YAPI: Yet Another Path Index for XML searching

Giuseppe Amato[1], Franca Debole[1], Pavel Zezula[2], and Fausto Rabitti[3]

[1] ISTI-CNR, Pisa, Italy,
{G.Amato,F.Debole}@iei.pi.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
[3] F.Rabitti@cnuce.cnr.it
WWW home page: http://www.isti.cnr.it

Abstract. As many metadata are encoded in XML, and many digital libraries need to manage XML documents, efficient techniques for searching in such formatted data are required. In order to efficiently process path expressions with wildcards on XML data, a new path index is proposed. 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.

Categories and Subject Descriptors: E.1 [Data Structures]: Records, H.2.2 - [Database Management]: Physical Design-Access Methods, H.3.3 [Information Storage and Retrieval]: Information Searching and Retrieval-Search process
Free Keywords: XML searching, Path Index, XML query processing

1 Introduction

Efficient management of metadata is an important issue in Digital Library systems. Simple flat solutions to metadata collections such as the Dublin Core cannot be applied to complex metadata models requested by advanced multimedia digital libraries. These complex metadata models often include nested structures, hierarchies, multiple views, and semistructured information, which cannot efficiently be handled by techniques based on a simple term inversion or by application of the relational databases. Instead, efficient technology for the management of such data should be developed.

Recently, several interesting trends to processing complex metadata by means of XML structures have been observed. Specifically, new generation digital libraries, such as ECHO [1] or OpenDlib [5], have chosen to encode their supported metadata with XML. Recent standards, for instance the MPEG-7 [9], require to encode metadata in XML. Moreover, some digital libraries consider XML documents as their native format of documents.

The obvious advantage of encoding metadata 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, 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 [7]and XQuery [8], 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 experiments have shown an evident superiority of our 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 also 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* [10]. 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 [4]. 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 [3,6]. Rotated versions of all terms in the original lexicon are mapped to the posting list associated with the original term.

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*.

Then the transformed query terms are used to search in the rotated lexicon. For example, suppose that 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.

A drawback of this technique is the memory overhead due to the rotation. In fact, an average memory overhead observed in [13] is about 250%. A memory reducing variant of this method is discussed in [2]. The memory overhead is reduced by representing the rotated lexicon as an array of pointers, one for each position in the original lexicon. This array of pointers is sorted accordingly to the rotated form of each term. By using this technique, [13] reports the memory overhead of about 30%.

3 Rotated Path Index

Wildcards are also frequently used in XPath expressions. These expressions are typically processed by multiple *containment joins* [12], which can be very inefficient 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

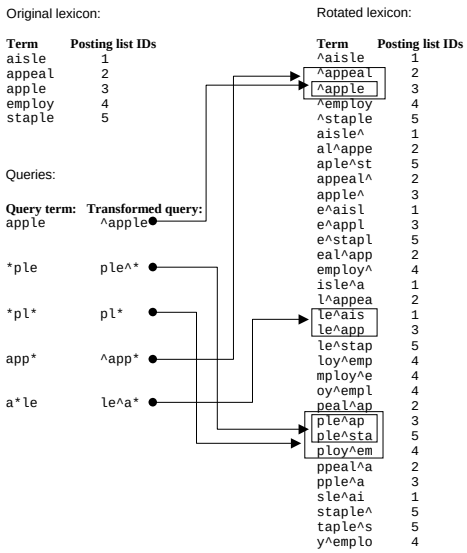

Fig. 1. The original lexicon, the rotated lexicon, and some queries processed on the rotated lexicon

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.

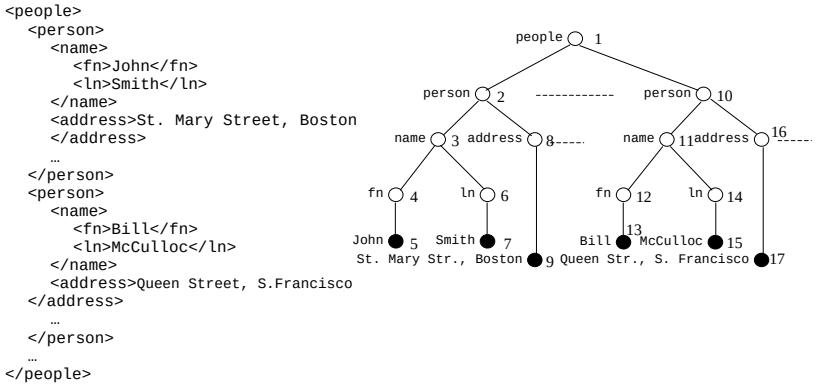

Fig. 2. An example of XML data and its tree representation

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 need of containment joins. We call the *rotated path* a path generated through the rotation. Note that names of attributes can 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.

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[4] (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 $//P//$[5]. 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 combining the obtained results through containment joins.

[4] 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 `*`.

[5] To be precise, note that P// and //P// alone are not syntactically valid XPath expression. In fact, they should be completed as P//`node()` and //P//`node()`, for instance. In this paper, we simplify the notation by omitting the `node()` function.

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 XML example in Figure 2.

The amount of storage memory required for the rotated path lexicon can be large, especially in case of entries consisting of long pathnames. To reduce the storage space, 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 list), associated with each pathname in the example, is shown as well.

Example Suppose the XPath expression `//person/name//`. The element name identifiers associated with `person` and `name` are 2 and 3, respectively, and the encoded query is //2/3//. According to the query translation rules, this query is processed by searching the rotated lexicon for /2/3//. Rotated paths that qualify for this query are /2/3/0/1, /2/3/4/0/1, and /2/3/5/0/1, corresponding to path names `/people/person/name`, `/people/person/name/fn`, and `/people/person/name/ln`. By merging their posting lists, we decide elements relevant to this XPath expression as those associated with the element instance identifiers {3,4,6,11,12,14}.

4 Indexing XML data structures and their content

Suppose the XPath expression `/people//name[fn="Bill"]/fn`. This returns all `fn` elements that are children of a `name` element, and that are descendants of a `people` root element, whose content is exactly the string `Bill`. 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`. In Section 3, we proposed a similar technique to index names of attributes.

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 the list of 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 attributes. The database administrator can decide, tacking into account performance issues, which elements and attributes should have their content indexed.

Suppose in our example that we decide to index the content of elements `fn`. In this case, pathnames `/people/person/name/fn/<Bill` and `/people/person/name/fn/<John` are added to the path lexicon. The rotated path lexicon is also updated with the corresponding rotated paths. The corresponding posting lists are {4} and {12}. Note that in this example the posting lists contain just one element. However, if several persons whose first name is `Bill` occur, the corresponding posting lists would be larger.

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

and 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 predicates on content is to organize the element instance identifiers of posting lists by using specific access methods. For instance, each posting lists corresponding to frequently searched elements can be indexed with a different B^+-Tree that uses

[6] We use < as flag since content of an element or attribute cannot start with it.

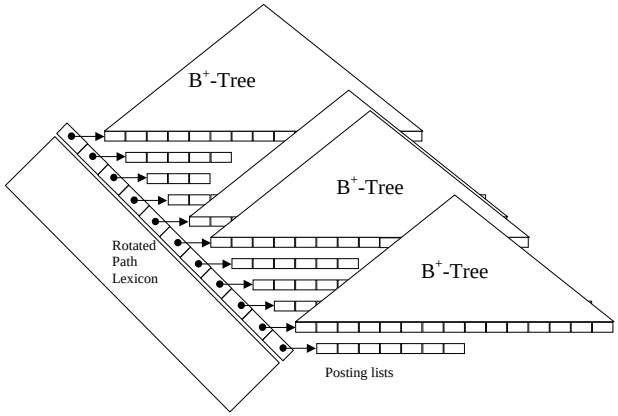

Fig. 4. Some posting lists are indexed by a dedicate B^+-Tree, so content predicates on elements of these posting lists can be processed efficiently.

content of elements as keys. This implies that 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, using the content predicate. Posting lists that are not indexed should be searched checking the content of each element.

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.

The advantage of this technique is that all predicates supported by the access method used to index the posting list, such as $<$, $>$, $\leq$, $\geq$, and $=$, in case of B^+-Tree, can be processed efficiently.

5 Experiments

The path index was implemented by using BerkeleyDB. Specifically, the search structure was implemented as a B^+-tree with multiple keys, so posting lists are automatically managed by BarkeleyDB. Elements in posting lists contain, in addition to the element instance identifier, the start and the end position of the corresponding element – start and end positions are, respectively, the

	Small Dataset	Medium Dataset	Large Dataset
# XML files	430	4300	43000
<article> occurrences	430	4300	43000
<prolog> occurrences	430	4300	43000
<authors> occurrences	405	4037	40432
<author> occurrences	10047	98523	989752
<contact> occurrences	10047	98523	989752
<email> occurrences	9224	90729	910420
# total elements	625726	5684115	56505321

Table 1. Number of files, occurrences of specific elements, and total number of elements in the three generated datasets.

positions of the start and the end tags of elements in XML documents. In this way, containment joins can still be used to process queries that are not compliant to the supported path expressions.

We compared our path index with the containment join according to the implementation from [12]. Accordingly, an *Element Index* was used that associates each element with its (start,end) position and the containment join was implemented as the Multi Predicate MerGe JoiN (MPMGJN). The element index was developed as a B^+-tree with multiple keys in BerkeleyDB.

Retrieval of the posting list associated with a key (a rotated path in case of the path index, an element name in case of the element index) was implemented with the bulk retrieval functionality, provided by the BerkeleyDB. Everything was implemented in Java, JDK 1.4.0 and run on a PC with a 1800 GHz Intel pentium 4, 512 Mb main memory, EIDE disk, running Windows 2000 Professional edition with NT file system (NTFS).

We have used a benchmark from XBench [11] to run our experiments. Specifically we have used the Text Centric Multiple Documents (TC/MD) benchmark whose Schema diagram is shown in Figure 5. This benchmark simulates a repository of text documents similar to the Reuters news corpus or the Springer Digital library. It consists of numerous relatively small text-centric documents with explicit structure description, looseness of schema and possibly recursive elements. We have modified the XBench Perl scripts to be able to control the number of generated XML files. Then, we have generated three different datasets with increasing size, to test the scalability of the path index with increasing number of elements. Statistics of the three generated datasets can be seen in Table 1.

We have run the experiments using various path expressions based on the query patterns supported by the path index. We have coded query names as Q<*p*><*l*>, where <*p*> takes values between 1 and 4, in correspondence of the query pattern tested, <*l*> indicates the length of the path expression, in terms of number of element names, and can be L(ong) or S(hort). Table 2 details the test queries, while the number of occurrences in the datatses of the element names that we have used in the queries is reported in Table 1. We have processed Q1<*l*> and Q2<*l*> both with path index and containment join. The other queries, Q3<*l*>

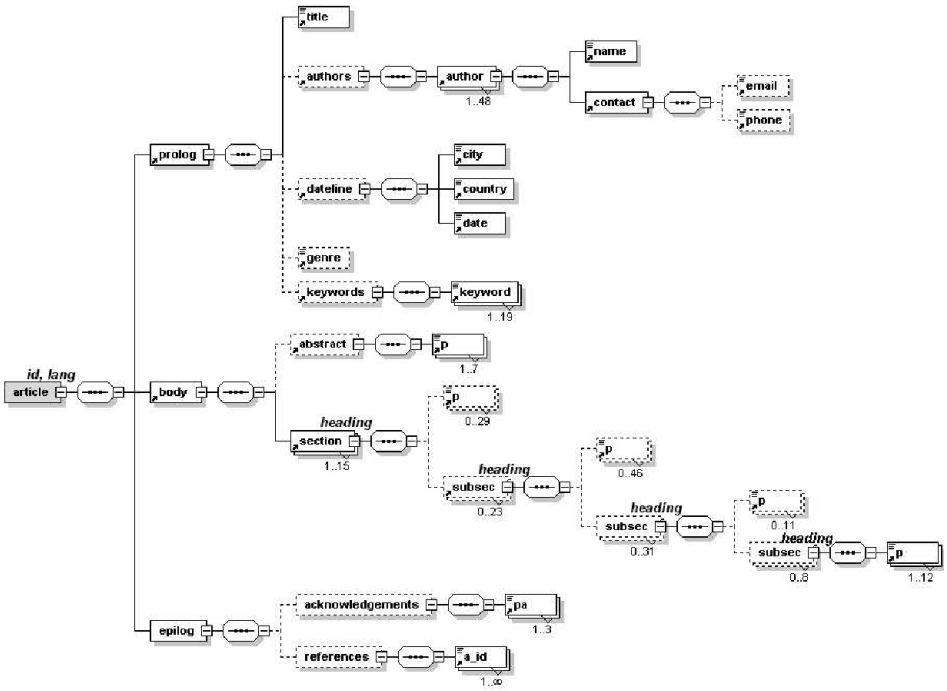

Fig. 5. Schema Diagram of the used benchmark

and Q4<*l*>, were only processed with the path index, since processing them with the containment join only is not possible.

Performance comparison between path index and containement join is shown in Table 3 were the processing time, expressed in milliseconds, and the number of elements retrieved by each query is reported. Processing time includes access to the path index and retrieval of the posting lists. In case of containment join, processing time includes access to the element index and execution of the join algorithm (which also include retrieval of the needed posting lists).

The experiments we have performed have shown an evident superiority of the path index with respect to the containment join. The difference in performance can be justified by observing that, even if not reported, the dominant cost is due to the retrieval of the posting lists, for both path index and containment join. In fact, the cost of accessing the corresponding search structure is negligible. However, while in case of the path index just the final posting lists should be retrieved, the containment join has to retrieve a posting list for each element name specified in the path expressions, in order to join them. Thus as expected, performance of the path index is practically independent on the length of the path expression, while performance of the containment join degrades with longer path expressions. This is evident when the size of the intermediate posting lists, corresponding to the number of occurrences of the intermediate element names

Query	Pattern	XPath expression
Q1L	//P	//authors/author/contact/email
Q1S		//contact/email
Q2L	P1//P2	/article/prolog//contact/email
Q2S		/article//email
Q3L	P//	/article/prolog/authors/author//
Q3S		/article/prolog//
Q4L	//P//	//authors/author//
Q4S		//author//

Table 2. Queries used for the experiments

Query	Dataset	Path index	Cont. join	#Retr. el
Q1L	Small	19	86	9224
	Medium	75	612	90729
	Large	1079	62601	910420
Q2L	Small	19	81	9224
	Medium	75	421	90729
	Large	1074	48268	910420
Q3L	Small	106	-	46269
	Medium	906		454239
	Large	5056		4562725
Q4L	Small	106	-	46269
	Medium	906		454239
	Large	5020		4562725
Q1S	Small	19	53	9224
	Medium	75	194	90729
	Large	1100	25686	910420
Q2S	Small	19	36	9224
	Medium	75	132	90729
	Large	1092	3351	910420
Q3S	Small	267	-	53119
	Medium	590		521753
	Large	5435		5242441
Q4S	Small	105	-	46269
	Medium	904		454239
	Large	4982		4562725

Table 3. Performance comparison. Time is expressed in milliseconds.

specified in the query, is large. The path index does not need to access these huge posting lists since it directly accesses the posting lists associated with entire paths that match the query. As a consequence, the path index has also the property of better scaling when the number of XML elements, and consequently the size of posting lists associated with elements names, increase. In our exper-

iments, this is particularly evident for queries `Q1L`, `Q2L`, and `Q1S`, where, using the large dataset, performance of the containment join becomes more than one order of magnitude worse than the path index.

6 Conclusions

We have proposed a path index that supports efficient processing of typical path expressions containing wildcards. The proposed index structure can be easily extended to also support path expressions containing content predicates in addition to constraints on structural relationships. Extensive evaluations have demonstrated the superiority of our approach to the previously proposed techniques. Currently, we are integrating the path index in the OpenDlib [5] system.

References

1. Giuseppe Amato, Claudio Gennaro, and Pasquale Savino. Indexing and retrieving documentary films: managing metadata in the ECHO system. In *4th Intl. Workshop on Multimedia Information Retrieval December 6, Juan-les-Pins, France, in conjunction with ACM Multimedia*, 2002.
2. G. Gonnet anf R. Baeza-Yates. *Handbook of data structure and algorithms.* Addison-Wesley, Reading, Massachussets, second edition, 1991.
3. Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.
4. P. Brately and Y. Choueka. Processing truncated terms in document retrieval systems. *Information Processing & Management*, 18(5):257 – 266, 1982.
5. Donatella Castelli and Pasquale Pagano. Opendlib: A digital library service system. In Maristella Agosti and Constantino Thanos, editors, *6th European Conference, ECDL 2002, Rome, Italy, September 16-18, 2002, Proceedings*, volume 2458 of *LNCS*, pages 292–308. Springer, 2002.
6. Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
7. World Wide Web Consortium. XML path language (XPath), version 1.0, w3c. Recommendation, November 1999.
8. World Wide Web Consortium. XQuery 1.0: An XML query language. W3C Working Draft, November 2002. http://www.w3.org/TR/xquery.
9. N. Day and J.M. Martnez. Introduction to MPEG-7 (v4.0). working document N4675, 2002. Available at: http://mpeg.telecomitalialab.com/working_documents.htm.
10. Gerald Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, 1983.
11. 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.
12. 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.

13. Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 290–301. Morgan Kaufmann, 1993.