A Method for Evaluating Full-text Search Queries in Native XML Databases

Roman Pastukhov

Institute for System Programming of the Russian Academy of Sciences
ignatich@mail.ru
Ph.D. advisor: Grinev M. N.

Abstract

In this paper we consider the problem of efficiently producing results for full-text keyword search queries over XML documents. We describe full-text search query semantics and propose a method for efficient evaluation of keyword search queries with these semantics suitable for native XML databases. Method uses inverted file index which may be efficiently updated when a part of some XML document is updated.

1 Introduction

One of the main features of XML databases is ability to store semi-structured data as well as structured data. XQuery and XPath languages allow addressing parts of XML documents and querying them. These languages are convenient for querying regularly structured data. If data is semi-structured or its structure is unknown making queries using these languages becomes difficult. In such cases it is easier to use keyword search queries. One of the key advantages of keyword search queries is its simplicity – users do not have to learn a complex query language and can issue queries without prior knowledge about the structure of the underlying data.

Keyword searching over XML introduces new challenges. The result of a keyword search query is not always the entire document, but can be a deeply nested XML element. In general XML keyword search results can be arbitrarily nested elements, and returning the "deepest" node containing the keywords usually gives more context information (see [1, 2])

In this paper we describe semantics of full-text queries over XML documents and a method for evaluating such queries in an XML database system that supports XQuery data model [3].

Proceedings of the Spring Young Researcher's Colloquium On Database and Information Systems SYRCoDIS, Moscow, Russia, 2007

2 Related work

The recent increase in the number of XML repositories [4] has motivated extensive work on designing languages for XML full-text search [5, 6, 7, 8, 9].

There has been extensive research in information retrieval on the efficient evaluation of full-text queries [3], including structured full-text queries [10] and of XML queries such as XQuery/IR [11], XSEarch [5], XIRQL [7], XXL [8] and Niagara [12]. However, these works develop algorithms for specific full-text predicates in isolation.

The idea of computing the most specific elements for conjunctive queries has been actively explored using deepest common ancestors [13, 14, 15]. We extend this idea to support the efficient evaluation of queries with complex full-text predicates.

3 Data Model & Query Semantics

3.1 XML Data Model

The eXtensible Markup Language (XML) is a hierarchial format for data representation and exchange. An XML document consists of nested XML elements starting with root element. Each element can have attributes and text values, in addition to nested sub-elements.

3.2 Full-text Search Query Semantics

Full-text search queries concerned in this paper are composed of keywords and four operations – conjunction, disjunction, proximity and order. Conjunction and disjunction operations combine several (at least 2) sub-queries into a single query. Proximity and order operations are applied to a single query to produce another query.

Consider a query Q consisting of several keywords and a mapping M that maps some of these words to occurrences of these words in an XML document. If Q has a sub-query q than $M|q$ denotes a restriction of M to the set of keywords in q. Lets define a predicate matches(Q, M) to be true when one of the following conditions is true:

```
Op-And(lists)
  Current_match[i] = <empty list> for all i = {0, …, lists.count - 1}
  while lists are not completely processed:
    Choose i, such that lists[i] is the list with the lowest numbering scheme number;
    Elem = lists[i].next;
    If nodes in current_match have no common ancestor with elem:
      If all lists in current_match are not empty:
        Put values from current_match to output stream
      Current_match[i] = <empty list> for all i = {0, …, lists.count - 1}
    Add elem to current_match[i];
  If all lists in current_match are not empty:
    Put values from current_match to output stream;
```

Listing 1

```
Op-Or(lists)
  while lists are not completely processed:
     choose i, such that lists[i] is the list with the lowest numbering scheme number;
     output l[i].next to the output stream;
```

Listing 2

- If *Q* is a single keyword and *M* has a mapping for this word.
- If Q is a conjunction query with sub-queries *S*={q0, q1, … , qn} and {∀q∈S | matches(q, M|q)}
- If Q is a disjunction query with sub-queries *S*={q0, q1, … , qn} and {∃q∈S | matches(q, M|q)}
- If Q is a proximity query with distance d and sub-query q, such than matches(q, M) is true and the maximal distance between the words in image of M is no more than (d-1).
- If Q is an order query with sub-query q, such that matches(q, M) is true, M is injective and the order of the words in q matches the orders of their images defined by M.

The result of query Q consists of elements E such that E is the "deepest" common ancestor of image of some mapping M that makes predicate matches(Q, M) true.

4 Query processing

In order to evaluate the result of a query Q we will translate this query into a query plan tree. Each node corresponds to one of the four operations (conjunction, disjunction, proximity or order), leaves correspond to keywords in the query and edges correspond to relations between queries and sub-queries.

Each operator node receives a list of all matches from its child nodes and produces a result to its parent.

Output of the root operator (corresponding to the whole query) is transformed to a set of nodes.

4.1 Index structure

To allow efficient query evaluation we will use an inverted index which contains a list of all word occurrences (including position of word in node text) in nodes along with their identifiers and numbering scheme labels.

A numbering scheme assigns a unique label to each node of an XML document according to some scheme-specific rules. The labels encode information about relative position of the node in the document. Thus, the main purpose of numbering scheme is to provide mechanisms to quickly determine the structural relationship between a pair of nodes.

Most native XML databases use some sort of numbering scheme.

We will require numbering scheme to provide these mechanisms:

(1) determining ancestor-descendent relationship between to nodes;
(2) comparing nodes by document order;
(3) determining whether two nodes have a common ancestor;

Practically every numbering scheme used in native XML databases provides mechanisms (1) and (2). If it doesn't provide mechanism (3) we can easily modify it by adding document root identifier to numbering scheme labels. This modified numbering scheme will provide mechanism (3).

4.2 Data used by operations

To represent a list of all matches that is transferred between operators we will use tuples that consist of lists of word occurrences similar to the contents of inverted files in the index. Operators in the query plan tree leaves corresponding to keywords in the query will simply read an inverted file for this word and return tuples consisting of a single list describing some node and word positions of the keyword in the text of this node. Just like inverted files, lists in tuples contain numbering scheme labels for nodes.

```
Op-Order(list)
  while list is not completely processed:
    elem = list.next;
  if  elem.width == 1:
    put elem to output stream;
  else:
     create a list ord_list of all word occurrences in lists of elem along with list number (list contains pairs <list number, word occurence>) in document order;
     initialize array of pointers to ord_list p, to make p[i] point to the first occurrence of a word from elem’s i’th list, which is after p[i-1] in the ord_list (for i>0);
    p[n] points to the end of ord_list, n = elem.width
    while all elements of p are defined:
       output tuple i’th list of which consists of elements of  elem’s i’th list between p[i] and p[i+1] in ord_list;
          p[0] = first element belonging to 0’th list which is after p[1] in ord_list;
      for each i in [1..n-1]:
          p[i] = first element belonging to i’th list which is after p[i-1] in ord_list;
```

Listing 3

If a tuple consists of n lists, then we will say that width of this tuple is n (denoted as `t.width` in the pseudo-code, here t is a variable referencing some tuple).

A tuple represents a set of matches that is Cartesian product of sets of word occurrences in each list (i.e. if we choose one word in each list of a tuple we will get one of the matches represented by this tuple)

All nodes in a tuple always have a common ancestor. Tuples in an output stream of some operator are returned in the document order of their respective “deepest” common ancestors of nodes in each tuple.

4.3 Conjunction operation

This operator simply combines tuples from its sub-queries to a single tuple. Pseudo-code for this operation is shown in listing 2.

4.4 Disjunction operation

Disjunction operation returns all tuples produced by its sub-query operators in document order. Pseudo-code for this operation is shown in listing 1.

4.5 Order operation

Order operation filters tuples returned by its sub-query operator, so that all word occurrences in matches are in the correct order (it’s always corresponds to the orders of lists in tuple). This may split one tuple into several tuples. Pseudo-code for this operation is shown in listing 3.

4.6 Proximity operation

Proximity operation filters tuples returned by its sub-query operator, so that all word occurrences in matches are within a window of specific size. Pseudo-code for this operation is shown in listing 4.

Since we do not have information about indices of the first word in the nodes, this operator may produce false matches which should be checked at later stages of query execution by computing exact values of node staring word index array (sw array in the pseudo-code). This is not needed if all words in the match that need to fit in some window are in the same node.

```
Op-Window(list, window):
  while list is not completely processed:
    elem = list.next
    if elem.width == 1:
      put elem to output stream;
    else:
       create a list L of all different nodes in elem with a maximal word number for each node. List is ordered by document order of nodes;
        for each node U in the list L compute sw[U] as the sum of maximal word numbers of nodes before node u;
      while all lists elem[i], i={0..elem.width-1} are not empty:
         choose word W with minimum S(W) = sw[node that contains W] + the ordinal number of word W in the text of node;
          if all lists elem[i], i={0..elem.width-1} contain words W1, such that S(W1) < S(W) + window:
           output a tuple that consists of all words W1 from lists of elem, such that S(W1) < S(W) + window (if word W1 is in i’th list of elem, it will be in the i’th list of the resulting tuple) to the output stream;
```

Listing 4

5 Conclusion

In this article we described semantics for full-text search queries over XML data and proposed query evaluation method suitable for native XML databases. Method uses inverted file indices which can be effectively updated: if node changes do not affect its ancestor or sibling nodes in the index.

Proposed method allows efficient evaluation of most full-text queries described in [9].

6 Future work

Investigating the following problems may lead to improving the proposed query evaluation method:

- devise an efficient compression method suitable for proposed inverted file index (and a fixed numbering scheme);
- add relevance calculation and see how can index be changed to allow efficient evaluation of ranked queries;
- if the result of a full-text query will be presented as a set of nodes, some operations may not need to return a full set of matches that include word occurrences, instead they can return just a set of matching nodes. This may be used to improve query evaluation performance;
- see whether the proposed method can be modified to allow "not" or "mild not"[9] queries;

References

[1] N. Fuhr, K. Grobjohann, "XIRQL: A Language for Information Retrieval in XML Documents", SIGIR Conf., 2001.

[2] A. Schmidt, M. Kersten, M. Windhouwer, "Querying XML Documents Made Easy: Nearest Concept Queries", ICDE Conf., 2001.

[3] XQuery 1.0 and XPath 2.0 Data Model (XDM) http://www.w3.org/TR/xpath-datamodel/

[4] Initiative for the Evaluation of XML Retrieval. http://inex.is.informatik.uni-duisburg.de/2005/

[5] S. Cohen, J. Mamou. Y. Kanza, Y. Sagiv. XSEarch: A Semantic Search Engine for XML. VLDB 2003.

[6] D. Florescu, D. Kossmann, I. Manolescu. Integrating Keyword Search into XML Query Processing. WWW 2000.

[7] N. Fuhr, K. Grossjohann. XIRQL: An Extension of XQL for Information Retrieval. SIGIR 2000.

[8] A. Theobald, G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT 2002.

[9] The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text. Working draft. http://www.w3.org/TR/xquery-full-text/.

[10] E. W. Brown. Fast Evaluation of Structured Queries for Information Retrieval. SIGIR 1995.

[11] J. M. Bremer, M. Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. WebDB 2002.

[12] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD 2001.

[13] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

[14] A. Schmidt, M. Kersten, M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. ICDE 2001.

[15] Y. Xu, Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. SIGMOD 2005.