M. Sc. Engineering Thesis

MODIFIED INVERTED FILES AND ALGORITHMS FOR PHRASE QUERY AND NOT QUERY

by

Tuhin Paul

Submitted to

the Department of Computer Science and Engineering

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

বাংলাদেশ প্রকৌশল বিশ্ববিদ্যালয়
নং. 107529
তাং. 25.1.10
লাইব্রেরী, ঢাকা

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology (BUET)

Dhaka-1000

December 20, 2009

The thesis titled "**MODIFIED INVERTED FILES AND ALGORITHMS FOR PHRASE QUERY AND NOT QUERY**", submitted by Tuhin Paul, Roll No. 100605048P, Session October, 2006, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents.

The Examination was held on December 20, 2009.

26.12.2009

Board of Examiners

1. _____ Chairman (Supervisor)
Dr. Md. Humayun Kabir
Associate Professor
Department of CSE
BUET, Dhaka 1000

2. _____ Member (Ex-Officio)
Dr. Md. Monirul Islam
Professor & Head
Department of CSE
BUET, Dhaka 1000

3. _____ Member
Dr. A. S. M. Latiful Hoque
Associate Professor
Department of CSE
BUET, Dhaka 1000

4. _____ Member
Dr. Md. Mostofa Akbar
Associate Professor
Department of CSE
BUET, Dhaka 1000

5. _____ Member (External)
Dr. M. Ameer Ali
Assistant Professor
Department of CSE
East West University
Dhaka

Declaration

This is to certify that the research work entitled "Modified Inverted Files and Algorithms for Phrase Query and Not Query" is the outcome of the investigations carried out by me under the supervision of Dr. Md. Humayun Kabir in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka-1000. It is also declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree of diploma.

Tuhin Paul

Candidate

Acknowledgements

First I thank the Almighty God for providing me with the knowledge, determination, and capability to successfully complete this thesis. I owe my deepest gratitude to my Supervisor Dr. Md. Humayun Kabir, Associate professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology for his encouragement, magnificent guidance and witty suggestions. I would like to show my gratitude to Dr. A. S. M. Latiful Hoque and Dr. Md. Mostofa Akbar for their precious suggestions and support. It has been a great experience to work with them. Their erudite guidance and advice, profound knowledge, great experience and amusing support have given me a great momentum to accomplish this thesis work. I also want to thank the other members of the board of examiners of my thesis: Dr. Md. Monirul Islam and Dr. M. Ameer Ali for their valuable suggestions. Finally, I express my gratitude to my parents, my sister and brother, and to my wife who have inspired me and provided all necessary support in all of my academic endeavors.

Abstract

Inverted files, equivalent to database indices, are used to speed up the search of both Hyper Text Markup Language (HTML) and eXtensible Markup Language (XML) files in the web. Searching XML files differs from that of HTML in two ways: inverted files for XML need to be compressed because of their large size and the query evaluation against XML files requires keyword searching both in the structure and in the values. XML queries are often composed of multiple keywords with logical relations. XML queries with conjunction, disjunction, ancestor-descendant, and preceding-following relations among the multiple keywords have already been evaluated successfully. Multiple keywords often appear in the XML queries as a phrase. Phrase Query in a single XML document has already been evaluated. However, the method to evaluate phrase query in a large or small collection of XML documents does not exist. Additionally, a special type of query where keywords or phrases must not be present in the evaluated XML documents is also required in many applications. As per our study, the method to evaluate this NOT queries does not exist either. XML document retrieval will not be complete without evaluating these two important types of queries. New solutions are required to process both phrase and NOT queries efficiently. In this thesis, we introduce the methods to evaluate both phrase and NOT queries proposing necessary changes in the inverted file structure and query processing algorithms. We have used pull parser to parse the XML documents. We have developed a prototype query processor which is capable of creating inverted files and evaluating all types of queries including phrase and NOT queries. Our experimental results using this prototype query processor show the effectiveness of our proposed query evaluation methods.

Acronyms

Abbreviation	Description
ASCII	American Standard Code for Information Interchange
CDATA	Character Data
CSS	Cascading Style Sheets
DTD	Document Type Definition
FLWOR	FOR, LET, WHERE, ORDER BY, RETURN
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
LF	Line Feed character
PCDATA	Parsed Character Data
RSS	Really Simple Syndication
SMIL	Synchronized Multimedia Integration Language
SOAP	Simple Object Access Protocol
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WML	Wireless Markup Language
WSDL	Web Service Description Language
XHTML	eXtensible Hyper Text Markup Language
XML	eXtensible Markup Language
XPath	XML Path Language
XQuery	XML Query language
XSLT	eXtensible Stylesheet Language Transformations

Nomenclatures

Nomenclature	Description
f_t	Number of documents in a collection, that contain a term t
$f_{d,t}$	Number of times a term t appears in a document d
k	Maximum allowable distance between successive keywords of a phrase
$w_{q,t}$	Query vector for a term t
$\boldsymbol{W}_q$	Query vector for a search query q
$w_{d,t}$	Document vector for a term t in a document d
$\boldsymbol{W}_d$	Document vector for a document d
θ	The angle between the vectors $\boldsymbol{W}_q$ *and* $\boldsymbol{W}_d$
n	Number of keywords in a phrase
p	Allowed number of absence of phrase keywords

Table of Contents

Acknowledgements iii

Abstract iv

Acronyms v

Nomenclatures vi

List of Figures x

List of Tables xi

List of XML Structures xii

List of Algorithms xiii

1 Introduction 1

1.1 Motivation 2

1.2 Contributions 5

1.3 Organization of Thesis 5

2 XML Basics 7

2.1 Structure of a HTML Document 7

2.2 XML Document Structure and Syntax 9

2.2.1 XML Declaration 10

2.2.2 Root Element 10

2.2.3 Element 10

2.2.4 Attribute 11

2.2.5 Comments and Text Data 12

2.2.6 Special Characters 13

2.2.7 White Space and New Line in XML 13

2.3 Tree Structure of an XML Document 14

2.4 Extensibility Feature of XML 15

2.5 Namespaces in XML 16
2.6 Well Formed and Valid XML 18
2.7 Displaying XML 18
2.8 Languages for Querying XML Documents 19
2.9 Applications of XML 21
2.10 Summary 23

3 Literature Review 24

3.1 HTML Inverted File Structure and Ranking 24
3.2 XML Inverted File Structure 26
3.3 XML Inverted File Compression 30
3.4 Processing Text Search Queries in XML 32
3.5 Query Processing and Query Plan Tree 33
3.6 PIX - Phrase Matching in XML 34
3.7 Ranking and Weighting in Querying XML documents 36
3.8 Limitations 38
3.9 Summary 39

4 Proposed Solutions for Phrase and NOT Queries 40

4.1 Reasons for New Solution to Phrase Query in XML 40
4.2 New Processing Method for Partial Phrase Matching 41
4.2.1 New Inverted File Structure for Phrase Query 41
4.2.2 Determining Where to Search the Given Phrase 43
4.2.3 Finding presence of a phrase within a domain element 45
4.3 New Processing for NOT Query 53
4.4 Phrase in NOT Queries 58
4.5 Summary 58

5 Experimental Results 59

5.1 Experimental Test Cases 59
5.2 Effect of Proposed Modification on Inverted File Size 62
5.3 Comparison of Relevancy in Phrase Query 62
5.4 Comparison of Efficiency in Phrase Query 64

5.5 Analysis of Proximity (k) in Phrase Query 65
5.6 Comparison of Relevancy in NOT Query 68
5.7 Comparison of Efficiency in NOT Query 70
5.8 Summary 71
6 Conclusions and Future Works 72
6.1 Conclusions 72
6.2 Future Works 72
7 References 74

List of Figures

Figure 2.1 A Simple HTML Document 8
Figure 2.2 Output of the HTML Document in Figure 2.1 8
Figure 2.3 Tree Structure for the XML Document in XML Structure 2.3 14
Figure 3.1 Uniquely Identifying Nodes in XML Document Tree Structure 27
Figure 3.2 Tree Structure for XML Structure 3.1 29
Figure 3.3 Query Plan Tree 34
Figure 4.1 Representing NOT Query in Query Plan Tree 53
Figure 4.2 ($doc^1_i = doc^2_j$) and ($path^1_i = path^2_j$) in NOT Query 54
Figure 4.3 ($doc^1_i = doc^2_j$) and ($path^1_i \subset path^2_j$) in NOT Query 55
Figure 4.4 ($doc^1_i = doc^2_j$) and ($path^2_j \subset path^1_i$) in NOT Query 55
Figure 4.5 ($doc^1_i = doc^2_j$) and ($path^1_i \neq path^2_j$) in NOT Query 56
Figure 5.1 Comparison of Inverted File Sizes 62
Figure 5.2 Phrase Query Relevancy Comparison with the Conjunction Operation 63
Figure 5.3 Phrase Query Relevancy Comparison with the Disjunction Operation 64
Figure 5.4 Phrase Query Efficiency Comparison with the Conjunction Operation 64
Figure 5.5 Phrase Query Efficiency Comparison with the Disjunction Operation 65
Figure 5.6 Effect of Proximity on Relevancy in Test Set 2 66
Figure 5.7 Effect of Proximity on Relevancy in Test Set 3 66
Figure 5.8 Effect of Proximity on Efficiency in Test Set 2 67
Figure 5.9 Effect of Proximity on Efficiency in Test Set 3 67
Figure 5.10 NOT Query Relevancy Comparison with Conjunction Operation 69
Figure 5.11 NOT Query Relevancy Comparison with Disjunction Operation 69
Figure 5.12 NOT Query Efficiency Comparison with Conjunction Operation 70
Figure 5.13 NOT Query Efficiency Comparison with Disjunction Operation 71

List of Tables

Table 2.1 Encoding Information Using Attributes and Elements in XML 11
Table 2.2 Special Characters in XML 13
Table 2.3 XPath Example 19
Table 2.4 XQuery Example 21
Table 3.1 Inverted File Structure for Text Database 25
Table 3.2 Regular Expressions for Inverted Lists for XML 28
Table 3.3 A Sample Inverted List 29
Table 3.4 Semi-compressed Inverted List for Table 3.3 31
Table 3.5 XS Tree Representation for Figure 3.2 32
Table 3.6 Conjunctive and Disjunctive Queries 33
Table 3.7 Assigning (start, end) to Elements and Textual Words in PIX 35
Table 3.8 Index Structure for Table 3.7 in PIX 35
Table 4.1 Keyword Positions in *positional paths* (Node Names and Sequence Indexes). 42
Table 4.2 Modified Regular Expressions for Inverted Lists of XML Inverted Files 42
Table 4.3 Sample Inverted List for a Term in the Inverted File 43
Table 4.4 Tree Structures for the Keywords in Phrase Query 45
Table 4.5 Descendants of `tales[1]/tale[1]` in XML Structure 4.1 45
Table 4.6 Keyword Positions in Descendants of `tales[1]/tale[1]` 46

List of XML Structures

XML Structure 1.1 XML Document with Mixed Nodes 3
XML Structure 1.2 Simple XML Documents to Illustrate NOT Query 4
XML Structure 2.1 A Simple XML Document 9
XML Structure 2.2 Using CDATA in XML Documents 12
XML Structure 2.3 Sample XML Document 14
XML Structure 2.4 XML Document with Conflicting Element Names 16
XML Structure 2.5 Using Namespaces to Resolve Name Conflicts 17
XML Structure 2.6 Declaring Namespaces at the Root Element 17
XML Structure 3.1 Sample XML document 28
XML Structure 4.1 XML Document for Phrase Query 44

List of Algorithms

Algorithm 4.1 Algorithm for Partial Phrase Matching in XML 52
Algorithm 4.2 Algorithm for NOT Query Processing 58

বাংলাদেশ প্রকৌশল বিশ্ববিদ্যালয়
নং 107529
তাং 25.1.10
লাইব্রেরী, ঢাকা

Chapter 1

Introduction

XML (eXtensible Markup Language) [1] is applied mainly in data storage and exchange. It has been a World Wide Web Consortium (W3C) recommended markup language since February 10, 1998. XML standard has been developed amazingly quickly and a large number of software vendors have also adopted the standard. It became the primary tool for data transmissions between all sorts of applications and is finding ever more importance in the area of storing and describing information.

An XML document collection can be considered as a semi-structured database. Contrary to Hyper Text Markup Language (HTML), where markups can be ignored because they are used only for presentation purpose and do not bear any special information, tags or markups in XML or structured documents play important roles and hence they cannot be ignored [2]. To search over a collection of XML or HTML documents inverted files [3][4][5], which are equivalent to database indices, are used to store the index of the search keywords. Inverted files speed up the searching process. Inverted file structure for XML document collection is different [6][7] from that of text or HTML documents. XML inverted files store more information, such as tags and the document structure, than that is done by HTML inverted files. As a result, the sizes of the XML inverted files become larger than those of HTML inverted files. For this reason, inverted files for XML need to be compressed [2][6]. Again, including the structure in XML document searching makes it more challenging than plain text or HTML document searching, where structure is not included.

1.1 Motivation

BitCube [8][9] is an early system for XML document retrieval using bitmap indexing where the keywords are mapped into a multi dimensional array of bits which states in which documents and tags the keywords are found. However, BitCube can not identify these tags uniquely. BitCube does not encode the number of occurrences of keywords in these tags or documents either.

In [6] an inverted file structure for XML is proposed that uniquely identifies the location of a keyword in an XML document and employs compression methods such as run length encoding for integers, to eliminate common data in subsequent location information of keywords. For further compression, the authors of [6] have proposed a method of storing the XML document structures in a separate file and using references to these structures in the inverted file to determine keyword locations within XML documents. The research work in [2] also proposed to store the index file, compressed structural and textual information of XML documents in separate files. Though the inverted file in [6] uniquely identifies the locations where a keyword appears, it does not encode the word positions of the keyword in those locations.

The relationships between keywords are important in matching XML documents against query keywords. While processing a conjunctive (AND) query, an XML document is accepted if the document contains all of the keywords in the query. To match a disjunctive (OR) query, a document needs to contain at least one of the keywords. The relationships between keywords are not limited to 'AND' and 'OR' only - other conditions are also possible. As an example, a keyword may need to precede another one in the resultant documents which contain both of these keywords (preceding-following relationship) or the keywords may appear in an ancestor-descendant hierarchy. Methods to process XML queries with conjunction, disjunction, ancestor-descendant, and preceding-following relationships among multiple keywords have been proposed in [10]. Multiple keywords often appear as a phrase in the XML queries. This special type of query is called Phrase Query which searches an explicitly specified phrase and identify the XML documents in which the keywords appear in the same order and pattern as they

appear in the query. Though Phrase Query techniques [3][11] are available for text or HTML documents, these cannot be used to evaluate the phrase queries in XML documents. This is because the evaluation of Phrase Query for text or HTML documents matches the phrase only in the plain texts, which are enclosed by the tags or markers. It ignores the tags or markers completely while evaluating the phrase. However, in XML documents a phrase might be comprised of both plain texts and markers. Again, partial matching of a phrase in an XML document is also very important. As an example, if we consider the query "Author Rudyard Kipling wrote Jungle Book" the XML document in XML Structure 1.1 does not contain an exact match of the phrase in the query; the keyword 'Rudyard' is absent in the document. The rest of the keywords and their order in the query are matching and the document also contains the relevant information. So it is reasonable to include the document in the search result for the query.

```
<?xml version="1.0" encoding="utf-8" ?>
<catalog>
  <book>
    <author>Kipling</author> wrote <name>Jungle Book</name> in 1894
  </book>
</catalog>
```

XML Structure 1.1 XML Document with Mixed Nodes

Additionally, a special type of queries, NOT query, contains keywords or phrases that are expected not to be present in the resultant XML documents. If we search for "*berry, not black*" ('*not*' is not a keyword here - it denotes that the keyword '*black*' should not be present in the resultant documents. For the following XML data in XML Structure 1.2, the result should return the first document but not the second one:

(a) First Document

```
<?xml version="1.0" encoding="utf-8" ?>
<fruits>
  <fruit name="berry">
    <color>pink</color>
  </fruit>
</fruits>
```

(b) Second Document

```
<?xml version="1.0" encoding="utf-8" ?>
<fruits>
  <fruit name="berry">
    <color>black</color>
  </fruit>
</fruits>
```

XML Structure 1.2 Simple XML Documents to Illustrate NOT Query .

The availability of the appropriate order of the keywords of the phrase can not be determined from the XML inverted file structures in [2][6][12]. For this reason the above inverted file structure and the query processing algorithms in [10] cannot be applied to evaluate phrase queries in XML documents. Moreover, how to process NOT queries has not been proposed yet.

PIX (Phrase Matching in XML) [12] extends the W3C recommended XQuery standard [13][14][15][16] to provide a phrase matching technique. However, PIX allows searching a phrase only in a single XML document. This search also requires the prior knowledge of the internal structure of the XML document being searched to specify the list of the tags that can be ignored during the search. PIX also fails to search a phrase over a collection of XML documents.

New solutions are required to process both phrase queries with full or partial phrase matching and NOT queries in a collection of XML documents, without the knowledge of the internal structures of the documents.

In this thesis, we introduce new methods to evaluate both phrase and NOT queries in a collection of XML documents using modified inverted file structures and algorithms. We develop a prototype query processor which is capable of creating modified inverted files and evaluating all types of queries including phrase and NOT queries. Our experimental results using this prototype query processor show the effectiveness of our proposed query evaluation methods

1.2 Contributions

The followings are the contributions of this thesis:

1. We proposed the required modification to enable Phrase Query in XML documents
2. We introduced the solution for Phrase Query over a collection of XML documents.
3. We introduced the solution for NOT query over a collection of XML documents
4. We compared the performance of our proposed solutions with the conjunction and disjunction operations and noticed enhanced performance for Phrase and NOT queries over XML document collections

1.3 Organization of Thesis

The rest of the thesis is organized as follows. In Chapter 2, the overviews of the syntax and the structure of an XML document are presented. We have also discussed the important features of XML and their applications along with few basic query languages for XML.

Chapter 3 describes related research works those are done on defining the structure and compression techniques of XML inverted files and retrieval and ranking of XML documents. We also describe the limitations of processing phrase and NOT queries over a collection of XML documents in Chapter 3.

In Chapter 4, we propose our solutions to overcome these limitations. We present our proposed modification in XML Inverted File Structure and the new query processing algorithms in Chapter 4.

We discuss our experiments and results in Chapter 5. We compare the performance of our algorithms with the previously available conjunction and disjunction operations and show how our proposed solutions perform better in Phrase and NOT queries over a collection of XML documents.

Finally we conclude our thesis with shedding some lights on our future works in Chapter 6.

Chapter 2

XML Basics

The structure of an XML document is similar to that of a HTML document. Both HTML and XML documents consist of tags or markups. A tag represents an element in a HTML or an XML document. The tags in a HTML document describe how the document content should be displayed on browsers. Tags normally have attributes or properties and these attributes or properties of the tags provide more information on how to lay-out the data of a HTML document. However, instead of providing any presentation information, XML document markups or tags are used to group the data of a document and the attributes or properties describe additional properties of the data in a group. The set of tags or markups in a HTML document is pre-determined but an application itself determines the markups and the functional meanings of the tags or markups for an XML document which is used by the application. In this chapter, we briefly discuss the structure of an XML document and some special aspects of the XML standard.

2.1 Structure of a HTML Document

In a HTML document, an element named *html* wraps all other elements. An *html* element contains two direct child elements - first a *head* and then a *body*. The *head* element contains HTTP related information and the *body* contains all the data to be displayed. Figure 2.1 is an example of a simple HTML document. The output of this document is shown in Figure 2.2.

An element starts with an opening tag and ends with a closing tag. An opening tag is specified as the tag name inside angle brackets (e.g. `<html>` at line 1 in Figure 2.1) and a closing tag is specified as `</TagName>` (e.g. `</html>` at line 15). But the *Line Break* element ('`<br />`') at line 9 is expressed using a short form and does not have a closing

tag; instead it ends with the '/>' characters at the end of the opening tag instead of just the '>' character. This is because the *Line Break* element is defined to be empty (i.e. it can not contain any other element or text). It is also correct to write a *Line Break* element as '`<br></br>`' but an element which can contain other elements or text cannot be expressed using the short form (e.g. writing `<div />` is wrong). The title of the document defined in the *head* element can be seen in the title bar of the browser (in Figure 2.2).

```
1  <html>
2      <head>
3          <title>Intoducing HTML</title>
4      </head>
5      <body>
6          <h3>Hello HTML</h3>
7          <div style="color:Green; padding:2px">
8              This text will be displayed in Green color.
9              <br />
10             <span style='color:red'>
11                 This is a line after a line break.
12             </span>
13         </div>
14     </body>
15 </html>
16
```

Figure 2.1 A Simple HTML Document

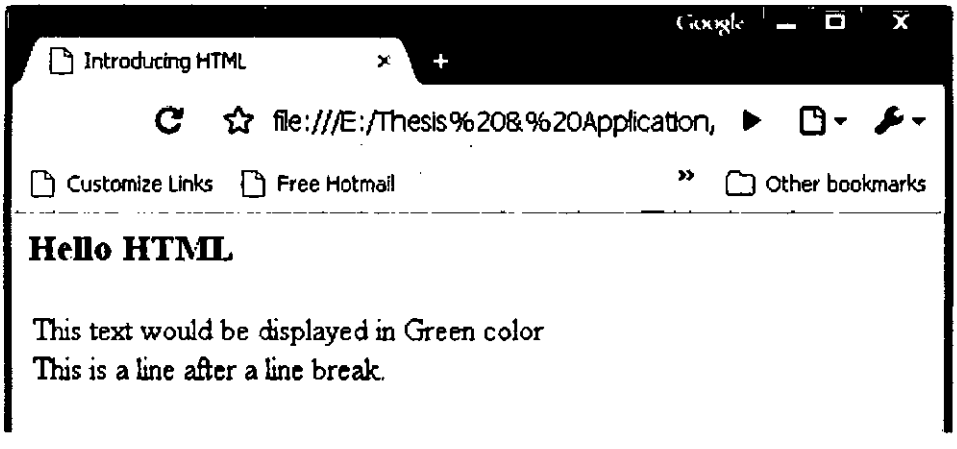

Figure 2.2 Output of the HTML Document in Figure 2.1

The body in Figure 2.1 declares a level 3 header at line 6. The *div* element at line 7 has a property named *style* and the value of this property (`'color:Green; padding:2px'`) says that the color of the text inside the *div* element would be green. At line 10 there is a *span* element which overwrites the text color defined in its parent *div* element by redefining the *style* property.

The specifications of any tag in HTML are given in the HTML DTD (Document Type Definition) that defines the structure and constraints of the documents which reference the DTD. A DTD describes which attributes can be associated with the tag, what can be the values of these attributes or what other tags can be nested within a given tag etc. But there is no pre-specified DTD for XML. The DTD for an XML document is determined by the user of the document. In an XML document it is also possible not to use any DTD and use an arbitrary structure. The basic structure of an XML document is pretty simple and very much similar to the above HTML file.

2.2 XML Document Structure and Syntax

An XML document mainly consists of:

- The XML declaration at the beginning of the document
- One Root element containing all other elements of the document
- Element(s)
- Attributes of Elements (name - value pairs)
- Comments and Text Data

To explain the structure of an XML document we can use a simple XML document:

```
<?xml version="1.0" encoding="utf-8" ?>
<customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
      <error statusCode="403" redirect="NoAccess.htm" />
      <error statusCode="404" redirect="FileNotFound.htm" />
</customErrors>
```

XML Structure 2.1 A Simple XML Document

2.2.1 XML Declaration

The first line is the XML declaration of the document:

```
<?xml version="1.0" encoding="utf-8" ?>
```

The declaration must appear at the beginning of an XML document. Here the declaration states that the version of the XML standard being used is 1.0 and the document is using utf-8 encoding. The encoding specified in the XML declaration has to match the encoding of the XML document.

2.2.2 Root Element

The root element comes after the XML declaration (`<customErrors>` element). XML documents must contain only one root element which contains all other elements. A root element is also called a document node. All features of the root element are same as those of any other element in the XML document.

2.2.3 Element

An XML element spans from its opening tag to its closing tag and can contain other elements, simple text or the mixture of both. XML element names can contain letters, numbers, and other characters but names cannot start with a number or punctuation character, the letters xml (or any string which is equal to *xml*, ignoring case) and can not contain spaces. There is no reserved word, so any name can be used. Since XML is a case sensitive language, opening and closing tags must maintain same case in each character. For example, the tag or element `<correct>` is different from the tag or element `<Correct>`.

The root element in the above sample XML document has 2 child elements (`<error>` elements). Similar to HTML, since the content (content refers to whatever appears between opening and closing tags - not attributes) of the `<error>` element is empty in the above example, it does not have a closing tags and the declarations end with the '`/>`' sequence.

An element is called a mixed node if it contains text and other nested element(s). An example of a mixed node (the `<book>` node in the example) is:

```
<book>
  <author>Kipling</author> wrote <name>Jungle Book</name> in 1894
</book>
```

The elements in an XML document must be properly nested. The following XML data is not proper because its elements overlap:

```
<nesting>This is <error>wrong nesting</nesting></error>
```

The following is an example of correct nesting of elements:

```
<nesting>This is <correct>correct nesting</correct></nesting>
```

2.2.4 Attribute

An element in an XML document can have zero or more attributes. An attribute must have a value (possibly empty string) and the value must be properly enclosed with either single or double quotes. In the sample XML file of XML Structure 2.1 the root element has an attribute named `mode` with the value `RemoteOnly`.

In an XML document same information can be encoded as an element or as an attribute. The following two XML documents bear the same information:

`<?xml version="1.0" ?>`
`<customErrors mode="RemoteOnly">`
`  <error statusCode="403"`
`  redirect="NoAccess.htm" />`
`</customErrors>`
	`<?xml version="1.0" ?>`
	`<customErrors>`
	`  <mode>RemoteOnly</mode>`
	`  <error>`
	`  <statusCode>403</statusCode>`
	`  <redirect>`
	`  NoAccess.htm</redirect>`
	`  </error>`
	`</customErrors>`

Table 2.1 Encoding Information Using Attributes and Elements in XML

Using elements rather than attributes has some benefits - an attribute can not contain multiple values whereas multiple values can be easily encoded using multiple elements of the same name. Attributes can not be easily expanded for future changes or extension and they do not contain any tree structure.

2.2.5 Comments and Text Data

The syntax of writing a comment in XML is similar to the syntax of a comment in HTML. A comment in XML is written as:

```
<!-- This is a comment in XML -->
```

When an XML parser parses an element it also parses the text (if present) between the opening and closing tags of the element. The text data that is parsed by the XML parser is called PCDATA (*Parsed Character DATA*). It is possible to mark some text as CDATA (*Character DATA*) which will not be parsed by an XML parser. A parser ignores everything inside a CDATA section which starts with "`<![CDATA[`" and ends with "`]]>`". Special characters like '<' or '&' can be used in a CDATA section without using the corresponding entity references - but entity references must be used to use these characters in a PCDATA section. A common application of CDATA in XHTML (Extensible HTML) is to wrap JavaScript code which contains a lot of '<', '&' or other special characters. To avoid errors script code should be defined as CDATA. An example of doing this is:

```
<script type="text/javascript">
  <![CDATA[
    function MaxPositive(a,b) {
      if (a <= b) return (b < 0) ? 1 : b;
      else if(b < a && a > 0) return a;
      else return 1;
    } ]]>
</script>
```

XML Structure 2.2 Using CDATA in XML Documents

2.2.6 Special Characters

Some characters have special meanings in XML. These are listed in Table 2.2. These characters can be used directly in a CDATA section but entity references (a group of characters used as a substitute for a single specific character) must be used to use them in any other part of the XML document. For example, placing the character '<' inside an XML element will generate an error because it represents the start of a new element - so the '<' character must be replaced with its entity reference.

Erroneous:

```
<error>
  Less than character can not be used as <-this is an error
</error>
```

Correct:

```
<ok>Less than character should be used as &lt; as is used in HTML</ok>
```

In XML there are five predefined special characters:

Character	Meaning	Entity Reference	Numeric Reference
<	less than	<	<
>	greater than	>	>
&	ampersand	&	&
'	apostrophe	'	'
"	Quotation mark	"	"

Table 2.2 Special Characters in XML

2.2.7 White Space and New Line in XML

Contrary to HTML, XML does not truncate multiple white space characters to a single space (ASCII 0x20) character. This is because HTML is used for presentation and XML

is used for data storage and transmission. XML uses the LF character (UNIX style) to store a New Line.

2.3 Tree Structure of an XML Document

XML documents form a tree structure which starts at the root element and branches to the leaf nodes (nodes which contain text or nothing). To describe the relationships between elements, three terms are used: *parent*, *child*, and *sibling*. A Parent element has children and the children on the same level are called siblings. Consider the following XML data:

```
<?xml version="1.0" encoding="utf-8" ?>
<catalog>
  <book>
    <author name="Rudyard Kipling" />
    <title>The Jungle Book</title>
    <year>1894</year>
  </book>
  ... ...
  ... ...
</catalog>
```

XML Structure 2.3 Sample XML Document

In the above XML data `<author>`, `<title>` and `<year>` are siblings. They are the children of `<book>` and `<catalog>` is the parent of `<book>`. The tree structure of the XML data in XML Structure 2.3 can be presented as:

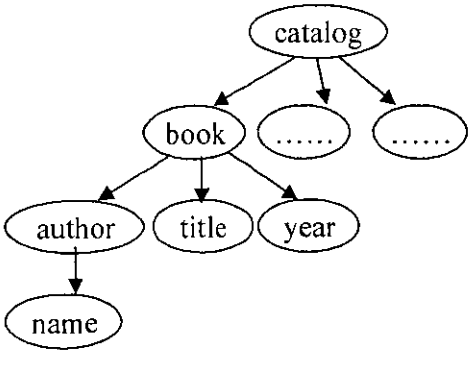

Figure 2.3 Tree Structure for the XML Document in XML Structure 2.3

In the above tree structure, a node represents an element or an attribute. This is why the term *node* is used in the literature to refer to elements or attributes. With the knowledge of the tree structure of an XML document, any node can be identified with the path of the node in the tree. All paths start with the root element. For example, in the above tree structure (for the XML data in XML Structure 2:3) the paths of the *name* and *year* nodes are 'catalog/book/author/name' and 'catalog/book/year' respectively.

2.4 Extensibility Feature of XML

XML elements can be extended to carry more information without causing any problem to an application. Suppose an application is using the following XML document:

```
<?xml version="1.0" encoding="utf-8" ?>
<customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
      <error statusCode="403" redirect="NoAccess.htm" />
      <error statusCode="404" redirect="FileNotFound.htm" />
</customErrors>
```

Later it may be required to add more `<error>` elements to the XML document for some other application:

```
<?xml version="1.0" encoding="utf-8" ?>
<customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
      <error statusCode="403" redirect="NoAccess.htm" />
      <error statusCode="404" redirect="FileNotFound.htm" />
      <error statusCode="500" redirect="ServerError.htm" />
</customErrors>
```

The new file will not create any problem for the first application because the first application still gets what is needed in the document.

2.5 Namespaces in XML

XML namespaces provide a simple method for uniquely naming elements and attributes in an XML document by associating them with namespaces identified by URI references. Namespaces are used to resolve element name conflicts. In the following XML there are two model elements but they are not same because of the difference between their structures:

```
<?xml version="1.0"?>
<models>
  <model>
    <name>F-91W</name>
    <cost currency="BDT">280</cost>
    <manufacturer>Casio</manufacturer>
    <country>Malaysia</country>
  </model>
  <model>
    <name>LifeBook E8420</name>
    <manufacturer>Fujitsu</manufacturer>
    <cpu>Core 2 Duo 2.2 GHz</cpu>
    <memory>2 GB</memory>
  </model>
</models>
```

XML Structure 2.4 XML Document with Conflicting Element Names

In the above example, the *model* elements are different in structure but they use the same element name. Such name conflicts can be solved using XML namespaces. A Namespace in XML is defined by the *xmlns* attribute in the opening tag of an element. Namespace declaration uses the syntax `xmlns:prefix="URI"` where *prefix* is the reference to the namespace and is used in conjunction with the name of an element or attribute to give a unique name to that element or attribute. Introducing namespaces to the above document produces a document which is similar to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<models>
  <watch:model xmlns:watch="http://www.mythesis.net/xml/ns/watch/">
    <watch:name>F-91W</watch:name>
    <watch:manufacturer>Casio</watch:manufacturer>
  </watch:model>
  <laptop:model xmlns:laptop="http://www.mythesis.net/xml/ns/laptop/">
    <laptop:name>LifeBook E8420</laptop:name>
    <laptop:manufacturer>Fujitsu</laptop:manufacturer>
  </laptop:model>
</models>
```

XML Structure 2.5 Using Namespaces to Resolve Name Conflicts

Namespaces can be declared either in the elements where they are used or in the root element. We can define a default namespace for an element and this saves us from using prefixes in all of its child elements. It uses the same namespace declaration syntax except the *prefix* (`xmlns="URI"`). If we shift the namespace declarations to the root element the above XML document turns into:

```
<?xml version="1.0" encoding="utf-8"?>
<models xmlns:watch="http://www.mythesis.net/xml/ns/watch/"
        xmlns:laptop="http://www.mythesis.net/xml/ns/laptop/">
  <watch:model>
    <watch:name>F-91W</watch:name>
    <watch:manufacturer>Casio</watch:manufacturer>
  </watch:model>
  <laptop:model>
    <laptop:name>LifeBook E8420</laptop:name>
    <laptop:manufacturer>Fujitsu</laptop:manufacturer>
    <laptop:cpu>Core 2 Duo 2.2 GHz</laptop:cpu>
  </laptop:model>
</models>
```

XML Structure 2.6 Declaring Namespaces at the Root Element

2.6 Well Formed and Valid XML

An XML document may reference a DTD (*Document Type Definition*) which defines the structure and constraints of the XML document. In an XML document a reference to an external DTD file is made using the DOCTYPE declaration. An example of declaring a DTD is (here "URIreference" is the URI of the external DTD file):

```
<!DOCTYPE rootElement SYSTEM "URIreference">
```

An XML document is *well formed* if it conforms to the XML syntax rules. A document which does not conform to the XML syntax rules (that is, not well-formed) is not XML. A *conforming parser* is disallowed from processing a document which is not *well formed.*

A *well formed* XML document is called *valid* if the document references a DTD and conforms to the syntactic constraints expressed in that DTD. A *validating parser* is required to report errors which arise due to violation of constraints expressed in the corresponding DTD.

2.7 Displaying XML

Displaying data is not the purpose of XML. XML documents do not normally carry information about how to display the data. Since there is no information about how to display the data, a browser will decide how it would display an XML document if the document is opened in the browser. For viewing on browsers CSS (*Cascading Style Sheets*) and XSLT (*Extensible Stylesheet Language Transformations*) can be used. CSS adds display information to an XML document. But XSLT is recommended by W3C (*World Wide Web Consortium*) for displaying XML data in browsers. XSLT is used to transform XML into HTML or XHTML. Transformation with XSLT can be done on the client side or on the server side. While transforming XML with XSLT in the browser, different browsers may produce different results, so the safer approach is do the transformation on the server side.

2.8 Languages for Querying XML Documents

Early XML query languages have the motivation to query contents into XML documents using a query language of pure XML structure. This led to Structural XML query languages like XPath [17] and XQuery [13] where the structural information of an XML document is required to query into it. XPath and XQuery work on a single document. XPath and XQuery are now W3C (*World Wide Web Consortium*) standards.

XPath is a simple language to find information in an XML document. XPath uses a given path expression to return either a node-set, or a string, or a true or false, or a number. The path expression contains a path which is used to navigate into the XML document, and can contain operators, functions and predicates. A predicate is a condition which is used to find specific node(s) that satisfies the predicate. XPath includes a set of standard operations on different data types. The following is an XPath example:

```
<?xml version="1.0"?>
<bookstore>
  <book category="CHILDREN">
    <title lang="en_us">
      Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title lang="en">
      Learning XML</title>
    <author>Ray et. al.</author>
    <price>39.95</price>
  </book>
</bookstore>
```

XPath Query:

/bookstore/book[last()-1]

Result:

```
<book category="CHILDREN">
  <title lang="en_us">
    Harry Potter</title>
  <author>
    J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
```

Table 2.3 XPath Example

In the above example, the XPath query is evaluated against the XML document on the left side. The path expression in the query is "/bookstore/book" and this will evaluate to all book elements under the bookstore element and the predicate '[last()-1]' selects the next to last book node.

XQuery is similar to XPath but provides more power and flexibility. XQuery supports conditional expressions, element constructors, a special *FOR, LET, WHERE, ORDER BY, RETURN (FLWOR)* expressions. XQuery shares some similar concepts of Structured Query Language (SQL) of Database System. Like SQL join operation on two database tables XQuery has join operation on two XML documents. Like SQL, XQuery also provides filtering nodes with the *WHERE* clause and sorting with the *ORDER BY* clause. Function calls and user-defined functions are also allowed in XQuery as it is allowed in SQL. A query in the XQuery language contains one or more query expressions. The following is an XQuery example:

Document Name: A.xml

```
<?xml version="1.0"?>
<bookstore>
  <book category="COOKING">
    <title lang="en">
      Everyday Italian</title>
    <author>Giada et. al.</author>
    <price>30.00</price>
  </book>
  <book category="WEB">
    <title lang="en">
      XQuery Kick Start
    </title>
    <author>James et. al.</author>
    <price>49.99</price>
  </book>
  <book category="WEB">
    <title lang="en">
```

Query:

FLWOR (For, Let, Where, Order by, Return) Expression

```
for $x in doc("A.xml")/bookstore/book
where $x/price>30
order by $x/title
return  if ($x/year <2004)
        then
<before>{data($x/title)}</before>
        else <after>{data($x/title)}</after>
```

Result:

```
  <before>
```

```
    Learning XML
  </title>
  <author> Erik et. al.</author>
  <price>39.95</price>
 </book>
</bookstore>
```

```
  <title lang="en">
    Learning XML</title>
</before>
<after>
  <title lang="en">
    XQuery Kick Start</title>
</after>
```

Table 2.4 XQuery Example

In the above example, all *book* elements under the *bookstore* element are selected by the 'for' clause. The selected *book* elements with a *price* element with a value greater than 30 are filtered by the 'where' clause. The filtered *book* elements are sorted by the value of the *title* element as indicated in the 'order by' clause. The 'return' clause specifies what should be returned. Here it returns the title elements with surrounding markups according to the result of the 'if-else' ladder.

2.9 Applications of XML

XML simplifies data storage and sharing. In the real life, different database systems store data in incompatible formats. Data sharing among these systems is difficult because of these incompatible formats. XML fills the gap of sharing data between such systems. Different applications or systems can share data through XML. Changes in the data centric platforms require large amount of data to be converted - in this process incompatible data are always lost. But since XML uses simple text format to store data, the data can be easily adapted to the change of the application or the system. Being independent of software and hardware, XML makes data more available because different applications can access the data.

As the name includes XML is extensible and it is a generic specification to create data formats. XML is used to create new languages and define protocol messages for the purpose of exchanging or sharing data. One example of extending XML to create a data format is news feeds on web, which are very popular these days - RSS (*Really Simple*

Syndication or sometimes translated as *Rich Site Summary*) is a family of formats used to publish frequently updated news. XML is used to specify RSS formats which have evolved since March 1999.

WSDL or *Web Services Description Language* is an XML based language and it is used to describe web services. WSDL is used in combination with SOAP (*Simple Object Access Protocol*) and an XML Schema. SOAP is a protocol that allows applications to access web services over HTTP. The XML schema is used to describe special data types embedded in the WSDL file. A client program that connects to a web server reads the WSDL to determine what functions are available on the server. The client can then use SOAP to call a function listed in the WSDL.

XHTML (eXtensible HTML) versions are recent improvements of HTML document types. XHTML conforms to XML syntax but has the same functions as HTML. Where HTML requires a relatively complex, lenient and custom parser, standard XML tools can perform the automated processing of a true XHTML document.

WAP (*Wireless Application Protocol*), an open international standard for application layer network communications in a wireless communication environment, is used to enable access to the mobile web from a mobile handheld device. WAP is used to access WML (*Wireless Markup Language*) files by the wireless devices. WML is developed based on XML.

SMIL (*Synchronized Multimedia Integration Language*), a W3C recommended XML markup language, is used to describe multimedia presentations. SMIL is written in XML. It defines markup for layout, timing, visual transitions, animations, and media embedding, among other things and allows the presentation of media items such as audio, video, images and text as well as files from multiple web servers and links to other presentations in SMIL.

XML's software and hardware independent way of storing data in plain text format allows different incompatible systems to share data without the need of passing them through many layers of conversion. The XML standard provides a basic syntax to share information between diverse types of computers, applications, and organizations. The consequence is ease in expanding or upgrading to new systems or applications without losing any data.

2.10 Summary

In this chapter we have discussed the structures and rules of the XML standard. We have shown how an XML document can be presented as a tree structure and how the extensibility feature makes XML a great tool for storage and sharing of information. We have also discussed structured query languages like XQuery and XPath in querying XML documents. Since XML promises many such benefits towards storing data and sharing information, many researchers have devoted their efforts to develop techniques for searching in XML documents. In the next chapter, we discuss some prominent search techniques and their features.

Chapter 3

Literature Review

Searching techniques for XML documents are different from those for plain text due to the structure of the XML standard, where tags bear special importance regarding document content. The information retrieval techniques in XML evolved from the information retrieval techniques used for HTML or plain text files. In this chapter, we first discuss techniques for information retrieval in text or HTML files in brief. Then we discuss the adaptations of those techniques for querying into XML documents- which includes structures and compression techniques of indexes for searching XML documents, desirable features of XML query languages for information retrieval, current state of text and phrase queries in XML and weighting or ranking of XML documents during search. We finally summarize the required changes for more flexible searching methods over a collection of XML documents.

3.1 HTML Inverted File Structure and Ranking

During processing a query over a collection of text or HTML documents the, following principles [3] are taken into account:

1. Less weight is given to:
    a. Query terms which appear in many documents
    b. Documents which contain many terms
2. More weight is given to terms which appear many times in a document

Inverted files are used for high performance text indexing [3][4][11][18]. An inverted file is associated with a vocabulary of query terms or keywords. In the simplest implementation an inverted file maps a keyword from the vocabulary to an inverted list (also called *postings list*) which is a list of the identification numbers of the documents

that contain the keyword. Every document in the collection is identified with a unique identifier; a simple approach is to use ordinal numbers. Zobel et al. [3][11] proposed an inverted file structure where the inverted list is a list of $<d, f_{d,t}>$ pairs where d is the document which contain the keyword, and $f_{d,t}$ is the *within-document frequency* (frequency of the keyword t in the document d) of the keyword. For efficient query processing the inverted file also holds the value f_t (the number of documents that contain a term t) for a term t. So the complete inverted file structure is a list of the tuple: $(t, f_t, \{<d, f_{d,t}>\})$ where t is the term in the tuple, f_t is the count of documents containing the term t and $\{<d, f_{d,t}>\}$ is the inverted list of the term t. An example is presented below:

Term	Frequency	Inverted List
black	1	(1, 2)
coil	2	(1,3) (3,2)
inward	1	(3,1)
...	...	... ...
t	f_t	$\{<d, f_{d,t}>\}$

Table 3.1 Inverted File Structure for Text Database

Documents are ranked while searching. For any keyword there are two types of ranking. One is ranking in the query and the other is the ranking in a document. These two rankings are combined to compute the final ranking. Ranking in the query indicates the importance of a keyword among all query keywords. In the searching process the value f_t is crucial for ranking the query terms. Whereas $f_{d,t}$ allows computing the weight of a term in a document.

Ranking involves assigning a document a numeric score which indicates similarity of the document to the query, and the highest ranked documents are displayed as the answers. Vector Space Model is a prominent method for ranking- in Vector Space Model, documents and queries are both considered as vectors in a n_t dimensional space where n_t is the no. of keywords in the query. $\boldsymbol{Wq}$ is the query vector with components being $w_{q,t}$

values. Similarly W_d is the document vector with components being $w_{d,t}$ values. To measure similarity or relevance of a document to a query the cosine of the angle between the vectors W_q and W_d is computed using cosine formula:

$$\cos ine(q,d) = \frac{\sum_t (w_{q,t}.w_{d,t})}{\sqrt{\sum_t w_{q,t}^2}.\sqrt{\sum_t w_{d,t}^2}} \quad \textbf{(3.1)}$$

Here q is the query, d is the document, and $w_{x,t}$ is the weight (or importance) of a word t in document or query x. The answers to a query q are the r documents with the highest C_d = *cosine (q, d)* values. The cosine formula was deduced from scalar product of vectors in [19].

Let, θ be the angle between W_d and Wq. So *cosine* (θ) = *cosine (q, d)*. If $\vec{W}_x$ is a vector then its value is expressed as:

$$|\vec{W}_x| = \sqrt{\sum_t w_{x,t}^2} \quad \textbf{(3.2)}$$

The cosine formula in (3.1) can be deduced from the Vector Scalar Product:

$$\vec{W}_q.\vec{W}_d = |\vec{W}_q|.|\vec{W}_d|.\cos(\theta)$$

$$\Rightarrow \cos(\theta) = \frac{\vec{W}_q.\vec{W}_d}{|\vec{W}_q|.|\vec{W}_d|}$$

$$\Rightarrow \cos(\theta) = \cos ine(q,d) = \frac{\sum_t (w_{q,t}.w_{d,t})}{\sqrt{\sum_t w_{q,t}^2}.\sqrt{\sum_t w_{d,t}^2}}$$

3.2 XML Inverted File Structure

To process queries in XML documents it is required to encode the path information of the occurrences of a keyword in XML documents in the inverted file. The inverted list is the

description of the occurrences of a keyword in a collection. The path information is stored as part of the entries in the inverted lists - this method is called the *Paths in Inverted Lists (PIL)* [6] approach. An occurrence in the inverted list, therefore, records the document id, no. of occurrences of the keyword in the term and the different paths (structural information) of the occurrences.

If a *path* is simply a list of nodes starting from the root node to the node concerned then the path can not uniquely identify a node in the tree representation of the document. Consider the three #pcdata nodes with dotted borders in the following tree structure:

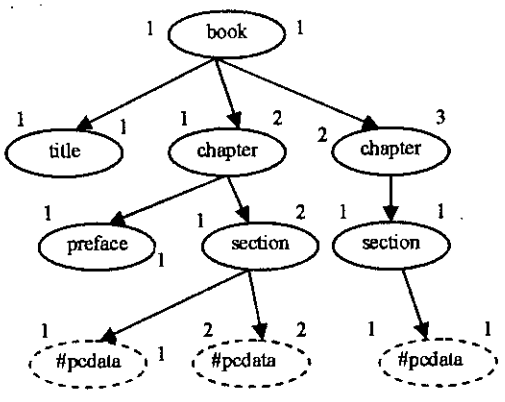

Figure 3.1 Uniquely Identifying Nodes in XML Document Tree Structure

The sequences of nodes in the paths for all of these #pcdata nodes are 'book/chapter/section/#pcdata' - to uniquely identify these nodes two additional indexes are used: one is *element index* and the other is *sequence index*. *Sequence index* is the index of a node among its siblings (sibling nodes are children of the same parent node). Element index is the index of a node among siblings with the same name. In the above example the element indexes are shown on the left side of the corresponding node and the sequence indexes are shown on the right side. When each node is represented as '*node-name [element index, sequence index]*' the path of the node with blue dashed border in the above example becomes 'book [1, 1]/chapter [1, 2]/section [1, 2]/#pcdata [2, 2]'.

A path in an inverted list of an XML inverted file is specified as the length of the path followed by three lists which are the element names, the element indices, and the

sequence indices of the path. Using regular expressions we can describe the abstract data structure as follows:

invlist -> docentry+
docentry -> docid, num_occ, occurrence+
occurrence -> path_length, element+, element_index+, sequence_index+, weight?

Table 3.2 Regular Expressions for Inverted Lists for XML

Weights can be ignored in describing the XML inverted file structure and compression because they add constant factors in terms of space and time. To explain XML inverted file structure we consider the following XML document:

```
<?xml version="1.0" encoding="utf-8" ?>
<book>
  <title>Unicode Explained</title>
  <author name="Jukka Korpela" />
  <chapter name="Characters as Data">
    <section>
      <sub-section>Sub Section A</sub-section>
      <sub-section>Sub Section B</sub-section>
    </section>
  </chapter>
  <chapter name="Writing Characters">
    <section>Writing Short Characters</section>
    <section>Writing Long Characters</section>
  </chapter>
</book>
```

XML Structure 3.1 Sample XML document

The tree structure of the XML document in XML Structure 3.1 can be presented as the following figure:

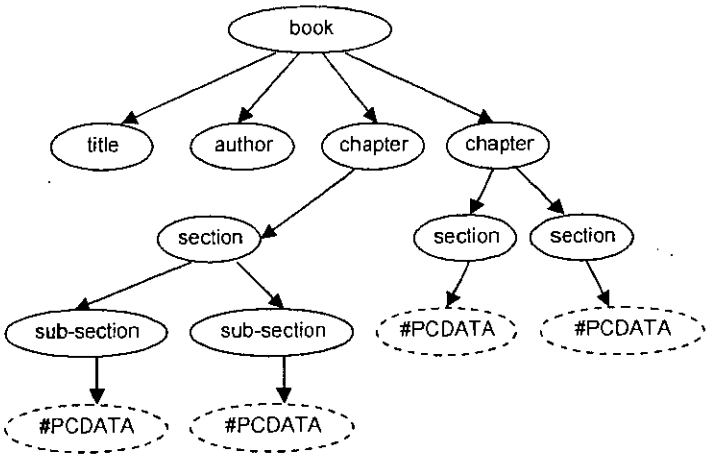

Figure 3.2 Tree Structure for **XML Structure 3.1**

As a simple example, we can consider an inverted list of a keyword occurring in documents numbered 7 and 16. Let, both the documents share the same structure as displayed in Figure 3.2 and the keyword occurs twice in each of the documents: within each of the two #PCDATA nodes descendant from sub-section node in document numbered 7 and within both of the two #PCDATA nodes descendant from section nodes in document numbered 16. The information pertaining to the keyword would look like the following in the inverted file (diamond brackets are used for grouping and illustration):

```
<keyword 2
<7 2
<5 <book chapter section sub-section #PCDATA> <1 1 1 1 1> <1 3 1 1 1>>
<5 <book chapter section sub-section #PCDATA> <1 1 1 2 1> <1 3 1 2 1>>
>
<16 2
<4 <book chapter section #PCDATA> <1 2 1 1> <1 4 1 1>>
<4 <book chapter section #PCDATA> <1 2 2 1> <1 4 2 1>>
>
>
```

Table 3.3 A Sample Inverted List

There are alternative indexing structures for XML documents collection. One alternative is to use bitmap indexing for XML documents. A bitmap index uses array of bits (bitmaps). BitCube [8][9] is a bitmap indexing approach, where term occurrences are associated with corresponding paths and document identifiers. The BitCube is a three dimensional data structure where the dimensions are the set of documents, the set of paths and the set of terms. Document Identifier is the first dimension of the BitCube. An entry into this dimension is a pointer to a two-dimensional structure defined by the set of paths and the set of terms. This two-dimensional structure contains an entry for a path if the path is present in the document which is pointing to this 2D structure. An entry in the path is a pointer to an array of occurrences of terms. The values of the elements of this one dimensional array are either 0 or 1. The value 1 indicates that the term is present and 0 indicates that the term is absent in the path and the corresponding document. The path in BitCube only considers the names of the nodes in the paths. But no means of inclusion of sequence indexes or weights are defined in BitCube. So the paths in the BitCube can not uniquely identify the occurrences of a term. Moreover, where a smallest element in the BitCube tells about the presence/absence of a term in a path in a document, the information of how many times the term appears in that path in the document is not encoded.

3.3 XML Inverted File Compression

In the PIL approach path information is encoded as part of the inverted list entries but a straight forward implementation of this requires a very large amount of disk space because the index size would grow larger than the original files. Redundancy is introduced in the inverted list entries because some paths share common prefix- hence the common prefixes of subsequent path entries can be eliminated to compress the inverted file. In this process another counter is added which tells about the length of the common prefix or the number of common path steps with the previous path entry. So the regular expression for *occurrence* is changed as follows:

Previous Regular Expression for *occurrence*:

```
occurrence -> path_length, element+, element_index+, sequence_index+,
weight?
```

New Regular Expression for *occurrence*:

```
occurrence -> path_length_diff, prefix_length, element+,
element_index+, sequence_index+, weight?
```

Here *path_length_diff* is the difference between the path length and the prefix length. The document numbers are compressed using run length encoding and sequence indices are encoded using the differences from the corresponding element indices. These steps convert the inverted list in Table 3.3 to the following:

```
<7 2
<5 0 <book chapter section sub-section #PCDATA> <1 1 1 1 1> <0 2 0 0 0>>
<2 3 <sub-section #PCDATA> <2 1> <0 0>>
>
<9 2
<3 1 <chapter section #PCDATA> <2 1 1> <2 0 0>>
<2 2 <section #PCDATA> <2 1> <0 0>>
>
```

Table 3.4 Semi-compressed Inverted List for Table 3.3

To further compress this data Fuhr et al. [6] proposes using universal codes [20] for the numbers and canonical Huffman codes [21] for the element names.

XML Structure Tree (XS Tree) [6] is an alternative to encoding compressed path information in inverted files. The XS tree is an additional data structure which describes the structure of an XML document. Using XS trees, the path occurrence entry in the inverted list is replaced with a path handle or pointer which is a single number pointing to a position in the corresponding XS tree. The corresponding XS Tree can be found from the document ID in the inverted file. A linear representation of the XS tree can be achieved using level numbers of the document nodes. Run length encoding is used to

compress the tree. The XS tree for the document tree structure in Figure 3.2 can be expressed as follows (indenting is used for illustration purpose):

```
<0 book>
    <1 title>
        <2 #PCDATA>
    <1 author>
        <2 @name>
    <1 chapter>
        <2 section>
            <3 sub-section>
                <4 #PCDATA>
            <3 sub-section>
                <4 #PCDATA>
    <1 chapter>
        <2 section>
            <3 #PCDATA>
        <2 section>
            <3 #PCDATA>
```

Table 3.5 XS Tree Representation for Figure 3.2

3.4 Processing Text Search Queries in XML

For structured documents processing a query should not require knowledge of the structure of the document or the possible locations of the result - most relevant results should be returned irrespective of the underlying structures of the XML documents. Determining the most specific elements for conjunctive queries has been explored in [22][23]. In a conjunctive query, the keywords in the query are related with the AND relationship - that is, if the tree rooted at an element contains all the keywords of the query then the element is a specific result for such a query. Processing queries with disjunction (OR) relations have been discussed in [10]. For a disjunction query, a node is a result if one or more terms of the disjunction relationships are descendants from the node.

```
<?xml version="1.0"?>
  <mobile>
    <models brand="nokia">
      <model>Black Berry</model>
      <model>N Series</model>
    </models>
    <models brand="apple">
      <model>IPhone</model>
    </models>
  </mobile>
      First Document
```

```
<?xml version="1.0"?>
  <mobile>
    <models brand="nokia">
      <model>Black Berry</model>
      <model>N Series</model>
    </models>
    <models brand="apple" />
  </mobile>
      Second document
```

Table 3.6 Conjunctive and Disjunctive Queries

For the documents in Table 3.6, processing a text search query for "apple and iPhone" would return the first document because both the keywords are present in the first document but the term 'iPhone' is missing in the second one. But the query "apple or iPhone" would return both documents because the second document contains at least 1 term from the disjunction relationship

Conjunctive and disjunctive searches are examples of value oriented searches in XML documents but queries can combine values and structures. If structure of XML documents is specified (e.g. parent child or preceding-following relationships) then these structures must be taken into account and only results which satisfy these structures would have to be accepted.

3.5 Query Processing and Query Plan Tree

A Query Plan Tree [10] is a method for expressing text search queries in XML. In a query plan tree the keywords are placed in the leaves and each internal node corresponds to an operation that is applied on a keyword or a group of keywords which are children of the node. Each operator node receives the list of all matches from nodes which are its

children and produces a result which is passed similarly to its parent. The final output of the root operator node (corresponding to the whole query) is a set of nodes which, in effect, is the result of the query. A Query Plan Tree helps in understanding the processing of a query. The following is an example of a Query Plan Tree:

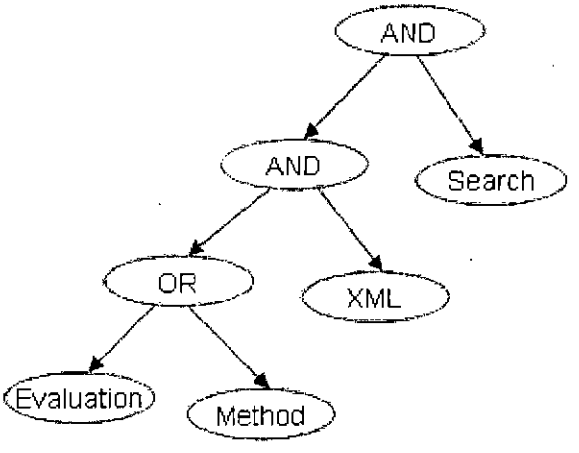

Figure 3.3 Query Plan Tree

The above Query Plan Tree represents the processing of a query which searches for XML documents which must contain the keywords 'Search' and 'XML' and either 'Evaluation' or 'Method'- so it is searching for the text 'XML Search Method or Evaluation'.

3.6 PIX - Phrase Matching in XML

Search techniques which require strict contiguity or close proximity of keywords in a text database cannot be applied directly to XML because text remains interleaved with arbitrary markups. As a phrase matching technique for XML documents, PIX [12] requires specification of both the markups to be ignored and the arbitrary tags which would be used as the search context. PIX supports phrase matching with strict contiguity or within *k*-proximity which means that an XML document where two successive keywords of the phrase are separated by no more than *k-1* words is accepted. The PIX implementation is a user defined XQuery function. In PIX one inverted index is built for every tag and word in the input XML document. This is an offline process. A (start, end) interval is assigned to each element and textual word of an input document. Start and end

values are the within-document word positions of a word considering all tags as distinct words. For a particular occurrence of a word the start and end positions are same. In PIX, the list of (start, end) intervals for each term are sorted by start position and can be accessed sequentially. An example of such an index structure is given below:

```
<book(1, 15)>
  <title(2, 5)>Unicode(3, 3) Explained(4, 4)</title>
  <author(6, 9)>Jukka(7, 7) Korpela(8, 8)</author>
  <description(10, 14)>
    author(11, 11) explained(12, 12) Unicode(13, 13)</description>
</book>
```

Table 3.7 Assigning (start, end) to Elements and Textual Words in PIX

The index for the above XML fragment is as follows:

book	Title	author	Unicode	explained	jukka	korpela	Description
(1, 15)	(2, 5)	(6, 9) (11, 11)	(3, 3) (13, 13)	(4, 4) (12, 12)	(7, 7)	(8, 8)	(10, 14)

Table 3.8 Index Structure for Table 3.7 in PIX

There are some limitations of PIX in phrase matching:

- To use PIX, it is necessity to know the granularity of the XML files being searched in advance.
- PIX is applicable to a single document, not over a collection of XML documents.
- Partial phrase matching is absent in PIX
- Path of the fragment of the document that contains the keywords are not readily available.

3.7 Ranking and Weighting in Querying XML documents

There are some desired properties of ranking functions [22] for searching into XML documents:

- Specificity of result
- Proximity of keywords
- Awareness of hyperlinks

Specificity of result means how specific the result is to the searched keywords. The closest outer node covering the searched keywords is the most specific one. As we move towards ancestors of this closest outer node, specification of result decreases. Proximity of keywords refers to the distance between keywords within a location. XRANK [22] considers the hyperlinked structure of XML documents. In this process, awareness of hyperlinks means that more referenced documents should be ranked higher.

For keyword search queries over XML documents XRANK employs *ElemRank(v)* which represents the weight in the inverted list and is called the objective importance of an XML element v with respect to a keyword or a group of keywords. This function is defined at the granularity of an element and utilizes the nested structure of XML. If a keyword search query is expressed as $Q = (k_1, k_2 \ldots\ldots k_n)$ where k_i represents the ith keyword and if the result of this query is $R = Result(Q)$ then for a result element v_l in R, the ranking of v_l with respect to one query keyword k_i, $r(v_l, k_i)$ is computed first before the overall rank, $rank(v_l, Q)$ is computed.

An XML document can be defined as a Directed Graph $G = (N, CE)$ where the set of nodes can be expressed as $N = NE \cup NV$, Here NE is the set of elements, and NV is the set of values, tag names and attribute names. CE is the set of containment edges. The edge (u, v) is an element of CE or (u, v) $\in$ CE iff v is the value of u or u is the parent of v. For a node v_l in R, v_l may not directly contain the key k_i. In that case there is a sequence of containment edges of the form $(v_1, v_2), (v_2, v_3) \ldots\ldots, (v_t, v_{t+1})$ in CE where v_{t+1}, which contains the keyword k_i, is an element that has some proper free text and that

text does not belong to any of its children or descendants. Such an element (e.g. v_{t+1} here) is called a textual node [2]. Given the above sequence of nodes,

$$r(v_1, k_i) = \text{ElemRank}(v_t) \times \text{decay}^{t-1} \quad \textbf{(3.3)}$$

To adjust the specificity of the result, the rank of a node v_1 with respect to a keyword k_i is scaled down as it moves higher in the XML document tree structure. In the above equation v_t is the parent element of the value node which directly contains the keyword k_i. When the result element v_1 directly contains the keyword k_i the rank of v_1 is its *ElemRank* without any scaling. Otherwise when the result element indirectly contains the keyword ($v_1 \neq v_t$), the rank of v_t is scaled down by a factor *decay* for each level – *decay* is a value within the range 0 to 1.

A textual node can also contain multiple occurrences of a keyword k_i. If the node v_1 contains m relevant occurrences of the keyword k_i the rank of each occurrence is computed first. Then the combined rank is determined as an aggregation function of these individual ranks. If the individual ranks of the occurrences are $r_1, r_2, \ldots \ldots, r_m$ then the combined rank is

$$r(v_1, k_i) = f(r_1, r_2, \ldots \ldots, r_m) \quad \textbf{(3.4)}$$

Here f is the aggregation function. Examples of such function are *sum* or *max*. The overall ranking of a result element v_1 for query $Q = (k_1, k_2, \ldots, k_n)$ is computed as follows:

$$R(v_1, Q) = \left(\sum_{1 \le i \le n} r(v_1, k_i) \right) \times p(v_1, k_1, k_2, \ldots\ldots\ldots, k_n) \quad \textbf{(3.5)}$$

The final ranking is computed as the sum of the ranks for each query keyword. The individual ranks are multiplied by a measure of *keyword proximity* which is denoted as $p(v_1, k_1, k_2, \ldots, k_n)$ in this formula. The keyword proximity function $p(v_1, k_1, k_2, \ldots, k_n)$ defines proximity of keywords in the node - due to the highly structured nature, distance

between XML query keywords is not an important factor. This proximity function, therefore, can be ignored or set to 1.

In some cases it may become necessary to assign different weights to different keywords, and then the individual keyword ranks are weighted accordingly. A typical scenario where this is desired is ranking query keywords. All keywords in a query are not equally important. As discussed in section 3.1, less weight is given to keywords which appear in many documents. For this reason $r(v_l, k_i)$ is weighted with the ranking of keyword k_i with respect to the query $r(k_i, Q)$ as follows:

$$R(v_1, Q) = \left(\sum_{1 \le i \le n} r(v_1, k_i) \times r(k_i, Q) \right) \times p(v_1, k_1, k_2, \ldots\ldots\ldots, k_n) \qquad \textbf{(3.6)}$$

In the above equation ranking of a node or element with respect to a keyword is again weighted with the ranking of the keyword in the query. After the weights or scores of the textual nodes are calculated these scores are propagated upwards. When a node is found that contains all the keywords this propagation is stopped and the path of the node is returned as a result element with its score.

3.8 Limitations

From the literature review, we have found that the available solution to Phrase Query needs to specify the list of nodes where the phrase has to be searched. This implies that the searching is dependent on document structure. Moreover, the available Phrase Query solution works only on a single document. We can improve Phrase Query in XML by adding the support for searching over a collection of XML documents. Phrase querying in XML would be more flexible to users if the process can be made independent of the awareness of the underlying document structures. Literature review also reveals that there's no available solution for NOT queries over a collection of XML documents. So we can implement this new feature of querying XML documents.

3.9 Summary

In this chapter, we have discussed inverted file structures of text and XML documents, the compression techniques of XML inverted files and information retrieval techniques for querying into text, HTML and XML documents. We have found that new solutions are required for document-structure unaware phrase querying over a collection of XML documents. Also, new solution is required for NOT queries over a collection of XML documents. In the next chapter, we present our proposed solutions which would enable us to execute Phrase queries and NOT queries over a collection of XML documents without requiring the knowledge of the underlying document structures.

Chapter 4

Proposed Solutions for Phrase and NOT Queries

In the previous chapters we have discussed existing research works on text search queries in XML documents. We also found that modification in inverted file structure and new processing methods are required for Phrase and NOT queries. In this chapter, we discuss our proposed modification in XML inverted file structure and new query processing algorithms for phrase and NOT queries.

4.1 Reasons for New Solution to Phrase Query in XML

We explain how phrase searching in XML documents is different from phrase searching in plain text files below. Suppose we are searching the phrase: "*package System.Configuration provides configuration settings access methods*".

```
<?xml version="1.0" encoding="utf-8"?>
<library>
  <packages>
    <package>
      The package &quot;System.Configuration&quot; provides
      configuration settings access methods
    </package>
  </packages>
</library>
```

Searching in the location `library/packages/package` in the above XML data is similar to searching phrase in plain text.

```
<?xml version="1.0" encoding="utf-8"?>
<library>
  <packages>
    <drawing>
      The package <package name="System.Drawing" /> provides
      <feature>Property</feature>,
      <feature type="public">method</feature>
        to draw in either <level>two</level> or <level>three</level>
        dimensional surface.
    </drawing>
  </packages>
</library>
```

However, if we need to search the phrase "*package System.Drawing provides Property, method to draw two or three dimensional surface*" in the above XML data then it can not be processed with phrase matching techniques of simple text. Phrase matching technique for simple text ignores the markup names and their attribute names in searching this phrase within the location `library/packages/package`. For this reason, it will not produce the correct result. We have proposed new techniques to search phrase in XML documents to overcome the above limitation.

4.2 New Processing Method for Partial Phrase Matching

Our proposed solution to phrase searching in XML documents includes following steps:

- Changing inverted file structure to add positions of keyword occurrences
- Determining the locations to which we should limit phrase search
- Transforming keyword positions for each location in the domain
- Determining the presence of the phrase using the above transformed positions

4.2.1 New Inverted File Structure for Phrase Query

Keywords appear at random positions within any textual node in an XML document. To determine the presence of the pattern of phrase keywords and to find out the relative distance between two keywords, we need to know the positions of the keywords within a

textual node. Position is an integer value which describes the ordinal position of a keyword within the data of a *textual node*. The following is an example of position information for textual nodes in an XML document:

```
<drawing>
  The packages
  <package name =
    "System.Drawing"/>
  <package name =
    "System.Drawing2D"/>
  provides features
  <feature>
    Method</feature>
  For Drawing
</drawing>
```

Term	Positional Path	Positions
The	/drawing[1]/#PCDATA[1]	1
Packages	/drawing[1]/ #PCDATA[1]	2
Provides	/drawing[1]/ #PCDATA[2]	3
System	/drawing[1]/package[1]/ @name[1]/#PCDATA[1]	1
System	/drawing[1]/package[2]/ @name[1]/#PCDATA[1]	1
Drawing	/drawing[1]/package[1]/ @name[1]/ #PCDATA[1]	2
Drawing2D	/drawing[1]/package[2]/ @name[1]/ #PCDATA[1]	2
Feature	/drawing[1]/feature[1]/	1

Table 4.1 Keyword Positions in *positional paths* (Node Names and Sequence Indexes)

We propose to use keyword positions in the inverted list, which will enable Phrase Query searching in XML documents. Therefore, the modified regular expression for inverted list in inverted files will be as follows:

```
invlist -> docentry+
docentry -> docid num_occ occurrence+
occurrence -> path_length element+ element_index+ sequence_index+
position+ weight?
```

Table 4.2 Modified Regular Expressions for Inverted Lists of XML Inverted Files

4.2.2 Determining Where to Search the Given Phrase

For a given keyword, we find a list of the structure described in Table 4.3, from the inverted file. For simplicity weight has been omitted in this structure. If the query consists of a single keyword, this data itself would be the search result. So the search result of a keyword is a list of (document id, path length, path node names, element indexes, sequence indexes, positions) tuples- each tuple is a result element.

<doc_no> <no_of_occurances>
<path_length$_1$> <path$_1$> <element_indexes$_1$> <sequence_indexes$_1$> <position$_1$>
… … … … … … … …
<path_length$_n$> <path$_n$> <element_indexes$_n$> <sequence_indexes$_n$> <position$_n$>

Table 4.3 Sample Inverted List for a Term in the Inverted File

To determine the presence of a phrase, our first task is to know the search context or the set of nodes to which we have to limit our search. Let this set of nodes be called the *domain* for searching the phrase. We propose following process to construct the domain of a search phrase:

Let, n is the no. of keywords in the phrase. We read results of these n keywords from the inverted file. If all keywords of the phrase must be present in the final result, then only the result of the keyword which has the least no. of occurrences should be considered to construct the domain. If p keywords are allowed to be absent in the search result, we consider the occurrences of $(p+1)$ keywords whose no. of occurrences are minimum. We use the Union of the results of these $(p+1)$ keywords. The list of locations thus found, is used to construct the domain.

Now consider the following XML data:

```
<?xml version="1.0" encoding="utf-8"?>
<tales>
      <author>Rudyard Kipling</author>
      <tale>
            In <year>1890</year> Kipling started his
            <theme>jungle</theme> story where he showed
            <role>Mowgli</role> in jungle in 1894.
      </tale>
</tales>
```

XML Structure 4.1 XML Document for Phrase Query

Let, the phrase to be searched is "*Mowgli in Jungle*" and all keywords must be present. Let, the document with the above XML data is the only document in our XML document collection. Here the keyword '*Mowgli*' appears only once within the location `tales[1]/tale[1]/role[1]`. But if we search in this location we are going to miss a result because other keywords appear outside this `<role>` node. The path, expressed as string, of the occurrences of other keywords do not contain the substring "`tales[1]/tale[1]/role[1]`". We propose to move one step up the XML tree and halt at `tales[1]/tale[1]`. This is the right place to start searching because the occurrences of other keywords contain this path as substring.

We propose to use the tree structures shown in Table 4.4 for the keywords from the inverted file (only the node names are shown for the ease of illustration). For a given keyword, it is a list of documents that contain the keyword and each document entry in the list points to the occurrences of the keyword in that document. Thus it will be easy to find if any occurrence of any keyword starts with a specific path or not.

For each location in the list selected for the construction of domain, we check if the tree structures of at least (n-p) keywords contain a subset of the location. If we find such a subset for (n-p) keywords we include the deepest common subset in the domain. If not found, we exclude this location from considering for the domain. We use the generic name 'domain element' to refer to each location included in the domain.

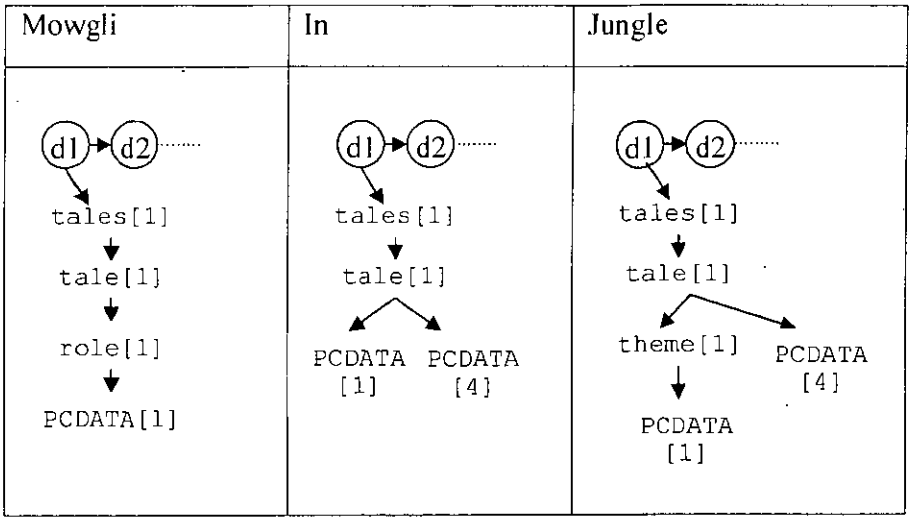

Table 4.4 Tree Structures for the Keywords in Phrase Query

4.2.3 Finding presence of a phrase within a domain element

Using the previous example, the descendants of the domain element `tales[1]/tale[1]` node, in order of appearance, are as follows:

`tales[1]/tale[1]/PCDATA[1]`	Contains keyword 'in'
`tales[1]/tale[1]/year[1]`	Contains no keyword of the phrase
`tales[1]/tale[1]/year[1]/PCDATA[1]`	Contains no keyword of the phrase
`tales[1]/tale[1]/PCDATA[2]`	Contains no keyword of the phrase
`tales[1]/tale[1]/theme[1]`	Contains no keyword of the phrase
`tales[1]/tale[1]/theme[1]/PCDATA[1]`	Contains keyword 'jungle'
`tales[1]/tale[1]/PCDATA[3]`	Contains no keyword of the phrase
`tales[1]/tale[1]/role[1]`	Contains no keyword of the phrase
`tales[1]/tale[1]/role[1]/PCDATA[1]`	Contains keyword 'mowgli'
`tales[1]/tale[1]/PCDATA[4]`	Contains keywords 'in' and 'jungle'

Table 4.5 Descendants of `tales[1]/tale[1]` in XML Structure 4.1

We propose to consider only the descendants that contain at least one keyword while searching the `tales[1]/tale[1]` node in the above document. The positions of the phrase keywords are as follows in the respective descendants:

Descendant	**Positions**$_{mowgli}$	**Positions**$_{in}$	**Positions**$_{jungle}$
`tales[1]/tale[1]/PCDATA[1]`	-	1	-
`tales[1]/tale[1]/theme[1]/PCDATA[1]`	-	-	1
`tales[1]/tale[1]/role[1]/PCDATA[1]`	1	-	-
`tales[1]/tale[1]/PCDATA[4]`	-	1, 3	2

Table 4.6 Keyword Positions in Descendants of `tales[1]/tale[1]`

From the above positions, it can not be determined whether the keywords appear in proper sequence of the phrase. To find out whether the phrase exists in the location, we first transform the positions of the keywords using a *Position Transformation* method described below. In *Position Transformation* method we scan the descendants in order of their appearance. At the beginning of the transformation we initialize a counter *prevmax* with zero.

While scanning descendants, we do the followings:

- Find out which keywords are present in a descendant
- Record the positions of the keywords (augmented by *prevmax*) in the descendant node.
- Update the value of prevmax by *prevmax* + *maxpos*, where *maxpos* is the maximum position value of all keywords which appear in the current descendant node.
- Repeat above 3 steps for the remaining descendants.

When the scanning of descendants is complete, we record the keyword positions. For the above example, the transformation steps are as follows:

Keywords: mowgli, in, jungle

prevmax=0

Descendant 1: `tales[1]/tale[1]/PCDATA[1]`

Contained keyword(s): in

$TransformedPositions_{in} = 1$

prevmax = 1

Descendant 2: `tales[1]/tale[1]/theme[1]/PCDATA[1]`

Contained keyword(s): jungle

$TransformedPositions_{in} = 1$

$DescendantPositions_{jungle} = 1$

$TransformedPositions_{jungle} = 2$ (augmented by *prevmax*)

prevmax = 2

Descendant 3: `tales[1]/tale[1]/role[1]/PCDATA[1]`

Contained keyword(s): mowgli

$TransformedPositions_{in} = 1$

$TransformedPositions_{jungle} = 2$

$DescendantPositions_{mowgli} = 1$

$TransformedPositions_{mowgli} = 3$ (augmented by *prevmax*)

prevmax = 3

Descendant 4: `tales[1]/tale[1]/PCDATA[4]`

Contained keyword(s): in, jungle

$TransformedPositions_{mowgli} = 3$

$DescendantPositions_{in} = 1, 3$

$TransformedPositions_{in} = 1, 4, 6$ (descendant positions are augmented by *prevmax*)

$DescendantPositions_{jungle} = 2$

$TransformedPositions_{jungle} = 2, 5$ (descendant positions are augmented by *prevmax*)

prevmax = 6

Above transformation method produces the following transformed positions for the keywords.

$TransformedPositions_{mowgli} = 3$

$TransformedPositions_{in} = 1, 4, 6$

$TransformedPositions_{jungle} = 2, 5$

In this example all the keywords appear in the domain element '`tales[1]/ tale[1]`' but in the real life this may not happen; some keywords may be missing. We determine the presence or absence of the phrase in the domain element from these transformed positions. In order to determine the presence of a phrase, we execute an iterative process for each keyword present in the phrase, in order of their appearance, i.e., the ith iteration is for the ith keyword.

We start the iterative process with a two dimensional empty array: *final-positions*. In the first iteration, we take the transformed positions of the first keyword as distinct rows of *final-positions*. After the first iteration the *final-positions* array becomes:

$$final\text{-}positions = \begin{bmatrix} 3 & * \\ * & * \end{bmatrix}$$

Since the first keyword of the phrase 'mowgli in jungle' has only one transformed position, after the first iteration the *final-positions* array has only one non-empty row and column.

In the subsequent iteration, all the transformed position values of the current keyword are appended to all the rows of *final-positions* array of the previous iteration. We need to keep the values in a row in *final-positions* array in the ascending order. For this reason, all the row values which are greater than the appended value are marked as null or absent. Null or absent is represented with a * sign. If any row of the previous iteration fails to append a larger value due to its unavailability in the current iteration, we simply repeat

the row appending a null value. After the second iteration we get the array elements as follows.

$$\textit{final-positions} = \begin{bmatrix} * & 1 \\ 3 & 4 \\ 3 & 6 \end{bmatrix}$$

We declare a row in the *final-positions* array invalid if the number of absent elements in that row is greater than the allowed number of absent keywords in the search result of the Phrase Query. A row also becomes invalid if the difference between two adjacent row elements becomes higher than k, where k is the measure of the proximity of the keywords or the maximum allowable distance between two successive keywords in a phrase. We remove the invalid rows at the end of iteration.

In our example, the elements of the 3rd row {[3, 6]} of *final-positions* array indicates that 2 words appear between 'mowgli' (at transformed position 3) and 'in' (at transformed position 6). If we allow at most one interleaving word (2-proximity), we need to remove the third row ([3, 6]) from the array.

$$\textit{final-positions (2-proximity)} = \begin{bmatrix} * & 1 \\ 3 & 4 \end{bmatrix}$$

In the third iteration, first we append the transformed position value **2** to each of the row in the *final-positions* array. Appending the value **2** to the first row ($[* \ 1]$) gives the new row $[* \ 1 \ 2]$. Appending the transformed position value of 2, to the remaining row ($[3 \ 4]$), we get the row $[* \ * \ 2]$. But we do not include the row $[* \ * \ 2]$ in *final-positions* because the row ($[* \ 1 \ 2]$) in *final-positions* covers it as follows:

- The non-null values in $[* \ * \ 2]$ match the corresponding values in $[* \ 1 \ 2]$
- For each null value in $[* \ * \ 2]$, there is a corresponding null or non-null value in $[* \ 1 \ 2]$

Next we append the transformed position value **5** to the rows of the *final-positions* array found from the previous iteration. After the third iteration we find the following *final-positions* array:

$$\textit{final-positions} = \begin{bmatrix} * & 1 & 2 \\ * & 1 & 5 \\ 3 & 4 & 5 \end{bmatrix}$$

Now, if we allow at most one interleaving word (2-proximity), we get:

$$\textit{final-positions (2-proximity)} = \begin{bmatrix} * & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

If we further impose the condition that all keywords must be present in the result,

$$\textit{final-positions (2-proximity, max. 0 absent terms)} = \begin{bmatrix} 3 & 4 & 5 \end{bmatrix}$$

If the *final-positions* array is not empty after the last iteration, we can conclude that the *domain element* contains the phrase. An empty array indicates the absence of the phrase within the *domain element*. At the end of the process, each row of a non-empty *final-positions* array represents one occurrence of the phrase and the transformed position of each keyword in that occurrence.

The algorithm to process the Phrase Query is given in the form of pseudo-code in Algorithm 4.1.

```
allowed_absence <- allowed no. of absent keywords in
                   the search results;
proximity <- expected proximity of keywords in search result;

//    DOMAIN CONSTRUCTION
Np <- no. of terms in the phrase;
```

```
asc_loc_keyword_ordinal[Np] <- an array with values 1 to Np;

/* if i<j then (asc_loc_keyword_ordinal[i])-th term has less
locations than (asc_loc_keyword_ordinal[j])-th term */

initial_list <- {}
limit <- min(allowed_absence+1, Np);
for i <- 0 to limit do
  initial_list <- initial_list U (asc_loc_keyword_ordinal[i]-th
  keyword locations);
end
domain <- {};
foreach location loc in initial_list do
  dom_element <- closest ancestor of loc, which contains
                 at least (Np-allowed_absence) keywords
  if dom_element is not null or empty
    add dom_element to domain
end
//   POSITION TRANSFORMATION AND SEARCHING
foreach location loc in domain do
  positions[Np][] = {}
  descendants <- descendants of loc
  prevmax <- 0
  foreach phrase keyword pk in descendants do
    updtae positions[pko[pk]]
    update prevmax
  end
  if (search-positions(positions))
    add loc to result
end

// Keyword Ordinal in Phrase
proc pko(keyword) do
  return keyword ordinal in the input phrase
```

```
end

// Append each element of keyword_pos to
// each row of final_2d_arr
proc append(final_2d_arr, keyword_pos) do
  nes <- {};
  foreach p in keyword_pos do
    foreach row e in final_2d_arr do
      ne <- p appended to e (items which are greater than p,
        in e are replaced with null or * mark);
      if nes contains ne
        continue;
      if(ne is covered by some other element in nes)
        continue;
      if(valid(ne))
        add ne to nes;
    end
  end
  return nes;
end

// Returns boolean
proc search-positions(positions) do
  final_2d_arr = {}
  for i = 1 to Np do
    final_2d_arr = append(final_2d_arr, positions[i])
  end
  return !final_2d_arr.empty?
end
```

Algorithm 4.1 Algorithm for Partial Phrase Matching in XML

4.3 New Processing for NOT Query

We introduce a new type of internal node NOT node in the query plan tree to explain the processing of a NOT query. Assume, we have a NOT node and two children which are leaves and contain two terms. Unless we make some assumption about which side should be present in the result and which side should not be present, it is not possible from the perspective of the parent NOT node to make the decision. Since the query plan tree is a left order traversal tree, we assume that the keyword or group of keywords on the left side of a NOT node should exist and keywords on the right side should not exist in the search result. When the NOT operation is carried out, we can do a filtering of documents because documents absent in result of the previous part of the query are not needed. For example, we consider the search query '*find T_1 and not T_2*'. The result would be a subset of the result of T_1.The query can be expressed as follows in a query plan tree:

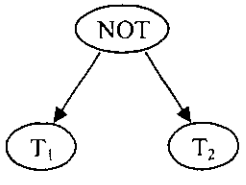

Figure 4.1 Representing NOT Query in Query Plan Tree

We read the (document id, path length, path node names, element indexes, sequence indexes) tuples for T_1 and T_2. We can use the term *path* for (path length, path node names, element indexes, sequence indexes). So we have (*document id, path*) pairs for each of these terms.

For simplicity let (*document id_i, $path_i$*) for T_1 be expressed as (doc^1_i, $path^1_i$) and (*document id_j, $path_j$*) for T_2 be expressed as (doc^2_j, $path^2_j$). To explain this, let T_1 has 2 (document id, path) pairs and T_2 has 3 (document id, path) pairs. The pairs of T_1 are expressed as the set $\{(doc^1_1, path^1_1), (doc^1_2, path^1_2)\}$ and those of T_2 are expressed as the set $\{(doc^2_1, path^2_1), (doc^2_2, path^2_2), (doc^2_3, path^2_3)\}$. To solve the NOT operation, we check each (doc^1_i, $path^1_i$) pair with each (doc^2_j, $path^2_j$) pair.

According to our observation, only five cases may appear while we are checking above two locations:

Case 1: The two locations actually point to the same location in the same document.
Case 2: The location of T_2 points to a descendant location of T_1 in the same document.
Case 3: The location of T_1 points to a descendant location of T_2 in the same document.
Case 4: The two locations are actually different locations in the same document.
Case 5: The two locations are two different locations in two different documents.

We initialize a flag *excluded* with the value *false* at the beginning of checking a (doc^1_i, $path^1_i$) pair of T_1 with the (doc^2_j, $path^2_j$) pairs of T_2. Below we describe how our proposed NOT query processing method handles the above mentioned scenarios while we check the (doc^1_i, $path^1_i$) pair of T_1 with a (doc^2_j, $path^2_j$) pair of T_2,

Case 1: ($doc^1_i = doc^2_j$) and ($path^1_i = path^2_j$)

Both T_1 and T_2 appear at the same location (doc^1_i, $path^1_i$) as shown in Figure 4.2 and the location can not be included in the result of the NOT operation. We update the flag *excluded* with the value true and stop checking the (doc^1_i, $path^1_i$) pair with the remaining (doc^2_j, $path^2_j$) pairs.

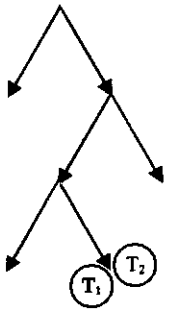

Figure 4.2 ($doc^1_i = doc^2_j$) and ($path^1_i = path^2_j$) in NOT Query

Case 2: ($doc^1_i = doc^2_j$) and ($path^1_i \subset path^2_j$)

The keyword T_2 appears in a node which is descendant from a node which contains the keyword T_1 as shown in Figure 4.3. In other words, (doc^1_i, $path^1_i$) contains T_2. This is why (doc^1_i, $path^1_i$) can not be included in the result of the NOT operation. We update the flag *excluded* with the value true and stop checking the (doc^1_i, $path^1_i$) pair with the remaining (doc^2_j, $path^2_j$) pairs.

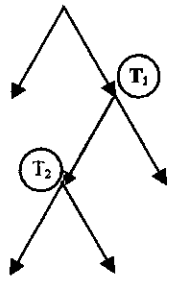

Figure 4.3 ($doc^1_i = doc^2_j$) and ($path^1_i \subset path^2_j$) in NOT Query

Case 3: ($doc^1_i = doc^2_j$) and ($path^2_j \subset path^1_i$)

The keyword T_1 appears in a node which is descendant from a node which contains the keyword T_2. This has been shown in Figure 4.4. (doc^1_i, $path^1_i$) should not be included in the result of the NOT operation. We update the flag *excluded* with the value true and stop checking the (doc^1_i, $path^1_i$) pair with the remaining (doc^2_j, $path^2_j$) pairs.

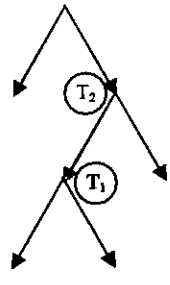

Figure 4.4 ($doc^1_i = doc^2_j$) and ($path^2_j \subset path^1_i$) in NOT Query

Case 4: ($doc^1_i = doc^2_j$) and ($path^1_i \neq path^2_j$)

(doc^1_i, $path^1_i$) and (doc^2_j, $path^2_j$) are two different locations in the same document. In other words, T_1 and T_2 appear at different locations as shown in Figure 4.5. We check the (doc^1_i, $path^1_i$) pair with the remaining (doc^2_j, $path^2_j$) pairs.

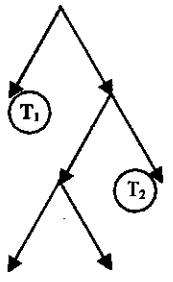

Figure 4.5 ($doc^1_i = doc^2_j$) and ($path^1_i \neq path^2_j$) in NOT Query

Case 5: ($doc^1_i \neq doc^2_j$)

(doc^1_i, $path^1_i$) and (doc^2_j, $path^2_j$) are two locations in two different documents. In other words, T_1 and T_2 appear at different documents. We check the (doc^1_i, $path^1_i$) pair with the remaining (doc^2_j, $path^2_j$) pairs of T_2.

Considering the above cases, when we have completed checking of a (doc^1_i, $path^1_i$) pair with all of the (doc^2_j, $path^2_j$) pairs, we check the value of the flag *excluded*. If the value of the flag is false, we add the (doc^1_i, $path^1_i$) pair to the search result of the NOT operation. Otherwise, the (doc^1_i, $path^1_i$) pair is excluded from the search result.

The algorithm to process the NOT operation 'T_1 and NOT T_2' is given in the form of pseudo-code in Algorithm 4.2.

```
proc NOT (T1, T2) do
  NOT_op_result <- {}

  {(doc^1_i, path^1_i)} <- search (T1)
  {(doc^2_j, path^2_j)} <- search (T2)

  for each pair (doc^1_i, path^1_i) of T1 do
    excluded <- false
    for each pair (doc^2_j, path^2_j) of T2 do

      if (doc^1_i = doc^2_j) and (path^1_i = path^2_j) do
        excluded <- true
        break
      end

      elsif (doc^1_i = doc^2_j) and (path^1_i ⊂ path^2_j) do
        excluded <- true
        break
      end

      elsif (doc^1_i = doc^2_j) and (path^2_j ⊂ path^1_i) do
        excluded <- true
        break
      end

      elsif (doc^1_i = doc^2_j) and (path^1_i ≠ path^2_j)
        continue

      elsif (doc^1_i ≠ doc^2_j)
        continue

    end // end of inner for loop

    if !excluded
```

```
        add (doc¹i, path¹i)} to NOT_op_result
    end // end of outer for loop
    return NOT_op_result
end // end of proc
```

Algorithm 4.2 Algorithm for NOT Query Processing

4.4 Phrase in NOT Queries

The query *"T_1T_2" and NOT T_3* can be processed as *T_4 and NOT T_3* where T_4 is the result of the phrase T_1T_2. The NOT portion of a query can contain a phrase. So, the query *T_1 and NOT "T_2T_3"* can be processed as *T_1 and NOT T_4* where T_4 is the result of phrase T_2T_3. When we search for the phrase T_2T_3 we can filter out documents which do not contain the term T_1.

4.5 Summary

In this chapter, we have discussed our proposed modification in XML inverted file structure and new query processing algorithms which are essential for Phrase and NOT queries over a collection of XML documents. In light of our proposed modification in XML inverted file structure and the new processing methods, we have carried out experiments using our modification and processing methods to verify the effectiveness of the modification and the processing methods. In the next chapter we discuss our experimental results.

Chapter 5

Experimental Results

In the previous chapter, we have proposed processing methods for Phrase and NOT queries over a collection of XML documents. We have introduced a change in the XML inverted file structure too. To test the correctness of our modification and proposed methods, we have developed a prototype query processor. Our prototype query processor is able to index XML documents incrementally and supports conjunction, disjunction operation, and the new operations of Phrase Query and NOT query.

We have compared the result of our proposed Phrase Query processing method or algorithm with the result of the conjunction and disjunction operations of the terms in the phrase because the presence of a phrase in a document implies that keywords of the phrase are present in the document. For the NOT query, we compare the result of our proposed processing method with that of the conjunction and disjunction of the keywords which are expected to be present in the result. The results of our experiment are given below.

5.1 Experimental Test Cases

We have verified the correctness of the proposed inverted file structure and Phrase and NOT query processing algorithms with incremental sets of XML documents and numerous queries. Test sets comprised of incremental and disjoint XML documents of various structures. We compared the efficiency and the relevancy of the results of our proposed algorithms with those of conjunction and disjunction algorithms. We define relevancy and efficiency as follows:

$$\text{Relevancy} = \frac{\text{No. of relevant results found}}{\text{Total no. of found results}}$$

$$\text{Efficiency} = \frac{\text{No. of relevant results found}}{\text{Total no. of expected results}}$$

Here, *total no. of expected results* is the number of results which are expected to match against the query to be relevant. *No. of relevant results found* is the number of relevant results actually found after matching the keywords of the query. The efficiency of a search algorithm defines what percentage of expected result is actually found by the algorithm.

The search algorithm may produce both relevant and non-relevant results. We express the total number of results (relevant and non-relevant combined) as *Total No. of found results*. We define the relevancy as the percentage of actual relevant results in the total search results.

We used three different test sets in our experiments. There were 1075 XML documents in the first test set. We gradually added more XML documents in the second and third test sets. The second and the third test sets consisted of 1297 and 1400 XML documents respectively.

Our proposed modification increases the XML inverted file size by adding ordinal keyword positions in the inverted list document entries. To test how it affects the inverted file size, we compared the sizes of the proposed inverted files with those of the previous structure.

Next using these three test sets, we compared the search results of our proposed Phrase Query processing algorithm with those of the conjunction and disjunction operations for the following measures:

1. Relevancy of Phrase Query
2. Efficiency of Phrase Query

We show the results of our comparisons using graphs. We plot the no. of keywords along the X axis and the relevancy or efficiency the along the Y axis in the graphs. There are two series along the Y axis in a graph. The yellow series shows the results found using our proposed processing method and the green series shows the results found from the conjunction or disjunction operation of the keywords in a Phrase Query.

After comparing the relevancy and efficiency of Phrase Query, we tested the effect of the value of proximity (k) on relevancy and efficiency in Phrase Query. We compare the relevancy and efficiency of conjunction operation, disjunction operation and our phrase processing algorithm for different values of k. We show the results of these comparisons using graphs. We plot the number of keywords along the X axis and the relevancy or efficiency the along the Y axis in the graphs. There are k numbers of series along the Y axis for each query in a graph. Each series represents the relevancy or efficiency value for the corresponding proximity value.

At the end, using the test sets we compared the search results of our proposed NOT query processing algorithm with those of the conjunction and disjunction operations for the following measures:

3. Relevancy of NOT Query
4. Efficiency of NOT Query

Similar to our experiments for Phrase Query, we show the results of our comparisons using graphs. We plot the no. of keywords along the X axis and the relevancy or efficiency the along the Y axis in the graphs. There are two series along the Y axis in a graph. The yellow series shows the results found using our proposed processing method and the green series shows the results found from the conjunction or disjunction operation of the expected keywords in a NOT query.

5.2 Effect of Proposed Modification on Inverted File Size

As we add the positions of the keywords in the inverted list document entries, the size of the inverted files grow. To test how our proposed modification affects the inverted file size, we created the inverted files for the three test sets with and without the position information of the keywords. In this comparison, we did not use any compression method for compressing the inverted files. The graph in Figure 5.1 shows our findings in this context. We place the total XML data size of the test sets along the X axis and the sizes of the inverted files along the Y axis. The green series represents the inverted file without any modification and the yellow series represents the inverted file with our proposed modification.

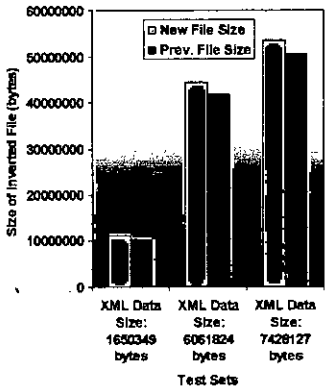

Figure 5.1 Comparison of Inverted File Sizes

We found that the extra position information increased the size of the XML inverted files by approximately 6.4%.

5.3 Comparison of Relevancy in Phrase Query

We compared the relevancy of our Phrase Query processing algorithm with that of the conjunction and disjunction operations.

The results of the comparisons with the conjunction operation are presented in Figure 5.2. The graphs in Figure 5.2 show that our proposed solutions have improved the relevancy of the search results. We carried out three Phrase queries over each of the test sets.

Using our proposed Phrase Query solution, we achieved 95.84%, 331%, and 177.6% improvement in relevancy for test set 1, test set 2, and test set 3 respectively. This high percentage of improvement is due to the fact that the conjunction operation processing algorithm does not take care of the sequence and positions of keywords, which are necessary for Phrase Query processing in order to find more relevant search results. Our proposed Phrase Query processing algorithm does consider both the sequence and positions of keywords in the query.

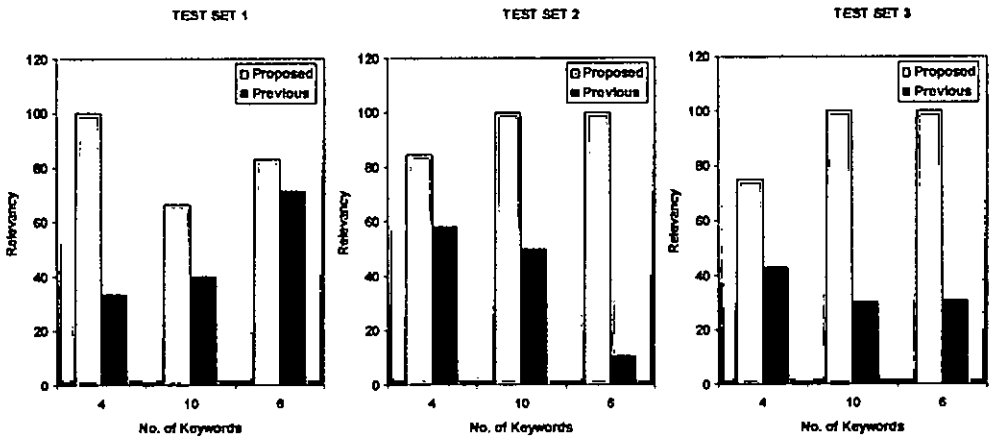

Figure 5.2 Phrase Query Relevancy Comparison with the Conjunction Operation

The graphs in Figure 5.3 show the comparison between our proposed Phrase Query solution and the disjunction operation. Here also, our proposed solution has improved the relevancy of the search results. We carried out three Phrase queries over each of the test sets.

Using our proposed Phrase Query solution, we achieved 210%, 363.8%, and 351.7% improvements in relevancy for test set 1, test set 2, and test set 3 respectively. Like conjunction operation, the disjunction operation also does not take care of the sequence and positions of keywords. Moreover, the disjunction operation accepts documents which

107529

contain any of the query keywords. For these reasons, the results of a Phrase Query using the disjunction operation are highly irrelevant.

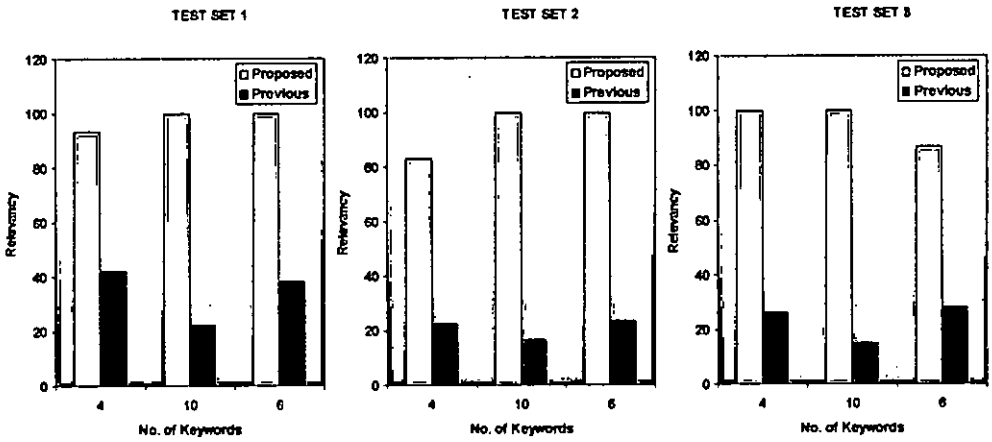

Figure 5.3 Phrase Query Relevancy Comparison with the Disjunction Operation

5.4 Comparison of Efficiency in Phrase Query

We carried out the same phrase queries of section 5.3 in order to compare the efficiency of our algorithm. The graphs in Figure 5.4 and Figure 5.5 show that our proposed algorithm for Phrase Query gives the same 100% efficiency as the conjunction or disjunction operation algorithm. This is due to the fact that our Phrase Query processing algorithm or the conjunction and disjunction operation processing algorithms do not miss any expected result.

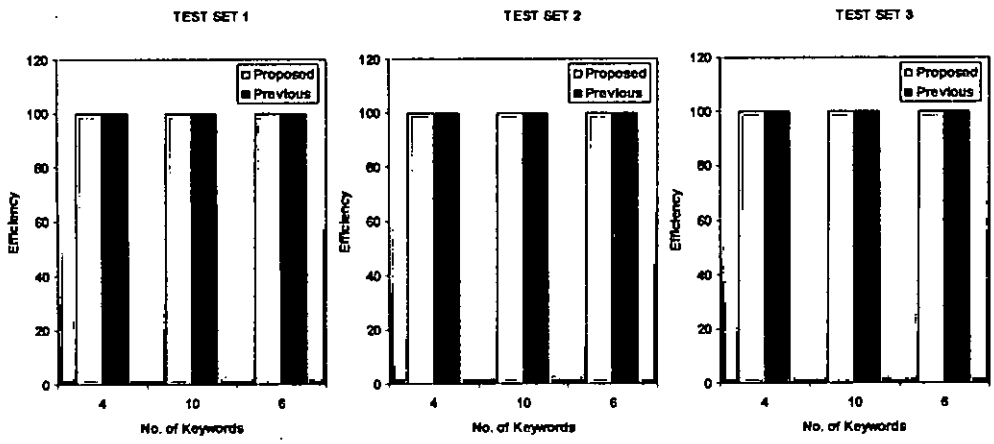

Figure 5.4 Phrase Query Efficiency Comparison with the Conjunction Operation

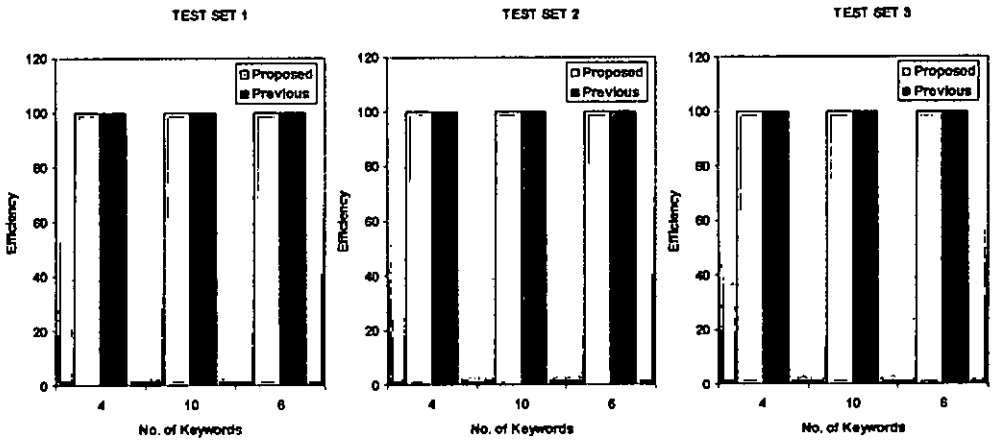

Figure 5.5 Phrase Query Efficiency Comparison with the Disjunction Operation

5.5 Analysis of Proximity (*k*) in Phrase Query

We used test set 2 and test set 3 to analyze the effect of the value of proximity on conjunction operation, disjunction operation and our proposed Phrase Query method. We used three queries for each operation and the proximity values were 1, 2, 3, and 4. We noticed increase in relevancy with the increase in the proximity value for all of the three algorithms as shown in Figure 5.6 and Figure 5.7.

Using the test set 2, for the conjunction operation the (minimum, maximum) pairs of values of relevancy were (44%, 55%), (33%, 67%), and (36%, 64%) for the first query, second query, and the third query respectively. For the disjunction operation the corresponding pairs were (31%, 38%), (12.5%, 25%), and (36%, 64%). For our proposed Phrase Query processing algorithm these pairs were (80%, 100%), (100%, 100%), and (80%, 100%). The corresponding graphs are shown in Figure 5.6.

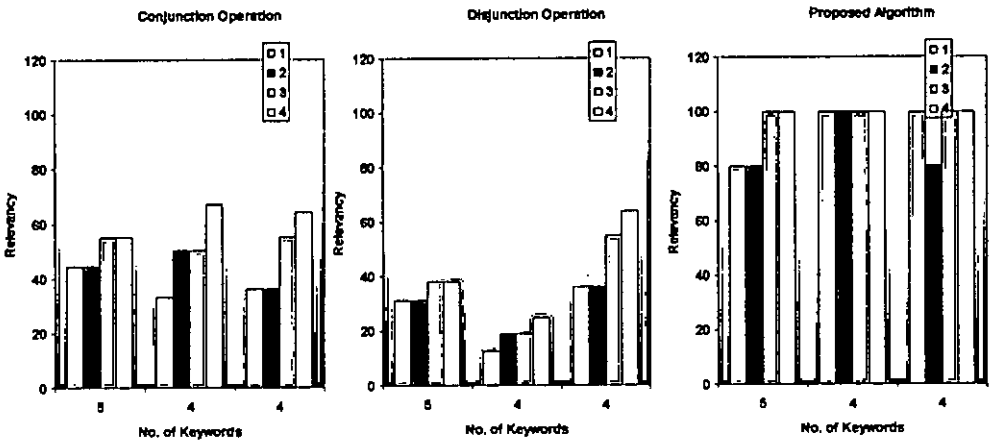

Figure 5.6 Effect of Proximity on Relevancy in Test Set 2

Using the test set 3, the (minimum, maximum) pairs of relevancy for the conjunction operation were (60%, 70%), (18%, 35%), and (44.4%, 66.7%) for the first query, second query, and the third query respectively. For the disjunction operation the corresponding pairs were (41%, 48%), (12.5%, 25%), and (30.8%, 46.1%). For our proposed Phrase Query processing algorithm these pairs were (100%, 100%), (100%, 100%), and (80%, 100%). The corresponding graphs are shown in Figure 5.7.

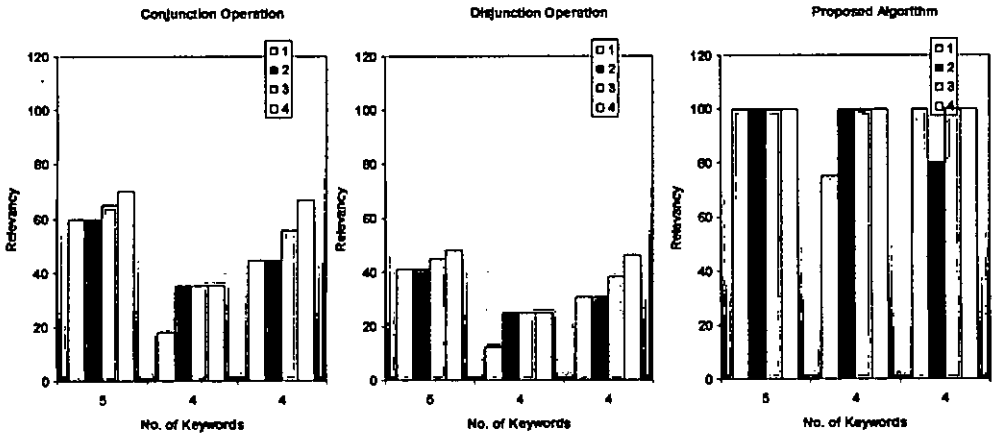

Figure 5.7 Effect of Proximity on Relevancy in Test Set 3

The efficiency of the conjunction, disjunction and the proposed Phrase Query processing method remain 100% as shown in Figure 5.8 and Figure 5.9.

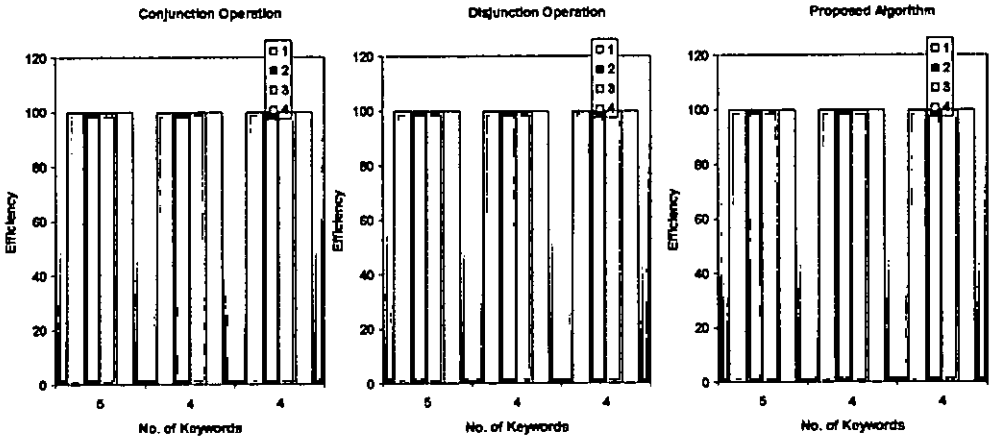

Figure 5.8 Effect of Proximity on Efficiency in Test Set 2

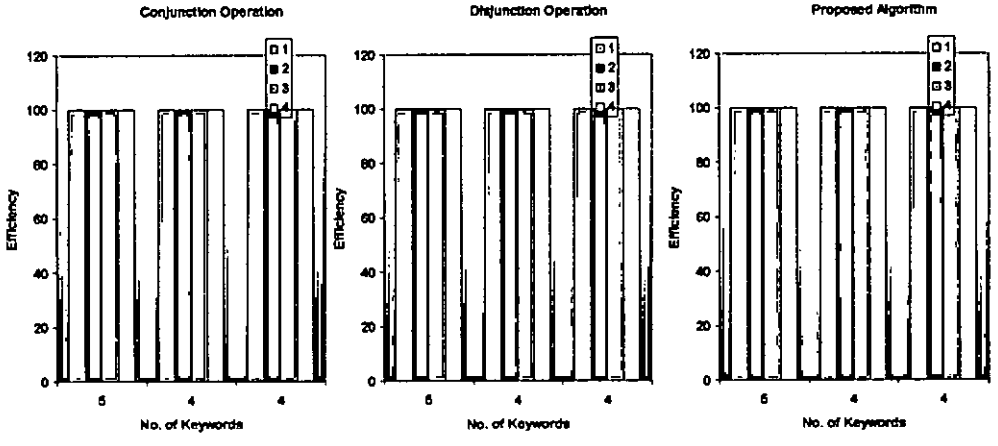

Figure 5.9 Effect of Proximity on Efficiency in Test Set 3

The relevant and non-relevant results of a search constitute the total results returned by the query. Using conjunction or disjunction operation for a Phrase Query introduces irrelevant results because of the following two reasons:

1. Proper sequence of phrase keywords is not maintained in the result
2. Proper sequence is maintained but the keyword positions violate proximity

If we use larger proximity values for a search query, those results where proper sequence is maintained but proximity is violated, start to become relevant. As a result, relevancy starts increasing. On the other side, for the results where the proper sequence is absent

still remain irrelevant. This is why relevancy increases as we increase the value of proximity but full relevancy is never achieved.

Increasing proximity does not affect the efficiency of conjunction or disjunction operation because these operations select all possible locations ignoring proximity. So changing proximity does not change the total number of results returned. If we increase proximity, some non-relevant results become relevant, thus no expected result is missed and efficiency remain same 100%.

In our Phrase Query processing method, proper sequence of phrase keywords is always maintained. In some results, the phrase keywords remain separated by a distance which is greater than *k*- so irrelevancy appears. But if we start increasing proximity or *k*, these non-relevant results start to become relevant. At the same time, the number of total found results may also increase because new locations may become relevant according to the increased proximity. As a result, we experience 100% relevancy for larger values of *k*. About efficiency, all of the new expected results become relevant. This is why 100% efficiency is maintained.

5.6 Comparison of Relevancy in NOT Query

We have compared the relevancy of our NOT query processing algorithm with that of the conjunction and disjunction operations.

To compare our proposed NOT query solution with the conjunction operation, we carried out three NOT queries over each of the the test set 1, test set 2 and test set 3 respectively. The graphs in Figure 5.10 show that our proposed solution provides more relevant results than the conjunction operation.

Using our proposed NOT query solution, we achieved 72.22%, 45%, and 202.76% improvements over the conjunction operation, in relevancy for test set 1, test set 2, and test set 3 respectively.

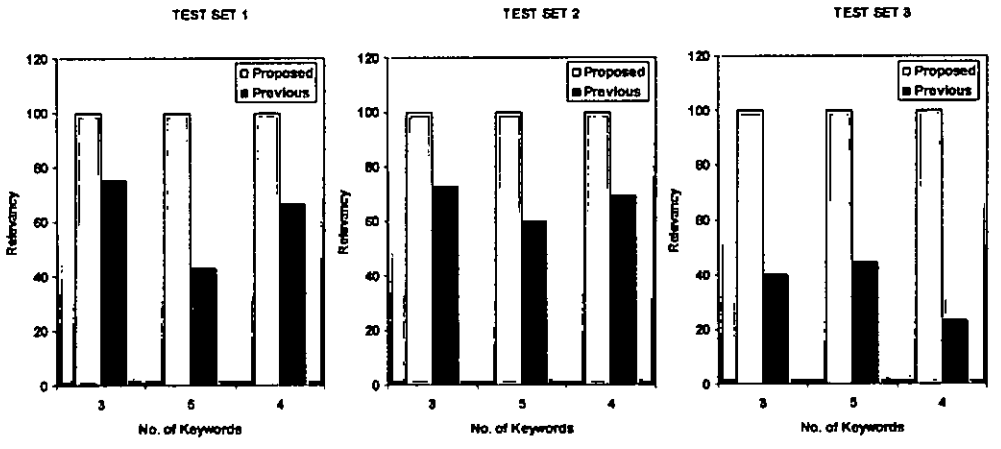

Figure 5.10 NOT Query Relevancy Comparison with Conjunction Operation

To compare our proposed NOT query solution with the disjunction operation, we carried out three NOT queries over the test set 1, test set 2 and test set 3 respectively. The graphs in Figure 5.11 show that our proposed solution provides more relevant results than the disjunction operation.

Using our proposed NOT query solution, we achieved 139.5%, 182.5%, and 375% improvements over the disjunction operation, in relevancy for test set 1, test set 2, and test set 3 respectively.

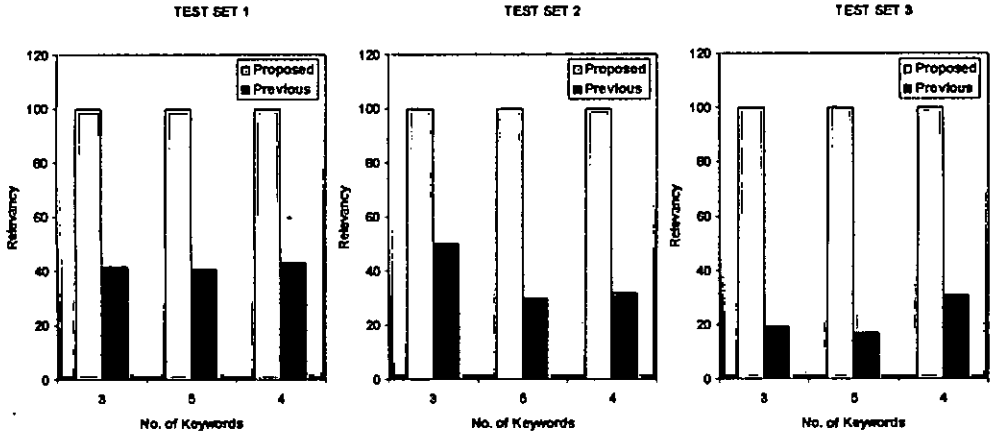

Figure 5.11 NOT Query Relevancy Comparison with Disjunction Operation

We achieved large improvements in relevancy because, the conjunction or disjunction operation cannot filter the documents where the keywords under NOT operation are present and produces more irrelevant search results regarding the NOT operation. Our proposed algorithm for processing NOT operation can filter the documents where the keywords under NOT operation are present and produces more relevant search results regarding the NOT operation.

5.7 Comparison of Efficiency in NOT Query

We carried out the same NOT queries of section 5.6 to verify the efficiency of our proposed solution. The graphs in Figure 5.12 show that our proposed solution for NOT query maintains the same efficiency of the conjunction operation. Like our proposed Phrase Query processing algorithm, our NOT query processing algorithm do not miss any expected result.

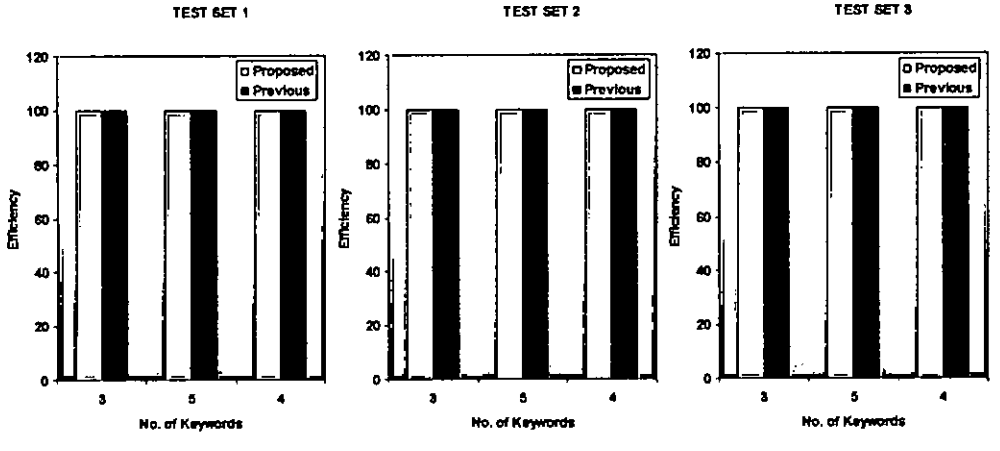

Figure 5.12 NOT Query Efficiency Comparison with Conjunction Operation

The graphs in Figure 5.13 show that the disjunction operation also achieves 100% efficiency because this operation does not miss any expected result.

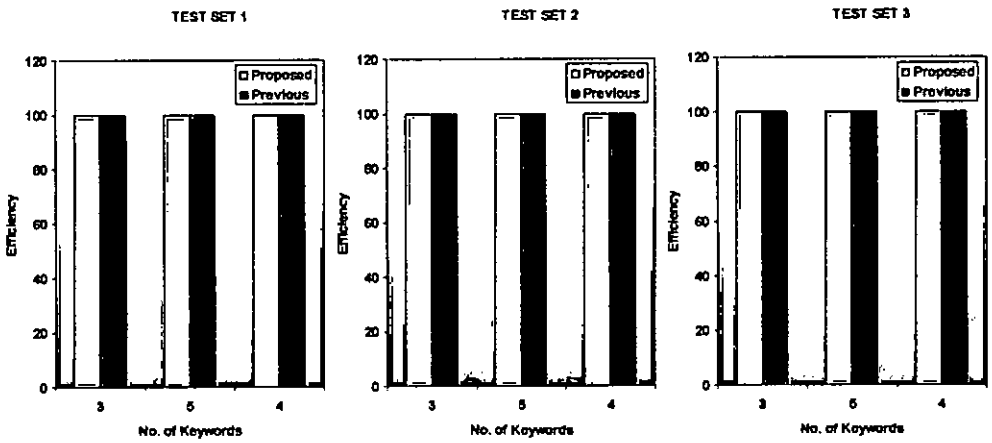

Figure 5.13 NOT Query Efficiency Comparison with Disjunction Operation

The experiments carried out show that our proposed solutions to process Phrase and NOT queries over a collection of XML documents provide much more relevant results than the conjunction or disjunction operation of query keywords.

5.8 Summary

In this chapter, we have presented the results of our experiments carried out with the help of our prototype query processor which is able to perform conjunction, disjunction, Phrase Query and NOT query operations. We presented the comparison between the output of our proposed Phrase Query processing method and that of the conjunction operation of the terms in the phrase. Similarly, for the NOT query, we compared the result of our proposed processing method with that of the conjunction of the keywords which are expected to be present in the result. The experimental results show that our proposed solutions enable Phrase and NOT query processing over a collection of XML documents and give more relevant results maintaining the same efficiency of the conjunction operation.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

The processing methods of Phrase Query over a collection of XML documents were not available earlier. Therefore, we proposed modification of the inverted file structure for XML documents and new processing methods to enable Phrase querying over a collection of XML documents. We have also developed a new solution to execute NOT Query over a collection of XML documents. NOT query processing algorithm was not available either.

We have developed a prototype query processor to verify the correctness of our proposed modification and new query processing algorithms. The prototype query processor was able to parse a collection of XML documents and to handle incremental documents without re-parsing the already parsed documents to produce inverted files according to our proposed modified structure. We carried out experiments on multiple document collections and compared the results of our proposed solution with those of the conjunction operation. We found that our processing methods of Phrase Query perform much better than the other in terms of relevancy of the result. We also found that our NOT query processing method provides all the relevant results.

6.2 Future Works

In our future research works we will focus on:

- We have observed that our proposed Phrase Query solution does not provide 100% relevancy always. We want to achieving 100% relevancy in Phrase Query

over a collection of XML documents. We will find out a method to eliminate all irrelevant search results.

- Developing a method to fetch those XML documents which contain synonyms of the search keywords.

References

[1] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F., "Extensible Markup Language (XML) 1.0 (Fifth Edition)", W3C Recommendation 26 November 2008, http://www.w3.org/TR/2008/REC-xml-20081126/, 2008

[2] Anh, V. N. and Moffat, A., "Compression and an IR Approach to XML Retrieval", Proceedings of the 1st INEX Workshop, pp 99-104, 2002

[3] Zobel, J. and Moffat, A., "Inverted files for text search engines", ACM Computing Surveys (CSUR), vol-38, No. 2, pp 6-61, 2006

[4] Singhal, A., "Modern Information Retrieval: A Brief Overview", Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol-24, No. 4, pp 35–43, 2001

[5] Vries, A. P. de, "Challenging Ubiquitous Inverted Files", Proceedings of the First DELOS Network of Excellence Workshop on Information Seeking, Searching and Querying in Digital Libraries, pp 71-75, 2000

[6] Fuhr, N. and Gövert, N., "Index compression vs. retrieval time of inverted files for XML documents", Proceedings of the 11th International Conference on Information and Knowledge Management, pp 662-664, 2002

[7] McHugh, J., Widom, J., Abiteboul, S., Luo, Q., and Rajaraman, A. "Indexing Semistructured Data", Technical report, Stanford University, February 1998

[8] Yoon, J. P., Raghavan, V. and Chakilam, V., "BitCube: A Three Dimensional Bitmap Indexing for XML Documents", Journal of Intelligent Information Systems, vol-17, No. 2, pp 241-254, 2001.

[9] Yoon, J. P., Raghavan, V. and Chakilam, V., "Bitmap Indexing-based Clustering and Retrieval of XML Documents", In ACM SIGIR'01 Workshop on Mathematical/Formal Methods in IR, 2001

[10] Pastukhov, R., "A method for evaluating full-text search queries in Native xml databases", Proceedings of the 4th Spring Young Researchers' Colloquium on Databases and Information Systems (SYRCoDIS-2007), 2007

[11] Moffat, A. and Zobel, J., "Self-indexing inverted files for fast text retrieval", ACM Transactions on Information Systems (TOIS), vol-14, No. 4, pp 349-379, 1996

[12] Yahia, S. A., Fernández, M., Srivastava, D. and Xu, Y., "Phrase Matching in XML", Proceedings of the 29th international conference on Very large data bases, pp 177-188, 2003

[13] Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J. and Siméon, J., "XQuery 1.0: An XML Query Language", W3C Recommendation 23 January 2007, http://www.w3.org/TR/2007/REC-xquery-20070123/, 2007

[14] Bremer, J. M. and Gertz, M., "XQuery/IR: Integrating XML Document and Data Retrieval", Proceedings of the 4th International Workshop on the Web and Databases (WebDB), pp 1-6, 2002

[15] Katz, H., "An introduction to XQuery - A look at the W3C's proposed standard for an XML query language", www.ibm.com/developerworks/xml/library/x-xquery.html, 2006

[16] King, J., "The Case for XQuery", November 2005 issue of SOA WebServices Journal, http://www.stylusstudio.com/whitepapers/why_xquery.html, 2005

[17] Clark, J., DeRose, S., "XML Path Language (XPath)", http://www.w3.org/TR/xpath, 1999

[18] Zobel, J., Moffat, A. and Ramamohanarao, K., "Inverted files versus signature files for text indexing", ACM Transactions on Database Systems (TODS), vol-23, No. 4, pp 453-490, 1998

[19] Salton, G. and Buckley, C., "Term-weighting approaches in automatic text retrieval", Information Processing and Management, vol-24, No. 5, pp 513-523, 1988

[20] Elias, P., "Universal codeword sets and representations of the integers", IEEE Transactions on Information Theory, vol-21, No. 2, pp 194–202, 1975

[21] Hirschberg, D. S. and Lelewer, D. A., "Efficient decoding of prefix codes", Communications of the ACM, vol-33, No. 4, pp 449–459, 1990

[22] Guo, L., Shao, F., Botev, C. and Shanmugasundaram, J., "XRANK: Ranked Keyword Search over XML Documents", Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp 16-27, 2003

[23] Xu, Y. and Papakonstantinou, Y., "Efficient Keyword Search for Smallest LCAs in XML Databases", Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pp 527-538, 2005

বাংলাদেশ প্রকৌশল বিশ্ববিদ্যালয়
নং 107529
তাং 25.1.10
লাইব্রেরী, ঢাকা