A path index for efficient XML path expression processing

Giuseppe Amato[1], Franca Debole[1], Pavel Zezula[2], and Fausto Rabitti[1]

[1] ISTI-CNR, Pisa, Italy,
{G.Amato,F.Debole}@iei.pi.cnr.it, F.Rabitti@cnuce.cnr.it
WWW home page: http://www.isti.cnr.it
[2] Masaryk University, Brno, Czech Republic,
zezula@fi.muni.cz
WWW home page: http://www.fi.muni.cz
(Extended Abstract)

Abstract. XML is a de fact standard for data representation and exchange on the Internet, therefore storing and querying XML repositories has become an important issue. Several XML query languages are based on the use path expressions containing optional wildcards. This poses a new problem, given that traditional query processing approaches have been proven not to be efficient in this case. We proposed a new path index to efficiently process path expressions with wildcards on XML data. Extensive evaluation confirms better performance with respect to other techniques proposed in the literature. An extension of the proposed technique to deal with the content of XML documents in addition to their structure is also discussed.

1 Introduction

The eXtensible Markup Language (XML) has become the standard for representing and exchanging data on the Internet. The obvious advantage of encoding data in XML is that they can easily be exported and imported. They can also be easily read by human user in their raw format. In addition to the documents' content, XML documents contain explicit information on their structures. However, efficient management of large XML document repositories is still a challenge. Searching for information in an XML document repository involves checking structural relationships in addition to content predicates, dealing with semistructured information, and the process of finding structural relationships has been recognized as the most critical for achieving the global efficiency. Several XML query languages, as for instance XPath [3] and XQuery [4], are based on the use path expressions containing optional wildcards. This poses a new problem, given that traditional query processing approaches have been proven not to be efficient in this case.

The aim of this paper is to propose a path index, that is an index structure to support evaluation of containment relationships for XML searching. The proposed index is able to efficiently process path expressions even in the presence of

wildcards, and our experiments have shown an evident superiority of the technique with respect to other approaches. An extension of the path index to deal with the content of elements or the value of attributes is also discussed.

The paper is organized as follows. Section 2, surveys the basic concepts, and Section 3 presents the idea of the path index. Section 4 discusses how the path index can be extended to deal with content. Section 5 presents a comparative evaluation of the proposed technique. Section 6 concludes the paper.

2 Preliminaries

In this section we briefly discuss some general concepts, necessary for the rest of the paper. We first introduce the inverted index as an access structure typically used for efficient text document retrieval. Then, we survey a technique for processing partially specified query terms.

2.1 Inverted index

Efficient text retrieval is typically supported by the use of *inverted index* [5]. This index structure associates terms, contained in text documents, with items describing their occurrence. An item can be just a reference to a text document containing the term or it might contain additional information, such as the location of the term in the text or the term frequency. An inverted index consists of two main components: a set of *inverted file entries* or *posting lists*, each containing a list of items corresponding to the associated term; and a *search structure* that maps terms to the corresponding posting lists. The set of terms indexed by the search structure, that is the set of terms contained in the whole text collection, is called the *lexicon*.

In order to search for text documents containing a specific term, the search structure is used first to obtain the posting list. Then the posting list is used to get the qualifying items.

2.2 Partially specified query terms

A technique for processing partially specified query terms (queries with wildcards) in text databases was proposed in [2]. This technique is based on the construction of a *rotated* (or *permuted*) *lexicon*, consisting of all possible rotations of all terms in the original lexicon.

Let us suppose that our original lexicon includes the term `apple`. The rotated lexicon will contain the terms `apple^`, `pple^a`, `ple^ap`, `le^app`, `e^appl`, `^apple`, where `^` is used as the string terminating character. The rotated lexicon is alphabetically ordered by using the sort sequence `^`, `a,b,c,...` and it is inserted in an inverted index using a search structure that maintains the rotated lexicon ordered. This can be obtained, for instance, by using a B^+-Tree. Rotated versions of all terms in the original lexicon are mapped to the posting list associated with the original term.

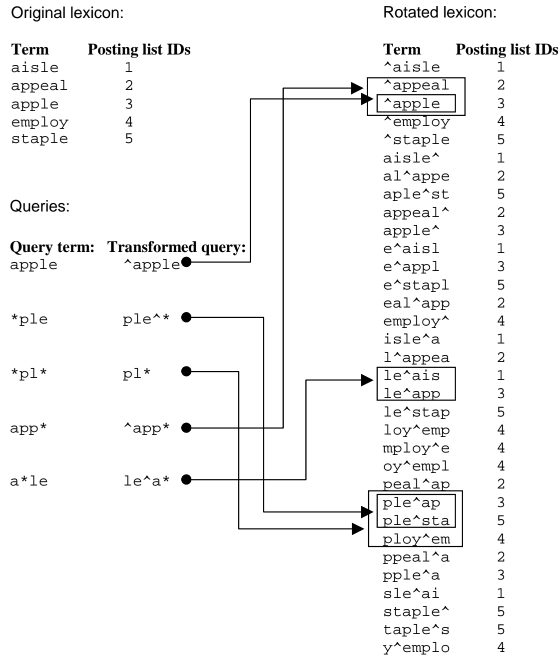

Fig. 1. The original lexicon, the rotated lexicon, and some queries processed on the rotated lexicon

By using the ordered rotated lexicon, query patterns A, *A, A*B, A*, and *A* (A, and B are sub-terms that compose the entire query term, and * is the wildcard) can be processed by transforming the query according to the following transformation rules:

I) A transforms to ^A; II) *A transforms to A^*; III) A*B transforms to B^A*; IV) A* transforms to ^A*; V) *A* transforms to A*.

The transformed query terms are used to search in the rotated term structure. For example, if our original lexicon contains `apple, aisle, appeal, employ, staple`. Figure 1 shows the obtained rotated lexicon, ordered alphabetically. Now consider the following queries: `apple`, `*ple`, `app*`, `*pl*`, and `a*le`. Figure 1 also shows how they are transformed and how the transformed query terms are matched against the rotated lexicon. For instance, the query `*pl*` is transformed into `pl*`, that matches the entries `ple^ap, ple^sta, ploy^em` corresponding to the terms `apple, staple, employ` of the original lexicon.

3 Rotated Path Index

Wildcards are also frequently used in XPath expressions. These expressions are typically processed by multiple *containment joins* [7], which can be very ineffi-

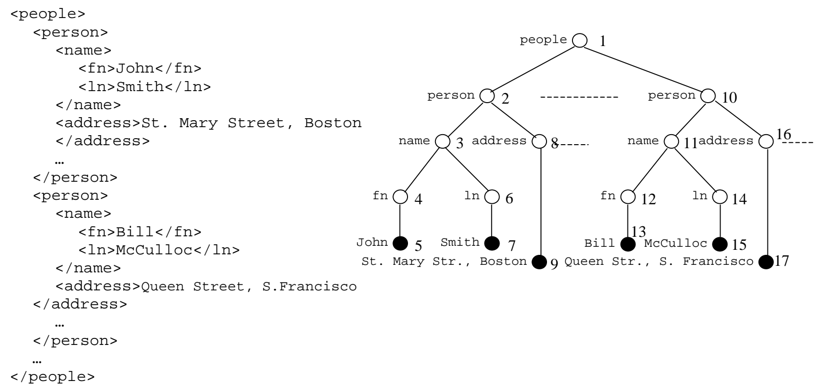

Fig. 2. An example of XML data and its tree representation

cient in case of long paths or element names with many occurrences. We propose an alternative approach that exploits the rotated lexicon technique above. In this way, typical XPath expressions, containing wildcards, are processed efficiently even in presence of long paths and high frequency of element names.

An XML document can be seen as a flat representation of a tree structure. For example, see Figure 2 for a portion of an XML document and its tree representation. In the figure, white nodes represent XML elements, and black nodes represent the XML content. Additional node types can be used to represent elements' attributes – in our example we omit them for the sake of simplicity. To identify specific elements in an XML document, nodes of the tree are also associated with a unique identifier, which we call *element instance identifier*, assigned with a preorder visit to the tree. In the remainder of this paper, XML document of Figure 2 and the corresponding tree representation will be used as a running example to explain the technique that we propose.

A simple possibility of indexing the structure of XML documents is to use an inverted index to associate each pathname, appearing in an XML document, with the list of its occurrences. For instance, in our example, the path `/people/person/address` is associated with a posting list containing element instance identifiers 8 and 16. This approach has some similarity to text indexing, considering that the paths play the role of terms and the names of elements play the role of characters. In text retrieval systems, each term is associated with the list of its occurrences. Here each path is associated with the list of elements that can be reached following the path. By analogy to the terminology used in text retrieval systems, we call *path lexicon* and *element lexicon*, respectively, the set of pathnames and the set of element names occurring in an XML document repository.

By exploiting this analogy, our proposal is to use the rotated lexicon technique from Section 2.2 to build a *rotated path lexicon*. In this way, we are able to efficiently process common path expressions containing wildcards, with no

Element lexicon:	Encoded element lexicon
Term. element	0
people	1
person	2
name	3
fn	4
ln	5
address	6

Path lexicon	Encoded path lexicon	Posting lists:
/people	/1/0	1->{1}
/people/person	/1/2/0	2->{2,10}
/people/person/name	/1/2/3/0	3->{3,11}
/people/person/name/fn	/1/2/3/4/0	4->{4,12}
/people/person/name/ln	/1/2/3/5/0	5->{6,14}
/people/person/address	/1/2/6/0	6->{8,16}

Rotated path lex.	Posting list ID:
/0/1	1
/0/1/2	2
/0/1/2/3	3
/0/1/2/3/4	4
/0/1/2/3/5	5
/0/1/2/6	6
/1/0	1
/1/2/0	2
/1/2/3/0	3
/1/2/3/4/0	4
/1/2/3/5/0	5
/1/2/6/0	6
/2/0/1	2
/2/3/0/1	3
/2/3/4/0/1	4
/2/3/5/0/1	5
/2/6/0/1	6
/3/0/1/2	3
/3/4/0/1/2	4
/3/5/0/1/2	5
/4/0/1/2/3	4
/5/0/1/2/3	5
/6/0/1/2	6

Fig. 3. Element lexicon, path lexicon, rotated path lexicon, and posting lists relative to the running example.

need of containment joins. We call the *rotated path* a path generated through the rotation. Note that names of attributes can be also be indexed by using this technique. In fact, they can be considered as children of the corresponding elements and managed similarly to the elements. The special character `@`, accordingly to the XPath syntax, is added in front of each attribute name to distinguish them from elements.

In fact, XPath uses two different types of wildcards. One is // and stands for any descendent element or self (that is 0 or more optional steps). The other is * and stands for exactly one element[3] (that is exactly one step). Let P, $P1$, and $P2$, be *pure path expressions*, that is path expressions containing just a sequence of element (and attribute) names, with no wildcards, and predicates. In addition to pure path expressions, the rotated path lexicon allows processing the following path expressions containing wildcards: $//P$, $P1//P2$, $P//$, and $//P//$[4]. This is obtained by using the query translation rules discussed in Section 2.2.

With a small additional computational effort, we can also process paths $*P$, $P1*P2$, $P*$, and $*P*$. The idea is to use again the query translation rules and filter out, from the result obtained by searching the rotated path lexicon, paths whose length is not equal to the length of the query path. Other generic XPath expressions can be processed by decomposing them in sub-expressions, consistent with the patterns described above, and combing them by using containment joins.

[3] Note that in text retrieval systems * is typically used to substitute any sequence of characters, as we said in Section 2.2, so the XPath correspondent is // rather *.

[4] To be precise, P// and //P// should be completed as P//`node()` and //P//`node()`, to be valid XPath expressions. For simplicity we omit the `node()` function.

To reduce the storage space required for the rotated path lexicon[5], each element name is encoded by an unique identifier (not to be confused with element instance identifiers introduced before) implemented, for instance, as an integer. Thus, pathnames are represented as *encoded pathnames* consisting of sequences of encoded elements, instead of strings. A specific identifier is reserved for the *path terminating element*.

To illustrate, Figure 3 shows the element lexicon, the path lexicon, and the rotated path lexicon, obtained from our example, along with their respective encoding. The list of element instance identifiers (the posting lists), associated with each pathname in the example, is shown as well.

4 Indexing XML data structures and their content

Suppose the XPath expression `/people//name[fn="Bill"]/fn`. To process this query, content predicates, in addition to structural relationships, should be separately verified. This can be inefficient since either the access to the document, in case of non indexed content, or additional containment joins, in case of indexed content, are required. However, the rotated path index technique can be extended in such a way that content predicates and structural relationships can be handled simultaneously. In the following, two different implementation directions are discussed.

4.1 Structure+content queries by extending the path index

Content of an element can be seen as a special child of the element so it can also be included as last element of a path. We add a special character `<`[6] in front of the content string to distinguish it from name of elements and attributes. For instance, `Bill` will be `<Bill`, and the path from the root to the content is `/people/person/name/fn/<Bill`. Suppose $P/<cont$ is a pathname, where *cont* is the content and P the path from the root. The posting list associated with $P/<cont$ (and its rotations) contains all elements reachable via P that have content *cont*. This is a subset of the posting list associated with P. Of course, it does not make sense to index content of all elements and values of all attributes. The database administrator can decide, tacking into account performance issues, which elements and attributes should be indexed.

By using this extension, our original XPath expression can simply be processed by a single access to the path index as:

1. let R_1 = *pathIndexSearch*(`/people//name/fn/<Bill`);
2. return R_1

[5] The number of entries in the rotated path lexicon is $\#PL \times (avg_PL_len + 1)$, where $\#PL$ is the cardinality of the path lexicon and avg_PL_len is the average length of paths in the path lexicon.

[6] We use `<` as special flag to distinguish content since content of an element or attribute cannot start with this.

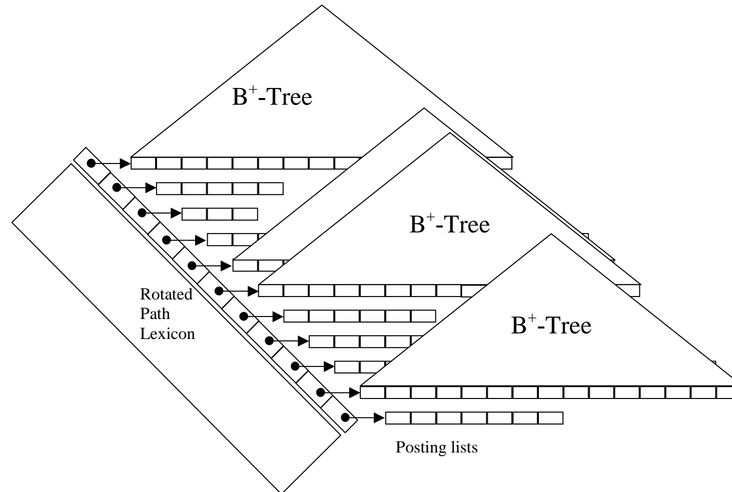

Fig. 4. Some posting lists are indexed by a dedicate B^+-Tree, so content predicates on elements of these posting lists can be processed efficiently.

so we are able to process XPath expressions that contain equality predicates on specific elements or attributes, with just one access to the path index.

4.2 Structure+content queries by indexing posting lists

Another possibility to support efficient processing of path expressions with content predicates is to organize the posting lists by using specific access methods. For instance, each posting list corresponding to frequently searched elements can be indexed with a different B^+-Tree that uses content of elements as keys. Elements satisfying a predicate can be efficiently retrieved from these posting lists. This idea is illustrated in Figure 4. In this approach, a path expression can be processed in two steps. First, the path index is searched to find the posting lists satisfying the structural part. Then, the obtained indexed posting lists are searched through the associated access method. The advantage of this technique is that all predicates supported by the access method used to index the posting list, can be processed efficiently.

Our query example `/people//name[fn="Bill"]/fn` can be processed with just two index accesses as:

1. let R_1 = *pathIndexSearch*(`/people//name/fn`);
2. let R_2= *contentIndexSearch*(R_1,"`Bill`");
3. return R_2

where we suppose that the posting list R_1 associated with `/people/person/name/fn` is indexed with a B^+-Tree, so it can be searched as a content index.

5 Experiments

We compared our path index with the containment join, implemented as a Multi Predicate MerGe JoiN (MPMGJN) according to the specification from [7]. To run our experiments, we have used the XBench [6] benchmark. Specifically, we have used the Text Centric Multiple Documents (TC/MD) version. In order to test the scalability of the path index, we have modified the XBench Perl scripts and generated three different datasets of increasing sizes. The conducted experiments have demonstrated a systematic superiority of the path index. This can be explained by observing that in both cases the dominant costs are due to the retrieval of the posting lists. The path index just retrieves the final posting lists, while the containment join has to retrieve posting lists for each element name specified in the path expression. The path index has the property of better scaling when the number of XML elements, and consequently the posting lists associated with elements names, increases. Using the largest dataset, the performance of the containment join becomes more than one order of magnitude less efficient with respect to the path index. A detailed description of the experiments and a deep analysis of the results can be found in [1].

6 Conclusions

We have proposed a path index that supports efficient processing of typical path expressions containing wildcards. Extensive evaluations have demonstrated the superiority of our approach to the previously proposed techniques.

References

1. Giuseppe Amato, Franca Debole, Fausto Rabitti, and Pavel Zezula. YAPI: Yet another path index for XML searching. In *ECDL 2003, 7th European Conference on Research and Advanced Technology for Digital Libraries, Trondheim, Norway, August 17-22, 2003*, 2003.
2. P. Brately and Y. Choueka. Processing truncated terms in document retrieval systems. *Information Processing & Management*, 18(5):257 – 266, 1982.
3. World Wide Web Consortium. XML path language (XPath), version 1.0, w3c. Recommendation, November 1999.
4. World Wide Web Consortium. XQuery 1.0: An XML query language. W3C Working Draft, November 2002. http://www.w3.org/TR/xquery.
5. Gerald Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, 1983.
6. Benjamin Bin Yao, M. Tamer Özsu, and John Keenleyside. XBench - a family of benchmarks for XML DBMSs. Technical Report TR-CS-2002-39, University of Waterloo, December 2002. http://db.uwaterloo.ca/~ddbms/projects/xbench/index.html.
7. Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In Walid G. Aref, editor, *ACM SIGMOD Conference 2001: Santa Barbara, CA, USA, Proceedings.* ACM, 2001.