STRUCTURE BASED XML INDEXING

by

NIROJ MANANDHAR

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 AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2005

ACKNOWLEDGEMENTS

I would like to thank my supervising professor, Dr. Ramez Elmasri, for his guidance and support, and for giving me an opportunity to work on this thesis. I appreciate his insights and suggestions, which led to the successful completion of this work. I would also like to thank Dr. John Patterson and Dr. Leonidas Fegaras for their time and valuable suggestions.

I owe my sincere thanks to Sunit Shrestha for his constant support and encouragement throughout this thesis. I would like to thank friends who are working on XML project and friends at ITLab.

I am thankful to my parents and my grandmother for their support and encouragement throughout my academic career.

July 21, 2005

ABSTRACT

STRUCTURE BASED XML INDEXING

Publication No. _____

Niroj Manandhar, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: Ramez Elmasri

The increase in the usage and popularity of semi-structured data has received considerable attention, and a lot of research is going on for the efficient retrieval and storage of semi-structured data. A popular model and language for semi-structured data is XML. In this thesis we focus on the structure based indexing of XML. As a part of an ongoing XML indexing project, we study and implement A(k)-index, which is a structure based indexing technique; and propose the use of offset, length pair to retrieve nodes of interest. We record offset and length of every node using the SAX parser, and then we use Random Access File to retrieve nodes from a XML file using the A(k)-index. It can accurately support all path expressions of length up to k, and retrieve the

result directly from the XML file. We also compare the performance of the indexing technique when different k values are used.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In recent years, Extensible Markup Language (XML) has gained more and more support as a language for data exchange in the web. It is rapidly emerging as the de facto standard of exchanging and querying documents over World Wide Web [2]. Most of the data used in the web are semi-structured, and it does not follow any predefined schema. XML data is semi-structured in nature, and are often modeled by tree hierarchies.

With the increasing popularity of XML the need of efficient querying of XML data has received considerable attention. Several XML query languages [1, 3, 4, 10, 12, 21, 28] have been proposed. Among the proposed query languages, XQuery [3], XPath [28], Lorel [21] and Quilt [4] use path expression to traverse a XML data graph, and to retrieve the matching data nodes. The data nodes are extracted if some path to that node has a sequence of labels matched by the path expression, thus navigation of data graph is the most important part of processing XML queries. The similar looking nodes of data graph may be scattered at different locations in an XML file, so querying and retrieving interesting nodes may take extra time. To improve the performance of query execution, indexing can be used.

Indexes in general speed up the performance of querying and improve the efficiency of an XML database. An index on XML data helps retrieve any information

based on the tagged content. Traditional relational databases force all data to adhere to an explicitly specified schema but XML does not follow a specified schema, so additional flexibility is needed to query XML databases.

A lot of research is going on for the efficient indexing of XML data and several techniques have been proposed for it. Li et al. in [15] has surveyed XML indexing techniques, and in [22], Shalli Prabhakar has categorized those indexing techniques into sequence based indexes, structural indexes, dimension based indexes, and keyword based indexes. In sequence based indexing, the XML data is transferred into sequences using either Prufers sequence [11], depth-first traversal or breadth first traversal. Then a one to one correspondence is created between the XML tree and the sequence [22]. In dimension based indexing, the pre-order or the post-order rank of a particular node on the XML tree is mapped to either a linear interval [14] or to coordinates in two-dimension plane [23]. The keyword based indexing is based on proximity search. In this technique the focus is on keyword search in which users do not have to learn any schema or query language. In this approach the distance between keywords is used to find the most relevant result. The authors in [5, 9, 26] have suggested this approach. Finally in structural summary a summary graph is constructed based on similarity of nodes. Structural summary of graph has been proposed by authors in [6, 13, 17, 19, 24].

A simple evaluation that scans the whole database is very expensive and it is not practical. As in the case of relational database, we need to use some indexes to speed up the evaluation of XML queries. Keeping all these in mind we focus on structure based indexing in this thesis.

The structural summaries of graph structured data are based on the notion of bisimilarity [13]. Two nodes are bisimilar if the path labels into them are identical. Path expressions can be directly evaluated in the summary graph and they can retrieve label matching nodes without referring to the original data graph. A(k)-index [19] is our choice among the structure based indexing techniques. It is implemented and evaluated with various values of k. A(k)-index helps to find a particular node information in XML node quickly. Information on nodes and summary structure can be stored in the database [17], or they can be stored in main memory. The advantage of storing in the database is that it enables to store summary structure for large XML files. However, the use of database comes with the added overhead of database engine and database connection, which might reduce the performance of queries. In that respect, main memory storage seems to be attractive but it may be limited by the size of the main memory to process larger XML files. We propose a new technique of storing summary information of nodes in main memory. Each node in the XML file can be represented as a pair consisting of [offset, length]. "Offset" is the position of the starting node and "Length" is the length of the string that begins from "<" of the start node tag and ends after ">" of the end node tag. This pair of information enables to access any interesting nodes in the XML file through Random Access File API of Java.

The contributions of this thesis are:

- Implementation of structure based indexing, A(k)-index in main memory
- Proposal to use offset, length pair to store node information
- Design and implementation of query processing algorithm.

The remainder of this paper is organized as follows. In Chapter 2, we give an overview of XML, XPath and XQuery. In chapter 3, we discuss structure based indexing for XML. We proceed to describe the data structures and implementation of A(k)-index using offset, length pair, and query processing in chapter 4. In chapter 5, we discuss experimental results. We discuss related work in chapter 6 and conclude in chapter 7.

CHAPTER 2

XML OVERVIEW

XML data is a semi-structured data that does not conform to traditional data models like relational data models. Several languages have been proposed to process XML data, such as XPath and XQuery. In this section we will give an overview of XML, XPath and XQuery. We will also discuss XML data graph. This chapter is based on material from [13, 15, 27, 28].

2.1 Overview of XML

XML is a Markup Language for documents containing structured information developed by W3C (World Wide Web Consortium). XML looks like HTML but it is different in the sense that it is designed to describe data, not to display information. XML documents can be either data-centric or document-centric Data-centric XML documents have regular structure, so it is easier to process. Document-centric XML documents have less regular or irregular structure. An XML document should be well formed, i.e. it conforms to the XML syntax. A well-formed XML document has the following properties:

- XML document can have only one root element.
- XML element must be closed after its entire children element is closed.
- Any entities that are referenced must be well formed.

XML consists of hierarchically nested elements as shown in Figure 1.2, and it can be modeled as a labeled rooted graph as shown in Figure 2.2. The nodes of the graph represent elements, attributes and simple values, and the edges of the graph represent element-sub element, element-attribute, or element-value relationships between nodes. Each node in the graph has a label and a unique id. There is a single root element with the distinguished label, e.g. 'uta' in Figure 2.2. The structure in the Figure 2.2 is actually a tree, with edges representing element-sub element or element-value relationships between nodes.

```
<?xml version="1.0" ?>
<uta>
  <student>
    <undergrad>
      <name>Jacob</name>
    </undergrad>
    <dept>CSE</dept>
  </student>
  <student>
    <grad>
      <name>Emily</name>
    </grad>
    <dept>Math</dept>
  </student>
  <student>
    <grad>
      <name>Andrew</name>
    </grad>
    <dept>Math</dept>
  </student>
</uta>
```

Figure 2.1 Sample XML Document

Let us take an example data graph G given in Figure 2.2. In the figure nodes are represented by v_0 v_1 …… v_n and edges are labeled l_0 l_1…… l_n. Paths are represented by path p = $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$.

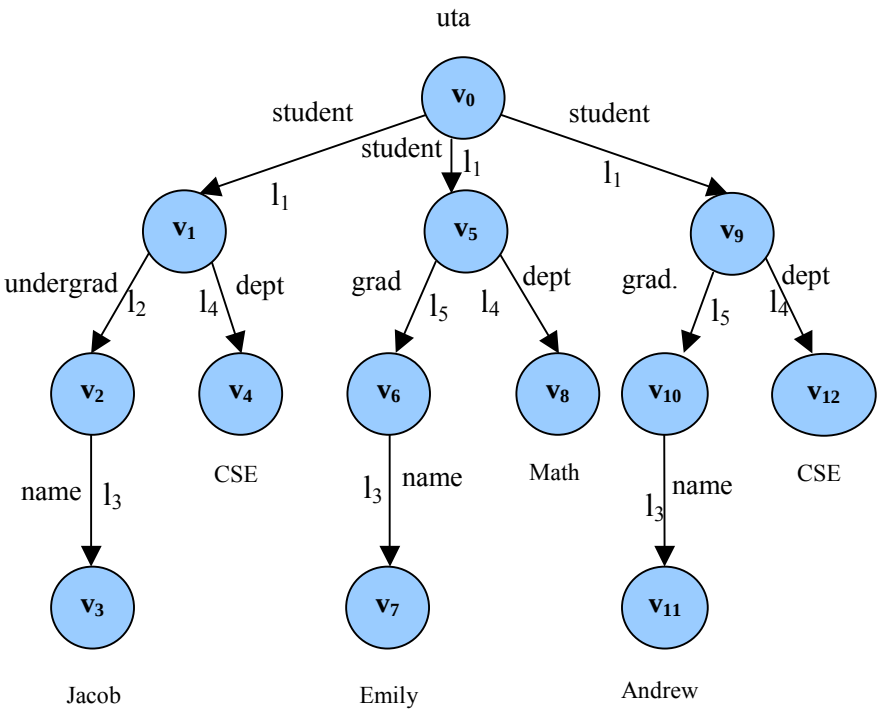

Figure 2.2 Sample XML Database

A node path in the data graph G is a sequence of nodes v_0 v_1 …… v_n, such that there exists an edge between nodes v_i and v_{i+1}, for $0 \leq i \leq n-1$. A label path is a sequence of labels l_1 l_2 …… l_n. A node path matches a label path if label $(v_i) = l_i$, for $0 \leq i \leq n$. A label path l_1 l_2 …… l_n matches a node v if there is some node path ending in node v that matches l_1 l_2 …… l_n. A regular path expression, R, is defined in the usual way in terms of sequence (.), alternation (|), repetition (*) and optional expression (?), as follows[13]:

R = ΣG|_|R.R|R|R|(R)|R?|R*

2.2 Overview of XPath

XPath is a well accepted language used for selecting nodes from XML documents. It uses path expressions like a URL address to navigate an XML document. It follows the matching nodes and may travel to multiple directions. It indicates nodes by position, relative position, type, content and several other criteria.

1. /uta/student/grad/name → gives name of all grad who are students of uta.
2. /student/*/name → give name of all students whether grad or undergrad.
3. //name → gives all names from data graph.

Figure 2.3 Sample XPath

Some examples of XPath expressions are shown in Figure 2.3. XPath specifies a tree traversal via two parameters: context node and sequence of location steps. Context node is a starting point of traversal, which need not be the root of the document. A sequence of location steps is set of nodes relative to the context node. It is separated by backslash '/' and evaluated from left to right. For example given the expression '/student/grad/name', the first segment expression 'student' is evaluated to locate the node-set 'Node Set 1' consisting of all student nodes; next, the segment expression 'grad' is applied to 'Node Set 1' to locate the node-set 'Node Set 2' consisting of all grad children of student nodes; and finally the segment expression 'name' is applied to 'Node Set 2' to locate all name children of grad nodes.

XPath can also be used as a query language for XML. XPath query consists of location path and output expression, e.g. '/student [gpa > 3.5]/name'. A location path is a path from the context node to a desired node, and it may be relative or absolute. A relative location path is a sequence of one or more locations steps separated by backslash '/'. Each location step selects a set of nodes relative to a context node. The selected set may be empty or it may contain multiple nodes, which are evaluated from left to right. The location step has two parts: an axes and a node test. The axes returns node sets reachable from the context node and defines a direction to travel. The node test identifies a node within an axis.

The result of evaluating an XPath expression on an XML data graph is the set of nodes from the data graph that matches the XPath expression. For example, the path expression, 'student/grad/name' in Figure 2.2 returns v_7 and v_{11}; the more complicated path expression, 'student.*.name' returns v_3, v_7 and v_{11}, i.e. the name of all students regardless of grad or undergrad. XPath operates on an XML document as a tree. There are seven XPath node types [28]. They are root node, element node, attribute node, namespace nodes, processing instruction nodes, text nodes and comment nodes.

2.3 Overview of XQuery

XML queries are queries over data that conforms to a labeled data graph or labeled tree. As most of the query languages an XML query forms a pattern language. An XML query retrieves sub-structures of a data graph that match the query structure.

XQuery is a query language for XML released by the World Wide Web Consortium. It is designed to be broadly applicable across many types of XML data

sources [27]. It is closely related to XPath, so it is also seen as the superset of XPath. It supports the same data models, operators and functions as XPath. XQuery, like XPath also contain values and wildcards '*' and '//'. Unlike XPath, XQuery does not support some of the less common axes: ancestor, ancestor-or-self, following-sibling, preceding, preceding-sibling and namespace.

XQuery is an expression based query language, which reads a sequence of atomic values and returns a sequence. There are two kinds of expressions, single path expressions and branching path expressions. A single path expression defines queries on one element, e.g. it evaluates to 'find all the name of students'. A branching path expression defines queries on two or more elements, e.g. it evaluates to 'find all the name of students who have taken the database course'. XQuery is case sensitive; and keywords in XQuery are not reserved and are lower case characters.

1. document("uta.xml")//grad/name → gives name of grad only.
2. document("uta.xml")//student/*/name → gives name of all students whether grad or undergrad.

Figure 2.4 Sample XQuery

There are 7 types of expressions in XQuery: path expression, element constructors, FLWOR expressions, expressions involving operators and functions, conditional expressions, quantified expressions and expressions that test or modify data types [27]. The FLWOR expression is similar to the select-from-where construction in SQL and forms the skeleton of the XQuery expression. A FLWOR expression consists

of: For (binds variables to a value and iterates over values), Let (binds variable without iterating), Where (contains one or more predicates that filters), Order by (imposes an order) and Return (generates the result of the FLWOR expression). Some examples of path expressions are shown in Figure 2.4. In addition to the built-in functions, XQuery also supports user-defined functions whose definitions appear in the query prolog. The function parameters and results could be sequences, nodes or primitive values [22].

CHAPTER 3

STRUCTURE BASED INDEXING FOR XML

We mentioned in Chapter 2 that an XML query extracts sub-structures of the data graph that match the query structure. The sub-structures or data nodes that we want to retrieve may be scattered over the data graph causing slower extraction. A simple grouping of data nodes can give a good index graph. Structural summary is one of the techniques that groups nodes by label and construct efficient index. Structural summaries can speed up query evaluation on XML data reducing the search space. In this section we will explain structure based XML indexing. Then we will discuss construction of a type of structural index called A(k)-index in detail. We will also look at the definition of bisimulation and k-bisimilarity. This chapter is based on materials from [2, 13, 17, 18, 19, 24].

3.1 Structure Based Indexing for XML

In Chapter 2 we explained that an XML query use path expressions to traverse an XML data graph; and nodes that have a sequence of labels and conditions that matches the path expressions are selected. So traversal of sequences of labels on the data graph is essential in query processing. A structural summary can be constructed to speed up the traversal and evaluation process. Structural summary prunes the search space and preserves paths and properties. All data nodes in the data graph that match a

particular path expression will be grouped under a single node on the index graph. All data nodes belonging to a grouped single node become the extent of the index node.

Structural summaries speed up query evaluation on XML data by restricting the search to only relevant portion of the XML data [2]. A structural summary for the data is a labeled summary graph with fewer numbers of edges and nodes. All data nodes in the data graph that match a particular path expression are grouped together and represented as a single node on the index graph. This allows evaluation of path expression on the summary graph instead of the original data graph**.** The summary graph can be achieved with a refinement of the data graph. A partition P_1 of the data nodes is a refinement of another partition P_2 if the following condition holds: whenever two nodes are in the same equivalence class P_1, they are in the same equivalence class P_2 as well. If P_1 is a refinement of P_2, then P_2 is coarser than P1 [18]. A structural summary makes query processing efficient by associating an extent with each node in the summary to produce an index graph. Figure 3.2 and Figure 3.3 shows examples of structural summary for data graph of Figure 3.1.

According to [19], let us assume that there is a data graph G that has a structural index graph denoted by I_G. The result of executing a path expression, P on I_G is the union of the extents of the index nodes in I_G that matches P. The mapping from the data nodes to index nodes is required to be safe: if l_1 l_2 …… l_k is a label path that matches a path to node v in G, then there is some node A in index graph I_G for which v ε extent (A). Extent is a set of data nodes in the data graph, with a single node in the summary

graph. This guarantees that the evaluation result of any path expression, P, on G is contained in the result of evaluating P on the index graph, I_G.

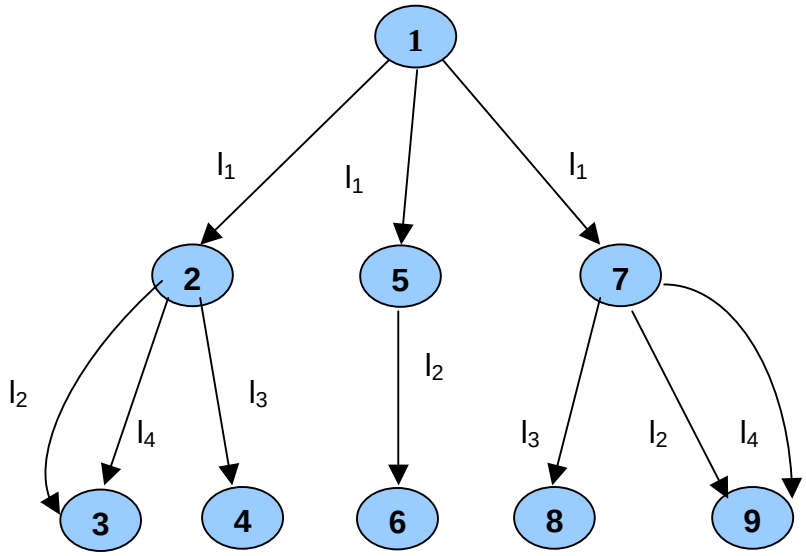

Figure 3.1 Data Graph

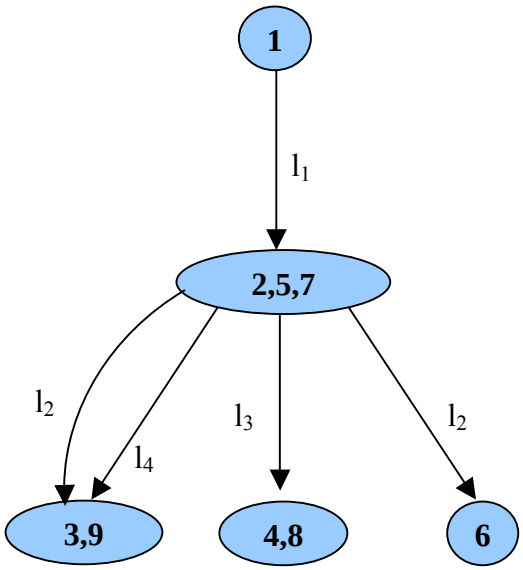

Figure 3.2 1- index

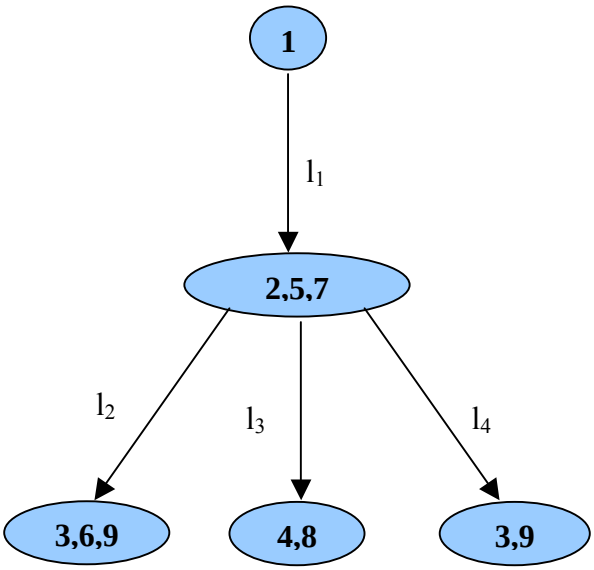

Figure 3.3 Data Guide

Similarly, an index graph that is precise for simple path expressions has the property that if v ε extent(A) and l_1 l_2 …… l_k is a valid label path for A in an index graph I_G, then l_1 l_2 …… l_k is a valid label path for every node in extent(A) in G.

Index graph I_G can be obtained by associating an index node with each equivalence class in the data graph, G. Then each index node's extent is defined to be the equivalence class that formed it. Finally if there is an edge from some data node in extent (A) to some data node in extent (B), an edge is added from index node A to index node B in I_G.

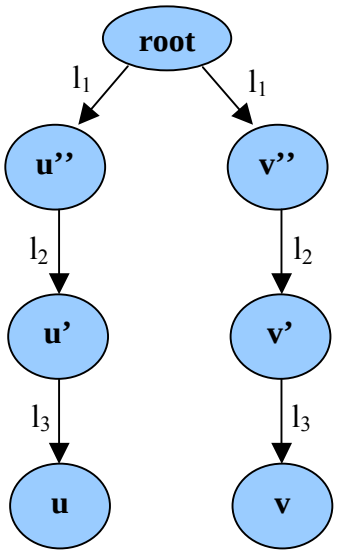

Figure 3.4 Bisimulation

3.2 Overview of Bisimulation

Bisimulation technique is used to compress the state space graph in a manner that preserves some properties and behaviors of the state space [13]. Existing index structures for semi-structured data or XML are based on the notion of bisimulation.

Definition of bisimilarity: A symmetric, binary relation ≈ on V_G is called a bisimulation if, for any two data nodes u and v with u ≈ v, we have that

a. u and v have the same label, and
b. if u' is a parent of u, then there is a parent v' of v such that u' ≈ v', and vice versa.

Two nodes u and v in data graph G are said to be bisimilar, denoted by $u \approx^b v$, if there is some bisimulation ≈ such that u ≈ v.

For example, in Figure 2.2, node v_7 and v_{11} (the name nodes) are bisimilar. Nodes v_3 and v_7 (the name nodes) are not bisimilar, since there parents nodes v_2 and v_6 are not bisimilar. In simple terms we can say that two nodes are bisimilar if the set of incoming paths to the nodes are the same. Bisimulation can be computed in time O(m log n) where n is the number of nodes and m is the number of edges in the data graph, using an algorithm proposed by Paige and Tarjan [20].

3.3 A(k)-index

Structural indexing can be constructed as an accurate structural summary or approximate structural summary of the semi-structured databases. A structural summary is accurate if it is the true reflection of the structure of the original data graph. Some examples of accurate structural summaries are strong Data Guides [17] and 1-index [24].

A structural summary is approximate if the nodes of an XML tree are grouped according to the local structure, i.e., the incoming path of length of a certain parameter. Local similarity required for a node is obtained by checking the maximum size of a

query. In approximate structural summary only paths shorter than given parameter are of significance. Some examples of approximate structural summaries are Approximate Data Guides [17], A(k)-index [19], D(k)-index [13], and M(k)-index [6].

Among all the approximate structural indexing we focus on A(k)-index. In [19], A(k)-index was proposed based on the observation that long and complex paths tend to contribute disproportionately to the complexity of an accurate structural summary. The main idea of A(k)-index is to group nodes based on their local similarity instead of the global path information. It groups nodes according to a local path up to path length k. This reduces the storage overhead, and allows faster extraction of interesting nodes. A(k)-index technique groups database objects into equivalence classes containing objects that are indistinguishable with respect to a class of paths defined by a path [19]. Bisimulation concept discussed above is used to find the equivalence classes. It constructs non deterministic automata whose states represent the equivalence classes and whose transitions correspond to edges between objects in those classes.

It assumes that most queries involve short path expressions, so it relaxes the equivalence condition and considers only incoming paths whose lengths are no longer than k. It groups paths of length up to k, based on a property called k-bisimilarity, which is defined below. A(k)-index can accurately support all path expressions of length up to k, but path expressions longer than k must be validated in the data graph. Since A(k)-index only considers local similarity it is significantly smaller than a fully accurate structure and it is significantly faster for shorter path expressions. The local similarity can be varied by changing the value of parameter k.

k-bisimilarity ($\approx^k$) is defined inductively:

1. For any two nodes, u and v, $u \approx^0 v$ iff u and v have the same label;
2. Node $u \approx^k v$ iff $u \approx^{k-1} v$ and for every parent u' of u, there is a parent v' of v such that $u' \approx^{k-1} v'$, and vice versa.

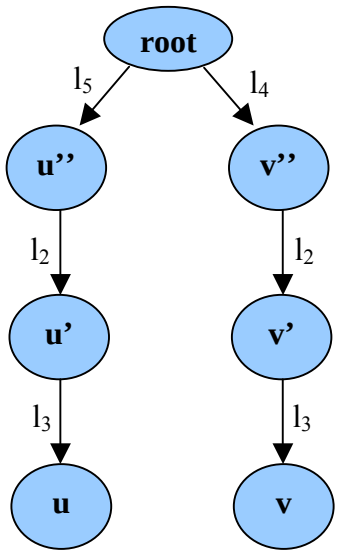

Figure 3.5 k-bisimilarity

In Figure 3.5 k-bisimilarity is shown. Node u'' and v'' are $\approx^0$ similar since they have same label. Node u' and v' are $\approx^1$ similar since parent of u' and v' are $\approx^0$ similar. Similarly node u and v are $\approx^2$ similar since parent of u and v are $\approx^1$ similar. In other word we can say that paths of length up to 2 are similar for node u and v. The k-bisimilarity shown in Figure 3.5 is different than the bisimilarity shown in Figure 3.2. In Figure 3.2 all the paths up to u is same as all the path up to v, but in Figure 3.5 the first location path from root to u" and root to v" is different, so the similarity in Figure 3.5 is considered local. It must be kept in mind that the evaluation result of the A(k)-index is

accurate if the length of a path expression is less than or equal to k. The A(k)-index can be constructed in O(km) time, where m is the number of edges in the data graph.

Given a data graph and a refinement ≈, the A(k)-index is a rooted, labeled graph defined as follows. Its nodes are equivalence classes [v] of ≈; for each v $\rightarrow^{a}$ v' in data graph there exists and edge [v] $\rightarrow^{a}$ [v'] in index graph. The root r in data graph would be root [r] in index graph.

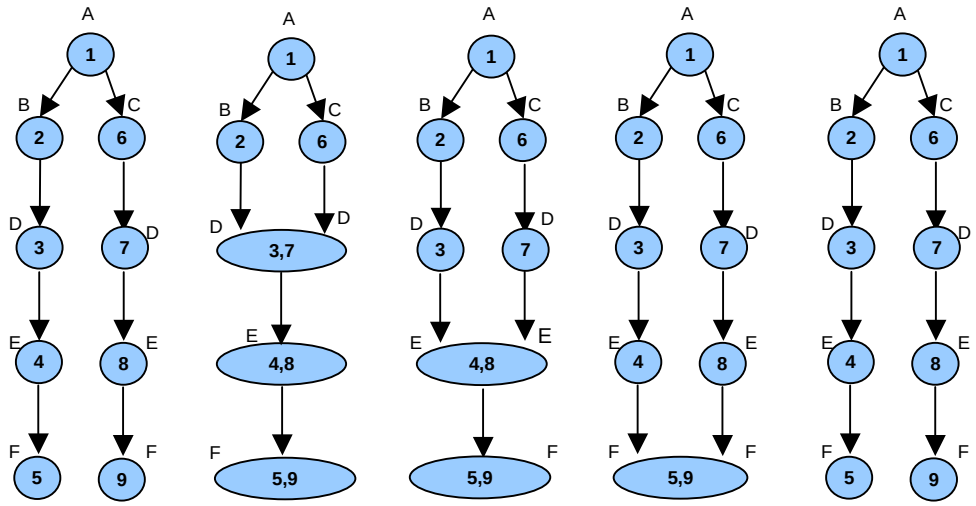

Figure 3.6 A(k)-index
(a)A(1)-index, (b)A(2)-index, (c)A(3)-index, (d)A(4)-index,
(e)A(5)/A(max)-index

The A(k)-index has following properties

1. If nodes u and v are k-bisimilar, then the set of label paths of length ≤ k into them is the same.
2. The set of label-paths of length m(m ≤ k) into an A(k)-index node is the set of label paths of length m into any data node in its extent.

3. The A(k)-index is safe i.e. , its results on a path expression always contain the data graph results for that query.
4. The A(k)- index is sound for any path expression of length less than or equal to k.

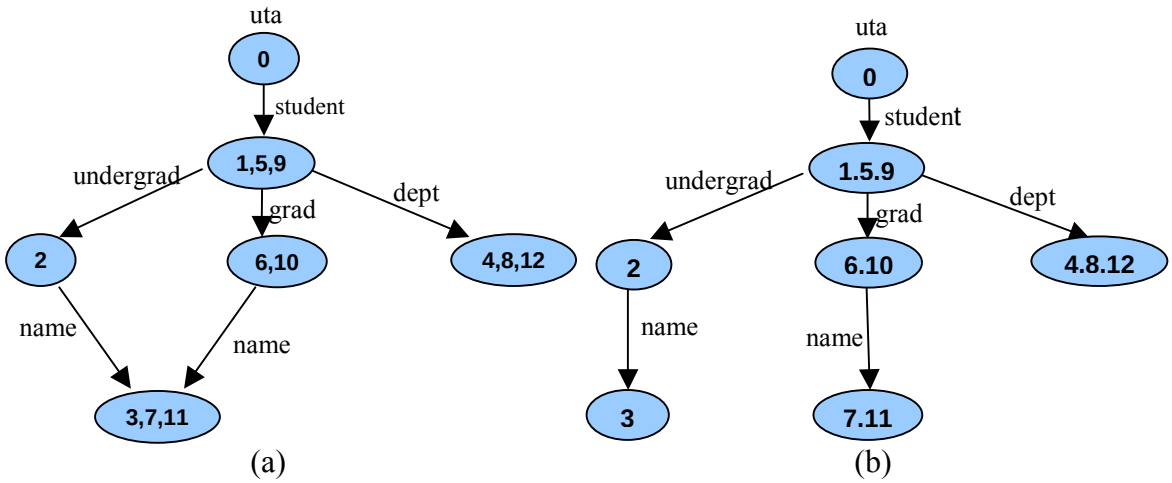

Figure 3.7 A(k)-index for Figure 2.1
(a)A(1)-index, (b)A(2)-index

Finding objects reachable by a given labeled path through the database is important part of query processing [8]. To evaluate a regular path expression, automaton evaluation is used and A(k)-index result set is obtained from a data graph. Then nodes in the result set are validated against the original data graph to extract the accurate result set for the path expression. The length of a path, i.e. parameter k determines the performance and the size of the A(k)-index. If the value of k is small, the size of index will be small, but k does not cover longer paths so more validation is required. If the value of k is large the size of index will be large too, but it will require fewer validation. For example in the Figure 3.7 when k is 1 all names are in same

equivalence class. The index graph is small but the query result of '/grad/name' will also contain names of undergrads, so validation of the result is required. In the same data if k is 2 then index graph will be as shown in Figure 3.8. Index shown in Figure 3.7(b) is larger than index shown in Figure 3.7(a). For query '/grad/name', the result set will be correct, it does not require validation. As k increases in A(k)-indexing, the partition induced by equivalence relation keeps getting refined and at some value of k it would be maximal bisimilarity as shown in Figure 3.6. In Figure it is clearly seen that every time k is increased the index graph is getting large requiring less and less validation. So there is a trade off between index size and validation required.

CHAPTER 4

DESIGN AND IMPLEMENTATION OF A(K)-INDEX

Various structural indexing techniques have proposed different ways to store and retrieve summary graphs. In this thesis we implement A(k)-index in main memory, and nodes are retrieved using a [offset, length] pair. "Offset" is the position of the starting node and "Length" is the length of the string that begins from "<" of the start node tag and ends after ">" of the end node tag. This pair of information enables to access any nodes of interest in the XML file through the Random Access File API of Java. This is very effective when the data graph is static. This concept is based on the location of nodes in original data graph. We record the starting position of a node and its length in the XML file. Then when we query the XML file we find the result set and retrieve individual results using the offset and length stored for nodes that satisfy the query. In this chapter we will discuss the design and implementation of the A(k)-index data structure. Section 4.1 presents the data structures used in the indexing, and section 4.2 describes our implementation.

4.1 Data Structures

4.1.1 Start of Tag

XML data has a tag name for every node. Every element starts with a tag and ends with a tag. As shown in Figure 4.1, there is start of tag and end of tag for every node. Start of tag means the offset in a file where a node starts. The SAX parser is used

to get the exact position of the start tag of a node. This start position serves as pointer into an XML file, which will be used to retrieve nodes later. In Figure 4.1 the starting position of tag ‘name’ is ‘A’.

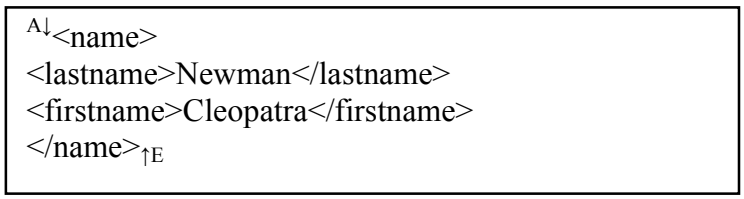

Figure 4.1 Example of Start and End Position of Node

4.1.2 Length of Node

The length of a node is the second component of node information. It is the length of the string starting from the beginning of the node to the end of the node. When the SAX parser encounters the character string after start and end of a node tag, it returns the position of that character. The position of the character encountered after the end of node tag is used to find the offset next to the end of node. Then the length of node is found by subtracting the position of start tag by position after end of tag. What ever comes in between start tag and end tag is used to update the position of pointer in the XML file. This information about each node is used to get the length of all the nodes in a file. In Figure 4.1 a start tag position is ‘A’ and end tag position ‘E’, so the length can be found by subtracting A from E.

4.1.3 Node Object

Node objects are created to record information about nodes. Node object stores information about nodes like node name, starting position and length. A node number is

assigned to each node so that nodes with the same element name have different identity. Node objects also have a successor vector consisting of all successors of a node. Besides this information, a node object also has an access flag. Access flags will be used during query processing so that the same node is not visited twice.

4.1.4 Initial Equivalence Class Hash Table

Table 4.1 Initial Equivalence Class Hash Table

Key	Value(vector)
student	1,5,9
dept	4,8,12

As the name suggests initial equivalence class hash table is a hash table that contains initial equivalence classes. Each unique element name is a hash key and vector containing node number of all nodes with that element names is the value for that key. Figure 4.1 shows the sample initial hash table for the data graph of Figure 2.2.

4.1.5 Successor Vector

Table 4.2 Successor Vector

Node	Nodes in successor vector
0	1,5,9
5	4,8,12

Every non leaf node has set of successors. A successor vector is created for each node and children nodes are in the vector. The table 4.2 shows a sample successor vector for data graph of Figure 2.2.

4.1.6 Final Node Object

Table 4.3 Final Node Object

Equivalence class node ID	Node name	Member nodes objects in vector
2	Student	1,5,9
5	Dept	4,8,12

A final node object that represents the final equivalence class is created before the index tree is created. Each final equivalence class is given a new node number. Then a final node object is created that has node number, node name and vector containing member node objects of this equivalence class. The final node object also has an access flag that will be used in query processing so that the same final node is not traversed twice. A sample final node object is shown in Figure 4.3 for the data graph of Figure 2.2.

4.1.7 Index Tree

Table 4.4 Index Tree

Key (node no. , node name pair)	Successors in vector
0:uta	2:student
2:student	3:undergrad 4:grad 5:dept

After creating the final node object, the index tree is created. Index tree is a hash table, which has a unique key made up of element name and final equivalence class node id. Actually a hash key is an object of final node type. For simplicity it can be represented as a combination of element name and node id. A final node type is chosen

to represent a key because same element name can be shared by more than one equivalence class. The value in hash table is a vector that consists of objects of final node type representing successor final equivalence class nodes. The sample structure of an index tree is show in table 4.4 for data graph of Figure 2.2.

4.2 Implementation

A(k)-index and query processing is implemented in java programming language using SAX parser and Random Access File. In order to construct an A(k)-index, the bisimilarity partition algorithm proposed by the authors of [19] is followed. The algorithm is presented in Appendix A.

In order to examine the data source, data graph is traversed according to depth first method. Each time a new element name is encountered, a node object and an equivalence class for that element name is created. Then A(k)-index algorithm is implemented using the offset, length pair that we proposed earlier. The A(1) index and A(2) index that is constructed for the data graph of Figure 2.1 is shown in Figure 3.7 and Figure 3.8 respectively. The resulting A(k)-index satisfies the requirement that for each label, all nodes in the A(k)-index with such a label have a local similarity larger than or equal to the required one. The storage of A(k)-index consists of the index graph I and the sum of all extents.

In this sub section we will explain our implementation of A(k)-index in detail. Our implementation procedure is shown in Figure 4.2. Before explaining our implementation we will overview SAX parser and Random Access File.

4.2.1 Overview of SAX Parser

XML data is simply ASCII or Unicode data that can be envisioned as a logically structured text document [7]. XML records are stored in a file and it can be loaded in XML parsers like a DOM parser or a SAX parser. The DOM parser allows a random access of XML contents by selecting one or more XML nodes. It uses XPath pattern to get to the node. SAX parser provides fast sequential access by tokenizing the XML nodes in the file. So we use SAX parser to parse the XML document. The SAX parser is event oriented, its API provides a simple, lower level access to an XML document. The SAX events are shown in Table 4.5. SAX parser splits XML file into several chunks, each of which is usually of size 2048 characters. This property of SAX parser makes it possible to parse documents that are larger than system memory. The SAX parser calls procedure 'startElement' whenever it encounters an XML opening element tag. It has parameters uri, localname, raw and atts. The parameter local contains the name of an element and the parameter atts contains either the list of attribute names or nil. The SAX parser calls procedure 'endElement' whenever it encounters an XML closing tag. The procedure 'endElement' has parameters uri, localname and qname. The parameter of interest localname contains the name of the ending tag. The SAX parser calls a procedure 'characters' whenever it encounters a character string. This procedure contains parameters character array, start and length. The parameter length contains the length of the string being processed, and the parameter start is a starting position of the string in character array.

The three procedures of the SAX parser mentioned in the previous paragraph were modified to get the information that we require of all nodes. Besides the procedures mentioned above, SAX has a procedure ‘startDocument’ that is called when an XML document starts, and procedure ‘endDocument’ that is called when the XML document ends.

Table 4.5 SAX Events

XML Document	Events
<?xml version="1.0" ?> <undergrad> <name>Jacob</name> </undergrad>	start document start element: undergrad start element: name characters: Jacob end element: name end element: undergrad end document

4.2.2 Overview of Random Access File

One contribution of our thesis is the use of random access files to retrieve nodes of interest from a data graph. For this purpose we must have a quick access to the nodes of interest. The use of random access file is proposed to serve this purpose. The random access file class implements both data input and data output interfaces and it can be used for both reading and writing. It behaves like a large array of bytes stored in the file system, and it has an index into the implied array, called the file pointer. Random access file provides two methods that are used to access a file. First it has ‘seek’ procedure, which can take a pointer to the desired location in the file. Then it has ‘read’ procedure, which returns a string of given length beginning from the location of the pointer.

4.2.3 Node Object Implementation

The current position of a pointer is set to 0 in the beginning. Then the XML document is processed in depth first manner. Every time an element is encountered a node number is assigned to it and a node object is created. For every node object, a node name and starting position is set as element name and current position respectively. Then the node object is pushed in a stack and a start element flag is set to true.

If end of element is encountered it is compared with the element name of node object that is at the top of stack. If the comparison returns true the node object is popped and is added to the successor vector of node object that is below it in the stack. Then, an end element flag and extra character flag is set to true. Extra character flag signals that a character after end tag is being processed.

If the parser encountered a character string, the character buffer is checked to find if it has changed or not. If the character buffer has changed, the total buffer size (2048 * number of buffer) is added to the start value returned by procedure 'characters' of SAX parser. Then according to the updated value of start, the current position pointer is updated.

There is one more thing that needs to be accounted for properly, a position jumped. If the tag name does not fit in the buffer it goes to new buffer leaving some space in the previous buffer unused. The position jumped is calculated and is subtracted from current position pointer. A position returned from SAX parser and Random access file position may differ due to various reasons, so all the possible cases where buffer

can change are examined and appropriate calculations are done to keep the current position pointer up-to-date.

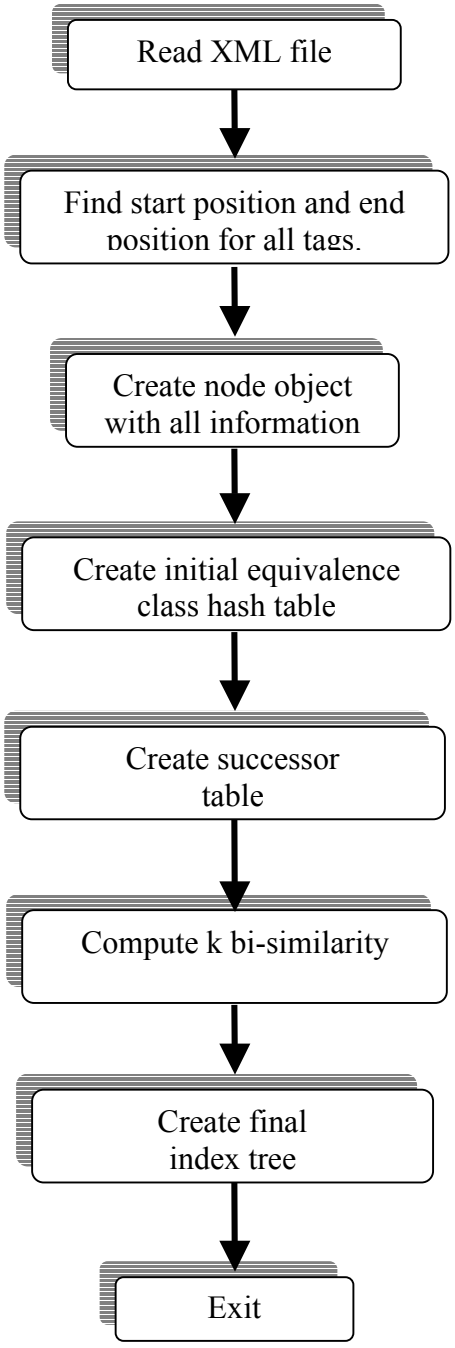

Figure 4.2 Construction of A(k)-index

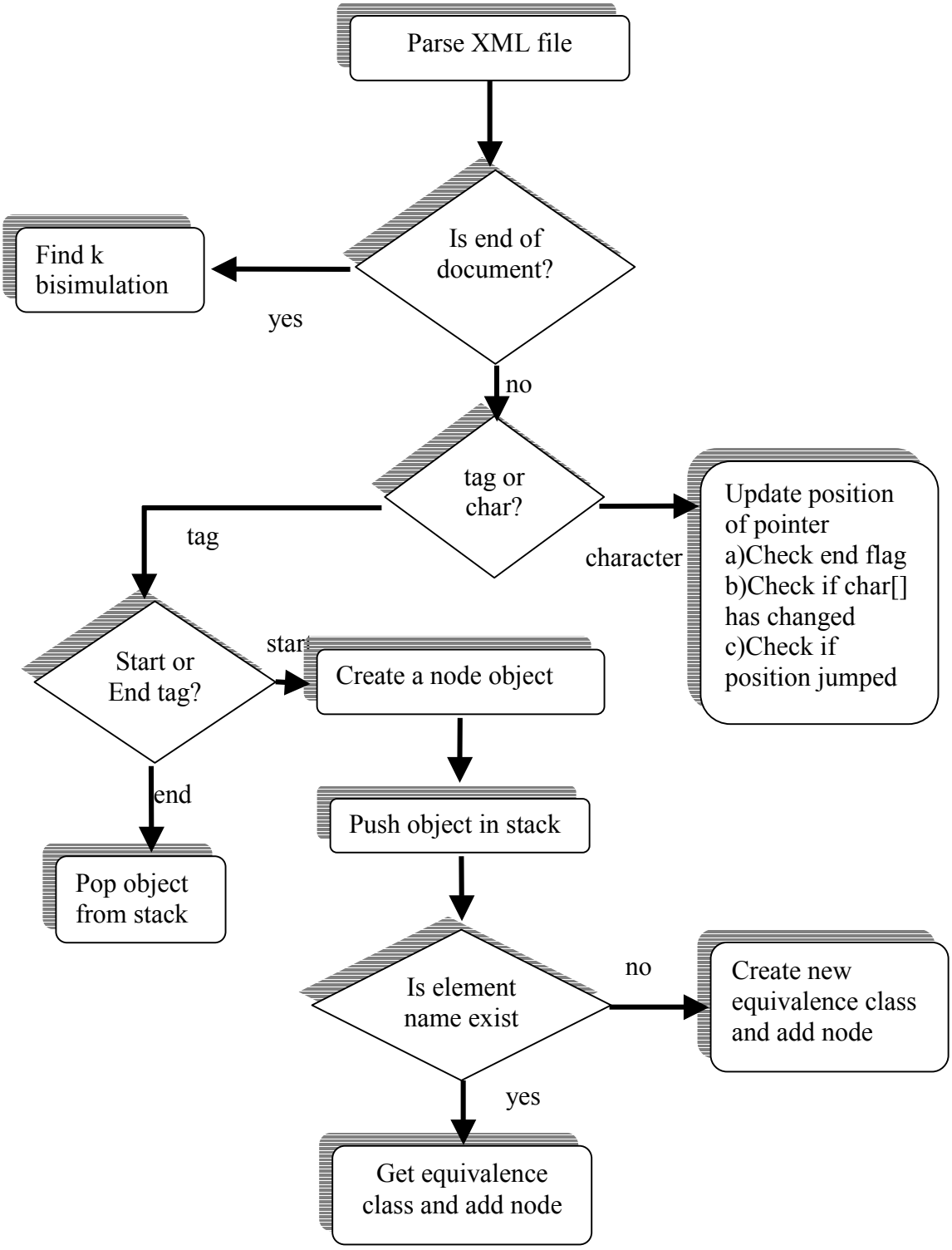

Figure 4.3 Constructions of Node Object and Equivalence Class

The value of the extra character flag is checked when a character string is encountered. If the value of the extra character flag is true, it means that the position behind the end tag is being processed. Once it is determined that the end position of the node has been encountered, the start position of that node object is retrieved. Then the length of the node is calculated and corresponding node object length is set. The length is actually the difference between current position and starting position retrieved. The value of the start element flag is also checked in the procedure 'characters'. If the value of the start element flag is true, the length is added to current position.

Beside node object, for each unique element name an equivalence class hash table is also created whenever a start element name is encountered. This hash table is called initial equivalence class hash table. The data graph is visited in depth first order and if an element is encountered the table is checked to find if that element is already present. If the element name does not exist in the hash table, a new element is created in hash table and the node number of that element is added into the vector that represents the group of elements belonging to that element name. If the element is already present in hash table, the equivalence class vector is retrieved and current node number is added to the vector. This gives the initial equivalence class, which is also be referred to as A(0)-index. The node object construction step is shown in Figure 4.3.

4.2.4 Query Implementation

An XPath query is evaluated on the index graph rather than data graph to improve performance. If several nodes in the index graph satisfy the query path, then the result is the union of all the nodes that are member of the satisfying nodes of index

graph. The Query is parsed and location paths are retrieved one location path at a time. The query path is examined to find if it is //, * or a regular path and processed accordingly. If the path location is //, the result is directly retrieved from the initial equivalence class hash table. If the query is *, all the successors of the final node are iterated. If the path is a regular path string, the final node object that has a path matching the location path is retrieved from index hash table and is processed further. For example if a query is '/grad/name', then the vector value from index hash for hash key 'grad' is retrieved. The substring 'name' and vector value that is retrieved for hash key 'grad' is passed to query procedure to process further. This is a recursive method which goes on until it finds the last location path of query. Once it gets to the last location path, all member node objects are retrieved from member vector; then the query result is extracted from initial hash table. In summary, the index tree, i.e. the index hash table is traversed to get to the result node, but the result is retrieved from the initial equivalence class hash table where start position and length of node resides as part of node object. Some final node objects in the index hash table and node objects in the initial hash table may be encountered more than once resulting in duplicate traversal. This can be prevented by use of a access flag. Every time a node is encountered the access flag is set to true; and if the same node is encountered again, then that node is skipped. In the implementation several queries can run sequentially, so the access flag is set to false before a new query is processed. The query processing step is shown in Figure 4.4.

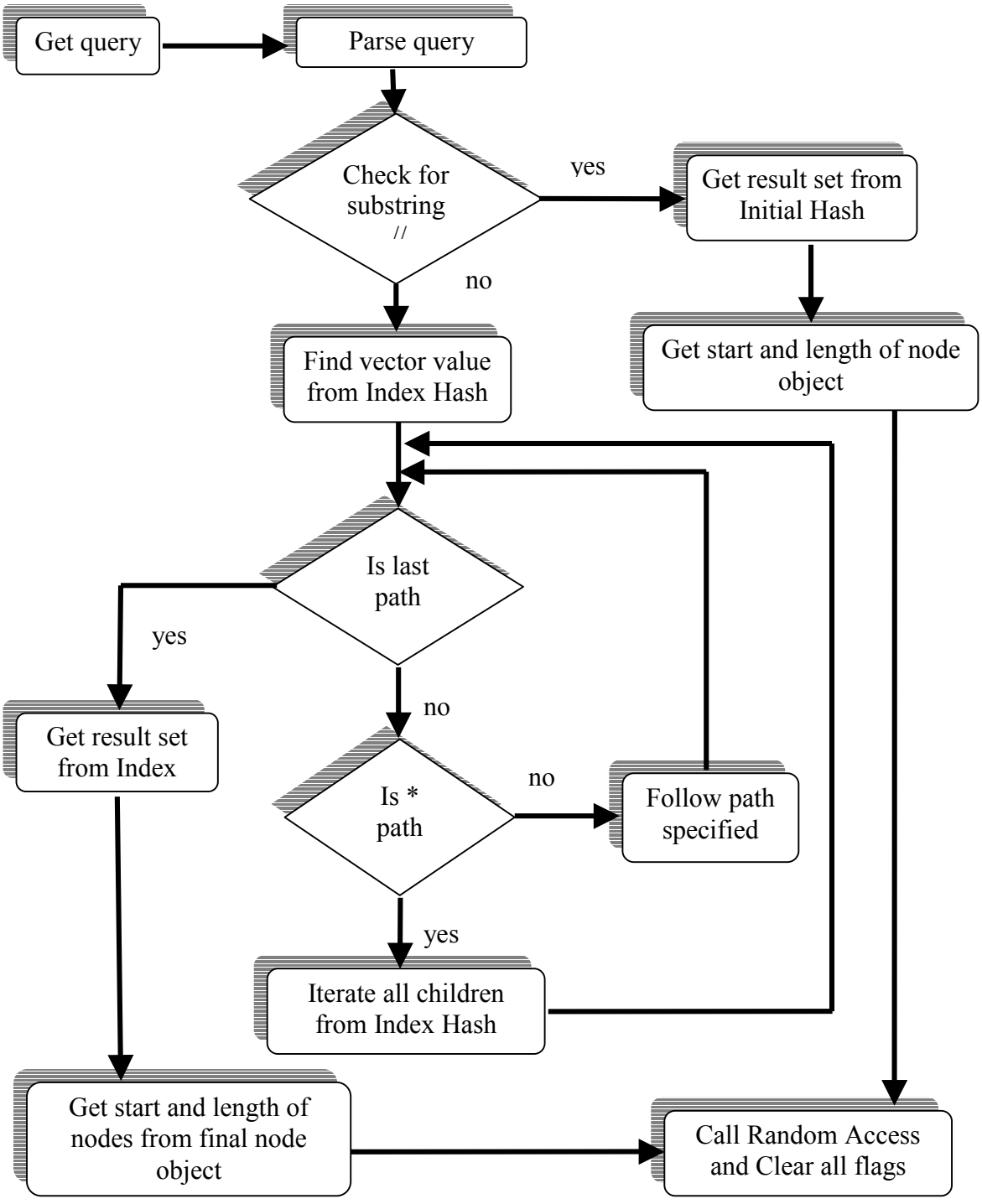

Figure 4.4 Query Processing Steps

CHAPTER 5

EXPERIMENTAL RESULTS

In this section, we describe the XML data that are used in our experiment, and list various results of the experiment. As for our experiment we constructed A(k)-index using different values for k. We obtained the cost for creating index, and noted the reduction in the size of data graph. We used several XPath queries to study performance of query on the index graph. We used a machine with Pentium 4 processor, 2.2 GHz, 512 MB of RAM and 28 GB hard drive. The experiment consists of (a) building an A(k)-index and (b) processing XPath queries given in table 4.4.

5.1 Data

Two different XML data sets are used for our experiment. The current implementation does not support attributes, so the XML data that are used for our experiment do not have attributes. The first XML data is drawn from the website of Niagara Experimental Data, The University of Wisconsin Madison [29]. A XML data 'medicine.xml' which has 2840 nodes is chosen for the purpose. It contains information about students, staff and faculty of a department.

The second XML data is 'othello.xml', which has been widely used in various experiments. It is a part of plays by Shakespeare, distributed by Jon Bosak [30]. It contains 6194 nodes, and it describes various acts and scenes of a play. The data is modified a bit so that the change brought by different values of k can be well noticed.

We notice that XML documents are built in different ways, for example in some files several start tags are in a same line and in some file start tag of each node starts in new line. In our approach the position of the character string encountered after an end tag is used to calculate the length of node in bytes. So the calculation of length would not be correct if a second tag starts after first tag with no space in between. The start tag of nodes in Medicine and Othello XML documents start in a new line. The number of nodes, labels, heights and size on disk of XML data used are shown in Table 5.1.

Table 5.1 XML Data Sets

Data Set	Nodes	Labels	Height	Size on disk
Medicine.xml	2840	18	4	88 KB
Othello.xml	6194	14	5	252 KB

5.2 Cost of Building Index

The cost of building an index is defined as the total time taken to build it. The A(k)-index can be constructed in O(km) time, where m is the number of edges in the data graph. The time taken to build A(k)-index with higher k is more than the time taken to create A(k)-index with smaller k, because higher k requires more partition. The total time taken to build index with various values of k is recorded, and it is presented in Table 5.2.

5.3 Index Size

The index size depends on number of distinct labels, and longest path in data graph. One of the benefits of indexing is reduction in graph size, which is measured by

counting the number of non leaf nodes in the index graph. The storage of A(k)-index consists of the index graph I and the sum of all extents. The difference between the number of data nodes and number of index nodes in Table 5.2 shows how the original data graph has been reduced. The medicine data graph reduced from 2840 nodes to 31 nodes in A(1)-index and reduced to 40 nodes in A(2) and A(3) index. In the Othello A(1), A(2), A(3), A(4), A(5), A(6) and A(7) reduced data graph from 6194 nodes to 27,31,43,55,56,56 and 56 nodes respectively. It is noticed that after a certain value of k, the index graph did not change. In the Othello data, the maximum bisimulation is reached in A(5), after which the index graph does not change even when k is increased.

Table 5.2 Result of Indexing

Index	Medicine			Othello		
	Data Nodes	Index Nodes	Time Elapsed(ms)	Data Nodes	Index Nodes	Time Elapsed(ms)
A(1)	2840	31	650	6194	27	812
A(2)	2840	40	702	6194	31	1180
A(3)	2840	40	806	6194	43	1619
A(4)	2840	40	923	6194	55	2096
A(5)	-	-	-	6194	56	2656
A(6)	-	-	-	6194	56	3094
A(7)	-	-	-	6194	56	3156

5.4 Performance of Queries

XML documents are queried using the simple XPath queries given in Table 5.3. The queries include search operators like wild cards. In the experiment the path notation '/' , '//' and '*' are used.

Table 5.3 Sample Queries

Query	Path Expressions	Datasets
Q1	/gradstudent/email	Medicine
Q2	//phone	Medicine
Q3	/gradstudent/address/city	Medicine
Q4	/department/*/phone	Medicine
Q5	/department/gradstudent/address/city	Medicine
Q11	//stagedir	Othello
Q12	/play/act1/acttitle	Othello
Q13	/play/*/acttitle	Othello
Q14	/play/act1/acttitle/scene	Othello
Q15	/play/act1/acttitle/scene/speech	Othello
Q16	/play/act1/acttitle/scene/speech/line	Othello
Q17	/act1/acttitle/scene/speech/line	Othello
Q18	/acttitle/scene/stagedir	Othello

```
<grad>
  <name>Emily</name>
</grad>
<grad>
 <name>Andrew</name>
</grad>
```

Figure 5.1 Query Result Format

In our experiment the correct result is obtained for most of the queries except for those queries with length greater than value k of A(k)-index. In the time proportional to the length of a label path, A(k)-index was able to access all nodes reachable via that path, independent of the size of the source. The result retrieved from queries not only contained value, it also contained starting tag and ending tag. For example the result for query '//grad' on XML document shown in Figure 1.1 in Figure 5.1.

The query results of different A(k)-index for medicine are presented in Table 5.4. The result of query Q3 and Q5 for A(1)-index included some incorrect result, because the query length is greater than 1. The incorrect results are highlighted by using bold and italic letters in Table 5.4. A(2)-index and A(3)-index index gave correct result for all queries. It is noticed that A(1)-index has the advantage of small index but it may return wrong results requiring validation. If most of the queries are of type Q1 and Q2 then A(1) is best.

Table 5.4 Query Performance on Medicine

Query	A(1)			A(2)			A(3)		
	Nodes visited	Time elapsed	Result Set	Nodes visited	Time elapsed	Result Set	Nodes visited	Time elapsed	Result Set
Q1	3	260	74	3	260	74	3	312	74
Q2	1	630	256	1	614	256	1	750	256
Q3	***4***	***640***	***238***	4	338	74	4	234	74
Q4	15	609	256	15	645	256	15	776	256
Q5	***5***	***604***	***238***	5	177	74	5	203	74

The query results of different A(k)-index for Othello are presented in Table 5.5, Table 5.6 and Table 5.7. The number of nodes visited by the query processor is significantly lower than the original graphs. A(1), A(2) and A(3) indexes are constructed in less time; and number of index nodes are fewer in those index than number of index nodes in A(4), A(5) and A(6) indexes. But query results for A(1), A(2) and A(3)-index included some incorrect result, those incorrect numbers are highlighted using bold and italic letters in Table 5.5.

Table 5.5 Query Performance on Othello (Nodes in result set)

Query	Nodes in result set					
	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)
Q11	208	208	208	208	208	208
Q12	5	5	5	5	5	5
Q13	26	26	26	26	26	26
Q14	***15***	3	3	3	3	3
Q15	***1181***	***1181***	163	163	163	163
Q16	***3556***	***3556***	***3556***	737	737	737
Q17	***3556***	***3556***	***3556***	737	737	737
Q18	129	129	129	129	129	129

Table 5.6 Query Performance on Othello (Time elapsed)

Query	Time elapsed (ms)					
	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)
Q11	625	633	677	671	645	672
Q12	250	213	208	265	271	265
Q13	2390	2157	3390	3141	3049	1822
Q14	2421	360	266	291	375	297
Q15	2289	1625	1053	1078	1305	1219
Q16	860	517	415	464	412	478
Q17	969	625	447	406	377	453
Q18	344	512	571	478	470	491

Table 5.7 Query Performance on Othello (Nodes visited)

Query	Nodes visited					
	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)
Q11	1	1	1	1	1	1
Q12	4	4	4	4	4	4
Q13	26	26	26	26	26	26
Q14	5	5	5	5	5	5
Q15	6	6	6	6	6	6
Q16	7	7	7	7	7	7
Q17	6	6	6	6	6	6
Q18	8	16	16	16	16	16

5.4.1 Elapsed CPU Time

Elapsed CPU time is defined as the total time taken by the query. In our experiment the total time taken to process the query are recorded, it is shown in Table 5.4 and Table 5.6. During the experiment it is found that the time taken to get access of result nodes is very small when compared to time taken to print the result. So to find actual time taken to access the result nodes two sets of experiment were performed for Othello. In the first experiment the total number of nodes in result set and nodes visited are recorded, and in second experiment the total time taken to print out maximum 5 nodes from result set are recorded. Only 5 nodes are printed so that the time recorded is close to the time taken to access it.

5.4.2 Number of Elements Scanned

The cost of a query is defined to be the number of nodes visited in the index graph during path expression evaluation. The number of nodes visited in the index data graph is recorded for each of our queries. The individual nodes within the extent of a matched index node(nodes belonging to traversed equivalence class nodes) are not counted. The experimental results of visited nodes are shown in Table 5.4 and Table 5.7.

To simplify our experiment result let us see the following example. Suppose we have a query '/department/gradstudent/address/city' for medicine.xml which has 2840 nodes. With out any summary graph, first all 'department' nodes have to be found from the root. Then all 'gradstudent' children of the department node have to be found, in turn all 'address' children of gradstudent have to be found, and finally return all 'city'

children of address. Here thousands of nodes have to be traversed. If A(3)-indexing is used, a index graph with 40 nodes is created. Then some index nodes, which are very few compared to original number of nodes, are traversed. In the query given, only 5 nodes have to be traversed to get the result as given in Table 5.4. When traversing the index tree, the path may not exist. For this type of query, such finding guarantees that the query result is empty.

From our experiment above we found that the smaller the value of k the smaller the size of index graph and the less time it takes to construct the index. But smaller k may include incorrect results of queries whose length is more than k. In Table 5.5 A(1), A(2), A(3)-index gave more result for queries Q17 than A(4), A(5), A(6)-index for the same queries, so it is clear that they have some incorrect result. More over it is noticed that A(1)-index gave incorrect result for 4 queries, A(2)-index gave incorrect result for 3 queries and A(3)-index gave incorrect result for 2 queries. So the larger the value of k is, the fewer incorrect results. The larger value of k gives larger index graph and it takes more time to construct, but it also ensures accurate result and no validation is required. So there is a trade off between larger index graph that gives correct result and smaller index graph that may give some incorrect result, requiring more validation. So if query length is studied and the appropriate value of k is chosen the A(k)-index will be very efficient and dependable.

CHAPTER 6

RELATED WORK

In this chapter we will discuss some other works which are closely related to the structural indexing that we have implemented.

6.1 DataGuides

In [17] the authors have proposed a method for extracting all possible path information from a data graph. DataGuides are dynamically generated and maintained to represent the state of the database. DataGuides introduces the concept of target sets, which is the set of nodes that are reachable via that path. DataGuide precisely encode all paths in the data graph, including long and complex paths. Each path in the data graph is represented exactly once regardless of the number of times it appears in the source data graph. Since each path is represented once, the target set of DataGuide overlaps for nodes that can be reached from two or more different parent nodes. Thus, even when two nodes are locally similar, they may be stored in different target sets due to a variety of complex paths. So the storage for DataGuide can be exponentially larger than the size of the data graph. It is constructed by using a non-deterministic finite automation (NFA) and equivalent deterministic finite automation (DFA) is obtained. Since single NFA can have multiple DFA, source data graph may have multiple DataGuides. It is also possible that multiple label paths lead to the same object in the DataGuide.

The improved version of DataGuide is a strong DataGuide. It induces a straight forward one-to-one correspondence between source target sets and DataGuide objects [17]. This property helps the incremental maintenance and query processing. As with the DataGuide, target set of a strong DataGuide may overlap. In the worst case, the number of index nodes in the strong DataGuide can be exponentially larger than the size of the data graph.

DataGuide can be constructed by performing a depth first exploration of the source data graph. When a new path is discovered, a node is created for that target set and stored in the hash table. If a path for a node is already present in the hash table, an edge is added in the DataGuide. DataGuides can be built and incrementally maintained as a comprehensive path index for all possible path expressions. Each object in the strong DataGuide is linked to its corresponding target set in the source data graph. So, the time taken to find source objects through a path is proportional to the length of a label path. DataGuide are not useful in complex queries with several paths.

DataGuide construction of cyclic databases is expensive, so in [16] authors proposed Approximate DataGuide. It relaxes certain aspects of DataGuides and allows some inaccuracy. It keeps important properties, but does not keep all DataGuide paths. DataGuides are stored directly in Lore as Object Exchange Model (OEM) objects. As with relational database systems, user may access and query the DataGuide through Lore's standard interfaces [17].

6.2 1-index

The authors in [24] have proposed a structural summary index called 1-index. This index is based on the computation of a bisimulation relation. If nodes have the same set of incoming paths, 1-index groups them together in an equivalence class. In the worst case, the total size of all extents (i.e. set of nodes in the equivalence class in 1-index) is exactly same as the number of nodes in the data graph. So 1-indexes can be constructed and stored more efficiently than DataGuides. It can be thought of as a non-deterministic finite automaton.

A(k)-index is better than 1-index because A(k)-index uses local similarity to reduce the size of index graph.1-index also precisely encodes all paths in the data graph, including long and complex paths. Even when two nodes are locally similar, they may be stored in different extents. The authors do not discuss how the node information is stored and accessed.

6.3 D(k)-index

In [13], the authors have proposed an indexing technique called D(k)-index. It is an adaptive structural summary for data graph. The main difference between this and the earlier structural indexing is that D(k) indexing assigns different local similarity to different nodes based on the query requirement. So index size in D(k)-index is smaller. D(k)-index generates equivalence classes of various length according to the current query workload requirement. In D(k)-index, local similarity of parent plus one, can not be less than the local similarity of its child. A larger k can be used for longer path

expressions, while a smaller k can be used for shorter path expressions. The authors do not mention how the information is stored and accessed.

6.4 M(k)-index

In [6], the authors have proposed an indexing technique called M(k)-index. It is also adaptive summary of data graph, but is an improvement over D(k)-index. Like D(k)-index, M(k)-index allows different index nodes to have different local similarity requirements, providing finer partitioning only for parts of the data graph targeted by longer path expressions. The difference between M(k)-index and D(k)-index is that M(k)-index does not over-refine the irrelevant data nodes, so M(k)-index has a smaller size. The authors noted that over-refinement is still possible due to over-qualified parent index nodes. So the M*(k)-index is proposed which maintains k-bisimilarity information for all k up to some desired maximum, which can be different across nodes and adjusted dynamically according to the query workload. This feature allows the M*(k)-index to avoid over-refinement due to overqualified parents and support both short and long path expression queries over the same data nodes at the same time[15].

CHAPTER 7

CONCLUSION

7.1 Conclusion

This thesis has been primarily motivated by the need of efficient retrieval of information from XML documents. We also want to gain the capability to retrieve a node from XML file directly using the position of the node in the file. We studied structure based XML indexing and implemented A(k)-index in main memory. We proposed the use of offset, length pair to record all nodes of a data graph. We used the SAX parser to get information of nodes and stored them in such a way that we were able to retrieve any of the nodes directly by using random access file. We investigated the cost of building A(k)-index and cost of querying using various values of k. We also found that for typical database it is easy and fast to create A(k)-index. The performance of A(k)-index depends directly on the structure of the database, but even so the direct access to nodes enables the query processor to retrieve interesting nodes very efficiently.

7.2 Future Work

As part of our on going project on various XML indexing, we plan to run more experiments using various benchmark databases, and compare cost and performance with other indexing techniques. As for the future work, we will conduct research on constructing the structural indexing when an entire XML data graph does not fit in main

memory. During our experiment we faced different kinds of XML documents that required different procedures for handling update of offset. We will identify some standard XML documents and run experiments on them. There are some special characters that are treated differently by SAX parser and random access file. For example &amp and &gt are treated as & and > respectively, which have different byte size. Also, after the special character the rest of the string is cut off. We plan to build a table of special characters and their corresponding update to offset value. We also plan to implement a validation procedure to validate the results of queries whose length is longer than the value k of an A(k)-index.

APPENDIX A

ALGORITHM FOR A(K)-INDEX CONSTRUCTION

1. Algorithm for finding equivalence class

Input: XML document

Output: In memory Initial Equivalence class hash table HT1

```
while not end of xml file
    when ever element with element name EA is encountered
    assign node number N to EA
    if the element name EA  is present in hash table
        get vector value VV for that element and add node number  N to vector
    else
        create a new vector NV and add node number N to it
        put element name EA and vector NV in hash table
end
```

2. Algorithm for finding successor, and building initial tree

Input: XML document

Output: In memory Tree hash table HT2

```
while not end of xml file
    when ever new node object NO is created push object in stack S
    when end element EE is encountered
        while stack S is not empty and corresponding node object NO is not found
            peek the stack, check top of stack and get node object NO1
            get node number NN1 of NO1
            if NO1 = NO
                set corresponding node object NO found
            else
                pop the stack, get the node object NO
                add NO1 into successor vector SS
        if successor vector size is greater than 0
            put node number of NN1 and successor vector SS in hash table
            set successor vector of NO1 = vector SS
end
```

3. Algorithm for computing k-bisim

Input: Set S1 obtained using values of Initial equivalence class hash table HT1

Tree hash table created by successor algorithm, i.e. Algorithm 2

and local similarity requirements of label nodes k

Output: In memory Final equivalence class set S2

```
copy set S1 to set S2, S1 = S2
for i =1 to i = k
    for each value of set S2
        compute successor set "succS2" of S2 using successor tree hash table
        for each value A of set S1
            set_intersect  =  A ∩ succS2
            set_minus =  A – succS2
            replace A by set_intersect and set_minus, resulting new set S1
        if A splits then
            break
    copy set  S1 to set S2
end
```

4. Algorithm for computing A(k)-index

Input: Set S2 created by Algorithm 3, which has final equivalence class

Tree hash table HT2 created by successor algorithm, i.e. Algorithm 2

Output: In memory Index tree

for each equivalence class in set "S2"

create an index node I for each equivalence class

ext[I] = data nodes in the final equivalence class stored in set S2

for each edge from u to v in tree hash table HT2

I[u] = index node containing u

I[v] = index node containing v

If there is no edge from I[u] to I[v] then

add an edge from I[u] to I[v]

end

APPENDIX B

ALGORITHM FOR QUERY

1. Algorithm for processing of query

Input: String XPath query

In memory Hash table index tree

In memory Hash table initial equivalence class

Output: Result of query

1.1 Main

get the query string and parse it at backslash /

S1 = substring after first slash

call procedure firstQuery, pass S1

1.2 Procedure firstQuery(String S1)

```
parse S1
S2 = current location path
S3 = remaining sub string
if S2 is empty, i.e. double slash encountered in the beginning
    Call procedure directToEnd, pass S3 to
else
    while index hash has more element
        get element final node NO
        if access flag of NO is false
            get element name EA of NO
            set access flag of NO to true
            if EA equals S2
                get vector V1 value for NO
                if length of S3 is greater than 0
                    call procedure processAkQuery, pass V1 and S3
```

1.3 Procedure directToEnd(String S3)

```
if initial equivalence class hash table contains key S3
    get vector value V2 for S3
    for each nodes object N in vector V2
        ST = start position of node N
        LT = length of node N
        call random access file method, pass ST and LT
```

1.4 Procedure processAkQuery (Vector V1,String S3)

```
parse S3
if S3 is last location path of query
    call procedure lastAkQuery, pass V1 and S3
else
    S4 = current location path
    S5 = remaining sub string
    while vector V1 has more elements
        get element E
        get final node object FN1 for element E
        if access flag of FN1 is false
            set access flag of FN1 to true
            if S4 equals '*'
                call procedure starQuery, pass FN1 and S5
            else
                get element name EN1 of FN1
                if S4 equals EN1
                    get vector value V2 of final node object FN1
                    for all elements of V2
                        recursive call procedure processAKQuery, pass V2 and S5
```

1.5 Procedure starQuery(Final node object FN1, String S5)

```
if index hash contains FN1
    get vector value V3 of FN1
    call procedure processAkQuery, pass V3 and S5
end
```

1.6 Procedure lastAkQuery(Vector V, String S)

```
while vector V has more elements
    get element E
    get final node object FN for element E
    if access flag of FN is false
        set access flag of FN to true
        set element name EN of FN
        if EN equals S
            get member vector value V1 for FN
            while V1 has more elements
                get element E1
                get node object NO for element E1
                ST = start position of node N
                LT = length of node N
                call random access file method, pass ST and LT
end
```

APPENDIX C

SAMPLE XML FILE

<department>
<deptname>cse</deptname>
<gradstudent>
<name>
<lastname>Papadopoulos</lastname>
<firstname>Johnathan</firstname>
</name>
<phone>755917</phone>
<email>Papadopoulos.Johnathan@foo.edu</email>
<url>http://www/~[Ljava.lang.String;@f44ebe</url>
<address>
<city>Janesville</city>
<state> WI</state>
<zip>53708</zip>
</address>
<office>6152</office>
<gpa>1.8668671963852623</gpa>
</gradstudent>
<gradstudent>
<name>
<lastname>Abiteboul</lastname>
<firstname>Ralf</firstname>
</name>
<phone>8987305</phone>
<email>Abiteboul.Ralf@foo.edu</email>
<address>
<city>Janesville</city>
<state> WI</state>
<zip>53708</zip>
</address>
<office>9533</office>
<gpa>0.43996358085274956</gpa>
</gradstudent>
<undergradstudent>
<name>
<lastname>Robertson</lastname>
<firstname> Joan </firstname>
</name>
<phone>8767892</phone>
<email>Robertson.Joan@foo.edu</email>
<address>
<city>Janesville</city>
<state> WI</state>

```
  <zip>53708</zip>
  </address>
  <gpa>0.717747091607051</gpa>
</undergradstudent>
<staff>
  <name>
  <lastname>Newman</lastname>
  <firstname>Cleopatra</firstname>
  </name>
  <phone>8415977</phone>
  <email>Newman.Cleopatra@foo.edu</email>
  <office>3896</office>
</staff>
<faculty>
  <name>
  <lastname>Tzavaras</lastname>
  <firstname>Ivan</firstname>
  </name>
  <phone>7765691</phone>
  <email>Tzavaras.Ivan@foo.edu</email>
  <office>871</office>
</faculty>
</department>
```

APPENDIX D

INTERMEDIATE RESULTS AS OBTAINED IN A(K)-INDEX CONSTRUCTION

The initial equivalence class hashtable

1) key =gpa [node 14 st 400 len 29 ; node 26 st 739 len 30 ; node 37 st 1083 len 28 ;]
2) key =address [node 9 st 278 len 93 ; node 21 st 617 len 93 ; node 33 st 978 len 101 ;]
3) key =deptname [node 1 st 14 len 24 ;]
4) key =url [node 8 st 225 len 49 ;]
5) key =state [node 11 st 318 len 19 ; node 23 st 657 len 19 ; node 35 st 1018 len 19 ;]
6) key =undergradstudent [node 27 st 787 len 345 ;]
7) key =phone [node 6 st 151 len 21 ; node 19 st 550 len 22 ; node 31 st 911 len 22 ; node 42 st 1249 len 22 ; node 49 st 1465 len 22 ;]
8) key =gradstudent [node 2 st 40 len 405 ; node 15 st 447 len 338 ;]
9) key =lastname [node 4 st 67 len 33 ; node 17 st 474 len 30 ; node 29 st 819 len 38 ; node 40 st 1155 len 35 ; node 47 st 1374 len 37 ;]
10) key =office [node 13 st 375 len 21 ; node 25 st 714 len 21 ; node 44 st 1318 len 21 ; node 51 st 1531 len 20 ;]
11) key =faculty [node 45 st 1351 len 212 ;]
12) key =staff [node 38 st 1134 len 215 ;]
13) key =email [node 7 st 176 len 45 ; node 20 st 576 len 37 ; node 32 st 937 len 37 ; node 43 st 1275 len 39 ; node 50 st 1491 len 36 ;]\
14) key =city [node 10 st 291 len 23 ; node 22 st 630 len 23 ; node 34 st 991 len 23 ;]
15) key =firstname [node 5 st 104 len 32 ; node 18 st 508 len 27 ; node 30 st 861 len 35 ; node 41 st 1194 len 40 ; node 48 st 1415 len 35 ;]
16) key =department [node 0 st 0 len 1578 ;]
17) key =zip [node 12 st 341 len 16 ; node 24 st 680 len 16 ; node 36 st 1041 len 24 ;]
18) key =name [node 3 st 57 len 90 ; node 16 st 464 len 82 ; node 28 st 809 len 98 ; node 39 st 1145 len 100 ; node 46 st 1364 len 97 ;]

The successor hashtable

1) key =33 [node 36 st 1041 len 24 ; node 35 st 1018 len 19 ; node 34 st 991 len 23 ;]

2) key =28 [node 30 st 861 len 35 ; node 29 st 819 len 38 ;]

3) key =27 [node 37 st 1083 len 28 ; node 33 st 978 len 101 ; node 32 st 937 len 37 ; node 31 st 911 len 22 ; node 28 st 809 len 98 ;]

4) key =21 [node 24 st 680 len 16 ; node 23 st 657 len 19 ; node 22 st 630 len 23 ;]

5) key =9 [node 12 st 341 len 16 ; node 11 st 318 len 19 ; node 10 st 291 len 23 ;]

6) key =46 [node 48 st 1415 len 35 ; node 47 st 1374 len 37 ;]

7) key =45 [node 51 st 1531 len 20 ; node 50 st 1491 len 36 ; node 49 st 1465 len 22 ; node 46 st 1364 len 97 ;]

8) key =16 [node 18 st 508 len 27 ; node 17 st 474 len 30 ;]

9) key =15 [node 26 st 739 len 30 ; node 25 st 714 len 21 ; node 21 st 617 len 93 ; node 20 st 576 len 37 ; node 19 st 550 len 22 ; node 16 st 464 len 82 ;]

10) key =39 [node 41 st 1194 len 40 ; node 40 st 1155 len 35 ;]

11) key =3 [node 5 st 104 len 32 ; node 4 st 67 len 33 ;]

12) key =38 [node 44 st 1318 len 21 ; node 43 st 1275 len 39 ; node 42 st 1249 len 22 ; node 39 st 1145 len 100 ;]

13) key =2 [node 14 st 400 len 29 ; node 13 st 375 len 21 ; node 9 st 278 len 93 ; node 8 st 225 len 49 ; node 7 st 176 len 45 ; node 6 st 151 len 21 ; node 3 st 57 len 90 ;]

14) key =0 [node 45 st 1351 len 212 ; node 38 st 1134 len 215 ; node 27 st 787 len 345 ; node 15 st 447 len 338 ; node 2 st 40 len 405 ; node 1 st 14 len 24 ;]

The original set Q

[0] [1] [27] [13 25 44 51] [10 22 34] [6 19 31 42 49] [5 18 30 41 48] [45] [8] [14 26 37] [2 15] [9 21 33] [4 17 29 40 47] [7 20 32 43 50] [12 24 36] [3 16 28 39 46] [11 23 35] [38]

The final equivalence class A(3)

[29] [27] [38] [24 12] [8] [0] [19 6] [3 16] [10 22] [44] [1] [42] [46] [45] [43] [49] [31] [11 23] [36] [37] [9 21] [17 4] [28] [32] [35] [39] [30] [48] [20 7] [40] [33] [41] [14 26] [50] [34] [2 15] [51] [18 5] [47] [25 13]

The equivalence class with edges(indexHash)

Hash Key 12:name
27:firstname :{ mem 48 st at 1415 len 35;}
38:lastname :{ mem 47 st at 1374 len 37;}

Hash Key 13:faculty
36:office :{ mem 51 st at 1531 len 20;}
33:email :{ mem 50 st at 1491 len 36;}
15:phone :{ mem 49 st at 1465 len 22;}
12:name :{ mem 46 st at 1364 len 97;}

Hash Key 20:address
3:zip :{ mem 24 st at 680 len 16; mem 12 st at 341 len 16;}
17:state :{ mem 11 st at 318 len 19; mem 23 st at 657 len 19;}
8:city :{ mem 10 st at 291 len 23; mem 22 st at 630 len 23;}

Hash Key 2:staff
9:office :{ mem 44 st at 1318 len 21;}
14:email :{ mem 43 st at 1275 len 39;}
11:phone :{ mem 42 st at 1249 len 22;}
25:name :{ mem 39 st at 1145 len 100;}

Hash Key 30:address
18:zip :{ mem 36 st at 1041 len 24;}
24:state :{ mem 35 st at 1018 len 19;}
34:city :{ mem 34 st at 991 len 23;}

Hash Key 35:gradstudent
32:gpa :{ mem 14 st at 400 len 29; mem 26 st at 739 len 30;}
39:office :{ mem 25 st at 714 len 21; mem 13 st at 375 len 21;}
20:address :{ mem 9 st at 278 len 93; mem 21 st at 617 len 93;}
28:email :{ mem 20 st at 576 len 37; mem 7 st at 176 len 45;}
6:phone :{ mem 19 st at 550 len 22; mem 6 st at 151 len 21;}
7:name :{ mem 3 st at 57 len 90; mem 16 st at 464 len 82;}
4:url :{ mem 8 st at 225 len 49;}

Hash Key 5:department
13:faculty :{ mem 45 st at 1351 len 212;}
2:staff :{ mem 38 st at 1134 len 215;}
1:undergradstudent :{ mem 27 st at 787 len 345;}
35:gradstudent :{ mem 2 st at 40 len 405; mem 15 st at 447 len 338;}
10:deptname :{ mem 1 st at 14 len 24;}

Hash Key 25:name
31:firstname :{ mem 41 st at 1194 len 40;}
29:lastname :{ mem 40 st at 1155 len 35;}

Hash Key 22:name
26:firstname :{ mem 30 st at 861 len 35;}
0:lastname :{ mem 29 st at 819 len 38;}

Hash Key 1:undergradstudent
19:gpa :{ mem 37 st at 1083 len 28;}
30:address :{ mem 33 st at 978 len 101;}
23:email :{ mem 32 st at 937 len 37;}
16:phone :{ mem 31 st at 911 len 22;}
22:name :{ mem 28 st at 809 len 98;}

Hash Key 7:name
37:firstname :{ mem 18 st at 508 len 27; mem 5 st at 104 len 32;}
21:lastname :{ mem 17 st at 474 len 30; mem 4 st at 67 len 33;}

REFERENCES

[1] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu "A Query Language for XML", In proceeding of the Eight World-Wide Web Conference, 1999.

[2] C.W. Chung, J.K. Min, K. Shim, "APEX: An Adaptive Path Index for XML data", ACM SIGMOD June, 2002.

[3] D. Chamberlin, " XQuery: An XML query language", IBM systems Journal, Vol 41, No. 4, 2002.

[4] D.Chamberlin, J.Robie and D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", Proceedings of WebDB, 2000.

[5] D. Florescu, D. Kossmann and I. Manolescu, "Integrating Keyword Search into XML Query Processing", The International Journal of Computer and Telecommunications Networking, Vol 33, Issue 1-6(119-135), 2000.

[6] H. He and J. Yang, "Multiresolution Indexing of XML for Frequent Queries". ICDE, 2004.

[7] H. Henseler, "Indexing XML databases WHITEPAPER Using a full-text search engine as a driver for native XML databases", Zylab, February, 2003.

[8] J. McHugh, J.Widom, S.Abiteboul, Q.Luo and A.Rajamaran, "Indexing Semistructured Data", Technical Report, Stanford University, January, 1998.

[9] L. Guo, F. Shao, C. Botev, and J. Shanmugasudaram, "XRANK: Ranked Keyword Search over XML Documents", SIGMOD, June, 2003.

[10] M. F. Fernandez, D. Florescu, A. Y. Levy and D. Suciu, “A query language for a Web-site management system”, ACM SIGMOD Record, 26(3):4-11,1997.

[11] P. Rao, and B. Moon, “PRIX: Indexing and Querying XML Using Prufer Sequences”, ICDE, 2004.

[12] P. Buneman, S. Davidson, G. Hillebrand and D. Suciu, “A Query Language and Optimization Techniques for Unstructured Data”, In Proceeding of the 1996 ACM SIGMOD International Conference on Management of Data, pages 505-516, June, 1996.

[13] Q. Chen, A. Lim, and K.W. Ong, “D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data”, SIGMOD, June, 2003.

[14] Q. Li and B. Moon, “Indexing and Querying XML Data for Regular Path Expressions”, Proceedings of the 27th VLDB Conference, 2001.

[15] Q. Li, R. Elmasri, S. Prabhakar, N. Manandhar, Do Y. Kim, “Survey of XML Indexing Techniques”.

[16] R. Goldman and J. Widom, “Approximate DataGuides”, In Proc. of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, pages 436–445, January, 1999.

[17] R. Goldman and J. Widom, “Dataguides: Enabling query formulation and optimization in semistructured databases”, In Twenty-Third International Conference on Very Large DataBases, pages 436–445, 1997.

[18] R. Kaushik, P. Bohannon , J. F. Naughton, P. Shenoy, “ Updates for Structural Indexes”, Proceedings of the 28th VLDB Conference, 2002.

[19] R. Kaushik, P. Shenoy, P. Bohannon and Ehud Gudes, "Exploiting Local Similarity for Efficient Indexing of Paths in Graph-Structured Data", ICDE, 2002.

[20] R. Paige and R. Tarjan, "Three Partition Refinement Algorithms, SIAM Journal of Computing", 16:973-988, 1987.

[21] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. L. Wiener, "The Lorel Query Languages for Semistructured Data", International Journal on Digital Libraries, 1:68-88, 1997.

[22] S. Prabhakar, "Indexing Techniques for XML Data", MS Thesis, CSE Department, University of Texas at Arlington, December, 2004.

[23] T. Grust, "Accelerating XPath Location Steps", ACM SIGMOD, June, 2002.

[24] T. Milo and D. Suciu, "Index structures for path expressions", In ICDT: 7th International Conference on Database Theory, 1999.

[25] V. Ganapathy, G. Balakrishnan, "On Computing k-Bisimilar Partitions of Split Graphs", http://www.cs.wisc.edu/~vg/writings/term_projects/cs764-report.pdf.

[26] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword Proximity Search on XML Graphs", ICDE, 2003.

[27] http://www.w3.org/.

[28] http://www.w3.org/TR/xpath.

[29] Niagara is available at http://www.cs.wisc.edu/niagara/data.html.

[30] Shakespeare is available at http://metalab.unc.edu/bosak/xml/eg/shaks200.zip.

BIOGRAPHICAL INFORMATION

Niroj Manandhar was born in Thimi, Bhaktapur, Nepal. He completed his undergraduate studies in Business Administration from Tribhuvan University, Kathmandu, Nepal in May 1997. He earned his second Bachelors degree in Computer Science and Engineering in May 2002, and then he joined Masters Program in Computer Science and Engineering at UTA in August 2003. His major areas of interest are Database System, Web Database and Networks. He earned his Masters in Computer Science degree in August 2005.