A TEXT SEARCH QUERY ALGEBRA

FOR XML DOCUMENTS

BY

HASAN AL-SHEBOUL

A thesis submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science

Major Subject: Computer Science

New Mexico State University

Las Cruces, New Mexico

August 2002
ABSTRACT

A TEXT SEARCH QUERY ALGEBRA

FOR XML DOCUMENTS

BY

HASAN AL-SHEBOUL, B.S.

Master of Science

New Mexico State University

Las Cruces, New Mexico, 2002

Dr. Enrico Pontelli, Chair

An algebra for querying structured text regions has been implemented as a Java package called “algebra,” along with a command line system called “xsrch.” The query algebra has sixteen operators that are based on text intervals, or regions. Their output also are regions. The operators are: union, intersection, difference, inner, outer, containing, not containing, contained in, not contained in, followed by, hull, extract, and four variants of the containment operators.

The implementation is divided into three phases: (a) document indexing, which depends on the parsing that utilizes the XML DOM parser which views
the document as a hierarchical structure of tree nodes, (b) forming text retrieval query expressions that are based on the query algebra operators, where their design is independent from the hierarchical structure of the document, but based on a document indexing and region sets, and (c) parsing and evaluation of query expressions.
CONTENTS

LIST OF FIGURES ...................................................... vii

1 INTRODUCTION ....................................................... 1

1.1 Document Indexing and Region Sets ................................. 3

1.2 Structured Text-Region Algebra ..................................... 6

1.3 XML Documents and The DOM Model ............................... 8

2 SOLUTION PROPOSED (System Overview) .......................... 10

2.1 Forming Regions ..................................................... 11

2.2 Nested Region Algebra Operators ................................. 12

2.2.1 The Ordinary Math Set Operators ............................... 13

2.2.2 The Unary Operators: Outer and Inner .......................... 13

2.2.3 The Containment Operators ..................................... 15

2.2.4 The Followed By Operator ....................................... 16

2.2.5 The Hull Unary Operator ......................................... 18

2.2.6 The Extract Operator ............................................. 19

2.2.7 The “@all” All elements ........................................ 19

2.3 Properties of Xsrch Query Expressions ............................. 20

2.4 Xsrch Query Expressions ........................................... 21

2.4.1 Query Examples .................................................. 21

iv
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>IMPLEMENTATION</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Facts, Behaviors, and Considerations</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Entities</td>
<td>25</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Apache Xerces2 for Java</td>
<td>25</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Some Considerations for Parsing</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Java Source Files Built in Package Algebra</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Indexing Classes Core</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Query Algebra Operators Classes</td>
<td>35</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Query Expression and Evaluation Classes</td>
<td>37</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Utility Classes</td>
<td>44</td>
</tr>
<tr>
<td>3.2.5</td>
<td>The Engine Class</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Class Design Objectives and Performance Issues</td>
<td>46</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Indexing Process</td>
<td>46</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Some Choices of Data Structures Implementation</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>Other Non-Java Files</td>
<td>49</td>
</tr>
<tr>
<td>3.5</td>
<td>The System Organization, Requirements and Execution</td>
<td>51</td>
</tr>
<tr>
<td>3.5.1</td>
<td>The System File Hierarchy</td>
<td>51</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Executing the System</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>SYSTEM EVALUATION</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Experiments Conducted on The System</td>
<td>56</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Experiments on booklist.xml Document</td>
<td>56</td>
</tr>
</tbody>
</table>
4.1.2 Experiments on hamlet.xml Document .......................... 62
4.2 Related Work-Like Implementation ................................. 66
   4.2.1 Comparing a One-to-One Correspondence Queries .......... 68
5 DISCUSSION AND CONCLUSIONS ................................. 72
   5.1 Supporting Attributes, Namespaces, and ProcessingInstructions . 72
   5.2 Supporting XML Documents that Contain Program Source Codes 73
   5.3 Text Database ........................................ 74
      5.3.1 Implementation ..................................... 75
   5.4 User Interface and Word Truncation ........................... 77
REFERENCES ............................................... 79
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Part of Booklist XML document.</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Indexing the document in Figure 1.</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>The syntax of query expression $E$.</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Forming regions for the <em>followed by</em> operator.</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>The BNF rules of xsrch query expression.</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Core classes inheritance hierarchy.</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>Inheritance hierarchy for expression and evaluation classes.</td>
<td>38</td>
</tr>
<tr>
<td>8</td>
<td>The file hierarchy of package algebra.</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>play.dtd for hamlet.xml</td>
<td>63</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

Textual documents have natural structure. Familiar structures may divide documents into books, chapters, sections, subsections, paragraphs, issues, volumes, and lines. Regular structures can not be assumed. On demand and continuous data processing create unlimited amounts of structured documents. Such documents may be more or less restricted in their structure, such as program source codes and HTML pages. Documents characteristics differ very greatly. A document may include a title, an author or editor, an abstract, a preface, a body, sections, quotes, and references. It is very necessary to be able to access the contents of such documents. Unix search tools, such as AWK [1] and grep [17] are of everyday use for searching structured text documents. Advent of the web, that acts as "a universal repository " of knowledge with free and inexpensive access, completely changed the areas of classical information retrieval and text databases search [5]. For all of this and other issues, there is a need for simple tools that make the documents’ structure available for accessing and updating their contents.

Text retrieval query expressions should be expressed in terms of the natural structure of the document being queried. Such natural structure is divided into text intervals that contains each other, such as lists within lists. In a Booklist document, we might be interested in the following queries:
1. Find all titles that contain the word “algorithms”.

2. Find the “Algorithms” books written by “Horowitz”.

3. Find the complete records of books written by “Horowitz”.

4. Find all titles and their authors that have been published in 2002.

5. Find all titles that are not published in 2002.

It is obvious that the results of these queries are also expressed in terms of the document natural structure. Their results will be titles, books or authors.

**Thesis Organization**

In this section, structured text documents and text-region sets are introduced. XML documents and the DOM model are also presented. The remaining sections of this report are organized as follows. Section 2 presents a system that is capable for handling text retrieval queries, for a single XML document, based on query algebra which is also presented. The implementation of the system and the query algebra operators are discussed in Section 3. Sample experiments performed on the system, along with two different implementation, are introduced in Section 4. Future works and plans that can be either incorporated into the system or added as layer modules are discussed in Section 5. The system API documentation generated via javadoc [16] are currently avialble at [2].
1.1 Document Indexing and Region Sets

A simple and powerful indexing scheme for manipulating structured text documents for text retrieval is to present the document as contiguous sequences of “concatenated symbols” [11]. Every word in the document should be indexed, except for a set of words that are most commonly used such as “for”, “and”, “the”, and “but”. This set of words is called stoplist. This ensures that the indexes, which are also called positions, are reduced so that words that are more relevant for the search criteria are indexed and referenced. This indexing scheme should capture the document structure. Indexing markup, tags specific to the document format, is a powerful approach for capturing document structure. In this way, it would be possible to access the structural elements of a document when formulating query expressions.

The symbols are drawn from the following alphabets: the text alphabet, the stoplist alphabet, and the markup alphabet, such that there are no common members among these sets. An index term, or index alphabet, is a word (or token) from the text and markup alphabets. Stoplist words are not indexed [11].

Figure 1 shows part of a booklist XML document. Indexing this document would attach positions to symbols from both the markup set, \{ \text{<author>, <book>, <first>, …, </author>, </book>, …} , and the text alphabet set, \{ 1978, algorithms, computer, ellis, fundamentals, horowitz, press, …\}. Symbols from the
stoplist alphabet, \{a, am, an, ..., of, ... \} would not be indexed. Figure 2 shows how this part would be indexed.

**Region Sets**

After Burkowski’s *Generalized Concordance Lists*, [7, 10] that refer to text intervals, it is more efficient to view the grouping of concatenated symbols as contiguous subgroups of text fragments. Such text fragments are called Generalized Concordance Lists (GC-Lists), [11], or text regions [13]. We will refer to these intervals throughout this text simply as regions.

Document text being modeled as a sequence of “concatenated symbols,” their associated positions have a finite universe \( U = \{0, 1, 2, ..., N - 1\} \). A non-empty
<book> <title> fundamentals of computer algorithms </title> 
34 35 36 37 38 39

<author> <name> <last> horowitz </last> <first> ellis </first> </author>
40 41 42 43 44 45 46 47

<author> <name> <last> Sahni </last> </author>
48 49 50 51 52 53 54

<author> <name> <last> Sartaj </last> </author> <publish> <year> 1978 </year> <publisher> computer science press </publisher> 62 63 64 65 66 67 68
</publish> </book>
69 70

Figure 2: Indexing the document in Figure 1.

interval of contiguous positions \{a, a + 1, ..., b\}, where 0 ≤ a ≤ N − 1, is a start position, and a ≤ b ≤ N − 1, is an end position, is called a region, and denoted by \([a, b]\). The notation \(r.\text{start}\) is used for the start position \(a\) of a region \(r = [a, b]\), and \(r.\text{end}\) for its end position \(b\). The power set \(P(U)\) is the set of all regions \(R = \{ [a, b]|0 ≤ a ≤ b ≤ N − 1 \}\). The length of a region \(r\) is \(r.\text{end} − r.\text{start} + 1\). The largest possible region, practically, would be the whole document, i.e. \([0, N − 1]\).

The smallest region is of length 1, that represents a single indexed term. The regions, in practice, could be lines, sentences, or any other kind that could fit within a document structural elements. Regions of a region set \(s \in R\) usually share some common property. For instance, they are at the same depth in the
document hierarchy, they are the same document’s elements type, or mostly, they are occurrences of the of a text pattern.

The universe $U$ forms the basis of the nested text-region algebra. A query algebra is needed for retrieving text region sets.

### 1.2 Structured Text-Region Algebra

An indexing function $I(p)$ for a text pattern $p$, where $p$ is an indexed term in our model, maps each symbol from a text alphabet and a markup alphabet into a set of positions, or regions, where that symbol occurs.

Query expressions for manipulating text regions for retrieval should be expressed in terms of the algebra operators and in terms of the document natural structure (the tagging scheme being used). These queries also expressed in terms of Boolean operators to select regions based on conditions on the regions relative containment and their ordering.

Since a region has both start and end position, regions can be compared, using usual set inclusion, and also they can be ordered according to either their start positions or their end positions. Regions in a region set can be classified to the following [13]:

Let $x = [x.start, x.end]$ and $y = [y.start, y.end]$ be two regions.

- **nested regions**: If $x$ starts as or after $y$ ($y.start \leq x.start$) and $x$ ends as or before $y$ ($x.end \leq y.end$), then the two regions are nested ($x \subseteq y$). If the
\[ E \rightarrow (E) \]
\[ | E \cup E | E \cap E | E \setminus E \] // Math set expressions
\[ | E \subseteq E | E \nsubseteq E \] // Containment expressions
\[ E \prec E \] // Ordering or "followed by"
\[ E \supseteq E \] // Extraction
\[ \text{hull}(E) \]
\[ \text{inner}(E) \]
\[ \text{outer}(E) \]
\[ I(p), \text{ where the pattern p is an indexed term} \]
\[ \{ [ \alpha_1, \omega_1 ], ..., [ \alpha_j, \omega_j ] \}, \]
where \( 0 \leq \alpha_i \leq \omega_i \leq N - 1 \) // Constant region set

Figure 3: The syntax of query expression \( E \).

Inclusion is proper, i.e. the two regions may share only one end point but not both, that is \( y.\text{start} \leq x.\text{start} \) and \( x.\text{end} < y.\text{end} \), or \( y.\text{start} < x.\text{start} \) and \( x.\text{end} < y.\text{end} \), then \( x \subset y \).

- **disjoint regions**: If region \( y \) starts after the end of region \( x \), \( x.\text{end} < y.\text{start} \), then the two regions are disjoint. And the set containing disjoint regions is called flat region set.

- **overlapping regions**: If two regions neither nested nor disjoint, they are overlapping.

The nested region algebra query expression, \( E \), is described by the grammar that appears in Figure 3.

The containment operators are divided into two groups, the first group is defined in terms of regular set inclusion (\( \subseteq \)) [11], and the second group is defined
with respect to proper set inclusion (⊂) [13]. Constant region set represents any fixed set of regions with fixed start and end positions. In our model, it allows to refer to the set of regions of “all” structural elements in a document. See Section 2.2.7.

1.3 XML Documents and The DOM Model

XML, the Extensible Markup Language [20], recommended by W3C (World Wide Web Consortium) in 1998, is a metalanguage that describes a class of data objects called XML documents, to integrate documents and data on the web. XML is a subset of SGML and it is more restricted form. XML document has both physical and logical structure. The physical structure of XML document made up of storage units called entities. The entities contain either parsed or unparsed data. An entity may refer to other entities. A document entity or “root” starts a document. The logical structure specifies the structural elements of a document; this structure is indicated by “explicit” markup, and it consists of declarations, comments, elements, processing instructions, and character references. For more details, please refer to the W3C XML recommendation [20].

The DOM Model Core

The XML recommendation partially specifies the interfaces of software that process the XML documents. Also one of XML design goals is that programs
could be easily written and developed for manipulating XML documents.

The Document Object Model (DOM) [18], is a standard that originated by the W3C. It is an Application Programming Interfaces (APIs) for well-formed XML and HTML documents, that allows programs to manipulate the content and the structure of the documents. DOM model is based on an object structure that precisely reflects the structure of the document it models. DOM uses the structure model to describe the document tree-like representation. DOM presents documents as a hierarchy of Element node objects. The DOM identifies these objects and interfaces, and their semantics, that used to represent and manipulate the document structure and content.

This is a very brief description of the DOM model core. The system that is described here is based on Apache Xerces project [4] implementation of DOM level 2 Core. Please refer to the W3C DOM specification [18]

Note: The Apache Xerces [4] implementation of the DOM model, keeps the document’s DOM implementation in the memory.
2 SOLUTION PROPOSED (System Overview)

A system, xsrc, has been developed for tackling the search problem within XML documents as a member of structured documents class. Xsrc is capable for indexing any well-formed, valid XML [20] document according to the indexing model described earlier, see Section 1.1. The system implements the Boolean operators on ordinary sets of text intervals: OR (set union), AND (set intersection) and NOT (set difference); and the nested region algebra operators that are based on text interval, GC-Lists, or text regions: the relative containment operators, the ordering operators (string concatenation), and region extraction.

The system can index any XML document that is well-formed and valid. First, it parses the document, using Xerces-2 for Java [4] DOM parser and creates a DOM model tree. The tree is kept in memory, and this ensures that the relationships between nodes are preserved. The DOM nodes hierarchy, with all elements and text contents, are mapped into an inverted list-like [15] data structure (map) of unique keys or identifiers, those that we refer to as indexed terms. The index map, inverted list, maps each index term in the document into a list of positions where that term occurs in the document's DOM nodes hierarchy. This is done and used by an indexer, to quickly identify an indexed term's position record. The position record keeps track of both the indexed term integer position within the document tree, that is being attached while traversing the tree, and a reference to the node
that it belongs to.

Indexing is done by creating an object of indexer for a document. The document can be located anywhere but with a valid URI. This means that the system can be plugged into a text database composed of XML documents, such that each document is well-formed, valid, and has a valid URI. And then it can create one indexer object, with its own DOM tree and its index map, for each XML document in the database. The system does not set any limits on the size of the document.

The query language of xsrch has three boolean operators, four containment operators and other four variants, one ordering operator, one extraction, and three unary operators. These operators are based on text intervals, or regions, data type. They take either one or two operands. The operands and the results all have only one data type: region sets. A region set could be formed from an argument, which is a text pattern. The argument could be an element name from the document tree element nodes, or it could be a text content of any child of an element. There are no limits on the length, overlapping, or nesting of region sets within a document.

2.1 Forming Regions

In Figure 2, the indexing function for the start tag, $I(<\text{last}>)$, returns the list of positions $\{42, 52\}$, where the symbol $<\text{last}>$ occurs in the document. The indexing function for the end tag $</\text{last}>$, $I(</\text{last}>)$, returns the list $\{44, 54\}$. A
text region, \( r \), takes the form \([r.\text{start}, r.\text{end}] = \{r.\text{start}, r.\text{start} + 1, \ldots, r.\text{end}\}\). The start tag \(<\text{last}>\) with position 42, its match end tag \(<\text{/last}>\), has the position 44. Thus, the corresponding region is the text interval \([42, 44]\). The text region for the end tag \(<\text{/last}>\) with the position 44, is also \([42, 44]\). For the text content “horowitz”, the indexing function, \( I(\text{“horowitz”}) \), will return the list containing only one position (43). And its text region will be \([42, 44]\). When forming a region set, the result will take the form \([[r.\text{start}, r.\text{end}], \ldots]]\). The region set of the start tag \(<\text{last}>\) is the same as for the end tag \(<\text{/last}>\), and it , would be be the region set \([[42, 44], [52, 54]]\). The region set for the text “horowitz” is \([[42, 44]]\). The regions for the element \(<\text{book}>\) and for the text “horowitz” are nested. The regions for the element \(<\text{title}>\) and for \(<\text{author}>\) are disjoint. The region set containing disjoint regions, e.g. \(<\text{title}>\) and \(<\text{author}>\), is a flat region set. The region for the element \(<\text{booklist}>\) is \([0, N - 1]\), has the maximum length, \(N\).

2.2 Nested Region Algebra Operators

Let \( U = \{0, 1, 2, \ldots, N - 1\}\) be the finite universe, where 0 is the first position in the indexed document, \( N - 1 \) the last index. Let \( R = P(U) \), be the set of all subsets of \( U \). And let \( A, B \in R \), be any two region sets.
2.2.1 The Ordinary Math Set Operators

The boolean operators, AND, OR, and NOT (complement with respect to another set), implemented in xsrch as ordinary math set operators: union, intersection, and difference, respectively. [11] refers to the AND as “Both Of” and to the OR as “One Of”. They take two region sets as their operand. The set union operator, OR, combines two region sets. The set intersection operator, AND, returns the common regions between two sets. The set difference returns the the regions in the first operand that are not members of the second operand. For any two region sets $A \in R$ and $B \in R$,

$$A \cup B = \{a|a \in Aora \in B\}$$

$$A \cap B = \{a|a \in Aanda \in B\}$$

$$A \setminus B = \{a|a \in Aanda \notin B\}$$

2.2.2 The Unary Operators: Outer and Inner

The unary operators outer and inner are utilized in the implementation of other binary operators in the nested region algebra of the xsrch query language. Both of them take only one region set as their operand.
The outer operator:

The outer of a region set $A$ is the outermost regions of $A$ [13]. The result is always a flat region set, the regions are disjoint. This means that there are neither nested nor overlapping regions in its result. Outer is defined by the formula

$$outer(A) = A \not\preceq A$$

The outermost regions of a complete XML document are composed of only one region, the region covered by the root element, that is the region $[0, N - 1]$. In Figure 2, the outermost regions of the elements inside, but not including, the $<book>$ element, are given by the set

$$\{[35, 39], [40, 49], [50, 59], [60, 69]\}$$

The inner operator:

The inner operator is the exact converse of the outer operator. For a region set $A$, the inner computes the innermost regions of $A$, [13]. The same as outer, the result is always a flat region set. Inner is defined by the formula

$$inner(A) = A \not\succeq A$$

For the Booklist XML document, the innermost regions of all elements (See Section 2.2.7 for the All elements), are composed of the regions covered by the elements $<title>$, $<last>$, $<first>$, $<edition>$, $<year>$, and $<publisher>$. For
the part in Figure 2, a subset of the result region set will be

\{ [35, 39], [42, 44], [45, 47], [52, 54], [55, 57], [61, 63], [64, 68] \}.

2.2.3 The Containment Operators

The containment operators select regions of the first operand based on conditions on their containment in the regions of the second operand. The result is a subset of the left operand. The right operand works as a filter that “filters out” the regions to be selected from the first operand. Two different groups have been implemented, based on either usual set inclusion (\( \subseteq \)), or set proper inclusion (\( \subset \)). The proper containment operators are defined below:

- A containing \( B \) : selects the regions of \( A \) that contain some region of \( B \),
  \[ A \triangleright B = \{ a | a \in A \text{ and } \exists b \in B \text{ such that } a \supset b \} \].

- A not containing \( B \) : selects the regions of \( A \) that do not contain any region of set \( B \),
  \[ A \not\supset B = \{ a | a \in A \text{ and } \neg \exists b \in B \text{ such that } a \supset b \} \].

- A contained in \( B \) : selects the regions of \( A \) that are included in some region of set \( B \),
  \[ A \subset B = \{ a | a \in A \text{ and } \exists b \in B \text{ such that } a \subset b \} \].

- A not contained in \( B \) : selects the regions of \( A \) that are not included in
any region of set $B$,

$$A \not\subseteq B = \{a | a \in A \text{ and } \neg \exists b \in B \text{ such that } a \subset b \}.$$ 

The containing and not containing operators make use of the inner operator to compute their results. The contained in and not contained in make use of the outer operator in the computation of the result.

The other four containment operators that are based on usual set inclusion [11], are the same as the above, but the condition in the above equation for region inclusion has to be replaced by $\subseteq$.

### 2.2.4 The Followed By Operator

The followed by operator produces a result region set that might not be a subset of its operands. In other words, the structure of the result might be different of that of its operands. This means that if the regions of the operands region sets are either nested or disjoint, the result of the followed by may contain overlapping regions. This is in contrast to the results produced by the boolean operators and the containment operators.

Text fragments in XML are determined by the elements start and end tags. Consider the regions in Figure 4. We consider the regions delimited by tags. We count them by the following order: [42, 44], [45, 47], [41, 48], and finally, [40, 49].

16
Figure 4: Forming regions for the followed by operator

We end up with the following structure:

\[ [40, [41, 42], [44, 45], [48, 49]] \]

The semantics of the followed by operator for forming the regions can be described as "matching delimiters together from inside out." The followed by operator is defined by the following formula [11]:

\[ A . B = \{ c \in R \mid c = [a.start, b.end] \text{ where } \exists [a.start, a.end] \in A \text{ and } \exists [b.start, b.end] \in B \text{ such that } a.end < b.start \} \]

but the order for selecting the regions \( c \) is important. And this means that [13] “result regions span from the beginning of a region of \( A \) to the end of the closest following unmatched region in \( B \).”

In general, the followed by operator works as follows:

- First, regions of \( A \) are sorted according to their start position total ordering and regions of \( B \) are sorted according to their end position total ordering.

- To compose the first region \( c \) of the result, it takes the first region of \( B \) with \( b.start \geq a.end \). Then \( c.end = b.end \), and \( c.start = a.start \). Thus
\[ c = [a._\text{start}, b._\text{end}] \]. This routine continues until one of the operands’ region set runs out.

As a consequence, it is obvious that each region of the operands may participate in the production of one result region or none. That is \(|A..B| \leq \min\{|A|, |B|\}\).

2.2.5 The Hull Unary Operator

The \textit{hull} is a function with \textit{cardinality} defined as \(\text{hull} : P(U) \rightarrow R\). It takes only one region set as its operand.

The hull of a region set \(A\) is a kind of \textit{reduction function} of the regions in set \(A\). It returns the minimal region set that covers the same positions of the regions in set \(A\) [13]. Recall that, a region \(r = [r._\text{start}, r._\text{end}] = \{r._\text{start}, r._\text{start} + 1, \ldots, r._\text{end}\}\).

The hull operator formula is given by

\[
\text{hull}(A) = \{[x, y] \in R \mid [x, y] \subseteq \bigcup_{a \in A} a \quad \text{and} \quad x - 1 \notin \bigcup_{a \in A} a \quad \text{and} \quad y + 1 \notin \bigcup_{a \in A} a\}.
\]

Similar to the unary operators inner and outer, the result of the hull of a region set is always a flat region set, the regions are disjoint. This means that there are neither nested nor overlapping regions in its result.

The hull of all elements, Section 2.2.7, of an XML document is the region covered by the \textit{root} element, that is the region \([0, N - 1]\). The hull of the regions in Figure 2, is the region covered by the \textless book\textgreater element, that is the region \([34, 70]\).
2.2.6 The Extract Operator

The implementation of the extract operator utilizes the hull operator. Therefore, there is some kind of reduction on the result of the extract operator, as generally stated by the hull operator. Similar to containment operators, the regions of the resultant region set of the extract operator are those that are covered by the left operand. The right operand works as a filter. The value of “$A$ extract $B$” is composed of regions of the positions covered by regions of $A$ by excluding any positions that overlap with any region of $B$. The definition of the extract operator is given by the formula, [13],

$$A \ominus B = \cup_{a \in A} hull(a \setminus \cup_{b \in B} b)$$

If the region set of the $<$publish$>$ element is extracted from the region set of the $<$book$>$ element in Figure 2, the resultant region set will be $\{[34,59],[70,70]\}$.

2.2.7 The “@all” All elements

This actually is not an operator of the algebra, but it is part of the query language of xqrch. @all is a kind of operand that means all elements in the document. For example, in Figure 2, if the booklist document contains only that $<$book$>$ element, then all elements, @all, is the region set

$\{[34,70],[35,39],[40,49],[41,48],[42,44],[45,47],[50,59],\,$

$[51,58],[52,54],[55,57],[60,69],[61,63],[64,68]\}$. 

19
2.3 Properties of Xsrch Query Expressions

The algebraic properties of the operators can be inferred from the previous discussion. The ordinary set operators, or the boolean operators, on text region sets are associative and commutative:

- \( A \cup B = B \cup A \)
- \( A \cap B = B \cap A \)
- \( A \cup (B \cup C) = (A \cup B) \cup C \)
- \( A \cap (B \cap C) = (A \cap B) \cap C \) and
- \( A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \).

The set difference is not associative:

\[
A \setminus (B \setminus C) \neq (A \setminus B) \setminus C
\]

The containment operators and the extract operator are neither commutative nor associative. For the containment operator, the right operand is a filter for the regions to be selected from the left operand based on the containment conditions. And it is very obvious for the extract operator that it is neither commutative nor associative.

The followed by operator is not commutative. From its definition, a region in the result of the followed by operator, its start delimiter considers the start
positions of the regions of the left operand, and its end delimiter considers the end positions of the regions of the right operand. But it is associative:

\[(A..B)..C = A.((B..C))\]

For the containment operators, \(\triangle, \triangledown \in \{\triangleright, \triangleleft, <, \triangleup\}\), the following commutativity holds

\[(A \triangle B) \triangledown C = (A \triangledown C) \triangle B\]

This means that the evaluation order of “containment conditions” is not important.

Other properties between the containment operators and the ordinary set operators can be derived. Clarke [10] discusses a group of such properties along with their proofs.

### 2.4 Xsrch Query Expressions

The BNF rules of xsrch query expressions are defined in Figure 5. The query expressions should be written carefully.

#### 2.4.1 Query Examples

The following examples are according to the Booklist XML document:

1. Find all titles that contain the word “Algorithms”.
   
   “<title> containing algorithms”
<expression> ::= <RegionSet>
    | ( <expression> )
    | <UnOp> <expression>
    | <expression> <BinOp> <expression>

<RegionSet> ::= <XMLStartTag>
    | <XMLEndTag>
    | <XMLEmptyElementTag>
    | <TextFragment>
    | @all

<BinOp> ::= <MathSetOp>
    | <ContainmentOp>
    | <OrderingOp>
    | <ExtractionOp>

<MathSetOp> ::= union OR
    | intersection AND
    | difference NOT

<ContainmentOp> ::= containing
    | notContaining lcontaining
    | containedIn
    | notContainedIn lcontainedIn
    | properlyContaining
    | notproperlyContaining lproperlyContaining
    | properlyContainedIn
    | notProperlyContainedIn lproperlyContainedIn

<OrderingOp> ::= followedBy | by | ..

<ExtractionOp> ::= extract

<UnOp> ::= inner
    | outer
    | hull

Figure 5: The BNF rules of xsrch query expression.
2. Find the “Algorithms” books written by “Horowitz”.

"(<book> containing (<title> containing algorithms))
intersection (<book> containing (<author> containing horowitz))"

3. Find the complete records of books written by "Horowitz".

"<book> containing horowitz"

This query returns a region set with one of its regions as shown in Figure 2.

4. Find all titles and their authors from the booklist that have been published in 2002.

"(<title> .. <author> ) .. 2002"

5. Find all titles that are not published in 2002.

"<title> extract ( <title> .. 2002 )"

Section 3.5.2 shows how to execute such query expressions.
3 IMPLEMENTATION

Xsrfch has been implemented as a Java package called algebra. Package algebra is a 100% pure Java. It is written in Java 2 Platform, j2sdk version 1.4.0, [16].

Package algebra, or algebra for short, supports the following DOM node types [18]:

- Document Node,
- Element Node,
- Text Node,
- CDATASection,
- Node,
- Entity Node,
- Entity Reference Node, and
- External Entity Reference Node.

All other node types are not supported. DocumentFragment Node type, as a valid XML elements, but not well-formed, since there is no root element, is already implemented through External Entity Reference.

There are 21 Java source files implementation in package algebra, divided as follows: 18 implemented as classes, two as interfaces, and one as exception. There
are also other source files of different types for the completeness of the package: stoplist.txt, build.xml, xsrch.sh, and package.html.

3.1 Facts, Behaviors, and Considerations

3.1.1 Entities

In the DOM interfaces there are no objects representing entities. Numeric character references and references to the predefined entities in XML are replaced by the single character that makes up the entity’s replacement. This is not true for the CDATA sections. See [18].

3.1.2 Apache Xerces2 for Java

The Apache xerces2 for Java, version 2.0.1 implementation of the W3C DOM model Level 2 Core has the following consideration regarding the whitespaces as the Node’s children:

Xerces seems to be DTD conscious, and this consideration is an accidental discovery. For example, consider the following XML Document Fragment:

```xml
<tagA>
  <tagB>
    ... some text contents ...
  </tagB>
</tagA>
```
If there is no DTD for this XML document, then xerces assumes the corresponding DTD as follows:

```xml
<!ELEMENT tagA (#PCDATA, tagB, #PCDATA)>
<!ELEMENT tagB (#PCDATA)>
```
as each newline "\n" character is a parsed character data child.

However, to ignore the whitespace nodes from the elements tagA and tagB, the following should be done:

1. a DTD is required with the following:

```xml
<!ELEMENT tagA (tagB) >
<!ELEMENT tagB (#PCDATA)>
```

2. the DOM feature “include-ignorable-whitespace” feature should be set to `false`,

### 3.1.3 Some Considerations for Parsing

**Empty Elements:**

Empty element takes start and end positions only. No further processing. For a *GenericIdentifier*, empty element has the form `<GenericIdentifier />`, will be indexed as both the *start tag* and the *end tag*: `<GenericIdentifier> </GenericIdentifier>`, and both of them take two consecutive positions.
Delimiters:

- Default Java delimiters: " \\n\\t\\r\\f", and punctuation marks also considered as delimiters: , ; : " ' " ! (, ), [ ] , ] , } } \rightarrow \{ comma, semicolon, colon, double quotes, single quote, tilde, exclamation mark, different parentheses styles \}

- The dot ".", if it is part of a number, e.g., 450.99, then it is part of the token. Otherwise, it is a delimiter and ignored.

More considerations for tokenization:

- The special characters @ # $ % ^ & * ? + = < > | / \ are tokens.

- Hyphen (-) and underscore (_) are part of the enclosing token if no space between two adjacent tokens (neither delimiters nor tokens), e.g. "shipping-address" or "shipping_address" each considered as one token. But if the hyphen or underscore exists with spaces before and after, it is a token, e.g. "shipping - address" is a three token string.

- The sequence of characters "@all" is reserved for referencing to all elements, see Section 2.2.7.

Note: All alphabetic characters inside the documents (actual text contents, CDATA section contents, tag names, etc.) are converted into lower case letters.
3.2 Java Source Files Built in Package Algebra

The source files are implemented as classes, interfaces, and exceptions. They are categorized according to their functions in the package. At [2], there is a full API documentation coverage of package algebra, generated via javadoc tool, [16].

Note: Indexed term, indexed token, index symbol, and index alphabet are references to the same concept, See Section 1.1, Indexing Scheme and Region Sets.

3.2.1 Indexing Classes Core

These are the core classes for implementing the indexing scheme and provide the necessary methods for constructing the text region sets.

> **DocumentContentFilter - Class**

This class implements the org.w3c.dom.traversal.NodeFilter interface [19]. More specific, it implements the acceptNode() method. It is used by the Indexer class to accept the supported node types and to filter out the other node from the DOM tree generated by the DOMParser. So that it creates a specific logical view of the DOM tree. This logical view is then traversed by a TreeWalker [19] implemented inside the Indexer class.
**PositionType - Class**

Every indexed term, as mentioned in Section 1.1, has an associated position. However, this position is actually a record with two components: the position itself and a reference to the node in the DOM tree hierarchy. An element start tag (indexed token), its node field component refers to that Element Node itself within the DOM tree logical view. The objects of this class form the base of this package.

A record of an indexed term has two fields:

- **position**: The integer position that is associated with the indexed term itself
- **node**: A reference to the Node. This reference is in the DOM tree logical view for which this indexed term belongs in the same view.

Notice that two position records are equal, *iff* their integer positions are equal and they belong to the same node. The latter condition is for assertion.

**Indexer - Class**

A class for indexing an XML document. This indexing scheme is as described in Section 1.1. The default, no-argument Indexer() constructor creates an empty `indexMap` instance field. The `Indexer(String)` constructor creates a new Indexer object. This constructor indexes an XML document that is passed as the String argument "uri" which is the URI of the XML document that is to be indexed.
Indexer works as follows

- The constructor `Indexer(String uri)` creates an empty `indexMap` instance field of Indexer object. This object invokes the `indexing Document(String uri)`. This method does the following:

  - sets a customized parser configuration, `parserConfiguration()`.

- parses the document, `parseURI(String uri)`, and loads the DOM tree into the instance field `document`,

- sets up the environment to create the document DOM tree logical view, `settingDOMTraversal()`, and sets up a `TreeWalker [19]` object to traverse the logical view. `settingDOMTraversal()` does the following:

  - creates a new `DocumentContentFilter` object. See `DocumentContentFilter` class above.

  - sets up the node types to be shown in the DOM tree logical view, by selecting the required flags that correspond to node type.

- The newly created `TreeWalker` object will be attached to the document DOM tree, starting from the document root element, then

  - sets up the required flags [19] to indicate which node types to be included in the logical view,
- plugs in the DocumentContentFilter [19] object, that also determines
  the node types to be accepted while doing the traversal, and
- sets up the ExpandEntityReference to true. So that any reference to
  an entity will be expanded in the DOM tree logical view.

This way, it will be guaranteed a very clean logical view of the DOM tree.

- It extracts the node types from the DOM tree logical view, attach position
  records for the indexed terms, attachPositions(TreeWalker).
- attachPositions(TreeWalker) method attaches positions while doing a
  recursive traversal of the DOM tree. It has three steps:

  1. When it encounters an Element node, it starts processing the cur-
     rent node and invokes nodeStartActions(TreeWalker). The method
     nodeStartActions(TreeWalker) attaches positions based on the node
     type as follows:

     (a) if the node type is a Node.ELEMENT_NODE, it composes a key
        that equals the start tag, key = <TagName>, by prefixing and
        suffixing the element node name by the start “<” and end “>” tag
        symbols. It creates a new PositionType object by assigning a new
        position and a node reference. This will be the value of the key.
        The key, along with its value are mapped into the indexMap, by

  31
invoking `adjustIndexMap(String key, PositionType).

(b) If the node type is either `Node.TEXT_NODE` or `Node.CDATA_SECTION_NODE`, then it invokes `processTextContents(Node).` Only the Node is passed, since it carries both the text as the node value, `Node().getNodeValue()`, and itself is the reference of any contents of the Node’s text value. This method does actual parsing for identifying the text contents and their semantics. And the rest are as described above with an indexed token.

(c) If the node type is either `Node.ENTITY_NODE` or `ENTITY_REFERENCE_NODE`, both of these cases have been taken care of when invoking the `parserConfiguration()` and `settingDOMTraversal()` methods.

2. Recursively invokes `attachPositions(TreeWalker)` on the children of the current Element node.

3. Whenever `attachPositions(TreeWalker)` finishes processing an element, recursively, and before transferring to the next sibling, it closes it session be calling a method `nodeEndActions()`.

- While doing this traversal, it fills up an instance field of type List, `documentList`, with all indexed tokens and their position records. This list will be used later with the `Extraction.extract()` operator.

32
Region - Class

A class that represents a mathematical closed interval that represents a text region set. This class has four fields, two fields represent the start-delimiter and end-delimiter tags: alpha and omega. The other two fields are class fields that represent start and end position total ordering.

Recall that a region \( r = [r.start, r.end] \), implemented here as \( [r.alpha, r.omega] \). The START_POSITION_ORDER and END_POSITION_ORDER are class variables. Both of them implement the Comparator.compare (Object, Object) method, and they provide sorting regions in a region set (implemented as either Set or List) by the their start position and their end position total ordering, respectively.

This class has methods for creating a new Region object, either empty, Region(), or constructed with the two position records passed as the arguments, Region(PositionType start, PositionType end). It also implements equals(Object) to compare two regions for equality. Two Region objects are equal iff their start (alpha) and their end (omega) position records are equal. See PositionType class.

Region class has two class methods for sorting a region set according to their start and end positions total ordering, respectively, by conforming to the general contract of the START_POSITION_ORDER and END_POSITION_ORDER
class variables. The methods are: `sortByStartPosition()` and `sortByEndPosition()`.

This class also implements two class methods for usual regions (intervals) inclusion. Method `containedIn(Region a, Region b)` for the usual set inclusion (⊆), that may share the two end points in their inclusion. And method `properlyContained(Region a, Region b)` for regions proper inclusion (⊂), that may share only one end point in their inclusion.

▷ **IndexingQueries - Interface**

This interface provides the necessary methods that are needed to complete the remaining process of indexing the document and makes the necessary constructions for the remaining of the package algebra.

In fact, this interface is for convenience than necessity. It has been written for two reasons. The first is to be included as part of the external APIs of the package algebra when it is needed to be used with other third parties applications. The second reason is regarding to multiple inheritance of class design. See Section 3.3.

▷ **QueryIndexer - Class**

This class extends (child of) Indexer class and implements `IndexingQueries` interface, for multiple inheritance. The inheritance hierarchy of these core classes of package algebra is demonstrated in Figure 6.
3.2.2 Query Algebra Operators Classes

These classes are the implementation of the nested-region query algebra operators. This group of classes are implemented as final classes, they can not be instantiated.

> **MathSet - Class**

This class that implements the boolean operators, has a set of methods that implement the ordinary set group of the nested text-region algebra operators: \texttt{union(Set, Set)}, \texttt{intersection(Set, Set)} and \texttt{difference(Set, Set)}. The corresponding boolean operators are: \texttt{OR}, \texttt{AND}, and \texttt{NOT} (the complement with respect to another set), respectively. The arguments to these methods are \texttt{Set}
objects. The Sets passed are of any Set class implementation: `HashSet()` or `TreeSet()`, as well as of any interface: `Set` or `SortedSet`.

> **Containment - Class**

This class implements the `Containment operators` group of the nested text-region algebra. It also implements the `unary operators outer` and `inner`. It implements the two variants of the containment operators, that are based on usual containment and the proper containment. See `Region` class. It has six methods, two for the unary operators outer and inner: `outer(SortedSet)` and `inner(List, boolean proper)`.

The other four methods for the eight containment operators: `containedIn`, `not-ContainedIn`, `containing`, and `notContaining`.

Each method has the following signature and header templates:

```
SortedSet containmentOpName(SortedSet, SortedSet, boolean proper)
```

where the boolean argument `proper` indicates if the containment is “proper” (`proper` is `true`), or usual (`proper` is `false`).

For the method `inner(List, boolean proper)`, when it is invoked by the containment operators, then the argument `proper` is indicated by the caller method. And when invoked as unary operator in a query expression, then `proper` is always has the value `false`, by the definition of the `inner` operator. See Section 2.2.2. Sets
passed as arguments, must be sorted according to their start position order. See Region class.

▷ Ordering - Class

This class implements the Ordering operator of the nested text-region algebra operators. The class has only one method `followedBy(Set, Set)`. The arguments for the this method could be of any `Set` class implementation: `HashSet()` or `TreeSet()`, as well as of any interface: `Set` or `SortedSet`.

▷ Extraction - Class

This class implements the Extraction operators group of the nested text-region algebra operators. The class has two methods: `hull(SortedSet)`, for the unary `hull` operator, and `extract(Set, Set)`, for the `extract` operator. The argument for the `hull()` method must be of type `SortedSet` (either `SortedSet` interface or `TreeSet()` class implementation). While the arguments for the `extract` method could be of any `Set` class implementation: `HashSet()` or `TreeSet()`, as well as of any interface: `Set` or `SortedSet`.

3.2.3 Query Expression and Evaluation Classes

These classes are divided into three groups: the first group is for parsing the query expressions. The second, for evaluating the query expressions, and the third
Figure 7: Inheritance hierarchy for expression and evaluation classes.

is an exception. Figure 7 shows the inheritance hierarchy among the Expression interface, the ExpNode abstract class, and the implementing classes.

▷ Expression- Interface

This interface specifies the constants that are required by the nested text-region algebra operators as short constants, to identify the semantics of the different tokens. So that these operators can be easily referenced by upper-case names wherever they are required. Also, this class is designed so that any newly added operators, that conform to the Nested Text-Region Algebra, can be easily added to this package. The same is applicable for any node types and features
that are not supported by the current implementation.

Constant values in the following ranges are reserved. Non-used constant values are also reserved, by this interface, for any future releases:

000 .. 099 – for tag-related tokens
100 .. 199 – for operators
200 .. 300 – for text fragments and others

Note: Parenthesis are actually not part of operators, but treated as part of operators, since they are used for grouping sub-expressions and operator precedences.

Expression is designed as interface so that its list of constants are made available for the implementing classes, and to allow for multiple inheritance See Section 3.3.

> ExpressionToken - Class

This class implements Expression interface. It defines the semantics of the tokens generated from parsing the command line arguments supplied as a nested text-region query expression. It is mainly used by ExpressionParser class that does the parsing process of the expressions.

The default, no-argument constructor, ExpressionToken(), constructs a new ExpressionToken object with meaningless values, to avoid the bugs that could be the results of referencing null objects and empty objects as well.
The copy constructor, `ExpressionToken(String word, short type)`, constructs a new `ExpressionToken` object. It has two arguments: `word` of type `String` is the parsed token from the query expression, and `type` of type `short`, to keep track of what is the type of token, its semantics, based on the constants defined in Expression interface.

`ExpressionToken` class has several boolean, class methods to test the type of a parsed token, based on token semantics, some action will be taken care of in `ExpressionParser` class.

> **ExpressionParser - Class**

It implements the `Expression` interface. `ExpressionParser` class parses the command line arguments supplied as a nested text-region query expression, using recursive-descent parser, according to the BNF rules given in Section 2.4. It defines the grammar rules of the syntax of the tokens that compose the query expression through class variables. It fills up the parsed tokens, extracted from the command line argument, in the instance field List object `tokenList`, by invoking the method `fillUpTokenList(String)`. The `String` argument of this method represents the query expression. See `Help` class and `Engine` Class. There are other utility methods in the class.

Note that the Backus-Nor Form, BNF, rules of the regular expressions for `LETTER`, `DIGIT`, `NAMECHAR`, `NAME`, `STAG`, `ETAG`, and `EMPTYELEMTAG` are subset
of their respective definitions at W3C XML 1.0, second edition [20].

The regular expressions here are according to the package java.util.regex. Pattern in j2sdk 1.4.0 [16, 8].

- **ExpNode - Abstract Class**

  Evaluation of a query expression is performed using some kind of binary tree implementation, expression trees. See Section 3.3.

  ExpNode is an abstract class representing any node in an expression tree. It has three concrete node subclasses. Text-region expressions and constant region expression, are presented as expression trees, whose leaves stand for calls of the method Engine.IndexingFunction(ExpressionToken), \( I(p) \) for some text pattern \( p \). Internal nodes of the expression trees are labeled by the binary and unary operators of the Nested Text-Region Algebra. Unary operators nodes have only one child sub-expression tree, the only one operand.

  ExpNode, and all its subclasses, have two methods: value() and printStackCommands(). In this class they are not implemented at all, they are defined to be abstract. Since ExpNode is just a concept of a node in the expression tree.

  The value() method represents the value of the node which is the region set that it contains. The method printStackCommands() is for convenience and tracing results for testing while evaluating the query expression. It shows the Stack
status for what commands (operators and/or operands) have been performed.

▷ ConstNode - Class

This is a subclass of ExpNode. It represents an expression node that holds a region set as its value(). It represents the leaves of the expression tree that stand for calls of the Engine.IndexingFunction(ExpressionToken), I(p) for some text pattern p.

This class has a copy constructor, ConstNode (Set r) that constructs a ConstNode object containing the region set of its passed argument, the region set r. The value() and printStackCommand() methods are as specified by the ExpNode parent abstract class.

▷ BinaryOpNode - Class

This is a subclass of ExpNode and implements Expression interface. It represents an expression node that represents the internal nodes of the expression tree. This set of internal nodes represents the set of binary operators of the nested text-region algebra.

The class has three instance fields:

- **left**: of type ExpNode, the expression for the operator's left operand,

- **right**: of type ExpNode, the expression for the operator's right operand,

and
• op: binary operator.

This class has a copy constructor, BinOpNode (short op, ExpNode left, ExpNode right) that constructs a BinOpNode object containing the specified data as its passed arguments. The value() and printStackCommand() methods are as specified by the ExpNode parent abstract class.

UnaryOpNode - Class

This is a subclass of ExpNode and implements Expression interface. It represents an expression node that represents the internal nodes of the expression tree, where a node of this type has only one child. These type of nodes in the expression tree represent the unary operators that are added to the nested text-region algebra.

UnaryOpNode has two fields

• operand: of type ExpNode, the expression for the unary operator.

• op: the nested text-region algebra unary operator.

UnaryOpNode class has a copy constructor, UnaryOpNode(short op, ExpNode operand) that constructs a UnaryOpNode object containing the specified data as its passed arguments. The value() and printStackCommand() methods are as specified by the ExpNode parent abstract class.
➤ **ParseError - Exception**

An exception that is thrown if there is an error while parsing the command line arguments supplied as a query expression. If there is an error, it exits with an error status.

### 3.2.4 Utility Classes

These are classes that are helpful for the package algebra.

➤ **NodeWriter - Class**

This class recursively writes the contents of a node to the standard output. It has one class field, DEFAULT_CANONICAL, and two instance fields: fCanonical and fOut. The DEFAULT_CANONICAL field indicates whether the output should be in *canonical format*. Its value is set to `false`, so the output is not restricted in that format. It has been included for any future requirements of the output.

The main method is `printNode(org.w3c.dom.Node node)`, which sets up the NodeWriter’s object, the encoding "UTF8", the canonical attribute, and finally calls the method `write(Node)` to recursively write the contents of the node.

To include the contents of regions in the result of a query expression, the option “-n” should be included in xsrch command line.
Help - Class

This class contains help methods. It contains the `printUsage()` message. See Section 2.4 (Xrch Query Expression). This class cannot be instantiated. It is called from within the Engine class and the Indexer class.

3.2.5 The Engine Class

This is the query engine of package algebra. The Engine is a subclass of its parent QueryIndexer, which also is a sub-subclass of Indexer class. See Figure 6.

Engine has an instance field `tokens`, which is an array of region-text query expression tokens, `ExpressionToken[]`, that have been parsed from the command line argument.

The one-argument constructor, `Engine(String commArgs)`, has all the command line arguments. The first argument, index zero, is the complete URI of the XML document to be queried, and then followed by the query that must be quoted either with a single or double quotes. See Section 2.4 (Xrch Query Expression).

Engine calls its parent passing to it the URI of the document, QueryIndexer (uri). QueryIndexer does its job by calling its parent with the URI, `Indexer(uri)`. Engine then parses the query expression by invoking the `ExpressionParser`, and it returns back the result of the parsed query tokens in its instance field `tokensList`, which is a List object.
It evaluates the query expression as follows: It creates an object of itself, this would index the document (argument at index 0), and parses the query expression (the second argument). Then it evaluates the expression by invoking the method expressionTree() applied on the instance object it has just created.

Engine class has the IndexingFunction(ExpressionToken) method that does the regional set indexing, by invoking the in-positional methods inherited from QueryIndexer class.

3.3 Class Design Objectives and Performance Issues

This section illustrates the reasons behind the class design structure that have been chosen. It also discusses some implementation of data structures that are used within the package and within classes.

3.3.1 Indexing Process

Indexer class is the absolute parent of the indexing process. It has been designed this way, for a pure object-oriented (OO) reasons. The document to be indexed is done by one and only object, an instance of Indexer class. Thus the DOM tree, the document object, and its associated indexMap, that keeps track of the indexed terms and their associated positions, are kept inside that object.

The indexing can be expanded easily to include new node types that are not supported by the current implementation, e.g. including the elements attributes
in the indexed document. This can be done by modifying the following:

The DocumentContentFilter class, thus the corresponding object inside Indexer, the flags inside settingDOMTraversal() method, and finally, the appropriate modification inside the methods attachPositions(), nodeStartActions(), and nodeEndActions(), if required. These changes depend on the node type to be included.

IndexingQueries interface, is an absolute interface that does not expand any higher level interface. The reason for this interface, could be:

- If a third party program needs to include package algebra in his own applications. There are two choices: either using the current implementation, that are already implemented inside QueryIndexer class, or he must implement his own, depending on the core indexing queries he needs. For instance, forming the text region sets as to contain only the text itself, and not the inclusion element.

- Multiple inheritance, as an object-oriented design choice. Again, for any third party, to use his own implementation, so that any modification that are not related to the indexing, are done external to Indexer. This way, all allowed methods and data from Indexer class are accessible, in addition to the new modification that are down in the hierarchy of both Indexer class and IndexingQueries interface.
**QueryIndexer**, which is a subclass of Indexer, and an implementation of IndexingQueries, somehow acts as a third party application discussed above. **Engine** is the leaf class in the hierarchy. As its name suggests, it is the engine of the system. It has the methods calls for organizing the way the system should behave.

**Region** class provides the methods that are necessary for dealing with regions and collection of region, e.g. sets. Region class depends only on the position records, **PositionType** objects.

The **nested-region algebra operators** are implemented as final classes. Their implementation, mostly depend on the Region class. This way, they can be used with any other implementation of indexing and region sets as long as these implementation are conformed with the indexing scheme and region sets in Section 1.1.

The design of the **expression and evaluation classes** also allows to add any new kind of token structure to algebra, e.g. new operator or other kinds of tags or operands. First, by assigning a short constant, that is not in use, to that new token in Expression Interface. Make the required code addition in both ExpressionToken and ExpressionParser. Finally, if it is an operator, then depending on its cardinality, unary or binary, so the its implementation can be called from within either UnaryOpNode or BinOpNode class. If it is not an operator, then only its pattern case after determining its exact semantics, should be added to the **Engine.IndexingFunction(ExpressionToken)**.
3.3.2 Some Choices of Data Structures Implementation

Package algebra implementation extensively utilizes the Java Collection Framework [9, 21, 8]. The inverted-list data structure has been implemented as SortedMap, which is a red-black tree based design. The search through this map is guaranteed to have complexity of \( \lg(n) \), where \( n \) is the total number of the indexed terms in the DOM tree logical view.

A lot of data structures that have been chosen are objects of interfaces Set or List. List is more efficient for addition and removal of objects than the other corresponding data structures, e.g. arrays. Whenever it is possible, an ArrayList implementation was chosen.

The Set interface, sometimes used as the data structure of some objects without specific implementation. HashSet implementation allows searching and insertion in constant time. SortedSet interface and the corresponding implementation TreeSet, which is a red-black tree based, allows dictionary operations to be done in \( \lg(n) \).

3.4 Other Non-Java Files

This section illustrates other, none Java files, that are used by package algebra.

stoplist.txt: The stoplist file “stoplist.txt”, contains the set of stoplist words, such as: “the”, “but”, and “for”. If Indexer class, before doing the traversal on the
logical view of the DOM tree, could not open this file, then it considers the stoplist words as indexed tokens and continues the indexing process. This file should be located in the same directory where the Java source files are. According to the current design, it should be located in algebraSrc directory. See Figure 8.

**build.xml:** This file is a kind of “make” file that is based on Ant, a java-based build tool [3]. In this file, the following targets are included:

- Initialization “init”,
- compilation “compile”,
- running the program, “execute”, which is based on some specific arguments, and
- generating javadocs "apidocs", using the standard doclet.

**xsrch.sh:** is a Unix shell script file. There are two ways to execute the system through the command line: using the java command [16] or using the xsrch as a command line. See Section 3.5.2.

**package.html:** A file that contains general description of package algebra. It is for generating the API documentation using the javadoc tool. It is included with the other source file, as a requirement for Javadoc tool.
3.5 The System Organization, Requirements and Execution

This section presents the organizational file structure of the system, the requirements for executing the system, and how to execute the system on the command line.

3.5.1 The System File Hierarchy

The files in package algebra are organized as shown in Figure 8:

```
./xmlquery - This is the base directory “basedir” \(^1\) of both xsrc and package algebra.

/algebraSrc - This source directory “sourceDir” that contains the Java source files, the stoplist.txt, and package.html files

/classes - The destination directory, “destDir” that contains a subdirectory algebra that contains the compiled, bytecode class files.
```

\(^1\)These quoted names: “basedir”, “sourceDir”, and “destDir” are the file names used in file build.xml.
build.xml - The Java-based ant build file.

xsrch.sh - The shell script file, which is the command line executable system.

3.5.2 Executing the System

The system was developed on Linux Box (RedHat 7.1, Kernel 2.4.2-2), and also has been tested on Microsoft Windows 2000. It was developed using the following softwares:

1. Java 2 Platform, j2sdk, version 1.4.0, [16]. Required.

2. Xerces 2 for Java, 2.0.1 Release, [4]. Required.

3. Ant build tool, 1.4.1 Release, [3]. Optional, needed for the build only.

To compile the sources using ant build file tool, i.e. to use the file build.xml, it is required to have the Jakarta Ant build tool installed, or accessible, on the machine running the system. Otherwise, it is required to compile each “.java” source file alone and put the “.class” files in subdirectory “classes” as shown in Figure 8.

To follow this way, it is required to avoid the file dependencies, by keeping in mind that every compiled file must be located in “classes” subdirectory before compiling the next one. The following sequence of file names must be followed when compiling the sources without the build.xml:

Before executing the system, the files: xercesAPIs.jar and xercesImpl.jar, must be in the classpath of the JVM. The system can be executed on the command line only. To execute the system, there are two ways, assuming the current working directory is the “basedir”:

- Using the java command:

  java algebra.Engine <the_uri_of_the_XML_file> <query_expression>

- Using xsrch: (Unix-systems only\(^2\))

  xsrch <the_uri_of_the_XML_file> <query_expression>

where

<the_uri_of_the_XML_file>: is the location of the XML document to be queried, which could be local on the machine, on a LAN, or any other valid URI.

<query_expression>: The query. The syntax of the query expression is given in Section 2.4.

\(^2\)For MS Windows systems, a file "xsrch.bat", the exact analogue for "xsrch.sh", must be written.
Notes about the query expression syntax in Section 2.4:

- The syntactic categories appear on the right-hand side of the BNF rule for 
  `<RegionSet>` given by:

  `<RegionSet> ::= <XMLStartTag> | <XMLEndTag> | ...`

  are subsets of their corresponding definitions in XML 1.0, see Section 3.2.3,
  ExpressionParser class.

- To include in the query one of the operators as a regular text pattern, i.e. to 
  escape the the meaning of the operators, such as `union`, `outer`, or `difference`, 
  the escaped operator must be prefixed and suffixed with double quotes. For 
  example, the query `xsrc`

  `xsrc <booklist.xml_uri> "<title> containing \"difference\""`  

  returns a region set, as its result, of all book titles that have the word 
  “difference”.

54
4 SYSTEM EVALUATION

The current implementation of the system, xsrch, supports text pattern matching that are case insensitive. For an indexing function $I(p)$, for a text pattern $p$, regions are formed according to every occurrence of $p$ in the DOM tree as described in Section 2.1, and formally described here:

- If $p$ is the name of an element node, such as $<\text{TAGID}>$, the returned region $r$ will be $r.\text{start} = I(<\text{TAGID}>)$, and $r.\text{end} = I(</>\text{TAGID}>)$

- Otherwise, where $p$ is a text fragment, the region will be the one that corresponds to the parent element, the one that includes the occurrence of $p$.

Forming regions according to the preceding strategy is already implemented in the system. Xsrch utilizes the nested elements feature found in XML. Therefore, to form a region for a single indexed term, there is no need to use the "followed by" operator, as the expression “$I(<\text{TAGID}>) .. I(</>\text{TAGID}>)$”. Thus, the approach implemented here is different than that found in the literature [11, 10, 13]. The latter approach is the one that the indexing function, $I(p)$, would return only the positions of where $p$ occurs in the document. For example, if $p = <\text{TAGID}>$, then the start position of the regions in the region set have the same as the end position. This alternative is discussed in more detail in Section 4.2.

The results found in this section are based on the following environment
**Operating System:** Linux - RedHat 7.1

**Kernel version:** 2.4.2-2

**Processor:** Intel SpeedStep 750 MHz

**RAM:** 128 MByte

**Software used:**
- Java 2 Platform, ver. 1.4.0,
- Xerces2 for Java 2.0.1 Release. See section 3.5.2.

### 4.1 Experiments Conducted on The System

The system has been tested on several XML documents. This section introduces samples of experiments that were performed on two XML documents: `booklist.xml` and Shakespeare work, `hamlet.xml`.

#### 4.1.1 Experiments on booklist.xml Document

The query expressions presented in section 2.4.1, are evaluated here:

1. *Find all titles that contain the word “Algorithms”.*

   
   "<title> containing algorithms"

   Each region in the result of the `<title>` expression, i.e. $I(<title>)$, will return all the regions covered by this element, thus building a region set of titles. Each region in the result of the text fragment `algorithms` exactly delimits a region
covered by the inclusion, parent element. Then the regions returned by the latter operand are used to select the title regions. The result of this query expression is

1. [[35: title], [39: title]]

2. [[118: title], [121: title]]

This result interpreted as follows: every region is given by the formula

$$[r.start, r.end]$$

where each component is a record to the associated position. The first region [[35: title], [39: title]], is a text region, The first component, \( r.start = [35: title] \), is the record of the region start position, the integer value of 35 is the position attached, and the string title is the associated node reference in the DOM tree, which is the element \(<title>\). The second component of the region, \( r.end = [39: title] \), is the record of the region end position. The second region, [[118: title], [121: title]] is interpreted in the same way.

When applying the option "-n"\(^3\) on the command line, which means print out the node contents of the result region set, we get, in addition to the region set obtained above, the following:

1. \(<title>\) Fundamentals of Computer Algorithms \(</title>\)

2. \(<title>\) Introduction to Algorithms \(</title>\)

\(^3\)This works only with \texttt{xarch} command line.
The first <title> element corresponds to the first region set, and the second <title> corresponds to the second region set in the booklist document.

2. Find the "Algorithms" books written by "Horowitz".

"(<book> containing (<title> containing algorithms )) and
(<book> containing (<author> containing horowitz ))"

The subexpression "<title> & algorithms" would return all regions of <title> that have the text fragment "algorithms". The left operand of the intersection operator, boolean AND, "(<book> & (<title> & algorithms))" will return all <book> elements with titles that have the word "algorithms". Similarly, the second operand of the intersection operator: <book> & (<author> & horowitz), that returns all books written by "Horowitz". Finally, when applying the intersection operator on the two region sets, the result returned would be the algorithms books written by Horowitz. The result of the query is

1. [[34: book], [70: book]]

This region set, which contains only one region, can be interpreted as in the first query. This region contains the complete <book> element. When applying the option "-n", the complete book listing will be printed out, which is exactly the same as the sample given in Figure 1.
3. Find all titles that have been published in 2002.

"<title> .. 2002".

The result, along with the contents:

1. [[172: title], [193: year]]

<title> The Java Developers Almanac 1.4, Volume 1 </title>

<year> 2002 </year>

The resultant region set can be formulated as

{[[r.start.position : title], [r.end.position : year]], ...}

Recall that a region in the result of the followed by operator starts from a region in the left operand and extends into the following matching region from the right operand. This interprets the formula, which says the region start position is from element <title> and its end position is from the element <year>. The end position also could be of text fragment.

For the whole query expression, a region from its result, will start from element <title> and will extend into a region from a text fragment, #text. In our example, it is “2002”, that is the contents of element <year>.

A second point worths to be mentioned is that the contents of the region [[172: title], [193: year]] covers all the positions from 172 to 193, inclusive. But when displaying the contents, the relevant are only displayed, that is the regions covered by the <title> element and the regions covered by the inclusion
element of the text fragment, “2002”, which is the \texttt{<year> element}.

4. \textit{Find all titles that are not published in 2002.}

\texttt{<title> extract ( <title> .. 2002 )}. 

Once again, the subexpression \texttt{“<title> .. 2002” returns all regions of titles of books that are published in year 2002. That is a region in the result of this subexpression starts from <title> and extends to the text fragment “2002”.

The left operand of the extract operator \texttt{<title>} returns all book titles from the booklist. Extract operator has a similar behavior as the containment operators, that is to filter out from the result some positions that satisfy some certain conditions. In this query, book titles that are published in the year 2002. The result of the above expression

1. [[2: title], [10: title]]
\texttt{<title> The Art of Computer Programming, Volume 3, Sorting and Searching </title>}

2. [[35: title], [39: title]]
\texttt{<title> Fundamentals of Computer Algorithms </title>}

3. [[72: title], [77: title]]
\texttt{<title> Compilers, Principles, Techniques, and Tools </title>}

4. [[118: title], [121: title]]
<title> Introduction to Algorithms </title>

5. [[192: title], [199: title]]

<title> A practical Introduction to Data Structures and Algorithm Analysis </title>

The same regions could be obtained by the query

"<title> extract ( <book> .. 2002 )".

Whereas the query "<book> extract ( <book> .. 2002 )" would return all the book listings except those that were published in year 2002. For testing purposes, this <book> element is stored in file j_almanac.xml as a valid document fragment, and is referenced in the booklist.xml as an external entity reference.

Miscellaneous Examples

To retrieve all elements in a document, the operand @all, Section 2.2.7, does this job. The outermost and the innermost regions of a nested region set are sometimes worthy to be selected. For example, to retrieve the root element of a document, the query expression

“outer ( @all )”

does this. For a document, say macbeth.xml, with root element <PLAY>, this expression would return [[0: PLAY], [20554: PLAY]]. The same result could be obtained by using the hull operator:
“hull ( @all )”

The hull operator is useful when a region set contains nested and/or overlapping regions. It returns a flat region set. The hull operator functions as a concatenation of region sets.

4.1.2 Experiments on hamlet.xml Document

The Shakespeare’s play, hamlet.xml, is a bit big file, its size is approximately 288 KByte. Shakespeare’s works as well-formed XML documents for testing XML tools are available at [6].

The play hamlet.xml is located at http://www.cs.nmsu.edu/~hsheboul/hamlet.xml, for testing purposes. The play.dtd is given in figure 9.

Since the play is moderately big, a preselect query expressions are demonstrated here.

1. Find the characters of the play.

It might be interested to find out the titles of the play, this could be expressed as "<title>". Also it might be of interest to someone else to find out the characters of the play, <persona> elements, the query expression

"<persona>"

would result in 26 <persona> elements, the characters. The first 3 of are listed:
Figure 9: play.dtd for hamlet.xml
1. [[62: PERSONA], [66: PERSONA]]

   <PERSONA> CLAUDIUS, king of Denmark. </PERSONA>

2. [[67: PERSONA], [74: PERSONA]]

   <PERSONA> HAMLET, son to the late, and nephew to the present

   king. </PERSONA>

3. [[75: PERSONA], [79: PERSONA]]

   <PERSONA> POLONIUS, lord chamberlain. </PERSONA>

2. Find text fragments that contains “Claudius” and “Hamlet”

"claudius intersection hamlet”

1. [[1782: STAGEDIR], [1795: STAGEDIR]]

   <STAGEDIR> Enter KING CLAUDIUS, QUEEN GERTRUDE, HAMLET, POLONIUS,

   LAERTES, VOLTIMAND, CORNELIUS, Lords, and Attendants </STAGEDIR>

   In this expression, there is no ordering specified. Also, the intersection operator

   could be replaced by the boolean AND.

   For some text pattern, if it is unknown, whether it is a text fragment or an

   element, for instance, the word “witch”, the query expression

   “witch or <witch>”

   would give

1. [[1661: LINE], [1670: LINE]]

   64
<LINE> No fairy takes, nor witch hath power to charm, </LINE>

Thus, it would be understood that the text pattern “witch” occurs in the document hamlet.xml as a text fragment. The boolean OR could be replaced by union operator.

The above query could also be expressed as

“<line> containing witch”.

3. Find the lines where the word “hamlet” is spoken by a lord.

To illustrate things, we need to divide the expression into subexpression, from bottom up, as follows:

exp1 ≡ <speaker> \ lord, specify speakers who are lords.

exp2 ≡ <speaker> .. <speech>, links speaker and speech.

exp3 ≡ <speech> \ hamlet, specify speakers that contain the word “hamlet”.

exp4 ≡ exp3 ⊕ exp2 ⊕ exp1, and the expression

Q1 ≡ hamlet ⊕ exp4, will select all text fragments where the word hamlet is spoken by a lord.

Finally, the expression <line> \ (Q1) will select the the specified lines. This could be expressed as the full expression:

"<line> containing ( hamlet containedIn ( ( <speech> containing
hamlet) containedIn ((<speaker> .. <speech>) containing
(<speaker> containing lord )) )
"
The result is
1. [[2303: LINE], [2312: LINE]]

<LINEx> But now, my cousin Hamlet, and my son,-- </LINEx>

When the usual containment operators are replaced by the proper containment
operators, e.g. containing is replaced by properlyContaining and so on, the
result is an empty region set. This points out that some regions of region subsets
of different subexpressions are equal.

4.2 Related Work-Like Implementation

This section describes another variant of xsrc implementation, where text
regions are formed according to that found by the literature, [11, 10, 13], and im-
plemented in sgrep [14, 12]. We would refer to this implementation as XMLGrep.
An index function, I(p) for a text pattern p, returns a list of positions of where p
occurs in the DOM tree. For a single word, w, a region for the \textsuperscript{i}th occurrence of
w starts and ends at the same position, say \textit{p}_i. Therefor, a region set of the word
\textit{w} has the formula \{[\textit{p}_k, \textit{p}_k] | k = 0, 1, 2, ..., \textit{j} and \textit{0} \leq \textit{j} \leq \textit{N} - 1 \}, where \textit{N} is
the number of indexed terms in the DOM tree logical view. Most queries require
regions that reflect the natural structure of the document. Therefore, an opera-
tor that matches start and end delimiters of tags and text contents, as the text value of some XML elements, must be used to form such regions. This matching operator is the followed by operator.

To compare XMLGrep with the xsrch, of how regions thus can be formed. First, to formalize the regions returned by an index function $I(p)$, for a text pattern $p$,

- if $p$ is the name of an element node in the DOM tree, such as `<genericId>`, the returned region $r$ is $r.start = I(<\text{genericId}>)$, and $r.end = I(<\text{genericId}>)$, for the same positional of `<genericId>`.

- if $p$ is a text fragment as a single token in the document, say textToken, then $r.start = I(\text{textToken})$ and $r.end = I(\text{textToken})$, for the same positional occurrence of textToken.

To form regions, in the first case, we need an expression like "<genericId> .. </genericId>". This means that "form regions that match the `<genericId>` start delimiter with its closing match end delimiter `</genericId>`". In the second case, regions are simply the ones that are returned by the index function, and can not be formed arbitrarily. Sometimes a preknowledge of the text fragment inclusion element should be known. And this requires an additional containment operator to be used int the expression designed for forming regions. The choice of the containment operator is based on the goal of the query.

67
For instance, in the booklist document, if it is of interest to make some query related to the book titles, then forming regions of the <title> element might be required. This can be accomplished as the query expression: "<title> .. </title>". In xsrch, this can be done as the simple query "<title>".

4.2.1 Comparing a One-to-One Correspondence Queries

In the play Hamlet, if our interest is to know some text fragment, e.g. "hamlet", then a preknowledge of where this text occurs in which element, depending on the query goal, is sometimes required. Since this word could occur in an element <scene>, <persona>, <line>, etc. Also, it depends on what we are interested in: is it the text itself or the inclusion element.

If the scene is the goal of the query, then the expression in XMLGrep

"(<scene> .. </scene>) containing hamlet"

this returns a region set of scene elements containing the word hamlet.

The exact corresponding expression in xsrch is

"<scene> containing hamlet"

If it is the text in a <scene> element is our goal, then the XMLGrep query should be

"hamlet containedIn (<scene> .. </scene>)"
This returns a region set that their regions are of length one, subsets of the region set of "hamlet", which is the index position of the word hamlet, e.g. [[34385: #text], [34385: #text]].

In addition, printing out the contents of these regions in XMLGrep will result in the output that repeatedly displays the same word: Hamlet. Unless a more customized implementation for writing the contents of regions, see NodeWriter class in Section 3.2.4, that must refer to the inclusion element. Total number of regions here, are 452 regions.

In xsrch, the query

"hamlet containedIn <scene>"

returns several types of regions: <LINE>, <SPEAKER>, <STAGEDIR>, etc. And these regions make more sense than the queries in this model, for a one-to-one correspondence, unless a more complex expression is written. And, hence printing out the node contents would give more meaningful results, e.g.

<LINE> Was’t Hamlet wrong’d Laertes? Never Hamlet: </LINE>

<SPEAKER> HAMLET </SPEAKER>

<STAGEDIR> LAERTES wounds HAMLET; then in scuffling, they change rapiers, and HAMLET wounds LAERTES </STAGEDIR>

<LINE> It is here, Hamlet: Hamlet, thou art slain; </LINE>
Number of regions produced by this query is 446 regions. Notice that in the last sample line of the printed results, the word “Hamlet” repeated two times.

The query expression for XMLGrep would require a preknowledge of the elements containing the word hamlet, e.g

"((<person> .. </person> ) or ( <scene> .. </scene> ) or ( <speaker> .. </speaker> ) or ( <speech> .. </speech> ) or ( <line> .. </line> ) ) containing hamlet ) containedIn ( <scene> .. </scene> )"

This is harmful and very complex to list all element names. When compared to the expression in the former model, the expression

"hamlet containedIn <scene>",

we notice the following:

1. The expressions in XMLGrep are very complex to write and require preknowledge and memorization of element names. As a consequence, more errors could be addressed either in the expression itself and/or in the results.

2. The time for XMLGrep, for documents that are big in size have been noticed but small ones, is more efficient than xsrch. For example, in the booklist document, no time differences. Whereas in hamlet.xml, it is about 1:9, for
the above query.

A more kindly useful substituant to list all elements is to implement some sort of expression that references to all elements in the document, as described in Section 2.2.7. Such implementation would require the same implementation that referencing elements start tags and their matching closing end tags as the one that already implemented in xsrch. According to this, theoretically, for general queries on big documents, there would be no timing differences.

A serious problem that works to be mentioned here, assume that there is a XML document fragment, e.g.

<someTag> textContents &lt;/someTag&gt; textContents </tagId>

This document fragment would be parsed as

<someTag> textContents </someTag> textContents </tagId>

Then to form regions, the expression <tagId> .. </tagId>, would mistakenly refer to that document fragment. This issue is already taken care of in xsrch. Since the text </tagId> is maintained as a text and not as a closing delimiter tag.
5 DISCUSSION AND CONCLUSIONS

In what follows, we present the most relevant of future work that can be incorporated into the system, such as supporting the DOM node types: Attr and Namespaces, how to deal with program source codes when they are exist as XML documents, and extend the system search facility for word truncation and pattern matching. How to plug a GUI into the system is also presented. The current implementation of the system is capable for handling one document at a time, but it can be extended to work with a text database is discussed along with some guidelines for implementation.

5.1 Supporting Attributes, Namespaces, and ProcessingInstructions

The Indexer class in package algebra, does not index all DOM node types, and hence, some of the unsupported nodes are not considered as part of the current implementation of query expressions and the search engine. The current implementation could map anything that is either direct or indirect text references as text regions.

First thing should be considered in the future work is the following node types: Attributes (Attr), Namespaces, and ProcessingInstructions. All other node types are considered as part of supporting the above mentioned node types. For instance, DocumentType node type could be implemented as part of supporting the Attr
node types. Comments are generally of less interest, but it is possible to include them as part of the text being searched.

To support the Attr nodes in the model, it is just need to be treated as part of text, unless other special attributes, such as the namespaces, doctype, etc. Since the Attr node type are not part of the hierarchical model of the DOM tree, generally, indexing the attributes is done while traversing the DOM tree and takes into consideration the elements attributes as indexed terms, the attribute names along with their respective values. The required code can be inserted in the nodeStartActions() method of Indexer class, with any other relevant code in the relevant places.

5.2 Supporting XML Documents that Contain Program Source Codes

When an XML document contains a program source code, or itself is a program source, then the following have to be treated well:

The stoplist words and the set of delimiters are no longer ignored from the indexing process. Some kinds of delimiter are considered to be tokens, and therefore should be indexed, as well as for the members of the stoplist. Whitespaces also have special treatment accordingly.
5.3 Text Database

When talking about text database, this means that we have a set of XML documents. Therefore, namespaces are now more relevant to be supported in the system. To include namespaces in the system, its prerequisite is to support the XML elements attributes (Attr).

Then the system can be plugged into a text database composed of XML documents, such that each document is well-formed, valid, and has a valid URI. Each document in the database is assigned an Id, as <documentId>. For efficiency, the datatype of the Id should be an Integer, although other datatypes could be used that is more suitable to the set of the documents in the database. This Id serves exactly as the indexes that are attached to the tokens inside the document, but the scope of the <documentId> is the set of documents in the database. Though, every document has an associated Id. When referencing a document, this would reference the document itself and its associated Id.

For queries conducted on the text database, we have now two different data types for the operands and the results of query expressions and subexpressions, depending on the scope of the query: either document set, when dealing with documents, or region set, when dealing with document fragments. Mixing the two datatypes in a query should be implemented. For example, the query

Find the set of documents that their titles contain the word “algo-
rithms”.

This would select the set of documents that are books, articles, or any other document type. This query has document set and region set as its input, and produces document set as its output.

Another query that would select region sets, as its output, for example

*Find the scenes that contain the word “lord” in the plays Hamlet and Macbeth.*

Such query expressions should be handled carefully, and a denotational semantics should be followed. Since mixing different datatypes in the results of queries or sub-queries, may cause “semantic problems” (qtd. in [11] 23). For instance, each subexpression results in a different datatype.

5.3.1 Implementation

To make use of the current implementation of the system and to avoid the semantic problems that may appear as a result of mixing different datatypes, another alternative is to treat the whole database as a single document, i.e. to view the whole database as a sequence of concatenated symbols. Then each document could be identified by an explicit markup that would refer to the Id for each document. This could be thought as of

- identifying the starting delimiter by some sort of element `<documentId>`

  with the document associated Id value, for instance, assigned as an attribute
value.

- the match closing delimiter is identified as the end-delimiter </documentId>, along with the associated matching Id value.

Following this approach, the operands and the results of query expressions have only one datatype: region set.

Then, creating one indexer object, with its own DOM tree and its index map, for each XML document in the database, will index the tokens inside each document and the surrounding context that would identify each document, the start tag <documentId> and its match closing tag </documentId>. The Indexer objects can be stored in some type of collection data structure. The unary outer operator may be used here with region sets representing documents. According to the current implementation of forming region sets, the query

"outer (<documentId>)"

would select all documents stored in the database. This allows to work on each document individually. To select, for example, the documents with titles containing the word “Analysis”,

"(outer(<documentId>)) ◁ (<title> ◁ analysis)"

Theoretically, the system does not set any limits on the size of the document. Other approaches for dealing with memory and disk management, to save space, can be implemented. Another point related to the memory management, is to use
caching for some query and subquery results. This will improve the evaluation of complex queries that are dealing with set of big documents.

5.4 User Interface and Word Truncation

The queries expressed using the nested region algebra require some level of sophistication to be composed properly. Also, most often, a preknowledge of the elements names, and in case of text data base, the document names, is required. For such cases, a graphical user interface layer might be more suitable to be plugged into the system. In the case of GUI, there is no need for the the command line parser implementation, ExpressionParser class. As the semantics of each token can be identified through the entries of the GUI layer.

Other Future Plans

An extension that could easily be supported into the existing algebra is operators that support searching for text matching of truncated words. This kind of searching can be supported as for regions match element node names or text fragments. The implementation is direct with the indexed tokens stored in the index map, by implementing a kind of string pattern matching.
Conclusions

The indexing scheme and the structured text algebra that have been implemented in the system did not set any limits on the length of the document. The structured text algebra does not assume any hierarchical structure of the document. However, the system utilizes the parsing utility used by DOM parsers in implementing the classes for indexing, since The DOM model views the document as a hierarchical structure of tree nodes. The design of classes implementing the algebra operators is independent from the hierarchical view of documents.

We have seen that when extending the system to work with a text database, this may require mixing different datatypes as results of sub-queries; and this may cause semantic problems. An alternative implementation was provided, which views the whole database as a single sequence of concatenated symbols. This design has two major benefits:

- The structured text algebra operators have only one datatype: region sets. Therefore avoiding the semantic problems. There is no limits on the nested or overlapping of regions.

- There is no need to change the current implementation of the system, as each document is viewed as a one region.
REFERENCES


