A Query Expression and Processing Technique for an XML Search Engine

Wol Young Lee, Hwan Seung Yong

Department of Computer Science and Engineering
Ewha Institute of Science and Technology
Seoul, Korea (wylee@ewha.ac.kr, hsyong@ewha.ac.kr)

Abstract. One of the virtues of XML is that it allows complex structures to be easily expressed. This allows XML to be used as an intermediate, neutral, and standard form for representing many types of structured or semistructured documents that arise in a great variety of applications. To support for efficient queries against XML data, many query languages have been designed. The query languages require the users to know the structure of the XML documents and specify path-based search conditions on the structure. This path-based query against XML documents is a natural consequence of the hierarchical structure of XML. However, it is also desirable to allow the users to formulate no path queries against XML documents, to complement the current path-based queries. In this paper, we have developed a query expression and processing technique supporting non-navigational content-based queries for an XML search engine.

1 Introduction

The past few years have been a dramatic increase in the popularity and adoption of XML, as it includes semantic information within documents describing semi-structured data. This makes it possible to overcome the shortcomings of existing markup languages such as HTML and support data exchange in e-business environments. The increasing adoption of XML has also raised new challenges. One of the key issues is the information retrieval against large collections of XML documents. To allow for numerous data search, many query languages have been developed [1-4]. These query languages require users to know the structure of XML documents, including all the element and attribute names, data types of the data values, and the hierarchical structure of the elements. Since a search condition needs to involve an element/attribute name, which appears somewhere in the hierarchical structure of an XML document, these query languages force the users to specify the navigational access paths to the elements/attributes that are involved in search conditions. This makes it difficult for the users to query XML documents.

To complement navigational path-based queries, there have been researches for a keyword-based search engine [6-9]. However, it is difficult for users to search exact results because the keyword-based search engine makes use of only limited informa-

This work was supported by Korea Science and Engineering Foundation (KOSEF) grants 20044014.

tion. For example, against the following document, suppose users want to search papers of ‘semi-structured data’ released by ‘Serge Abiteboul.

```
<dblp>
  <inproceedings key=”book/Abiteboul”>
    <title>Data on the web</title>
    <author>Serge Abiteboul</author>
  </inproceedings>
  <inproceedings key=”conf/beeri”>
    <title>SAL: An algebra for semi-structured data and XML</title>
    <author>Catriel Beeri</author>
    <author>Yariv Tzaban</author>
  </inproceedings>
  <inproceedings key=”conf/icd/charwathe”>
    <title>Representing and querying changes in semi-structured data</title>
    <author>Sudashan S. Chawathe</author>
    <editor>Serge Abiteboul</editor>
    <author>Jennifer Widom</author>
  </inproceedings>
</dblp>
```

Here, existing search engines mainly return elements of least common ancestor that satisfies two search conditions. Even though the engines consider the structure of the document, the search results include too wide ranges like *dblp.* It is a total of one document. Also, *editor* or *author* and “Serge Abiteboul” can be all matched because keyword-based search expressions do not express data names. On the other hand, if users would like to search in navigational query expressions as considering XML properties, users have to use very complex expressions according to document structure. For example, suppose users want the following query against Fig. 1-1.

Q1: Search book title information of which publisher is ‘Morgan Kaufmann’, release year is ‘1999’, and whose authors include ‘Serge Abiteboul’.

If the names are represented as element names and their values are directly assigned as element values like Fig. 1-1(a), then the users can get desired results by expressing as //year=”1999” and //publisher=”Morgan Kaufmann” and //author=”Serge Abiteboul”. However, the users have to differently express a path expression if the names are represented not as elements but attributes on documents like Fig. 1-1(b). In this case, the users have to represent as //@year=”1999” and //@publisher=”Morgan Kaufmann” and //@author=”Serge Abiteboul”. If arbitrary data are inserted between the data names and their values like Fig. 1-1(c) and (d) the users have to know the fact and query expressions are differently represented by a way describing data. The users have to express as //year/@yyyy=”1999” and //publisher/@name =”Morgan Kaufmann” and //author/@name=”Serge Abiteboul” or //year/@* =”1999” and //publisher/@* =”Morgan Kaufmann” and //author/@*=”Serge Abiteboul” against Fig. 1-1(c). Against Fig. 1-1(d), the users have to express as //year/yyyy=”1999” and //publisher/name=”Morgan Kaufmann” and //author/name=” Serge Abiteboul” or //year/*=”1999” and //publisher/*=”Morgan Kaufmann” and //author/*=”Serge Abiteboul”. Furthermore, the document may be created as a nested relationship between data names like Fig. 1-1(e) and (f). In this case, the users have to express the nested relationship among the data as //year=”1999”//publisher=”Morgan Kaufmann”//author=”Serge Abiteboul” against Fig. 1-1(e). Against Fig. 1-1(f), the users have to express an inverse nested relationship because the two documents are oppo-

site in the upper and lower relationship of *year* and *publisher* even though both Fig. 1-1(e) and (f) have a nested relationship.

```
<book>
  <isbn>1</isbn>
  <year>1999</year>
  <publisher>Morgan Kaufmann
  </publisher>
  <title>Data on the web</title>
  <author>Serge Abiteboul</author>
  <author>Peter Buneman</author>
  <author>Dan Suciu</author>
</book>
```

(a) elements

```
<book>
  <isbn no= "1"></isbn>
  <year yyyy="1999"></year>
  <title name="Data on the web"></title>
  <publisher name="Morgan Kaufmann">
  </publisher>
  <author name1="Serge Abiteboul"
          name2="Peter Buneman"
          name3= "Dan Suciu"></author>
</book>
```

(c) intervening of attributes

```
<year>1999
 <publisher>Morgan Kaufmann
  <book>
    <isbn>1</isbn>
     <title>Data on the web</title>
      <author>Serge Abiteboul </author>
     <author>Peter Buneman </author>
     <author>Dan Suciu</author>
  </book> …
 </publisher>…
</year>
```

(e) *publisher* is nested in *year*

```
<book>
   <general_info isbn= "1" year="1999"
       publisher="Morgan Kaufmann">
   </general_info>
    <detail_info  title=" Data on the web"
       author=" 'Serge Abiteboul'
       'Peter Buneman' 'Dan Suciu'">
   </detail_info>
</book>
```

(b) attributes

```
<book>
  <isbn><no>1</sibn</no></isbn>
  <year><yyyy>1999</yyyy></year>
  <title><name>Data on the web</name>
  </title>
  <publisher><name>Morgan Kaufmann
  </name></publisher>
  <author><name>Serge Abiteboul</name>
          <name>Peter Buneman</name>
       <name>Dan Suciu </name></author>
</book>
```

(d) intervening of elements

```
<publisher>Morgan Kaufmann
  <year yyyy="1999">
    <book>
      <isbn>1</isbn>
      <title>Data on the web</title>
       <author>Serge Abiteboul </author>
       <author>Peter Buneman</author>
       <author>Dan Suciu</author>
    </book> …
  </year>…
</publisher>
```

(f) *year* is nested in *publisher*

Fig. 1-1 various document representations

Insofar as XML gives rise to hierarchical structure, and navigational access to selected parts of an XML document is a natural access pattern, the navigational query expressions clearly make sense. However, for query Q1, we would like to be able to state the search conditions simply as

publisher = "Morgan Kaufmann" and year = "1999" and author = "Serge Abiteboul"

regardless of the structure of the XML documents[5]. We believe non-navigational content-based query expression for XML, and general semi-structured data, are also desirable. In other words, we believe that both navigational and non-navigational query expressions are necessary to support all query needs against XML databases. Specially, the non-navigational content-based queries provide the following contributions.

- The non-navigational content-based queries allow users to query about all document types regardless of document structures. That is, the users need not worry about path representations because the users query by using only data names and their values as SQL-like queries.
- On distributed environments or web data, users do not generally know the exact navigational paths of documents. Also every site may have different structures. But our approach allows users to query on these many sites without knowledge

about XML schema like search engines if users know data names and their values. Furthermore, even on local systems, our work can be applied to the case that imports documents created by many people without a common schema.

- Existing HTML web search engines support keyword-based searches because the past web data were expressed using HTML. However, data is increasingly represented as XML because of its various good points. Our research result can be applied to XML web search engines because XML uses tags to represent data. That is, our research can improve the accuracy of XML web search engine by using tag names, element names, besides values in query expressions.

2 Related Work

Up to now, many XML query languages [1-4] were developed. As a commercial product, Oracle XML DB query in an extended SQL because it deals with XML data and relational databases in a tightly coupled method. DB2 XML Extender uses 'contains' operator for information retrieval systems. Also, SQL 2000 server supports XPath-based query expressions and accesses SQL servers through URL. Navigational query languages mainly support XPath [4] and use regular path expressions to relax somewhat query expressions.

Query expressions of current XML query languages are very complex or difficult because of irregular properties of XML. Therefore, there were researches on keyword-based searches to overcome this difficulty [6-9]. These researches used a structure concept to search text data by connecting keyword searches with a structure. But it is difficult that keyword-based searches get exact results because keyword-based query expressions express only data values even though the techniques consider document structures.

3. A Query Expression for an XML Search Engine

Example 3.1 For example, suppose that users want to try the query Q1. If the users would like to query regardless of document structures, the users express the search conditions in a condition clause and a return condition in a result clause of as follows.

Condition:	**publisher = "Morgan Kaufmann"** **and year = "1999"** **and author = "Serge Abiteboul"**
Result:	**title**

---- CQ1

The non-navigational content-based query expression for an XML search engine can be formally defined as follows.

Expression ::= Predicate ("and" | "or")Predicate
Predicate ::= DataName "=" DataValue
DataValue ::= string

where DataName is either an element name or an attribute name. The DataName-DataValue pairs of the search condition can have various paths on documents. There-

fore, to process the search condition, first of all, we have to know all possible paths of the condition clause. For simplicity, we do not distinguish attributes from elements.

First, all possible paths among the DataName-DataValue pairs can have a non-nested or nested relationship on documents. That is, in CQ1, *publisher*, *year*, and *author* have a non-nested relationship like Fig. 1-1 (a)-(d) or a nested relationship like Fig 1-1(e) and (f). Second, there may be intervening elements and/or an attribute between an element name and its corresponding value. An XML document of the same content as that represented in Fig. 1-1(a) may easily be represented as in Fig. 1-1(c) and (d). Here, *yyyy* intervenes between *year* and "1999", *name* intervenes between *publisher* and "Morgan Kauffmann", and *name* intervenes between *author* and "Serge Abiteboul". In general, there may be an arbitrary number of element or attribute names between the element name and its corresponding value that match the search predicate in a query. Furthermore, there may be the other intervening elements between elements like *book* between *publisher* and *author* of Fig. 1-1(e) and (f). If we analyze all possible paths against the search condition of non-navigational content-based queries, they can be classified like Table 3-1.

Table 3-1 all possible paths among the data of a condition clause

(a) all paths using only data of the condition clause

Relationship among data	Paths
Non-nested	Path X
Nested	Path N

(b) all paths intervening arbitrary data

Relationship among data	Location of ntervening data	Paths
Non-nested	ancestor	Path *a*-X
	descendant	Path *d*-X
Nested	ancestor	Path *a*-N
	descendant	Path *d*-N

Here, the location of intervening data means whether the intervening element/attribute is located in ancestor or descendant of data of the condition clause. A query processor has to search all the analyzed paths to process the non-navigational content-based queries. We will show the processing steps in section 4.

4. A Query Processing Technique for an XML Search Engine

The processing of a content-based query expression consists of three steps.

Step (a): Evaluate each predicate (i.e., search condition) to find a matching (DataName, DataValue) pair in an XML document.

Step (b): Do the Boolean processing of all search conditions to find only those XML documents that satisfy all search conditions.

Step (c): For each matching document, process the result clause processing to return appropriate parts of the document.

Since the users not use the document structure while querying, the results may not be always be correct. But, in this paper, we will not report some solutions for the problem.

4.1 Node identifier for content-based queries

We propose region-numbering scheme [11] to process non-navigational content-based queries. The node identifiers of the numbering scheme represent a hierarchical

relationship between nodes. We can search all possible paths for non-navigational content-based queries if we make use of this relationship.

4.2 Base level for nearest common ancestor

If two nodes have a non-nested relationship, then the query processor has to compute nearest common ancestor (NCA) [10]. The reason is to prevent the tree traversal of a single subtree from "over-flowing" into another subtree by designating a special level during computing NCA. For example, suppose the root of an XML tree is named *books*, and its child node is named *book* like Fig. 4-1. Then the query evaluation involving *books* should be confined to within each *book* document instance, and, for example, the search predicate on *author* in one *book* document should not be combined with the search predicate on *year* in another *book* document. For this, we ignore the outside data of special regions by designating a base level.

```
<books>
  <book>
    <year>1999</year>
    <author>Serge Abiteboul</author>
    <title>…
    <publisher>…
  </book>
  <book>
    <year>1999</year>
    <author>Serge Abiteboul</author>
    <title>…
    <publisher> ...
  </book>
</books>
```

Fig. 4-1 Base level for NCA

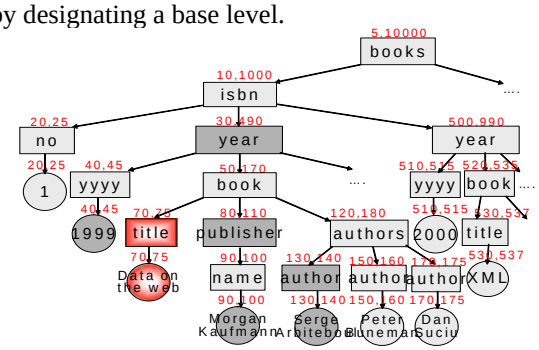


Fig. 4-2 tree representation of example document

4.3 Query processing algorithm

To search the results that satisfy all possible paths, our query processor processes three steps like Algorithm 1. Suppose we want to query CQ1 against Fig. 4-2.

First, like Fig. 4-3, our query processor searches the regions of "Morgan Kaufmann", "1999", and "Serge Abiteboul". It also searches the regions of *publisher*, *year*, and *author*. And then, select the regions of *publisher* (80-110), *year* (30-490), and *author* (130-140) including the regions of the values. Here, *publisher* and its value (or *year* and its value) become Path *d*-X and Path *d*-N.

RLV_1 : "Morgan Kaufmann"	
nv	region
1	90,100

RLV_2 : "1999"	
nv	region
1	40,45

RLV_3 : "Serge Abiteboul"	
nv	region
1	130,140

(a) after searching of data values("Morgan Kaufmann", "1994", "Serge Abiteboul")

RLC_1 : publisher	
nc	region
1	80,110

RLC_2 : year	
nc	region
1	30,490
2	500,990

RLC_3 : author	
nc	region
1	130,140
2	150,160
3	170,175

(b) after searching of data names(*publisher*, *year*, *author*)

RL_1 : publisher	
ncv	region
1	80,110

RL_2 : year	
ncv	region
1	30,490

RL_3 : author	
ncv	region
1	130,140

(c) after filtering of the data names having regions including the regions of data values

Fig. 4-3 Step (a): process each search condition separately

Second, the query processor has to do the AND-ing of *publisher* and *year*, *year* and *author*, and *publisher* and *author* like Fig. 4-4. In the case that two DataName-DataValue pairs have a nested relationship, the AND-ing of two predicates is to select the one DataName including the region of another DataName by comparing the region of two DataName-DataValue pairs. Therefore, the region of the AND-ing of *publisher* and *year* (or *year* and *author*) is 30-490. In the case that two DataName-DataValue pairs have a non-nested relationship, the AND-ing of two predicates is to search NCA of two DataName-DataValue pairs. Here, we make use of a base level. We designate the base level as 4 for NCA of *publisher* and *author*. Then, the query processor ignores the outsides of the region of the base level during computing NCA of *author* and *publisher*. Therefore, the result of AND-ing of *publisher* and *author* is NCA and its region is 50-170. Consequently, the result region of the condition clause that satisfies three predicates is 30-490. The query processor does merging the result regions of condition clause and stores the information into CR. Furthermore, we keep after merging even the region of NCA in NCAR in order to derive exact results. The path of *year* and *publisher* (or *author*) is Path *a*-N and the path of *publisher* and *author* are Path X. In the case of Path N or Path *a*-N, the query processor considers a bidirectional relationship between two node name-value pairs.

RL_1 : publisher	
ncv	region
1	80,110

RL_2 : year	
ncv	region
1	30,490

CR_1	
ncr	region
1	30,490

(a) CR_1 is the result of AND-ing of *publisher* and *year*(nested relationship)

RL_1: publisher	
ncv	region
1	80,110

RL_3: author	
ncv	region
1	130,140

$NCAR_1$	
ncr	region
1	50,170

(b) $NCAR_1$ is the result of AND-ing of *publisher* and *author* (non-nested relationship)

RL_2: year	
ncv	region
1	30,490

RL_3: author	
ncv	region
1	130,140

CR_2	
ncr	region
1	30,490

(c) CR_2 is the result of AND-ing of *year* and *author*(nested relationship)

CR	
ncr	region
1	30,490

(d) after merging of CR_1 and CR_2

NCAR	
nn	region
1	50,170

(e) after merging of $NCAR_1$

Fig. 4-4 Step (b): do Boolean processing of all search conditions

Third, to process the return clause, the query processor has to search the data of the return clause that satisfies all data of the condition clause like Fig. 4-5. The region of *title* that satisfies both the result region of condition clause (30-490) and the region of NCA (50-170) is 70-75. In the case that the region of *title* is 530-537, the query processor ignores this result because it does not satisfy both NCAR and CR. Therefore, the result of CQ1 is <title>Data on the web</title>.

R: title-	
nr	region
1	70,75
2	530,537

CR	
ncr	region
1	30,490

NCAR	
nn	region
1	50,170

RR	
nrr	region
1	70,75
	ignore

(a) RR is the result comparing R with CR and NCA

Fig. 4-5 Step (c): do the result clause processing

Algorithm 1 Query Processing Algorithm

Input: *n* element/attribute names (C_1, C_2,…, C_n) and their values (V_1, V_2,…, V_n)
// step (a): process each search condition separately
While (V_i) {
  Search regions of V_i, store the regions into RLV_i, and count the number of the regions (*nv*)
}
While (C_i) {
  Search regions of C_i, store the regions RLC_i, and count the number of the regions (*nc*)
}
For (*i*=1; *i* < *nv*; *i*++)
  For (*j*=1; *j* < *nc*; *j*++)
    If (RLC_i includes RLV_i) {
      Store the regions into RL_i
      Count the number of the regions (*ncv*)
    }
// step (b): do Boolean processing of all search conditions
For (*i*=1; *i* < *ncv*-1; *i*++)
  For (*j*=*i*+1; *j* < *ncv*; *j*++)
    If (RL_i and RL_j have a nested relationship)
      Store the upper region of RL_i and RL_j into CR_i.
    Else
      Compute NCA of RL_i and RL_j and store the regions of NCA into $NCAR_i$
// step (c): do the result clause processing
Search regions of R and count the number of the regions (*nr*)
If (R satisfies CR and NCAR)
    Store the R into RR
Else
    Ignore the R
Return the information of RR

5. Experimental Results

To compare non-navigational content-based queries with path-based navigational queries, we test on Xhive-6.0 known as a native XML-server [12] with 'book' data providing in X-Hive Corporation. We select XQuery as a path-based query language. The DTD of the 'book' data are as follows.

```
<?xml version="1.0" encoding="UTN-8"?>
<!ELEMENT article (para)>
<!ATTLIST article
  number CDATA #REQUIRED>
<!ELEMENT chapter (title, (section+ | article+))>
<!ATTLIST chapter
  number CDATA #REQUIRED>
<!ELEMENT item (para)>
<!ELEMENT list (item+)>
<!ELEMENT para (#PCDATA | list)*>
<!ELEMENT section (title, article+)>
<!ELEMENT title (#PCDATA)>
```

The system is providing 19 files of small size with slightly different document structures. We have extended the total data size by 10.9 MB. Query statements for testing are as Table 5-1. Q1 has one predicate and Q2 and Q3 has two predicate.

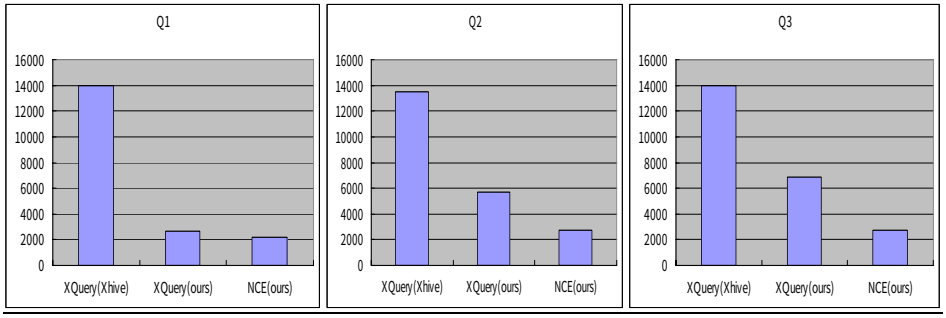
Table 5-1 Example queries for comparing with X-Hive server

Queries	XQuery	Content-based queries

Q1	for $c in doc (“UN Charter”)//chapter where $c/@number = “5” return $c/title/text()	Condition: chapter=”5” Result: title
Q2	for $c in doc (“UN Charter”)//chapter where $c/@number = “5” or $c/@number = “10” return $c/title/text()	Condition: chapter=”5” or chapter=”10” Result: title
Q3	for $c in doc (“UN Charter”)//chapter where $c//article/@number = “9” and $c//section/title = “COMPOSITION” return $c/title/text()	Condition: article=”9” and section=”composition” Result: title

We test both XQuery and content-based queries on our processor. And we perform X Query statements on X-Hive server to compare with our processor. The more detailed performance evaluation will not report on account of space considerations.

In the XQuery expression, the query processor searches two elements and one value for Q1, two elements and two values for Q2, and four elements and two values for Q3. In the non-navigational content-based query expression, the query processor searches one element and one value for Q1 and two elements and two values for Q2 and Q3. Our query processor generally has an effect on the time searching data and the size of results. The query processor does not traverse the path because users do not express a path in the condition clause. However, the query processor traverses the entire path because XQuery represent a complex path in the condition clause. Therefore content-based query time is the faster than it of XQuery expression. They take ordering of Q1, Q2, and Q3. We can also recognize that our query processing time for both content-based queries and XQuery is faster than it on X-Hive server.



* NCE: Non-navigational Content-based query Expression

Fig. 5-1 Performance evaluation on X-Hive and our query processor

The performance of the both non-navigational content-based queries and path-based navigational expressions are open to discussion. We believe that we can improve query speed by optimization such as an indexing technique, reduction in the number of searches and comparisons, and so on.

6. Conclusions and Future Work

XML data model can create many various document types because it allows users to create an irregular structure of documents. It is a reason requiring more knowledge about document structures to query. In this paper, for more easy queries, we devel-

oped a technique for schema independent queries. For this work, we designed a query expression called content-based queries for an XML search engine. Also, in order to process the query expression, we classified all possible paths among data and developed an algorithm capable of searching these paths. In the future, we will measure time updating documents and optimize non-navigational content-based query processing time.

Acknowledgement: We would like to thank Dr. Won Kim for his comments on this paper.

References

[1] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles, Extensible Stylesheet Language (XSL) Version 1.0, W3C Proposed Recommendation Aug. 2001

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener, The Lorel query language for semistructured data, International Journal on Digital Libraries, Apr. 1997

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Siméon, XQuery 1.0: An XML Query Language, W3C Working Draft 16 Aug. 2002

[4] W3C Consortium, XML Path Language (XPath) Version 1.0, W3C Recommendation 16 Nov. 1999

[5] W. Kim, W. Lee, and H. Yong, On Supporting Structure-Agnostic queries for XML, in *Journal of Object Technology*, vol. 3, no. 7, July-August 2004, pp. 27-35.

[6] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram, XRANK: Ranked keyword search over XML documents, Proc. of the ACM SIGMOD Int. Conf. on Management of Data, 2003

[7] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, XSEarch: A semantic search engine for XML, Proc. of the VLDB Conf., 2003

[8] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer, Searching XML documents via XML fragments, Proc. of the 26th Int. ACM SIGIR Conf., 2003

[9] V. Hristidis, Y Papakonstantinou, and A. Balmin, Keyword proximity search on XML graph, Proc. of Int. Conf. on Data Engineering, 2003

[10] Dov Harel and Robert Endre Tarjan, Fast algorithms for finding nearest common ancestors, SIAM Journal on Computing, v.13 n.2, p.338-355, May 1984

[11] M. P. Consens and T. Milo, Algebra for querying text regions: expressive power and optimization, Journal of computer and system sciences, 1998.

[12] X-Hive Corporation, X-Hive/DB 6.0, X-Hive Corporation, 2001, available at http://www.x-hive.com/