

# Index Compression vs. Retrieval Time of Inverted Files for XML Documents\*

Norbert Fuhr

Norbert Gövert

University of Dortmund, Germany

<http://ls6-www.cs.uni-dortmund.de/ir/projects/hyrex/>

## ABSTRACT

Query languages for retrieval of XML documents allow for conditions referring both to the content and the structure of documents. In this paper, we investigate two different approaches for reducing index space of inverted files for XML documents. First, we consider methods for compressing index entries. Second, we develop the new *XS tree* data structure which contains the structural description of a document in a rather compact form, such that these descriptions can be kept in main memory. Experimental results on two large XML document collections show that very high compression rates for indexes can be achieved, but any compression increases retrieval time. On the other hand, highly compressed indexes may be feasible for applications where storage is limited, such as in PDAs or E-book devices.

## Categories and Subject Descriptors

H.3.2 [Information Storage and Retrieval]: Information Storage—*File organization*

## 1. INTRODUCTION

XML allows for logical markup of texts both at the macro level (for example chapter, section, paragraph) and the micro level (e. g. MathML for mathematical formulas and CML for chemical formulas). In order to exploit this markup in information retrieval applications, an appropriate query language must be used. The XIRQL language [2] extends the XPath subset of XQuery by IR concepts such as weighting, ranking and relevance-oriented search.

In order to perform efficient retrieval on XML documents, appropriate index structures must be used, which include the structural information. In this paper we present two general approaches for making structural information needed for XML retrieval available through inverted files. Thom et al. describe a method to integrate structural information into inverted lists, and propose different compression methods [5]. Our *PIL* approach described in Section 3 is based on these ideas. Even if compression methods are

\*The full version of this article with detailed descriptions of the methods and experimental results presented here can be obtained from the web site depicted in the title.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'02, November 4–9, 2002, McLean, Virginia, USA.

Copyright 2002 ACM 1-58113-492-4/02/0011 ...\$5.00.

used, the index size grows significantly by adding structural information.

In contrast, the second approach presented here minimizes the index size: The structural information stored is reduced to a *path handle* which can be used along with our new *XS tree* data structure to reconstruct the structural information at retrieval time (Section 4).

For both approaches, the compressed index entries have to be decompressed during retrieval, thus savings in terms of storage space may lead to higher retrieval times. In order to show this tradeoff, we evaluated four variants of the two approaches, each with a different level of compression (Section 5).

First we specify the functionality of access methods for XML retrieval.

## 2. XML QUERYING

In order to process queries referring to the logical structure of documents, XML query languages must support the following types of conditions:

**Element names:** For high-precision retrieval, the queries must be allowed to specify the element name where certain values should occur (e. g. in an arts information system, ‘London’ in the title of an image vs. ‘London’ in the location of creation).

**Element index:** When the index has a specific semantics, then users may want to specify the value of an index (e. g. first chapter of a book, first author of a publication).

**Ancestor/descendant:** Since XML documents have a tree structure, the ancestor/descendant relationship describes the logical structure of a document (aggregation of document parts). Queries may refer explicitly to this relationship (e. g. find a chapter that gives a theorem and also contains the corresponding proof). Relevance-oriented retrieval needs this information in order to identify logical parts fulfilling best the conditions specified in the query.

**Preceding/following:** This type of conditions refers to the linear sequence within the document. For example, to find a document explaining the vector space model in terms of uncertain inference, we may look for a paper that talks about uncertain inference before mentioning the vector space model.

Given these different types of conditions, we may now think about access structures that contain all the necessary information for testing these conditions for (value-based) IR queries. A content-oriented query always contains values (e. g. terms), and structural conditions are used for increasing the precision of retrieval.

Developing access methods for XML retrieval means to consider both the values and the structural information. That is, we need additional information describing the within-document location of a

term or value. Considering XML documents as tree, this information is given by a so-called *path*. A path describes the sequence of nodes from the document root to a specific element. So it consists of a sequence of path steps, where each step corresponds to an element. In order to check for the different types of conditions listed above, each element must be described by its element name and index as well as its sequence index. This way conditions referring to element names or indexes can be answered directly, the ancestor/descendant relationship can be inferred from the sequence of path steps, and the sequence index allows for identifying preceding or following elements.

As an example, consider the path `/book[1,1]/chapter[1,3]/section[2,3]/p[3,4]`. Here the different steps are separated by a slash. For each step, we first give the element name and then a pair of indexes, namely the element index and the sequence index. The `chapter` element for example is located on the second level of the XML document tree. It is the first `chapter` node below the `book` root node, but the third child of the `book` node.

In order to have access to information provided by paths at retrieval time, this information must be directly included in or referenced to from the postings of the inverted lists. Using an XML-like notation for regular expressions, we can describe the abstract data structure for inverted lists as follows (here ‘?’ stands for an optional element, ‘+’ for at least one occurrence and ‘\*’ for an arbitrary, including zero number of occurrences):

```
invlist    -> docentry+
docentry   -> docid num_occ occurrence+
occurrence -> path_information weight?
```

In the following two sections we describe how the path information within the postings can be encoded. To decode the remaining items, methods like those described by Witten et. al [6] can be used. For most of the issues discussed in the following, we do not consider indexing weights explicitly, since they just add a constant factor in terms of space and time<sup>1</sup>.

### 3. PATHS IN INVERTED LISTS (PIL)

Within the *Path in Inverted Lists (PIL)* approach all path information is encoded as part of the entries of the inverted lists. However, a straightforward implementation of this idea requires vast amounts of disk space, i.e. the index will be larger than the original data. Thus, we are looking at methods for compressing the structural information. Concepts for this step have been proposed already in [5]. Here we briefly describe the basic ideas.

An obvious format for encoding such paths is described by the following syntax:

```
path -> path_length element+
      element_index+ sequence_index+
```

As a simple example, consider an inverted list entry which refers to a document position referenced by the path `/book[1,1]/chapter[1,3]/section[2,3]/p[3,4]`. The corresponding information to encode would look like this (here angular brackets and spacing are used for illustration purposes only):

```
< 4 <book chapter section p>
  <1 1 2 3> <1 3 3 4> >
```

Given that entries in inverted lists are sorted by position, there is some redundancy, which can be used for compression. As a first step, common prefixes of subsequent path entries can be eliminated. Instead of the path length, we only encode the difference between prefix length and path length. In order to compress the

<sup>1</sup>Also, some weighting methods may not need any additional space, since all the occurrence information they need is already present in the inverted list.

document numbers, run length encoding is used. For the sequence indexes (which were not considered by Thom et al. [5]), we only encode the difference to the corresponding element indexes. In order to compress this data, universal codes [1] are used for the numbers and canonical Huffman codes [3] for the element names.

## 4. THE XS TREE

As an alternative to compressed path information in inverted files, we have developed the XML Structure tree. The XS tree is an additional data structure that describes the structure of an XML document. It is highly compressed such that the XS trees of a whole document collection can be kept in main memory. Now the occurrence entry in the inverted list is reduced to a *path handle*, that is a single number pointing to a position in the corresponding XS tree (indicated by the document ID of the entry). Given this position, there is a resolution method that yields the corresponding path.

In order to achieve a rather compact representation of such a tree, compression methods should be used as much as possible. For creating such a linear representation of the XS tree, the following design was chosen.

**Enumerate nodes top-down (in preorder):** By choosing a top-down sequence, we can apply context-specific compression methods. Given the DTD, there is only a small set of elements that can occur as children of a specific element, so we only need a few bits for coding each of these alternatives.

**Parent-child relationship via level numbers:** Level numbers are compact. For compressing level numbers, we use run length encoding, thus only the relative differences between the level numbers are stored.

**Positions as element numbers:** Element numbers are compact, and therefore efficient encoding for use as path handles in the inverted lists is possible.

**Element and sequence indexes implicit:** Each element in a path has an element index and a sequence index that denote its relative position among the children of the parent node. These indexes could be encoded explicitly, but would require additional storage space; in contrast, by scanning the representation linearly, the indexes of each element can be computed on the fly.

Universal codes are used for compression of the level numbers. In order to encode the element names given for each node of an XML tree, we use Huffman coding.

To sum up, an access structure for XS trees has the following structure:

```
xs_access -> xstree+
xstree    -> node+
node      -> level element_name
```

## 5. EVALUATION

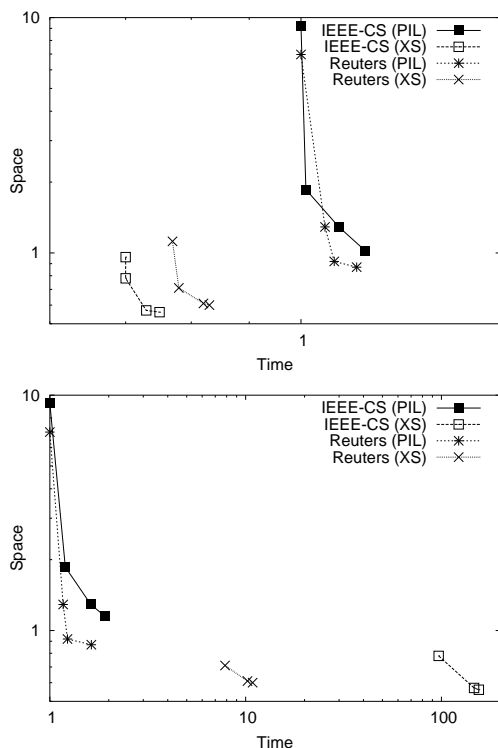
We performed an experimental evaluation of the approaches presented before. For both approaches four variants with different compression levels have been evaluated. For characterizing the efficiency of the different approaches, we considered the following parameters:

**Indexing time:** For applications with frequent updates, indexing time may be a crucial factor. Since we have not focused on an efficient implementation of the indexing process, our timings only indicate the relative effort for building the different index structures.

**Size of index:** Since our major topic is index compression, we are measuring the size of the inverted lists and XS trees.

**Retrieval time:** Following the standard tradeoff of space vs. time in computer applications, we want to know the effects of index compression on retrieval time.

For our experiments, we used two fairly large XML collections with quite different characteristics: The Reuters Corpus<sup>2</sup> which contains about 810 000 Reuters, English language news stories (2 369 megabytes). The structure of these documents is similar to those documents that have been used in many classical IR evaluations (e.g. the TREC collection). As another extreme, we chose the fulltexts of IEEE Computer Society publications during the years 1995–1997<sup>3</sup> (IEEE-CS). In terms of the number of documents (5 063 documents, 403 megabytes), this collection is quite small compared to the Reuters Corpus. But the IEEE-CS documents are typical journal papers, so their average size is much higher than that of the Reuters documents. Also, the structure of the IEEE-CS documents is rather complex and heterogeneous (about 2 000 nodes per document in the XML structure, at path level 6 on average), compared to the structure of the Reuters documents (about 100 nodes, path level 4).



**Figure 1: Space vs. time for indexing (above) and retrieval (at the bottom)**

In Figures 1, we have plotted the space-time tradeoff for both the indexing and the retrieval tasks. (Here space is relative to the size of the compressed document collection, and time is measured relative to the *PIL* variant without compression.) As mentioned before, in indexing, the *XS* tree approach is a nice exception; both space and time are reduced. For retrieval, we have the classical tradeoff. For IEEE-CS, the *XS* retrieval times clearly are too high. The major reason for this behavior is the large size of the documents. If we

<sup>2</sup><http://about.reuters.com/researchandstandards/corpus/>

<sup>3</sup>This collection can be ordered as CD set from <http://www.computer.org/cspress/catalog/cs-96.htm>.

could split up the documents in smaller parts, retrieval time would be reduced proportionally to the size of these parts (e.g. a split by a factor of 10 would result in retrieval times comparable to the Reuters retrieval times).

The space-time tradeoff also is confirmed by a closer look at the retrieval times, where decoding/decompression time is the dominant factor. For *PIL* without compression, decoding processes about 1 MB/s (and decompression needs more time per path), whereas I/O rates are about 20 MB/s. Whereas in [4], decompression and I/O time were in the same order of magnitude, the complex structure of XML documents leads to this imbalance. Thus, also additional mechanisms (like the skip lists described by Moffat and Zobel [4]) will not be able to overcome the space-time tradeoff observed.

The retrieval time differences between *XS* and *PIL* variants are reduced when we have multiple matches per document. In standard IR applications, those documents with multiple matches are most interesting. With linear retrieval functions, the top ranking documents usually contain several terms. Thus, it would be possible to implement a retrieval strategy where the candidates for the top ranks are determined before resolving of the *XS* path handles starts.

Another relevant application area are conjunctive queries, where only the documents containing all the query terms are considered (as in some Web search engines for example). We ran a small series of experiments with conjunctive queries generated by choosing  $n$  random terms from the same random document. It turned out that for Reuters, the average cooccurrence rate for two terms (and thus the fraction of *XS* path handles that must be resolved) is about 4 % (IEEE-CS: 16 %). In this situation, *XS* retrieval would be faster than the *PIL* variants.

## 6. CONCLUSIONS AND OUTLOOK

Most of the discussion in this paper has focused on single-term queries. For queries with multiple conditions, the difference between the two approaches can be reduced by applying appropriate retrieval strategies where first the candidates for the top ranks are determined, before decompressing the structural information. This strategy also can be used for the combination of the *XS* tree with other access structures that compute a superset of the result first (i.e. based on hashing or signatures), in order to determine the correct answers. We will continue our research along these lines.

## 7. REFERENCES

- [1] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2): 194–202, March 1975.
- [2] N. Fuhr and K. Großjohann. XIRQL: A query language for information retrieval in XML documents. In *Proceedings of the SIGIR*, pages 172–180, New York, 2001. ACM.
- [3] Daniel S. Hirschberg and Debra A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4):449–459, 1990.
- [4] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.
- [5] James A. Thom, Justin Zobel, and Bruce Grima. Design of indexes for structured document databases. Technical Report TR-95-8, Collaborative Information Technology Research Institute, Melbourne, Australia, 1995.
- [6] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 2nd edition, 1999.