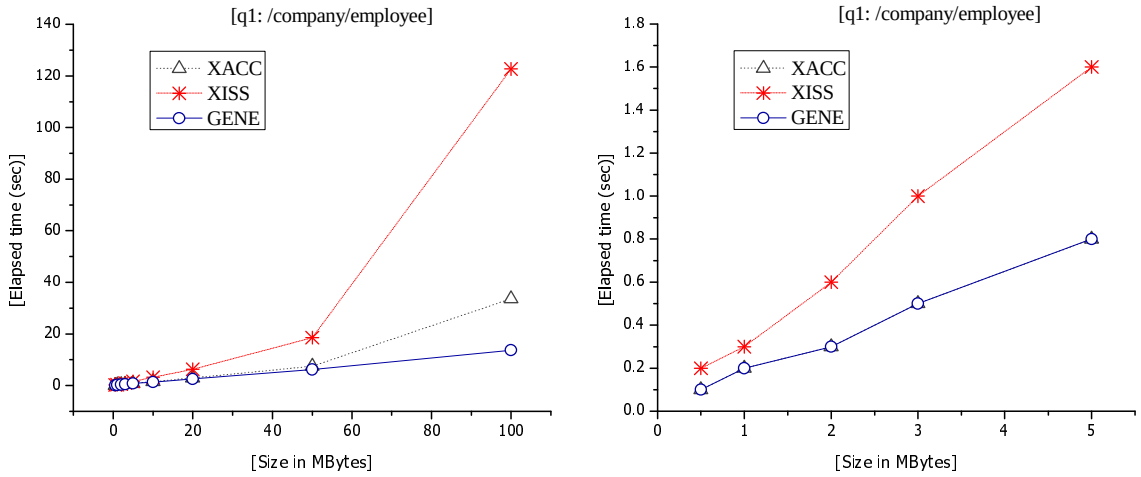
As illustrated in Figure 8.3(a), GENE shows a better performance in a large data. When we zoomed in on Figure 8.3(b), GENE and XACC show a similar query time up to 5MB. Overall, XISS shows the worst performance in both a large and small data. If a result content and data size is large, result reconstruction time linearly increases in GENE and XACC. In XISS, result reconstruction time is not much affected by result content and data size because Cache Manager caches previous results in Cache Directory (see Figure 6.1 in Section 6.1). As a result, GENE shows the best performance for a simple path query such as q1.

As depicted in Figure 8.4(a) and 8.4(b), XACC always shows a better performance than the others whether the data size is small or large. A q2 retrieves all dependents' names of all employees and has nearly two times more results than q1. In Table A.3 (see Appendix A), SQL query uses equality comparison unlike the others. This can be one of the reasons why XACC shows better performance. XISS is a lot faster for result reconstruction since it maintains cache for the previous results.

A q3 has a relatively large quantity because it retrieves all projects' numbers of all employees. It has five times more results than q1. Therefore, elapsed time of q3 takes more than q1 and q2. XACC shows consistently a better performance compared with others. When observing SQL queries in XISS and GENE (see Appendix A), both use

Table 8.6. Query time and reconstruction time details for q1 in shallow data set

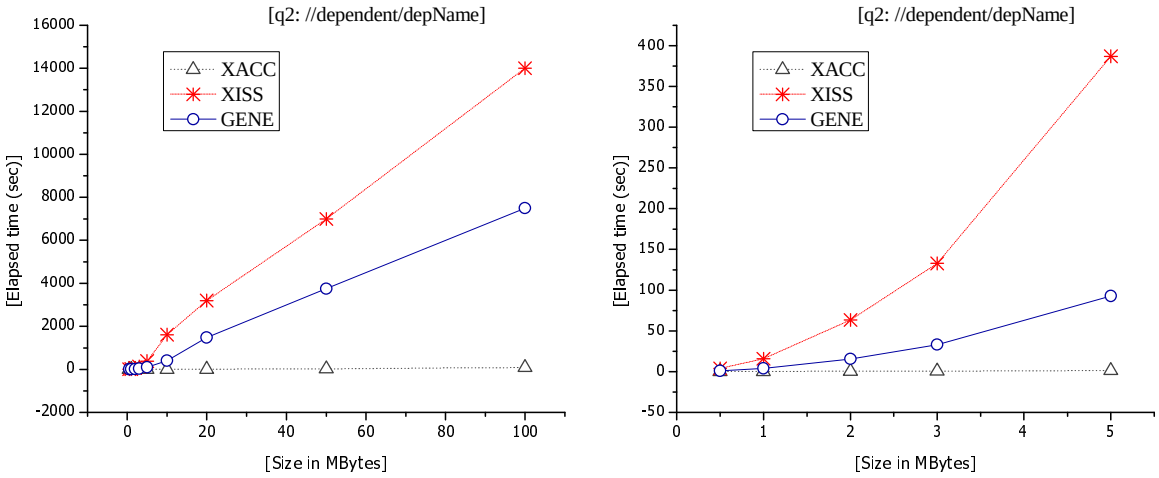
Dataset	q#	Data Size (MB)	XACC | Time (sec)		XISS | Time (sec)		GENE | Time (sec)	
			Querying	Reconstruction	Querying	Reconstruction	Querying	Reconstruction
Shallow	q1	0.5	100	42	173	12	123	405
		1	199	50	316	11	241	748
		2	327	31	643	18	275	1,585
		3	468	35	1,004	15	460	177
		5	755	240	1,557	10	753	3,083
		10	1,496	5,156	3,206	16	1,268	9,473
		20	2,996	7,351	6,322	57	2,526	14,129
		50	7,517	16,882	18,553	37	6,222	49,811
		100	33,727	125,487	122,654	165	13,733	66,482



(a) (b)

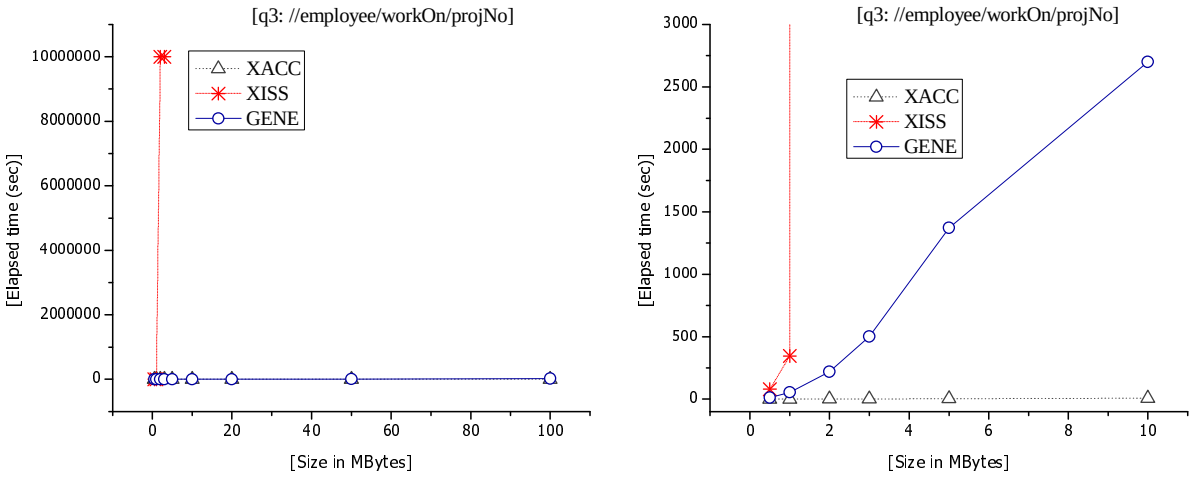
Figure 8.3. (a) [XPath Query Time: Shallow XML data] query1 (b) [Zoomed XPath Query Time: Shallow XML data] query1.

range queries due to indexing structure. However, XACC uses a equality comparison. This gives us a lot better performance whether the data size is small or large. As shown in Appendix A, XACC uses equality comparisons most. Moreover, XISS uses 8 self-joins containing 4 range conditions. GENE also uses 6 self-joins containing 4 range conditions. Those are the most important factors affecting XPath query responses.



(a) (b)

Figure 8.4. (a) [XPath Query Time: Shallow XML data] query2 (b) [Zoomed XPath Query Time: Shallow XML data] query2.



(a) (b)

Figure 8.5. (a) [XPath Query Time: Shallow XML data] query3 (b) [Zoomed XPath Query Time: Shallow XML data] query3.

Table 8.7. Query time and reconstruction time details for q2 in shallow data set

Dataset	q#	Data Size (MB)	XACC | Time (sec)		XISS | Time (sec)		GENE | Time (sec)	
			Querying	Reconstruction	Querying	Reconstruction	Querying	Reconstruction
Shallow	q2	0.5	126	22	3,882	2	1,084	38
		1	268	21	15,889	2	4,092	75
		2	489	11	63,508	2	15,578	140
		3	711	12	132,785	2	33,273	205
		5	1,215	347	386,798	2	92,824	340
		10	2,775	346	1,611,474	2	404,057	694
		20	5,444	312	3,200,000	3,000	1,482,022	800
		50	23,105	1,576	7,000,000	4,000	3,750,000	1,600
		100	86,300	71,710	14,000,000	4,000	7,500,000	2,000

Table 8.8. Query time and reconstruction time details for q3 in shallow data set

Dataset	q#	Data Size (MB)	XACC | Time (sec)		XISS | Time (sec)		GENE | Time (sec)	
			Querying	Reconstruction	Querying	Reconstruction	Querying	Reconstruction
Shallow	q3	0.5	341	25	79,664	2	12,486	39
		1	745	18	346,043	7	54,661	76
		2	1,527	12			219,005	144
		3	2,296	18			501,666	206
		5	3,927	317			1,372,146	342
		10	7,837	223			2,700,000	700
		20	15,832	305			5,400,000	1,400
		50	51,414	3,448			10,800,000	2,800
		100	680,614	109,932			21,600,000	5,200

8.4.3 Performance for Deep Tree

Apart from shallow data set, we prepared 100KB to 2MB for deep data set such that we can see if and how data size affects elapsed query time. We shrank data set size since we cannot measure query time within 20 minutes unlike the shallow data set. We ran three XPath queries, *i.e.*, q4, q5, and q6 for evaluating query performance of deep structured XML data set. Each query has generally deep path (see Section 8.3).

We describe performance results using graphs and discuss the reasons why it shows such features. Discussion of the results focuses on following factors:

- Which indexing technique shows a better performance in any case?
- Which indexing technique shows a better performance for a deep data?
- General query performance over varying sizes of data.
- Result reconstruction time.

A q4 starts from descendant and has 4 consecutive child elements. The number of result nodes are the same as in q5 and q6. As depicted in Figure 8.6(a), XISS shows 245 seconds over 100KB and, for the rest of data set, it took over 20 minutes. Meanwhile, GENE follows a equation below.

$$y = 117.4x^2 + 1.16x \tag{8.1}$$

which is a polynomial performance against the data size. In Appendix A.1, XACC has 4 self-joins and none of them are range conditions. Elapsed query times are expressed as follows:

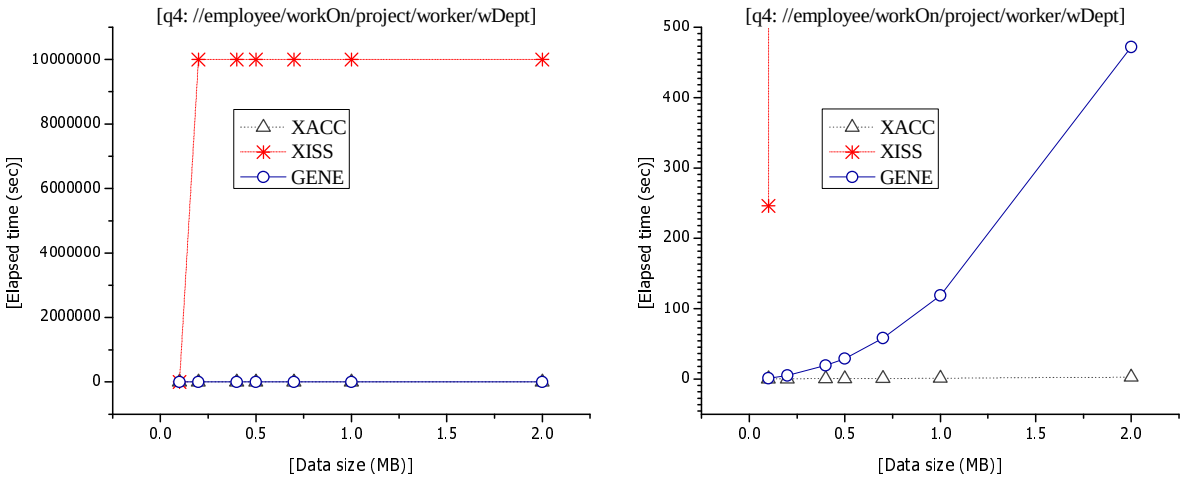
$$y = 1.115x + 0.04 \tag{8.2}$$

which is linear over the data size. Since the size of data set is relatively small, result reconstruction times are not affected by data size or query.

A q5 and q6 have exactly the same results even if XPath queries are slightly different. A q5 is composed of consecutive 2 descendants and 3 childs of instance. A q6 is the longest query and composed of 8 consecutive child elements. Thus q6 has more self-joins than q5 which significantly affect query performance. As depicted in Figure 8.7(a), GENE shows a better performance than XISS and XACC. GENE has 9 range conditions out of 12 self-joins. So far, XACC has not contained range conditions. However, in this query, it contains 3 range conditions out of 6 self-joins. Consequently, if we have descendants or arbitrary instances in XPath query, the performance of XACC is

Table 8.9. Query time and reconstruction time details for q4 in deep data set

Dataset	q#	Data Size (MB)	XACC | Time (sec)		XISS | Time (sec)		GENE | Time (sec)	
			Querying	Reconstruction	Querying	Reconstruction	Querying	Reconstruction
Deep	q4	0.1	155	22	245,841	2	1,260	9
		0.2	259	12			4,857	24
		0.4	471	44			19,479	46
		0.5	604	48			29,232	55
		0.7	850	22			58,326	76
		1	1,262	65			118,834	105
		2	2,706	168			471,835	216



(a) (b)

Figure 8.6. (a) [XPath Query Time: Deep XML data] query4 (b) [Zoomed XPath Query Time: Deep XML data] query4.

markedly reduced. XISS again shows the worst performance when executing q5. It has 8 range conditions out 14 self-joins (see Appendix A.1). This is one of the critical reasons for poor performance. Notice that elapsed time with regard to data size shows polynomial performance as shown in equation (8.3).

$$y = 356.3x^3 + 945.3x^2 - 23x + 1.51 \tag{8.3}$$

Table 8.10. Query time and reconstruction time details for q5 in deep data set

Dataset	q#	Data Size (MB)	XACC | Time (sec)		XISS | Time (sec)		GENE | Time (sec)	
			Querying	Reconstruction	Querying	Reconstruction	Querying	Reconstruction
Deep	q5	0.1	3,305	12	9,117	2	2,201	9
		0.2	23,796	35	41,906	2	9,135	18
		0.4	88,263	49	165,591	2	33,350	33
		0.5	155,517	58	264,962	2	50,509	39
		0.7	374,244	20	575,429	2	104,423	54
		1	678,050	21	1,279,321	2	205,458	82
		2	2,950,537	113			785,148	144

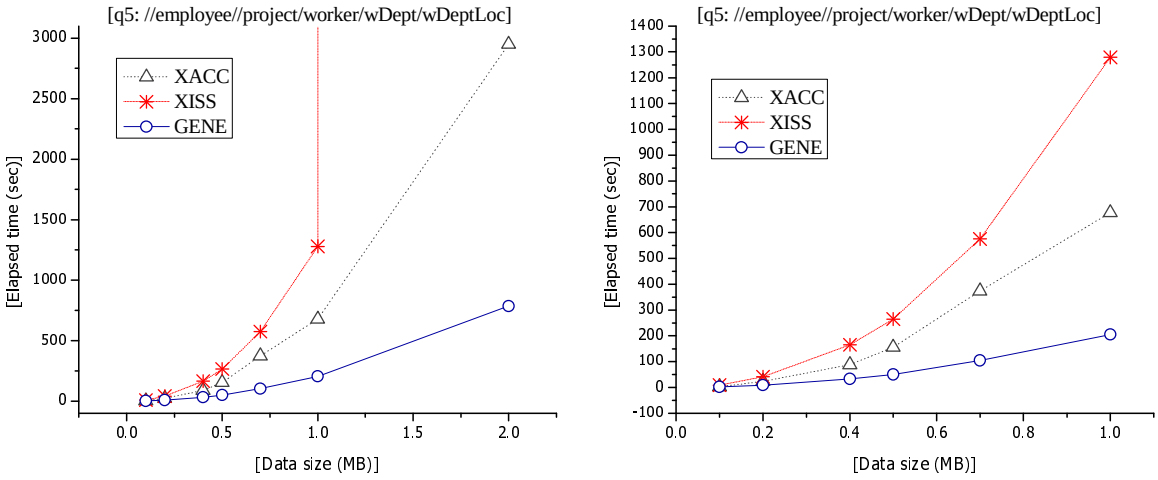


Figure 8.7. (a) [XPath Query Time: Deep XML data] query5 (b) [Zoomed XPath Query Time: Deep XML data] query5.

As is mentioned earlier in q5, q6 has the same number of results as q5. We come to know that different queries give us considerable difference in query time. In q6, XACC gives us the best performance (see equation 8.4) since it uses only equality comparison in

its SQL mapped from XPath (see Appendix A.2). GENE shows polynomial performance as shown in equation (8.5) because it has 14 range conditions out of 21 self-joins. Futher, XISS shows the worst performance since it has 15 range conditions out of 28 self-joins. This leads that we could not measure elapsed time even 100KB within 20 minutes.

$$y = 1.36x + 0.04 \tag{8.4}$$

$$y = -455x^3 + 1032x^2 - 250x + 12.4 \tag{8.5}$$

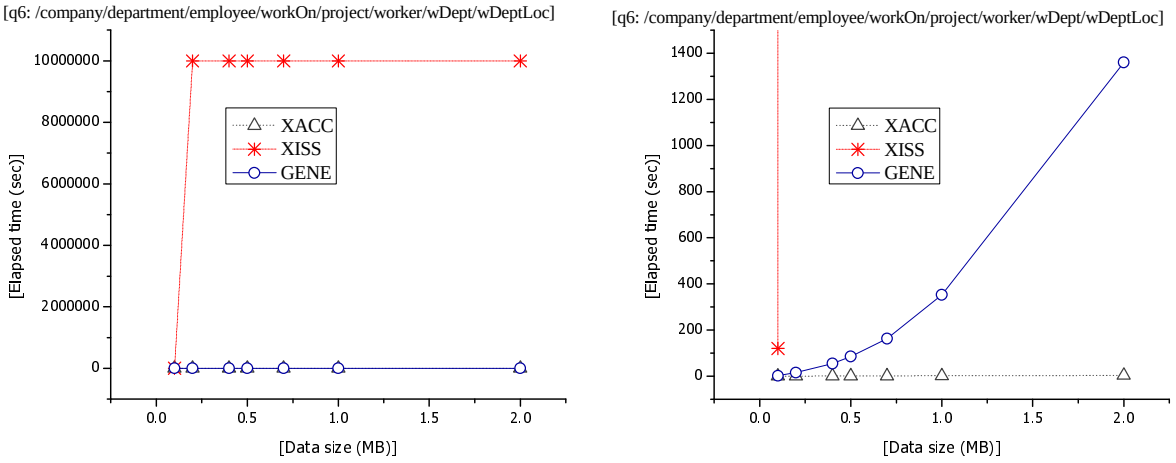
Table 8.11. Query time and reconstruction time details for q6 in deep data set

Dataset	q#	Data Size (MB)	XACC | Time (sec)		XISS | Time (sec)		GENE | Time (sec)	
			Querying	Reconstruction	Querying	Reconstruction	Querying	Reconstruction
Deep	q6	0.1	166	8	120,000		2,101	9
		0.2	345	35			15,167	19
		0.4	614	18			54,513	46
		0.5	778	21			86,034	49
		0.7	1,121	57			162,866	62
		1	1,645	62			353,162	89
		2	3,611	165			1,359,715	156

Since not all indexing techniques support predicates, we tested specific queries only on XACC and GENE. We will discuss the results in the next section.

8.4.4 Performance for Specific Queries

We prepared additional XPath queries containing predicates in XACC and GENE only since XISS supports predicates in a different way. We tested each query, *i.e.*, q7, q9 and q8, q10 on both shallow and deep data sets. Prepared queries consist of generally



(a) (b)

Figure 8.8. (a) [XPath Query Time: Deep XML data] query6 (b) [Zoomed XPath Query Time: Deep XML data] query6.

medium deep path and predicates (see Section 8.3). We describe performance results using graphs and discuss the reasons why they show such characteristics. Results discussion focuses on the following factors:

- ❍ Which indexing technique has a better performance in any case?
- ❍ General query performance over varying sizes of data.
- ❍ Result reconstruction time.

As depicted in Figure 8.9, up to 20MB, both show comparable performance. As data size gets larger, XACC gives us a better performance. An elapsed time with regard to data size in XACC can be expressed as an equation (8.6), which is linear. GENE shows exponential performance as shown in equation (8.7). The reason is that SQL in GENE includes 8 range conditions out of 12 self-joins while XACC contains none of the range conditions out of 4 self-joins. Elapsed times for reconstruction are not much longer because the result retrieves only short content of worker's information.

$$y = 0.07x + 0.16 \tag{8.6}$$

$$y = 343.5e^{-e^{1.96-0.03x}} \tag{8.7}$$

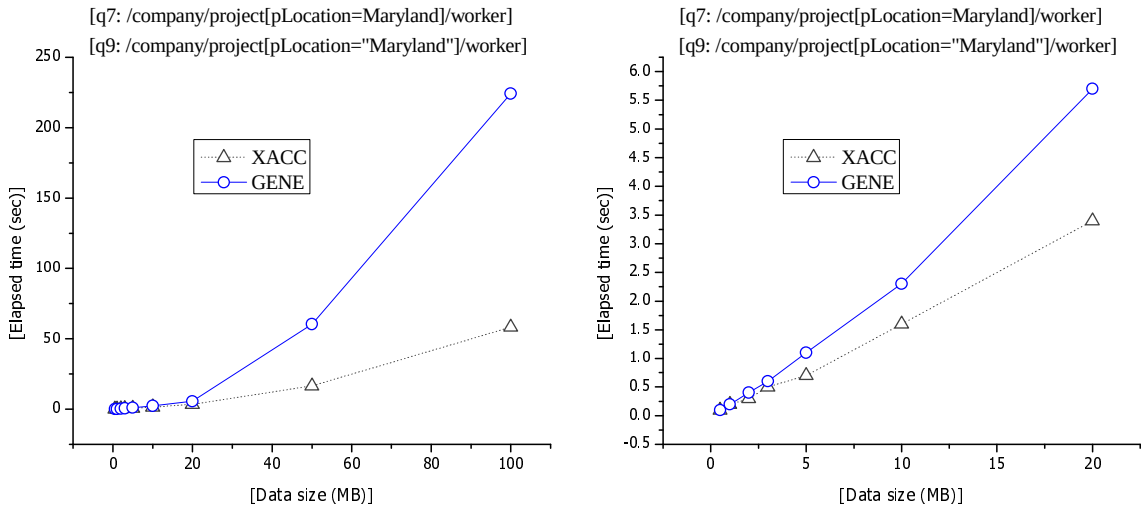


Figure 8.9. (a) [XPath Query Time: Shallow XML data] query7 and query9 (b) [Zoomed XPath Query Time: Shallow XML data] query7 and query9.

A q8 and q10 also have identical number of results over deep structured data. GENE shows a slightly better performance up to 400KB, yet it grows exponentially after 400KB unlike XACC. Therefore, as the data size gets larger, XACC will give us a much better performance. The reason is that SQL in GENE has 10 range conditions out of 15 self-joins. However, XACC has none of the range conditions out of 5 self-joins. Equation 8.8 and 8.9 represent performance behavior of XACC and GENE.

$$y = 0.07x + 0.16 \tag{8.8}$$

$$y = 343.5e^{-e^{1.96-0.03x}} \tag{8.9}$$

Consequently, even in specific queries, XACC shows better query responses than GENE. As far as the data size is small, both of them give us a comparable query time.

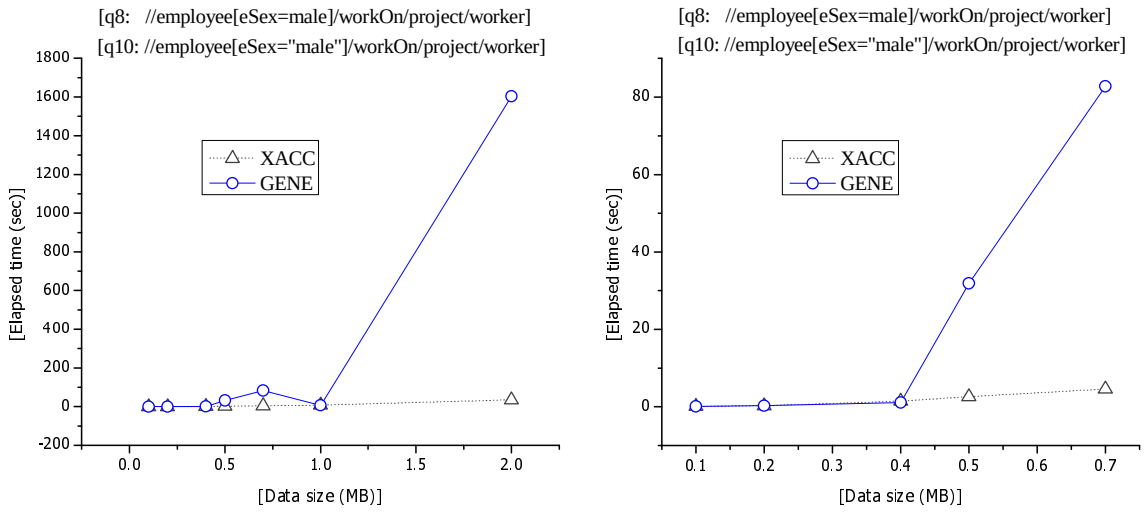


Figure 8.10. (a) [XPath Query Time: Deep XML data] query8 and query10 (b) [Zoomed XPath Query Time: Deep XML data] query8 and query10.

Yet given that the data size becomes larger, XACC always shows a better response time. Because XACC contains less self-joins in SQL statement and none of the range conditions within XPath queries other than GENE.

8.5 Summary

In this chapter, we described the experimental results and analyzed the reasons why we obtain such outcome in more detail. In the section of data loading, we found that XML data loading time significantly depends on whether we use DB connection pool or not. From the sections of shallow/deep data experiments, it turned out that query performance is mostly determined by the number of self-joins and the number of range condition in SQL statement. Generally, XACC showed a better query performance, irrespective of the data size of its structure. Additionally, XISS showed a shorter result reconstruction time in most cases since it caches precious results in Cache Directory.

CHAPTER 9

CONCLUSION AND FUTURE WORK

9.1 Conclusion

This thesis has been primarily motivated by the need of performance evaluation and comparison among numbering based XML indexing techniques. We generated shallow/deep structured XML data sets to see if its structures affect query response time using ToXgene. GENE (Generic numbering based indexing technique), XISS (Range based numbering technique), and XACC (Dimension based numbering technique) completely reside on top of a relational database utilizing mutually different numbering schemes. Since all three techniques convert XML documents into encoded numbers, we can conveniently put them into relational database as a structured format. As such, we can benefit from features such as B^+-tree index in relational database.

In Chapter 8, we analyzed the experimental results regarding three aspects, namely, varying size of the XML data, distinct XPath queries having different features, shallow/deep structured XML data sets. While querying, indispensably we had to use the number of joins in SQL statement, depending on the length of the XPath query. It turned out to be significant querying time bottleneck. Further, if there exists range conditions in SQL, we will confront another serious falling-off in querying performance. Above issues were adversely caused by using relational database. Result reconstruction time barely depends on the length of its context node. Hence to overcome limitations of numbering based XML indexing, we are required to use a numbering scheme accommodating less joins as well as less range conditions in SQL statement.

9.2 Future Work

This thesis is part of an ongoing research in a study of XML indexing techniques. Thus far, we have surveyed numerous XML indexing techniques. Then we classified them into four major categories, *i.e.,* sequence based, numbering based, structure based, keyword based. We evaluated and compared three techniques (XACC, XISS, GENE) among numbering based indexing techniques. Now we are left with a challenge to compare indexing techniques by inter-categories. For example, ViST [28], PRIX [29] in sequence based, A(k) [24], D(k) [25], and M(k) [26] in structure based could be compared in the next research. Before comparing indexing techniques, we should always consider architectures and system environments in an objective manner. Through those comparisons, we expect that we can find out a way of efficient indexing method of XML documents.

APPENDIX A

MAPPING XPATH QUERY TO RELATIONAL SQL

In this appendix, we present mapping common/specific XPath query to relational SQL in each indexing technique in tabular format.

A.1 Mapping XPath Query to SQL in XISS

Table A.1. Mapping common XPath query to relational SQL in XISS

Dataset	q#	XPath Query	SQL Query
Shallow	q1	/company/employee	SELECT DISTINCT v0.* FROM accel_company50m v1,accel_company50m v0 WHERE v1.tag="company" and v1.pre=v0.par and v0.tag="employee" and v0.kind="e";
	q2	//dependent/depName	SELECT DISTINCT v0.* FROM accel_company50m v1,accel_company50m v0 WHERE v1.tag="dependent" and v1.pre=v0.par and v0.tag="depName" and v0.kind="e";
	q3	//employee/workOn/projNo	SELECT DISTINCT v0.* FROM accel_company1m v2,accel_company1m v1,accel_company1m v0 WHERE v2.tag="employee" and v2.pre=v1.par and v1.tag="workOn" and v1.pre=v0.par and v0.tag="projNo" and v0.kind="e";
Dataset	q#	XPath Query	SQL Query
Deep	q4	//employee/workOn/project/worker/wDept	SELECT DISTINCT v0.* FROM accel_company100k v4,accel_company100k v3,accel_company100k v2,accel_company100k v1,accel_company100k v0 WHERE v4.tag="employee" and v4.pre=v3.par and v3.tag="workOn" and v3.pre=v2.par and v2.tag="project" and v2.pre=v1.par and v1.tag="worker" and v1.pre=v0.par and v0.tag="wDept" and v0.kind="e";
	q5	//employee/*/project/worker/wDept/wDeptLoc	SELECT DISTINCT v0.* FROM accel_company100k v4,accel_company100k v3,accel_company100k v2,accel_company100k v1,accel_company100k v0 WHERE v4.tag="employee" and v4.pre!=v3.par and v3.pre>v4.pre and v3.post<v4.post and v3.tag="project" and v3.pre=v2.par and v2.tag="worker" and v2.pre=v1.par and v1.tag="wDept" and v1.pre=v0.par and v0.tag="wDeptLoc" and v0.kind="e";
	q6	/company/department/employee/workOn/project/worker/wDept/wDeptLoc	SELECT DISTINCT v0.* FROM accel_company100k v7,accel_company100k v6,accel_company100k v5,accel_company100k v4,accel_company100k v3,accel_company100k v2,accel_company100k v1,accel_company100k v0 WHERE v7.tag="company" and v7.pre=v6.par and v6.tag="department" and v6.pre=v5.par and v5.tag="employee" and v5.pre=v4.par and v4.tag="workOn" and v4.pre=v3.par and v3.tag="project" and v3.pre=v2.par and v2.tag="worker" and v2.pre=v1.par and v1.tag="wDept" and v1.pre=v0.par and v0.tag="wDeptLoc" and v0.kind="e";

A.2 Mapping XPath Query to SQL in XACC

Table A.2. Mapping common XPath query to relational SQL in XACC

Dataset	q#	XPath Query	SQL Query
Shallow	q1	/company/employee	SELECT DISTINCT v0.* FROM accel_company50m v1,accel_company50m v0 WHERE v1.tag="company" and v1.pre=v0.par and v0.tag="employee" and v0.kind="e";
	q2	//dependent/depName	SELECT DISTINCT v0.* FROM accel_company50m v1,accel_company50m v0 WHERE v1.tag="dependent" and v1.pre=v0.par and v0.tag="depName" and v0.kind="e";
	q3	//employee/workOn/projNo	SELECT DISTINCT v0.* FROM accel_company1m v2,accel_company1m v1,accel_company1m v0 WHERE v2.tag="employee" and v2.pre=v1.par and v1.tag="workOn" and v1.pre=v0.par and v0.tag="projNo" and v0.kind="e";
Dataset	q#	XPath Query	SQL Query
Deep	q4	//employee/workOn/project/worker/wDept	SELECT DISTINCT v0.* FROM accel_company100k v4,accel_company100k v3,accel_company100k v2,accel_company100k v1,accel_company100k v0 WHERE v4.tag="employee" and v4.pre=v3.par and v3.tag="workOn" and v3.pre=v2.par and v2.tag="project" and v2.pre=v1.par and v1.tag="worker" and v1.pre=v0.par and v0.tag="wDept" and v0.kind="e";
	q5	//employee/*/project/worker/wDept/wDeptLoc	SELECT DISTINCT v0.* FROM accel_company100k v4,accel_company100k v3,accel_company100k v2,accel_company100k v1,accel_company100k v0 WHERE v4.tag="employee" and v4.pre!=v3.par and v3.pre>v4.pre and v3.post<v4.post and v3.tag="project" and v3.pre=v2.par and v2.tag="worker" and v2.pre=v1.par and v1.tag="wDept" and v1.pre=v0.par and v0.tag="wDeptLoc" and v0.kind="e";
	q6	/company/department/employee/workOn/project/worker/wDept/wDeptLoc	SELECT DISTINCT v0.* FROM accel_company100k v7,accel_company100k v6,accel_company100k v5,accel_company100k v4,accel_company100k v3,accel_company100k v2,accel_company100k v1,accel_company100k v0 WHERE v7.tag="company" and v7.pre=v6.par and v6.tag="department" and v6.pre=v5.par and v5.tag="employee" and v5.pre=v4.par and v4.tag="workOn" and v4.pre=v3.par and v3.tag="project" and v3.pre=v2.par and v2.tag="worker" and v2.pre=v1.par and v1.tag="wDept" and v1.pre=v0.par and v0.tag="wDeptLoc" and v0.kind="e";

Table A.3. Mapping specific XPath query to relational SQL in XACC

Dataset	q#	XPath Query	SQL Query
Shallow	q7	/company/project[pLocation=Maryland]/worker	SELECT DISTINCT v0.* FROM accel_company500k v4,accel_company500k v3,accel_company500k p3,accel_company500k c3,accel_company500k v0 WHERE v4.tag="company" and v4.pre=v3.par and v3.tag="project" and v3.pre=v0.par and p3.tag="pLocation" and v3.pre=p3.par and c3.pre=p3.pre+1 and c3.tag="Maryland" and v0.tag="worker" and v0.kind="e";
Dataset	q#	XPath Query	SQL Query
Deep	q8	//employee[eSex=male]/workOn/project/worker	SELECT DISTINCT v0.* FROM accel_company400k v5,accel_company400k p5,accel_company400k c5,accel_company400k v2,accel_company400k v1,accel_company400k v0 WHERE v5.tag="employee" and v5.pre=v2.par and p5.tag="eSex" and v5.pre=p5.par and c5.pre=p5.pre+1 and c5.tag="male" and v2.tag="workOn" and v2.pre=v1.par and v1.tag="project" and v1.pre=v0.par and v0.tag="worker" and v0.kind="e";

A.3 Mapping XPath Query to SQL in GENE

Table A.4. Mapping common XPath query to relational SQL in GENE

Dataset	q#	XPath Query	SQL Query
Shallow	q1	/company/employee	SELECT e2.* FROM Element e1, Element e2 WHERE e1.tagname = "company" and e1.level = 0 and e2.tagname = "employee" and e2.begin > e1.begin and e2.end < e1.end and e2.level = e1.level+1;
	q2	//dependent/depName	SELECT e2.* FROM Element e1, Element e2 WHERE e1.tagname = "dependent" and e2.tagname = "depName" and e2.begin > e1.begin and e2.end < e1.end and e2.level = e1.level+1;
	q3	//employee/workOn/projNo	SELECT e3.* FROM Element e1, Element e2, Element e3 WHERE e1.tagname = "employee" and e2.tagname = "workOn" and e2.begin > e1.begin and e2.end < e1.end and e2.level = e1.level+1 and e3.tagname = "projNo" and e3.begin > e2.begin and e3.end < e2.end and e3.level = e2.level+1;
Dataset	q#	XPath Query	SQL Query
Deep	q4	//employee/workOn/project/worker/wDept	SELECT e5.* FROM Element e1, Element e2, Element e3, Element e4, Element e5 WHERE e1.tagname = "employee" and e2.tagname = "workOn" and e2.begin > e1.begin and e2.end < e1.end and e2.level = e1.level+1 and e3.tagname = "project" and e3.begin > e2.begin and e3.end < e2.end and e3.level = e2.level+1 and e4.tagname = "worker" and e4.begin > e3.begin and e4.end < e3.end and e4.level = e3.level+1 and e5.tagname = "wDept" and e5.begin > e4.begin and e5.end < e4.end and e5.level = e4.level+1;
	q5	//employee/*/project/worker/wDept/wDeptLoc	SELECT e5.* FROM Element e1, Element e2, Element e3, Element e4, Element e5 WHERE e1.tagname = "employee" and e2.tagname = "project" and e2.begin > e1.begin and e2.end < e1.end and e2.level >= e1.level+1 and e3.tagname = "worker" and e3.begin > e2.begin and e3.end < e2.end and e3.level = e2.level+1 and e4.tagname = "wDept" and e4.begin > e3.begin and e4.end < e3.end and e4.level = e3.level+1 and e5.tagname = "wDeptLoc" and e5.begin > e4.begin and e5.end < e4.end and e5.level = e4.level+1;
	q6	/company/department/employee/workOn/project /worker/wDept/wDeptLoc	SELECT e8.* FROM Element e1, Element e2, Element e3, Element e4, Element e5, Element e6, Element e7, Element e8 WHERE e1.tagname = "company" and e1.level = 0 and e2.tagname = "department" and e2.begin > e1.begin and e2.end < e1.end and e2.level = e1.level+1 and e3.tagname = "employee" and e3.begin > e2.begin and e3.end < e2.end and e3.level = e2.level+1 and e4.tagname = "workOn" and e4.begin > e3.begin and e4.end < e3.end and e4.level = e3.level+1 and e5.tagname = "project" and e5.begin > e4.begin and e5.end < e4.end and e5.level = e4.level+1 and e6.tagname = "worker" and e6.begin > e5.begin and e6.end < e5.end and e6.level = e5.level+1 and e7.tagname = "wDept" and e7.begin > e6.begin and e7.end < e6.end and e7.level = e6.level+1 and e8.tagname = "wDeptLoc" and e8.begin > e7.begin and e8.end < e7.end and e8.level = e7.level+1;

Table A.5. Mapping specific XPath query to relational SQL in GENE

Dataset	q#	XPath Query	SQL Query
Shallow	q7	/company/project[pLocation=Maryland]/worker	SELECT e4.* FROM Element e1, Element e2, Element e3, Content c, Element e4 WHERE e1.tagname = "company" and e1.level = 0 and e2.tagname = "project" and e2.begin > e1.begin and e2.end < e1.end and e2.level = e1.level+1 and e3.tagname = "pLocation" and e3.begin > e2.begin and e3.end < e2.end and e3.level = e2.level+1 and c.text = "Montana" and c.position > e3.begin and c.position < e3.end and c.level = e3.level and e4.tagname = "worker" and e4.begin > e2.begin and e4.end < e2.end and e4.level = e2.level+1;
Dataset	q#	XPath Query	SQL Query
Deep	q8	//employee[eSex=male]/workOn/project/worker	SELECT e5.* FROM Element e1, Element e2, Content c, Element e3, Element e4, Element e5 WHERE e1.tagname = "employee" and e2.tagname = "eSex" and e2.begin > e1.begin and e2.end < e1.end and e2.level = e1.level+1 and c.text = "male" and c.position > e2.begin and c.position < e2.end and c.level = e2.level and e3.tagname = "workOn" and e3.begin > e1.begin and e3.end < e1.end and e3.level = e1.level+1 and e4.tagname = "project" and e4.begin > e3.begin and e4.end < e3.end and e4.level = e3.level+1 and e5.tagname = "worker" and e5.begin > e4.begin and e5.end < e4.end and e5.level =

APPENDIX B

COMPLETE TEMPLATE SPECIFICATION FILES

In this appendix, we present the fully specified TSL files for both shallow and deep tree-structured XML data sets.

B.1 A TSL File for the Shallow XML Data

Table B.1. A fully specified TSL file for Shallow tree

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE tox-template SYSTEM 'http://www.cs.toronto.edu/tox/toxgene/ToXgene2.dtd'>
<tox-template>
<!– enumerate different types of distribution –>
<!– generate normal distribution of the number of dependents –>
<tox-distribution name="n1" type="normal" minInclusive="0" maxInclusive="5" mean="2" variance="2"/>
<!– generate normal distribution of the number of workers in each project –>
<tox-distribution name="n2" type="normal" minInclusive="1" maxInclusive="20" mean="10" variance="3"/>
<!– generate normal distribution of the birth date of employees –>
<tox-distribution name="n3" type="normal" minInclusive="0" maxInclusive="11321" mean="5660"
variance="2000"/>
<!– generate exponential distribution of working hours of employees –>
<tox-distribution name="e1" type="exponential" minInclusive="10" maxInclusive="40" mean="23"/>

<tox-distribution name="c1" type="constant" minInclusive="1" maxInclusive="1"/>
<tox-distribution name="e2" type="exponential" minInclusive="1" maxInclusive="10" mean="3"/>
<tox-distribution name="u1" type="uniform" minInclusive="1" maxInclusive="10" mean="3"/>

<!– This distribution will be used to give discounts for some books –>
<tox-distribution name="salary" type="user-defined" minInclusive="0" maxInclusive="30">
<enumeration value="0" tox-percent="50"/>
<enumeration value="5" tox-percent="25"/>
<enumeration value="10" tox-percent="15"/>
<enumeration value="30" tox-percent="10"/>
</tox-distribution>

<tox-list name="department_list" unique="dName" readFrom="input/departments.xml">
<element name="dName" type="string"/>
</tox-list>
<simpleType name="dName_type">
<restriction base="string">
<tox-sample path="[department_list/dName]">
<tox-expr value="[!]"/>
</tox-sample>
</restriction>
</simpleType>

<simpleType name="dNum_type">
<restriction base="positiveInteger">
<minInclusive value="01"/>
<maxInclusive value="53"/>
</restriction>
</simpleType>

<tox-list name="sex_list" unique="sex" readFrom="input/genders.xml">
<element name="sex" type="string"/>
</tox-list>
<simpleType name="sex_type">
```

```
<restriction base="string">
<tox-sample path="[sex_list/sex]">
<tox-expr value="[!]"/>
</tox-sample>
</restriction>
</simpleType>
<tox-list name="relationship_list" unique="relationship" readFrom="input/relationships.xml">
<element name="relationship" type="string"/>
</tox-list>
<simpleType name="relationship_type">
<restriction base="string">
<tox-sample path="[relationship_list/relationship]">
<tox-expr value="[!]"/>
</tox-sample>
</restriction>
</simpleType>

<simpleType name="projectName_type">
<restriction base="string">
<minLength value="5"/>
<maxLength value="20"/>
<tox-string type="text"/>
</restriction>
</simpleType>

<simpleType name="projectNo_type">
<restriction base="string">
<pattern value="[0-9]7"/>
</restriction>
</simpleType>

<simpleType name="eSSN_type">
<restriction base="string">
<pattern value="[0-9]9"/>
</restriction>
</simpleType>

<simpleType name="start_date">
<restriction base="date">
<minInclusive value="1995-01-01"/>
<maxInclusive value="2006-31-01"/>
<tox-format value="MM/dd/yyyy"/>
</restriction>
</simpleType>
<!–
<simpleType name="birth_date">
<restriction base="date">
<minInclusive value="1950-01-01"/>
<maxInclusive value="1980-31-12"/>
<tox-format value="MM/dd/yyyy"/>
</restriction>
</simpleType>
–>
<simpleType name="birth_date">
<restriction base="date">
<tox-date start-date="1950-01-01" end-date="1980-31-12" tox-distribution="n3" format="MM/dd/yyyy"/>
</restriction>
</simpleType>
<simpleType name="eSal_type">
<restriction base="positiveInteger">
<minInclusive value="40"/>
```

```
<maxInclusive value="99"/>
</restriction>
</simpleType>
<simpleType name="proj_type">
<restriction base="string">
<pattern value="[0-9]3"/>
</restriction>
</simpleType>
<tox-list name="emp_list" unique="eName/fName,eName/lName">
<element name="eName" minOccurs="5000" maxOccurs="5000">
<complexType>
<element name="fName">
<simpleType>
<restriction base="string">
<tox-string type="fname"/>
</restriction>
</simpleType>
</element>
<element name="lName">
<simpleType>
<restriction base="string">
<tox-string type="lname"/>
</restriction>
</simpleType>
</element>
</complexType>
</element>
</tox-list>
<tox-list name="address_list">
<element name="address" minOccurs="5000" maxOccurs="5000">
<complexType>
<element name="No">
<simpleType>
<restriction base="positiveInteger">
<minInclusive value="1000"/>
<maxInclusive value="9999"/>
</restriction>
</simpleType>
</element>
<element name="Street">
<simpleType>
<restriction base="string">
<tox-string type="word"/>
</restriction>
</simpleType>
</element>
<element name="City">
<simpleType>
<restriction base="string">
<tox-string type="city"/>
</restriction>
</simpleType>
</element>
<element name="State">
<simpleType>
<restriction base="string">
<tox-string type="province"/>
</restriction>
</simpleType>
</element>
</complexType>
</element>
</tox-list>
```

```
<!– format department list –>
<tox-list name=“dept_list”>
<element name=“department” minOccurs=“53” maxOccurs=“53”>
<complexType>
<element name=“dName” type=“dName_type”/>
<element name=“dNumber” type=“dNum_type”/>
<element name=“dMgrSSN” type=“eSSN_type”/>
<element name=“dMgrStartDate” type=“start_date”/>
<element name=“dLocation” minOccurs=“10” maxOccurs=“10”>
<simpleType>
<restriction base=“string”>
<tox-string type=“city”/>
</restriction>
</simpleType>
</element>
</complexType>
</element>
</tox-list>

<!– format employee list –>
<tox-list name=“employee_list”>
<element name=“employee” minOccurs=“10000” maxOccurs=“10000”>
<complexType>
<element name=“eName”>
<simpleType>
<restriction base=“string”>
<tox-sample path=“[emp_list/eName]” duplicates=“yes”>
<tox-expr value=“[fName]#’ ’#[lName]”/>
</tox-sample>
</restriction>
</simpleType>
</element>
<element name=“eSSN” type=“eSSN_type”/>
<element name=“eSex” type=“sex_type”/>
<element name=“eSalary” type=“eSal_type”/>
<element name=“eDoB” type=“birth_date”/>
<element name=“eDno”>
<simpleType>
<restriction base=“string”>
<tox-sample path=“[dept_list/department]”>
<tox-expr value=“[dNumber]”/>
</tox-sample>
</restriction>
</simpleType>
</element>
<element name=“eSupervisorSSN”>
<simpleType>
<restriction base=“string”>
<tox-sample path=“[dept_list/department]”>
<tox-expr value=“[dMgrSSN]”/>
</tox-sample>
</restriction>
</simpleType>
</element>
<element name=“address”>
<simpleType>
<restriction base=“string”>
<tox-sample path=“[address_list/address]” duplicates=“no”>
<tox-expr value=“[No]#’ ’#[Street]#’ ’#[City]#’ ’#[State]”/>
</tox-sample>
</restriction>
</simpleType>
</element>
```

```
<element name="workOn" minOccurs="10" maxOccurs="10">
<complexType>
<element name="projNo" type="proj_type"/>
<element name="hours">
<simpleType>
<restriction base="positiveInteger">
<tox-number tox-distribution="e1"/>
<!–
<minInclusive value="10"/>
<maxInclusive value="40"/>
–>
</restriction>
</simpleType>
</element>
</complexType>
</element>
<element name="dependent" minOccurs="5" maxOccurs="5">
<complexType>
<element name="depName">
<simpleType>
<restriction base="string">
<tox-string type="fname"/>
</restriction>
</simpleType>
</element>
<element name="dSex" type="sex_type"/>
<element name="depDoB" type="birth_date"/>
<element name="relationship" type="relationship_type"/>
</complexType>
</element>
</complexType>
</element>
</tox-list>

<!– format project list –>
<tox-list name="project_list">
<element name="project" minOccurs="30" maxOccurs="30">
<complexType>
<element name="pName" type="projectName_type"/>
<element name="pNumber" type="projectNo_type"/>
<element name="pLocation" minOccurs="1" maxOccurs="1">
<simpleType>
<restriction base="string">
<tox-string type="province"/>
</restriction>
</simpleType>
</element>
<element name="pDnum">
<simpleType>
<restriction base="string">
<tox-sample path="[dept_list/department]">
<tox-expr value="[dNumber]"/>
</tox-sample>
</restriction>
</simpleType>
</element>
<element name="worker" minOccurs="20" maxOccurs="20">
<complexType>
<element name="SSN">
<simpleType>
<restriction base="string">
<tox-sample path="[employee_list/employee]">
<tox-expr value="[eSSN]"/>
```

```
</tox-sample>
</restriction>
</simpleType>
</element>
<element name="hours">
<simpleType>
<restriction base="positiveInteger">
<tox-sample path="[employee_list/employee/workOn]">
<tox-expr value="[hours]"/>
</tox-sample>
</restriction>
</simpleType>
</element>
</complexType>
</element >
</complexType>
</element>
</tox-list>

<!–
This is the main company XML document; it has one "company" root entry.
–>
<tox-document name="output/company">
<element name="company" minOccurs="1" maxOccurs="1">
<complexType>
<element name="department" minOccurs="10" maxOccurs="53">
<complexType>
<tox-scan path="[dept_list/department]" name="d">
<element name="dName">
<tox-expr value="[$d/dName]"/>
</element>
<element name="dNumber">
<tox-expr value="[dNumber]" format="00"/>
</element>
<element name="dMgrSSN">
<tox-expr value="[dMgrSSN]"/>
</element>
<element name="dMgrStartDate">
<tox-expr value="[dMgrStartDate]"/>
</element>
<element name="dLocation" minOccurs="1" maxOccurs="10">
<simpleType>
<restriction base="string">
<tox-scan path="[$d/dLocation]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
</element>
</tox-scan>
</complexType>
</element>

<element name="employee" minOccurs="4150" maxOccurs="4150">
<complexType>
<tox-scan path="[employee_list/employee]" name="e">
<element name="eName">
<simpleType>
<restriction base="string">
<tox-scan path="[$e/eName]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
```

```
</element>
<element name="eSSN">
<tox-expr value="[eSSN]"/>
</element>
<element name="eSex">
<tox-expr value="[eSex]"/>
</element>
<element name="eSalary">
<tox-expr value="[eSalary]*1000" format="00,000"/>
</element>
<element name="eDoB">
<tox-expr value="[eDoB]"/>
</element>
<element name="eDno">
<tox-expr value="[eDno]"/>
</element>
<element name="eSupervisorSSN">
<tox-expr value="[eSupervisorSSN]"/>
</element>
<element name="Address">
<simpleType>
<restriction base="string">
<tox-scan path="[$e/address]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
</element>
<element name="workOn" minOccurs="0" maxOccurs="10">
<complexType>
<element name="projNo">
<simpleType>
<restriction base="string">
<tox-scan path="[$e/workOn/projNo]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
</element>
<element name="hours">
<simpleType>
<restriction base="positiveInteger">
<tox-scan path="[$e/workOn/hours]">
<tox-expr value="[!]" format="00"/>
</tox-scan>
</restriction>
</simpleType>
</element>
</complexType>
</element>
<element name="dependent" minOccurs="0" maxOccurs="unbounded" tox-distribution="n1">
<complexType>
<element name="depName">
<simpleType>
<restriction base="string">
<tox-scan path="[$e/dependent/depName]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
</element>
<element name="dSex">
<simpleType>
```

```
<restriction base="string">
<tox-scan path="[$e/dependent/dSex]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
</element>
<element name="depDoB">
<simpleType>
<restriction base="string">
<tox-scan path="[$e/dependent/depDoB]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
</element>
<element name="relationship">
<simpleType>
<restriction base="string">
<tox-scan path="[$e/dependent/relationship]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
</element>
</complexType>
</element>
</tox-scan>
</complexType>
</element>

<element name="project" maxOccurs="unbounded">
<complexType>
<tox-scan path="[project_list/project]" name="p">
<element name="pName">
<tox-expr value="[pName]"/>
</element>
<element name="pNumber">
<tox-expr value="[pNumber]"/>
</element>
<element name="pLocation">
<tox-expr value="[pLocation]"/>
</element>
<element name="pDnum">
<tox-expr value="[pDnum]"/>
</element>
<element name="worker" minOccurs="1" maxOccurs="unbounded" tox-distribution="n2">
<complexType>
<element name="SSN">
<simpleType>
<restriction base="string">
<tox-scan path="[$p/worker/SSN]">
<tox-expr value="[!]"/>
</tox-scan>
</restriction>
</simpleType>
</element>
<element name="hours">
<simpleType> <restriction base="positiveInteger"> <tox-scan path="[$p/worker/hours]"> <tox-expr value="[!]"/>
</tox-scan> </restriction> </simpleType> </element> </complexType> </element> </tox-scan>
</complexType> </element> </complexType> </element> </tox-document> </tox-template>
```

B.2 A TSL File for the Deep XML Data

Table B.2. A fully specified TSL file for Deep tree

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE tox-template SYSTEM 'http://www.cs.toronto.edu/tox/toxgene/ToXgene2.dtd'>
<tox-template> <!– enumerate different types of distribution –>
<!– generate normal distribution of the number of dependents –>
<tox-distribution name="n1" type="normal" minInclusive="0" maxInclusive="5" mean="2" variance="2"/>
<!– generate normal distribution of the number of workers in each project –>
<tox-distribution name="n2" type="normal" minInclusive="1" maxInclusive="20" mean="10" variance="3"/>
<!– generate normal distribution of the birth date of employees –>
<tox-distribution name="n3" type="normal" minInclusive="0" maxInclusive="11321" mean="5660"
variance="2000"/>
<!– generate exponential distribution of working hours of employees –>
<tox-distribution name="e1" type="exponential" minInclusive="10" maxInclusive="40" mean="23"/>
<tox-distribution name="c1" type="constant" minInclusive="1" maxInclusive="1"/>
<tox-distribution name="e2" type="exponential" minInclusive="1" maxInclusive="10" mean="3"/>
<tox-distribution name="u1" type="uniform" minInclusive="1" maxInclusive="10" mean="3"/>
<!– This distribution will be used to give discounts for some books –>
<tox-distribution name="salary" type="user-defined" minInclusive="0" maxInclusive="30">
<enumeration value="0" tox-percent="50"/>
<enumeration value="5" tox-percent="25"/>
<enumeration value="10" tox-percent="15"/>
<enumeration value="30" tox-percent="10"/>
</tox-distribution>
<tox-list name="department_list" unique="dName" readFrom="input/departments.xml"> <element name="dName"
type="string"/> </tox-list> <simpleType name="dName_type"> <restriction base="string"> <tox-sample
path="[department_list/dName]">
<tox-expr value="[!]"/> </tox-sample> </restriction> </simpleType>
<simpleType name="dNum_type"> <restriction base="positiveInteger"> <minInclusive value="01"/>
<maxInclusive value="53"/> <!– <tox-number sequential="yes"/> –>
</restriction> </simpleType>
<tox-list name="sex_list" unique="sex" readFrom="input/genders.xml"> <element name="sex" type="string"/>
</tox-list> <simpleType name="sex_type"> <restriction base="string"> <tox-sample path="[sex_list/sex]"> <tox-
expr value="[!]"/>
</tox-sample> </restriction> </simpleType> <tox-list name="relationship_list" unique="relationship"
readFrom="input/relationships.xml">
<element name="relationship" type="string"/> </tox-list> <simpleType name="relationship_type"> <restriction
base="string"> <tox-sample path="[relationship_list/relationship]">
<tox-expr value="[!]"/> </tox-sample> </restriction> </simpleType>
<simpleType name="projectName_type"> <restriction base="string"> <minLength value="5"/> <maxLength
value="20"/> <tox-string type="text"/>
</restriction> </simpleType>
<simpleType name="projectNo_type"> <restriction base="string">
<pattern value="[0-9]7"/> </restriction> </simpleType>
<simpleType name="eSSN_type">
<restriction base="string"> <pattern value="[0-9]9"/> </restriction> </simpleType>
<simpleType name="start_date"> <restriction base="date"> <minInclusive value="1995-01-01"/> <maxInclusive
value="2006-31-01"/> <tox-format value="MM/dd/yyyy"/> </restriction>
</simpleType> <!– <simpleType name="birth_date"> <restriction base="date"> <minInclusive value="1950-01-
01"/>
<maxInclusive value="1980-31-12"/> <tox-format value="MM/dd/yyyy"/> </restriction> </simpleType> –>
<simpleType name="birth_date"> <restriction base="date"> <tox-date start-date="1950-01-01" end-date="1980-
31-12" tox-distribution="n3" format="MM/dd/yyyy"/> </restriction> </simpleType>
<simpleType name="eSal_type"> <restriction base="positiveInteger"> <minInclusive value="40"/> <maxInclusive
value="99"/>
</restriction> </simpleType>
```

<simpleType name=“proj_type”> <restriction base=“string”>
<pattern value=“[0-9]3”/> </restriction> </simpleType> <tox-list name=“emp_list” unique=“eName/fName,eName/lName”>
<element name=“eName” minOccurs=“5000” maxOccurs=“5000”> <complexType> <element name=“fName”>
<simpleType> <restriction base=“string”>
<tox-string type=“fname”/> </restriction> </simpleType> </element> <element name=“lName”>
<simpleType> <restriction base=“string”> <tox-string type=“lname”/> </restriction> </simpleType>
</element> </complexType> </element> </tox-list>
<tox-list name=“address_list”> <element name=“address” minOccurs=“5000” maxOccurs=“5000”> <complexType>
<element name=“No”> <simpleType>
<restriction base=“positiveInteger”> <minInclusive value=“1000”/> <maxInclusive value=“9999”/> </restriction>
</simpleType>
</element> <element name=“Street”> <simpleType> <restriction base=“string”> <tox-string type=“word”/>
</restriction> </simpleType> </element> <element name=“City”> <simpleType>
<restriction base=“string”> <tox-string type=“city”/> </restriction> </simpleType> </element>
<element name=“State”> <simpleType> <restriction base=“string”> <tox-string type=“province”/> </restriction>
</simpleType> </element> </complexType> </element> </tox-list>
<!– format worker’s department list –> <tox-list name=“wDept_list” unique=“wDept/wDeptNo”> <element name=“wDept” minOccurs=“53” maxOccurs=“53”> <complexType>
<element name=“wDeptNo” type=“dNum_type”/> <element name=“wDeptLoc” type=“string”> <simpleType>
<restriction base=“string”> <tox-string type=“city”/>
</restriction> </simpleType> </element> </complexType> </element>
</tox-list>
<!– format worker list –> <tox-list name=“worker_list” unique=“worker/SSN”> <element name=“worker” minOccurs=“100” maxOccurs=“100”>
<complexType> <element name=“SSN” type=“eSSN_type”/> <element name=“hours”> <simpleType> <restriction base=“positiveInteger”>
<tox-number tox-distribution=“e1”/> <!– <minInclusive value=“10”/> <maxInclusive value=“40”/> –>
</restriction> </simpleType> </element> <element name=“wDept”> <complexType>
<element name=“wDeptNo”> <simpleType> <restriction base=“string”> <tox-sample path=“[wDept_list/wDept]”>
<tox-expr value=“[wDeptNo]”/>
</tox-sample> </restriction> </simpleType> </element> <element name=“wDeptLoc”>
<simpleType> <restriction base=“string”> <tox-sample path=“[wDept_list/wDept]”> <tox-expr value=“[wDeptLoc]”/> </tox-sample>
</restriction> </simpleType> </element> </complexType> </element>
</complexType> </element> </tox-list>
<!– format project list –>
<tox-list name=“project_list”> <element name=“project” minOccurs=“20” maxOccurs=“20”> <complexType>
<element name=“hours”> <simpleType>
<restriction base=“positiveInteger”> <tox-number tox-distribution=“e2”/> </restriction> </simpleType>
</element>
<element name=“pName” type=“projectName_type”/> <element name=“pNumber” type=“projectNo_type”/>
<element name=“pLocation”> <simpleType> <restriction base=“string”>
<tox-string type=“province”/> </restriction> </simpleType> </element> <element name=“pDnum” type=“dNum_type”/>
<element name=“worker” minOccurs=“20” maxOccurs=“20”> <complexType> <element name=“SSN”>
<simpleType> <restriction base=“string”>
<tox-sample path=“[worker_list/worker]”> <tox-expr value=“[SSN]”/> </tox-sample> </restriction>
</simpleType>
</element> <element name=“hours”> <simpleType> <restriction base=“positiveInteger”> <tox-sample path=“[worker_list/worker]”>
<tox-expr value=“[hours]”/> </tox-sample> </restriction> </simpleType> </element>
<element name=“wDept”> <complexType> <element name=“wDeptNo”> <simpleType> <restriction base=“string”>
<tox-sample path=“[worker_list/worker/wDept]”> <tox-expr value=“[wDeptNo]”/> </tox-sample> </restriction>
</simpleType>
</element> <element name=“wDeptLoc”> <simpleType> <restriction base=“string”> <tox-sample path=“[worker_list/worker/wDept]”>
<tox-expr value=“[wDeptLoc]”/> </tox-sample> </restriction> </simpleType> </element>
</complexType> </element> </complexType> </element> </complexType>
</element> </tox-list>

<!– format workOn list –> <tox-list name=“workOn_list”>
<element name=“workOn” minOccurs=“1” maxOccurs=“1”> <complexType> <element name=“project” minOccurs=“20” maxOccurs=“20”> <complexType> <element name=“hours”>
<simpleType> <restriction base=“positiveInteger”> <tox-sample path=“[project_list/project]”> <tox-expr value=“[hours]”/> </tox-sample>
</restriction> </simpleType> </element> <element name=“pName”> <simpleType>
<restriction base=“string”> <tox-sample path=“[project_list/project]”> <tox-expr value=“[pName]”/> </tox-sample> </restriction>
</simpleType> </element> <element name=“pNumber”> <simpleType> <restriction base=“string”>
<tox-sample path=“[project_list/project]”> <tox-expr value=“[pNumber]”/> </tox-sample> </restriction>
</simpleType>
</element> <element name=“pLocation”> <simpleType> <restriction base=“string”> <tox-sample path=“[project_list/project]”>
<tox-expr value=“[pLocation]”/> </tox-sample> </restriction> </simpleType> </element>
<element name=“pDnum”> <simpleType> <restriction base=“positiveInteger”> <tox-sample path=“[project_list/project]”> <tox-expr value=“[pDnum]”/>
</tox-sample> </restriction> </simpleType> </element> <element name=“worker” minOccurs=“20” maxOccurs=“20”>
<complexType> <element name=“SSN”> <simpleType> <restriction base=“string”> <tox-sample path=“[project_list/project/worker]”>
<tox-expr value=“[SSN]”/> </tox-sample> </restriction> </simpleType> </element>
<element name=“hours”> <simpleType> <restriction base=“string”> <tox-sample path=“[project_list/project/worker]”> <tox-expr value=“[hours]”/>
</tox-sample> </restriction> </simpleType> </element> <element name=“wDept”>
<complexType> <element name=“wDeptNo”> <simpleType> <restriction base=“string”> <tox-sample path=“[project_list/project/worker/wDept]”>
<tox-expr value=“[wDeptNo]”/> </tox-sample> </restriction> </simpleType> </element>
<element name=“wDeptLoc”> <simpleType> <restriction base=“string”> <tox-sample path=“[project_list/project/worker/wDept]”> <tox-expr value=“[wDeptLoc]”/>
</tox-sample> </restriction> </simpleType> </element> </complexType>
</element> </complexType> </element> </complexType> </element>
</complexType> </element> </tox-list>

<!– format employee list –> <tox-list name=“employee_list”> <element name=“employee” minOccurs=“100” maxOccurs=“100”> <complexType> <element name=“eName”>
<simpleType> <restriction base=“string”> <tox-sample path=“[emp_list/eName]”> <tox-expr value=“[fName]#’ ’#[lName]”/> </tox-sample>
</restriction> </simpleType> </element> <element name=“eSSN” type=“eSSN_type”/> <element name=“eSex” type=“sex_type”/>
<element name=“eSalary” type=“eSal_type”/> <element name=“eDoB” type=“birth_date”/> <element name=“address”> <simpleType> <restriction base=“string”>
<tox-sample path=“[address_list/address]”> <tox-expr value=“[No]#’ ’#[Street]#’ ’#[City]#’ ’#[State]”/> </tox-sample> </restriction> </simpleType>
</element> <element name=“workOn”> <complexType> <element name=“project” minOccurs=“20” maxOccurs=“20”> <complexType>
<element name=“hours”> <simpleType> <restriction base=“positiveInteger”> <tox-sample path=“[workOn_list/workOn/project]”> <tox-expr value=“[hours]”/>
</tox-sample> </restriction> </simpleType> </element> <element name=“pName”>
<simpleType> <restriction base=“string”> <tox-sample path=“[workOn_list/workOn/project]”> <tox-expr value=“[pName]”/> </tox-sample>
</restriction> </simpleType> </element> <element name=“pNumber”> <simpleType>
<restriction base=“string”> <tox-sample path=“[workOn_list/workOn/project]”> <tox-expr value=“[pNumber]”/> </tox-sample> </restriction>
</simpleType> </element> <element name=“pLocation”> <simpleType> <restriction base=“string”>
<tox-sample path=“[workOn_list/workOn/project]”> <tox-expr value=“[pLocation]”/> </tox-sample> </restriction> </simpleType>
</element> <element name=“pDnum”> <simpleType> <restriction base=“positiveInteger”> <tox-sample path=“[workOn_list/workOn/project]”>
<tox-expr value=“[pDnum]”/> </tox-sample> </restriction> </simpleType> </element>
<element name=“worker” minOccurs=“20” maxOccurs=“20”> <complexType> <element name=“SSN”> <simpleType> <restriction base=“string”>
<tox-sample path=“[workOn_list/workOn/project/worker]”> <tox-expr value=“[SSN]”/> </tox-sample> </restriction> </simpleType>

</element> <element name=“hours”> <simpleType> <restriction base=“string”> <tox-sample path=“[workOn_list/workOn/project/worker]”>
<tox-expr value=“[hours]”/> </tox-sample> </restriction> </simpleType> </element>
<element name=“wDept”> <complexType> <element name=“wDeptNo”> <simpleType> <restriction base=“string”>
<tox-sample path=“[workOn_list/workOn/project/worker/wDept]”> <tox-expr value=“[wDeptNo]”/> </tox-sample> </restriction> </simpleType>
</element> <element name=“wDeptLoc”> <simpleType> <restriction base=“string”> <tox-sample path=“[workOn_list/workOn/project/worker/wDept]”>
<tox-expr value=“[wDeptLoc]”/> </tox-sample> </restriction> </simpleType> </element>
</complexType> </element> </complexType> </element> </complexType>
</element> </complexType> </element> <element name=“dependent” minOccurs=“5” maxOccurs=“5”> <complexType>
<element name=“depName”> <simpleType> <restriction base=“string”> <tox-string type=“fname”/> </restriction>
</simpleType> </element> <element name=“dSex” type=“sex_type”/> <element name=“depDoB” type=“birth_date”/> <element name=“relationship” type=“relationship_type”/>
</complexType> </element> </complexType> </element> </tox-list>
<!– format department list –> <tox-list name=“dept_list”> <element name=“department” minOccurs=“5” maxOccurs=“5”> <complexType>
<element name=“dName” type=“dName_type”/> <element name=“dNumber” type=“dNum_type”/> <element name=“dMgrSSN” type=“eSSN_type”/> <element name=“dMgrStartDate” type=“start_date”/> <element name=“dLocation” minOccurs=“20” maxOccurs=“20”>
<simpleType> <restriction base=“string”> <tox-string type=“city”/> </restriction> </simpleType>
</element> <element name=“employee” minOccurs=“100” maxOccurs=“100”> <complexType> <element name=“eName”> <simpleType>
<restriction base=“string”> <tox-sample path=“[employee_list/employee]”> <tox-expr value=“[eName]”/> </tox-sample> </restriction>
</simpleType> </element> <element name=“eSSN”> <simpleType> <restriction base=“string”>
<tox-sample path=“[employee_list/employee]”> <tox-expr value=“[eSSN]”/> </tox-sample> </restriction>
</simpleType>
</element> <element name=“eSex”> <simpleType> <restriction base=“string”> <tox-sample path=“[employee_list/employee]”>
<tox-expr value=“[eSex]”/> </tox-sample> </restriction> </simpleType> </element>
<element name=“eSalary”> <simpleType> <restriction base=“positiveInteger”> <tox-sample path=“[employee_list/employee]”> <tox-expr value=“[eSalary]”/>
</tox-sample> </restriction> </simpleType> </element> <element name=“eDoB”>
<simpleType> <restriction base=“date”> <tox-sample path=“[employee_list/employee]”> <tox-expr value=“[eDoB]”/> </tox-sample>
</restriction> </simpleType> </element> <!– <element name=“eDno” type=“dNum_type”/> –> <!– <element name=“eSupervisorSSN” type=“eSSN_type”/> –>
<element name=“address”> <simpleType> <restriction base=“string”> <tox-sample path=“[employee_list/employee]”> <tox-expr value=“[address]”/>
</tox-sample> </restriction> </simpleType> </element> <element name=“workOn”>
<complexType> <element name=“project” minOccurs=“20” maxOccurs=“20”> <complexType> <element name=“hours”> <simpleType>
<restriction base=“positiveInteger”> <tox-sample path=“[employee_list/employee/workOn/project]”> <tox-expr value=“[hours]”/> </tox-sample> </restriction>
</simpleType> </element> <element name=“pName”> <simpleType> <restriction base=“string”>
<tox-sample path=“[employee_list/employee/workOn/project]”> <tox-expr value=“[pName]”/> </tox-sample> </restriction> </simpleType>
</element> <element name=“pNumber”> <simpleType> <restriction base=“string”> <tox-sample path=“[employee_list/employee/workOn/project]”>
<tox-expr value=“[pNumber]”/> </tox-sample> </restriction> </simpleType> </element>
<element name=“pLocation”> <simpleType> <restriction base=“string”> <tox-sample path=“[employee_list/employee/workOn/project]”> <tox-expr value=“[pLocation]”/>
</tox-sample> </restriction> </simpleType> </element> <element name=“pDnum”>
<simpleType> <restriction base=“positiveInteger”> <tox-sample path=“[employee_list/employee/workOn/project]”>
<tox-expr value=“[pDnum]”/> </tox-sample>
</restriction> </simpleType> </element> <element name=“worker” minOccurs=“20” maxOccurs=“20”> <complexType>
<element name=“SSN”> <simpleType> <restriction base=“string”> <tox-sample path=“[employee_list/employee/workOn/project/worker]”>

<tox-expr value="[SSN]"/> </tox-sample> </restriction> </simpleType> </element> <element name="hours">
<simpleType> <restriction base="string"> <tox-sample path="[employee_list/employee/workOn/project/worker]">
<tox-expr value="[hours]"/> </tox-sample>
</restriction> </simpleType> </element> <element name="wDept"> <complexType>
<element name="wDeptNo"> <simpleType> <restriction base="string"> <tox-sample path="[employee_list/employee/workOn/project/worker/wDept]"> <tox-expr value="[wDeptNo]"/> </tox-sample> </restriction> </simpleType> </element> <element name="wDeptLoc"> <simpleType> <restriction base="string"> <tox-sample path="[employee_list/employee/workOn/project/worker/wDept]"> <tox-expr value="[wDeptLoc]"/> </tox-sample>
</restriction> </simpleType> </element> </complexType> </element>
</complexType> </element> </complexType> </element> </complexType>
</element> <element name="dependent"> <complexType> <element name="depName"> <simpleType> <restriction base="string"> <tox-sample path="[employee_list/employee/dependent]"> <tox-expr value="[depName]"/> </tox-sample> </restriction> </simpleType> </element> <element name="dSex">
<simpleType> <restriction base="string">
<tox-sample path="[employee_list/employee/dependent]"> <tox-expr value="[dSex]"/> </tox-sample>
</restriction> </simpleType> </element> <element name="depDoB"> <simpleType> <restriction base="date">
<tox-sample path="[employee_list/employee/dependent]"> <tox-expr value="[depDoB]"/> </tox-sample>
</restriction> </simpleType> </element>
<element name="relationship"> <simpleType> <restriction base="string"> <tox-sample path="[employee_list/employee/dependent]"> <tox-expr value="[relationship]"/> </tox-sample> </restriction>
</simpleType> </element> </complexType> </element> </complexType> </element> </complexType>
</element>
</tox-list>
<!– This is the shallow company XML document; it has one "company" root entry. –> <tox-document name="output/dCompany"> <element name="company" minOccurs="1" maxOccurs="1"> <complexType>
<!– Start of "department" element(s)–> <element name="department" minOccurs="1" maxOccurs="1">
<complexType> <tox-scan path="[dept_list/department]" name="d"> <element name="dName"> <tox-expr value="[$d/dName]"/> </element> <element name="dNumber"> <tox-expr value="[$d/dNumber]" format="00"/> </element> <element name="dMgrSSN"> <tox-expr value="[$d/dMgrSSN]"/> </element>
<element name="dMgrStartDate"> <tox-expr value="[$d/dMgrStartDate]"/> </element> <element name="dLocation" minOccurs="3" maxOccurs="10"> <simpleType> <restriction base="string"> <tox-scan path="[$d/dLocation]"> <tox-expr value="[!]"/> </tox-scan> </restriction> </simpleType>
</element> <!– Start of "employee" elements–> <element name="employee" minOccurs="40" maxOccurs="40">
<complexType> <tox-scan path="[$d/employee]" name="e"> <element name="eName"> <tox-expr value="[$e/eName]"/> </element> <element name="eSSN"> <tox-expr value="[$e/eSSN]"/>
</element> <element name="eSex"> <tox-expr value="[$e/eSex]"/> </element> <element name="eSalary">
<tox-expr value="[$e/eSalary]*1000" format="00,000"/> </element> <element name="eDoB"> <tox-expr value="[$e/eDoB]"/> </element>
<element name="address"> <tox-expr value="[$e/address]"/> </element>
<!– Start of "workOn/project" elements–>
<element name="workOn"> <complexType> <element name="project" minOccurs="10" maxOccurs="20">
<complexType> <tox-scan path="[$e/workOn/project]" name="p">
<element name="hours"> <tox-expr value="[$p/hours]"/> </element> <element name="pName"> <tox-expr value="[$p/pName]"/>
</element> <element name="pNumber"> <tox-expr value="[$p/pNumber]"/> </element> <element name="pLocation">
<tox-expr value="[$p/pLocation]"/> </element> <element name="pDnum"> <tox-expr value="[$p/pDnum]"/>
</element>
<!– Start of "worker" elements–> <element name="worker" minOccurs="10" maxOccurs="20"> <complexType>
<tox-scan path="[$p/worker]" name="w">
<element name="SSN"> <tox-expr value="[$w/SSN]"/> </element> <element name="hours"> <tox-expr value="[$w/hours]"/> </element> <!– Start of "wDept" elements–> <element name="wDept"> <complexType>
<tox-scan path="[$w/wDept]" name="wd"> <element name="wDeptNo"> <tox-expr value="[$wd/wDeptNo]"/>
</element> <element name="wDeptLoc">
<tox-expr value="[$wd/wDeptLoc]"/> </element> </tox-scan> </complexType> </element> <!– End of "wDept" elements–>
</tox-scan> </complexType> </element> <!– End of "worker" elements–> </tox-scan> </complexType>
</element> </complexType> </element> <!– End of "workOn/project" elements–>
</tox-scan> </complexType> </element> <!– End of "employee" elements–> </tox-scan> </complexType>
</element> </complexType> </element>
</tox-document> </tox-template>

REFERENCES

[1] Denilson Barbosa. (2003) ToXgene Template Specification Language. [Online]. Available: http://www.cs.toronto.edu/tox/toxgene/

[2] P. J. Harding, Q. Li, and B. Moon, "XISS/R: XML Indexing and Storage System using RDBMS." in *VLDB*, 2003, pp. 1073–1076.

[3] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions." in *VLDB*, 2001, pp. 361–370.

[4] P. F. Dietz, "Maintaining Order in a Linked List." in *STOC*, 1982, pp. 122–127.

[5] Wikibooks. (2006) Comparing XQuery and XPath. [Online]. Available: http://en.wikibooks.org/wiki/

[6] Sun Mi Shin and Hoe Jin Jeong and Sang Ho Lee. (2004) Design of an Integrated XML Data Generator for the Performance Evaluation of XML DBMSs. [Online]. Available: http://dblab.ssu.ac.kr/Publication/04-kips-ssm.pdf

[7] W3C. (1986) Standard Generalized Markup Language. [Online]. Available: http://www.w3.org/MarkUp/SGML/

[8] Leonidas Fegaras. (2005) Web Databases and XML. [Online]. Available: http://lambda.uta.edu/

[9] J. Clark and S. DeRose, "XML Path Language (XPath) version 1.0," World Wide Web Consortium, Techinical Report REC-xpath-19991116, November 1999.

[10] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler, "Extensible Markup Language (XML) 1.0, Second Edition," W3C Working Draft, October 2000.

[11] Q. Li, R. Elmasri, S. Prabhakar, N. Manandhar, and D. Y. Kim, "A Survey of XML Indexing Techniques," 2005.

[12] D. D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu, "XQuery: A Query Language for XML," World Wide Web Consortium, Techinical Report DCIS-TR-527, February 2001.

[13] Ellipsis, "DOMSafeXML," Online, June 2004, Commercial.

[14] S. AG, "Tamino," Online, November 2002, Commercial.

[15] SoftwareAG, "X-Hive/DB," Online, May 2005, Commercial.

[16] SleepycatSoftware, "Berkeley DB XML," Online, August 2003, Open Source.

[17] H. V. Jagadish, J. M. Patel, S. Al-Khalifa, and A. Chapman, "Timber," Online, October 2005, Open Source (for non-commercial users).

[18] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons, "ToXgene: A Template-based Data Generator for XML," in *SIGMOD Conference*, 2002, p. 616.

[19] T. Grust, "Accelerating XPath location steps." in *SIGMOD Conference*, 2002, pp. 109–120.

[20] D. Kim, "Multi-Dimensional Indexing for XML Data," Master, The University of Texas at Arlington, December 2005.

[21] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases." in *VLDB*, 1997, pp. 436–445.

[22] T. Milo and D. Suciu, "Index Structures for Path Expressions." in *ICDT*, 1999, pp. 277–295.

[23] R. Goldman and J. Widom, "Approximate DataGuides," 1999. [Online]. Available: http://citeseer.ist.psu.edu/goldman99approximate.html

[24] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting Local Similarity for Indexing Paths in Graph-Structured Data." in *ICDE*, 2002, pp. 129–140.

[25] C. Qun, A. Lim, and K. W. Ong, "D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data." in *SIGMOD Conference*, 2003, pp. 134–144.

[26] H. He and J. Yang, "Multiresolution Indexing of XML for Frequent Queries," in *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, p. 683.

[27] T. Grust, M. van Keulen, and J. Teubner, "Accelerating XPath evaluation in any RDBMS." *ACM Trans. Database Syst.*, vol. 29, pp. 91–131, 2004.

[28] H. Wang, S. Park, W. Fan, and P. S. Yu, "ViST: A Dynamic Index Method for Querying XML Data by Tree Structures." in *SIGMOD Conference*, 2003, pp. 110–121.

[29] P. Rao and B. Moon, "PRIX: Indexing and Querying XML Using Prüfer Sequences." in *ICDE*, 2004, pp. 288–300.

[30] H. Wang and X. Meng, "On the Sequencing of Tree Structures for XML Indexing." in *ICDE*, 2005, pp. 372–383.

[31] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked keyword search over XML documents," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2003, pp. 16–27.

[32] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword Proximity Search on XML Graphs." in *ICDE*, 2003, pp. 367–378.

[33] D. Florescu, D. Kossmann, and I. Manolescu, "Integrating keyword search into XML query processing." *Computer Networks*, vol. 33, no. 1-6, pp. 119–135, 2000.

[34] R. Bourret, "XML Database Products," XMLDatabaseProds," Techinical Report, 2001.

[35] D. Obasanjo, "An Exploration of XML In Database Management Systems," http://www.25hoursaday.com/StoringAndQueryingXML.html," Techinical Report, 2001.

[36] J. Clark, "XSL Transformations (XSLT) version 1.0," W3C, Tech. Rep. REC-xml-19980210, 1998, http://www.w3.org/TR/xslt. [Online]. Available: citeseer.nj.nec.com/bray98extensible.html

[37] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "XQuery 1.0: An XML Query Language," W3C Working Draft, April 2002.

[38] Object Data Management Group. (1998) ODMG OQL Users Manual. [Online]. Available: http://www.cis.upenn.edu/ cis550/oql.pdf/

[39] A. Aboulnaga, J. F. Naughton, and C. Zhang, "Generating Synthetic Complex-Structured XML Data." in *WebDB*, 2001, pp. 79–84.

[40] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management." in *VLDB*, 2002, pp. 974–985.

[41] K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa, "The Michigan benchmark: towards XML query performance diagnostics." *Inf. Syst.*, vol. 31, no. 2, pp. 73–97, 2006.

[42] Kanda Runapongsa and Jignesh M. Patel and H.V. Jagadish and Yun Chen and Shurug Al-Khalifa. (2002) The Michigan Benchmark. [Online]. Available: http://www.eecs.umich.edu/db/mbench/

[43] David Fallside. (2004) XML Schema Part 0: Primer Second Edition. [Online]. Available: http://www.w3.org/TR/xmlschema-0/

[44] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman, "On Supporting Containment Queries in Relational Database Management Systems." in *SIGMOD Conference*, 2001, pp. 425–436.

[45] D. Florescu and D. Kossmann, "Storing and Querying XML Data using an RDMBS." *IEEE Data Eng. Bull.*, vol. 22, no. 3, pp. 27–34, 1999.

[46] xmlpull.org. (2005) XML Pull Parsing. [Online]. Available: http://www.xmlpull.org/

[47] G. Xing and B. Tseng, "Extendible Range-Based Numbering Scheme for XML Document." in *ITCC (2)*, 2004, pp. 140–141.

[48] E. Cohen, H. Kaplan, and T. Milo, "Labeling Dynamic XML Trees." in *PODS*, 2002, pp. 271–281.

[49] Philip J. Harding and Quanzhong Li and Bongki Moon. (2003) XML Indexing and Storage System with RDBMS (XISS/R). [Online]. Available: http://xiss.cs.arizona.edu/

BIOGRAPHICAL STATEMENT

Chul Ho Ahn was born in Jeollabuk-do, Korea in 1974. He received his B.E. in the Faculty of Electrical, Electronic & Control Engineering at Chung-Ang University, Seoul in 2001. He worked as an Infrastructure Architect-System Engineer for LG CNS from 2001 to 2003. He began his study pursuing master's degree in the department of Computer Science and Engineering at the University of Texas at Arlington in 2004. His research interests include Web Databases and XML Indexing techniques. He earned his M.S. in Computer Science and Engineering in May 2006.