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

Norbert Fuhr   Norbert Gövert

University of Dortmund, Germany

Query languages for retrieval of XML documents allow for conditions referring both to the content and the structure of documents. In order to process these queries efficiently, inverted files must contain also structural information, thus leading to index sizes that exceed the storage space of the original data. In this paper, we investigate two different approaches for reducing index space. 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. We evaluate the efficiency of several variants of these two approaches on two large XML document collections. Results show that very high compression rates for indexes can be achieved. However, any compression increases retrieval time. Thus, retrieval time is minimized when uncompressed indexes are used. On the other hand, highly compressed indexes may be feasible for applications where storage is limited, such as in PDAs or E-book devices.

1 Introduction

In the past, most IR approaches have ignored document formats or structure, partly due to the diversity of formats, partly due to the limited availability of structured documents. With the increased use of the *Extended Markup Language* (XML) as standard format for documents, this will change very soon.

XML allows for logical markup of texts both at the macro level (for example chapter, section, paragraph) and the micro level (e. g. MathML[1] for mathematical formulas, CML[2] for chemical formulas). In order to exploit this markup in retrieval, an appropriate query language must be used. Unfortunately, the XQuery proposal [Chamberlin et al. 01] by the World Wide Web Consortium's working group on XML query languages offers very little support for information retrieval of XML documents; instead, it focuses on the data-centric view of XML, thus offering features similar to query languages for (object-oriented) databases. As a more IR-related approach, the XIRQL language [Fuhr & Großjohann 01] extends the XPath subset of XQuery by IR concepts such as weighting, ranking and relevance-oriented search.

'Classical' text retrieval regards documents as unstructured text units, and queries contain value conditions only (i. e. search for texts containing certain terms). In contrast, all approaches for XML retrieval also support structural conditions, even in combination with values. Predecessors of this type of retrieval can be found in commercial text retrieval systems developed since the 1970s, where documents could be split up e. g. in fields, paragraphs and sentences, and the corresponding query languages supported conditions with respect to (w. r. t.) this (fixed) structure. Today, XML allows for more flexible (even recursive) document structures.

In order to perform efficient retrieval on XML document collections, appropriate index structures must be used. Most of the old commercial text retrieval systems included all the necessary structural information within the inverted lists, thus creating indexes that were often larger than the original texts. As described below, experiments have shown that also in the case of XML documents, uncompressed indexes roughly require as much space as the documents themselves. However, standard compression schemes for XML documents reduce collection size by 80–90 %; therefore index size is a crucial factor when space matters.

Thus, it seems to be appropriate to apply methods for index compression. Work in this direction also has to take into account the classical space-time tradeoff in computer science: Since the compressed index entries have to be decompressed during retrieval, savings in terms of storage space may lead to higher retrieval times. In this paper, we investigate a range of possibilities for index compression with different compression rates and retrieval

---

[1] http://www.w3.org/Math/
[2] http://www.xml-cml.org/

performance; this way, we offer a system implementor (or database administrator) a variety of solutions, from which the most appropriate one can be chosen for a given application.

So far, there is very little literature on index structures for XML document retrieval; moreover, we are not aware of any experimentation using large, realistic XML collections. [Thom et al. 95] describe a method to integrate path information into inverted lists, and propose different compression methods. Our *PIL* approach described below is based on these ideas. [Moffat & Zobel 96] investigate methods for compressing inverted lists in the case of unstructured documents, and they also develop a new retrieval strategy that reduces overall retrieval time in comparison to uncompressed lists. As an alternative to inverted files, [Yoon et al. 01] use bitmap indexing for indexing structured documents. Term occurrences are associated with respective paths and document identifiers by means of a so-called *BitCube*: the set of paths, the set of terms, and the set of document identifiers each form one dimension. It is claimed that very efficient retrieval can be achieved with this approach. However, neither element indexes nor indexing weights are represented in the BitCube, and an extension towards inclusion of this data seems to be difficult at least.

Since inverted files [Harman et al. 92] [Witten et al. 99, chapter 3] have shown their advantages in comparison with other access methods proposed in the literature, we base our work on this access structure and seek for modifications that reduce the storage requirements. For this purpose, we are investigating two different approaches in this paper. In order to reduce index space for inverted files, we can either compress the index entries, or we can reduce the information stored in the entries (e.g. no structural or positional information). In the latter case, the inverted file yields a superset of the correct answers, and we need additional sources in order to determine the final result. The most obvious solution in this case would be scanning of the original documents, i.e. retrieving the candidate documents and testing for the fulfillment of the actual query condition(s); however, this solution yields very high retrieval times. As an alternative, we consider the additional *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. In this case, the structural description in an inverted file entry is replaced by a pointer to a position in the XS tree; thus, the overall storage space of the index is reduced.

In the remainder of this paper, we first specify the functionality of access methods for XML retrieval. Then we describe the *PIL* method for compressing inverted lists. As an alternative, Section 4 presents our new XS tree data structure that reduces the structural information in the inverted list entries to a pointer linking to this tree. The efficiency of both approaches are evaluated in Section 5. Finally, we give our conclusions and an outlook to further research.

2 XML querying

Queries that can be formulated for searching in an XML document collection can be classified in the following four types:

1. **Value-oriented** (e.g. "Find documents about XML retrieval"): This type of queries only relates to the content of documents, as in classical information retrieval.
2. **Structural** (e.g. "Find books having more than 10 chapters and an appendix"): This type of queries only relates to the structure of documents. Presumably, it will be very rare in document-oriented applications of XML, it may be more convenient in data-centric applications.
3. **Combination of value and structure** (e.g. in an arts encyclopedia, pictures *of* London vs. images *painted in* London). Here XML markup allows for searches with a higher precision, by restricting conditions of the query to specific elements.
4. **Relevance-oriented:** This type of queries is similar to the first type, but here the system is asked to search for relevant document *parts*—unlike the classical IR approach treating documents as atomic units. Passage retrieval [Wilkinson 94] [Callan 94] is a means for retrieving portions of a document, but it does not consider document structure. In contrast, relevance-oriented retrieval searches for those elements of an XML document that answer the query best.

For purely structural queries, the DataGuide structure described in [Goldman & Widom 97] is an efficient method for locating all elements in an XML document collection fulfilling structural conditions. Since IR is mainly value-based, its application to XML documents leads to queries of types 3 and 4. In the remainder of this paper, we only consider these two types of queries.

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 the example from above, '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 all 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. 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.


```
<book>
  <title>...</title>
  <author name="..."/>
  <chapter>
    <section>
      ...
      <list>
        <item>...</item>
        <item>...</item>
      </list>
    </section>
  </chapter>
  <chapter>
    <section>...</section>
    <section>...</section>
  </chapter>
</book>
```
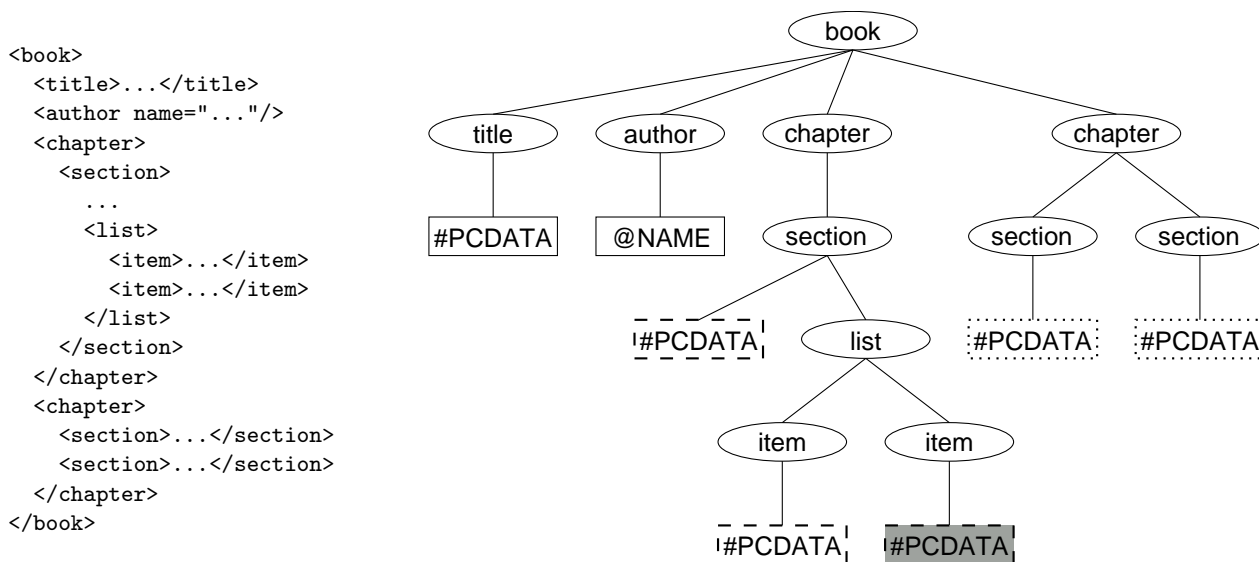
Figure 1: Example XML document structure: the textual representation on the left-hand side can be transformed into the tree notation on the right-hand side. Content can be found in those nodes depicted as rectangles in the tree notation (namely in attribute and `#PCDATA` / text nodes). These nodes are shown as ellipses in the textual notation.

As an example, consider the path leading to the grey #PCDATA node in Figure 1: `/book[1,1]/chapter[1,3] /section[1,1]/list[1,2]/item[2,2]/#PCDATA[1,1]`. 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 the first chapter in this document, but the third child of the `book` element).

3 Compressed paths in inverted lists

Inverted files have been used as the most efficient access method for text retrieval since many years. In order to process queries with structural conditions, path information has to be encoded as part of the entries in the inverted lists; we call this method the *PIL* approach (paths in 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 [Thom et al. 95]. Here we briefly describe the basic ideas and our extensions.

In principle, an inverted list describes occurrences of a term in a collection. Typically it consists of a sequence of document entries. In the most simple case, a document entry is just a document ID. In order to add structural information, we have to add all the different paths of occurrences of the specific term in the current document. For this purpose, we first store the number of occurrences, and then the occurrence entries themselves. Each of these entries specifies the length of the path first, followed by three lists containing the element names, the element indexes, and the sequence indexes, respectively. Using an XML-like notation for regular expressions, we can describe the abstract data structure 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_length element+ element_index+ sequence_index+ weight?
```

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[3]. As a simple example, consider an inverted list of a term occurring in documents number 34 and 40, which both share the same structure as displayed in Figure 1. Further assume that the term under consideration occurs three times in document 34, namely within each of the three #PCDATA nodes in dashed rectangles, and twice in document 40, namely within both of the two #PCDATA nodes in dotted rectangles. The corresponding inverted list would look like this (here angular brackets and spacing are used for illustration purposes only):

```
< 34 3
  < 4 <book chapter section #PCDATA> <1 1 1 1> <1 3 1 1> >
  < 6 <book chapter section list item #PCDATA> <1 1 1 1 1 1> <1 3 1 2 1 1> >
  < 6 <book chapter section list item #PCDATA> <1 1 1 1 2 1> <1 3 1 2 2 1> >
>
< 40 2
  < 4 <book chapter section #PCDATA> <1 2 1 1> <1 4 1 1> >
  < 4 <book chapter section #PCDATA> <1 2 2 1> <1 4 2 1> >
>
```

Obviously, due to the fact that the entries 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. For this purpose, another counter is added to the entry which gives the length of the common prefix (number of common path steps) with the previous entry, i. e.:

```
occurrence -> path_length_diff prefix_length element* element_index* sequence_index*
```

Instead of the path length, we only encode the difference between prefix length and path length. In order to compress the document numbers, run length encoding is used. For the sequence indexes (which were not considered in [Thom et al. 95]), we only encode the difference to the corresponding element indexes. Thus, we can compress the list of entries from above as follows:

```
< 34 3
  < 4 0 <book chapter section #PCDATA> <1 1 1 1> <0 2 0 0> >
  < 2 4 <list item #PCDATA> <1 1> <0 0> >
  < 2 4 <list item #PCDATA> <2 1> <0 0> >
>
< 6 2
  < 3 1 <chapter section #PCDATA> <2 1 1> <2 0 0> >
  < 2 2 <section #PCDATA> <2 1> <0 0> >
>
```

In order to compress this data, universal codes [Elias 75] are used for the numbers and canonical Huffman codes [Hirschberg & Lelewer 90] for the element names. Details are given in Section 5.2.

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 collection can be kept in main memory. Now the occurrence entry

---

[3]Also, some weighting methods may not need any additional space, since all the occurrence information they need is already present in the inverted list.

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, there are a number of design alternatives:

Top-down (preorder) vs. bottom-up (postorder): The latter organization would allow for restricting the decoding work to the specific path only, where we would work our way from the end of a path up to the root. 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 pointers or level numbers: The standard representation via pointers allows for faster access of specific nodes, but needs more space. Level numbers are more compact, especially when run length encoding is used. As an example of the latter strategy, the structure of document shown in Figure 1 can be described as follows (using preorder sequence):

```
<1 book> <1 title> <2 #PCDATA> <1 author> <2 @name> <1 chapter> <2 section>
<3 #PCDATA> <3 list> <4 item> <5 #PCDATA> <4 item> <5 #PCDATA> <1 chapter>
<2 section> <3 #PCDATA> <2 section> <3 #PCDATA>
```

Positions as bit addresses or element numbers: Bit addresses would allow for direct access, but could be used only when no context-dependent compression methods are applied. In contrast, element numbers are more compact, but require a linear scan through the representation

Element and sequence indexes explicit or 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.

In order to keep the XS trees in main memory, we have to minimize the storage requirements; thus, we should choose a representation that encodes the tree top-down, where the parent-child relationship is given via level numbers, positions are encoded as element numbers, and the element index is computed on the fly. So the tree is represented as a linear bitstream, without additional navigation pointers. As a consequence, the resolution method has to decompress this bitstream up to the position specified, in order to compute the corresponding path.

For compressing the tree structure, we use run length encoding, thus only the relative differences between the level numbers are stored. For our running example, this would give us:

```
<1 book> <1 title> <1 #PCDATA> <-1 author> <1 @name> <-1 chapter> <1 section>
<1 #PCDATA> <0 list> <1 item> <1 #PCDATA> <-1 item> <1 #PCDATA> <-4 chapter>
<1 section> <1 #PCDATA> <-1 section> <1 #PCDATA>
```

In order to use universal codes, we map these integers onto natural numbers. Since the level gaps are always less than or equal to one (descending in the XML tree structure is done one step at a time), we transform a gap z by means of the function $-(z-2)$:

```
<1 book> <1 title> <1 #PCDATA> <3 author> <1 @name> <3 chapter> <1 section>
<1 #PCDATA> <2 list> <1 item> <1 #PCDATA> <3 item> <1 #PCDATA> <6 chapter>
<1 section> <1 #PCDATA> <3 section> <1 #PCDATA>
```

It can be observed that the integers representing the level gaps are very small. Our experiments showed that unary codes give the highest compression compared to the γ and δ universal codes (other coding schemes for integers, like parameterized or local methods [Witten et al. 99, section 3.3] have not been considered within this work). For example γ codes would require about 15–17 % (depending on the collection) more space than the unary codes. On the other hand unary codes are quite simple and efficient also with regard to processing time: A number n is encoded as a sequence of $n-1$ 1-bits, followed by a single 0-bit. Encoding only the run lengths in our example, this would yield a bitstring of length 32.

With these few bits, we have encoded the shape of the document tree. In addition, we need information about element names. For encoding element names in compressed form, we have two design choices:

Huffman vs. arithmetic coding: The former method has the advantage that the codes of the element names can be inserted at the corresponding positions of the bitstring encoding the shape of the document tree. In contrast, for arithmetic codes, we would either have to decompress always the complete tree in order to access the arithmetic codes, or have to store an additional pointer to the start of this list; furthermore, the implementation of arithmetic codes is more complex (and thus decoding needs more time), especially when used with local coding schemes [Witten et al. 99, pages 51 ff]. For these reasons, we only implemented Huffman coding.

Global vs. local (context-dependent) coding scheme: With a context-dependent coding scheme, we can exploit context information for achieving higher compression rates. Given the DTD of the documents to be encoded, only a small number of elements may occur within a certain context; thus, a context-specific code must discriminate between these possible elements only. The only disadvantage is the higher complexity of the algorithms for encoding and decoding.

```
<!ELEMENT book      (author, title, chapter+)>
<!ELEMENT author    EMPTY>
<!ATTLIST author    name         CDATA #IMPLIED
                    affiliation CDATA #IMPLIED>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT chapter   (heading?, section+)>
<!ELEMENT heading   (#PCDATA)>
<!ELEMENT section   (#PCDATA | list)*>
<!ELEMENT list      (item+)>
<!ELEMENT item      (#PCDATA)>
```

Figure 2: DTD rules for the XML document example in Figure 1

As an example for illustrating the latter point, consider the DTD listed in figure 2. Here we have 12 different element / attribute names, which must be distinguished by a global coding scheme. In contrast, with a local coding scheme, we have at most three different elements in a context given by the production of an element. However, we can even do better: By considering the exact document syntax as specified in the DTD, we notice that no information has to be coded in many cases, namely when there is no choice at all and element / attribute names can be *predicted*. Based on the DTD, we can specify the following cases where the next element is determined uniquely by the context that has been read so far (here we use an abbreviated form of syntax rules, like in the definition of the data structures above):

- productions containing only one element—even if this one is optional or repetitive, like in the following examples:
    ```
    a1 -> b
    a2 -> c?
    a3 -> d*
    a4 -> e+
    ```
- productions that are pure sequences (without optional or repetitive elements), like e. g.
    ```
    a -> b c d
    ```
- combinations of the two previous cases, in the form where the last element of a sequence may be optional or repetitive:
    ```
    c1 -> d e f?
    c2 -> g h k+
    c3 -> l m n*
    ```
- As a further refinement, in a production starting with a pure sequence, only the remainder (starting with the first optional or repetitive element) has to be encoded, like e. g.
    ```
    r1 -> o p q? r?
    r2 -> s t u* v w
    ```
- For element attributes, we exploit the fact that here order does not matter. In the indexing process, we first group them into mandatory (`#REQUIRED`) and optional (`#IMPLIED`) attributes and then impose an implicit order on each of these groups (e. g. sort alphabetically by attribute name). Thus, we arrive at a content model like
    ```
    Attributes -> req1 req2 req3 imp1? imp2?
    ```
    Based on this content model, we can apply the strategies described above.

For our example DTD shown in Figure 2, there are only three contexts where we have to encode the element or attribute names, namely the attributes of `author` and the children elements of `chapter` and `section`. Continuing the above example we would represent the document structure as follows:

```
<1> <1> <1> <-1> <1 @name> <-1> <1 section>
<1 #PCDATA> <0 list> <1> <1> <-1> <1> <-4>
<1 section> <1 #PCDATA> <-1 section> <1 #PCDATA>
```

Thus we had to encode just seven from 18 element names for our example document. Applying context specific Huffman codes here would yield code word lengths of one bit per element / attribute name for our example DTD, which means that the structure of the document depicted in Figure 1 can be represented by 39 bits only.

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

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

5 Evaluation

In order to evaluate the approaches described in the previous sections, an analytical investigation would be most helpful. However, the complexity of the subject under consideration hardly allows for the application of this method. In contrast to methods for atomic documents (e. g. [Moffat & Zobel 96]) where a document can be viewed as a sequence of terms, we have to deal with nested structures of arbitrary depth. In combination with context-dependent compression methods, it is hardly possible to develop an appropriate analytical model that would allow for the estimation of the space and time requirements for the different approaches.

Thus, we have to restrict to experimental evaluation of the approaches presented before. The various experiments are described below. For characterizing the efficiency of the different approaches, we are considering 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. In addition, the size of the XS tree is very crucial, since it has to be kept in main memory.

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.

All experiments described below have been conducted with HyREX[4], the *Hy*per-text *R*etrieval *E*ngine for *X*ML, on a Linux PC with a 1.4 GHz AMD processor and one gigabyte of main memory.

5.1 Test collections

	Reuters	IEEE-CS
size in MB	2 369	403
compressed size in MB	369	44
# documents	806 791	5 063
∅ bytes / document	3 079	83 557
∅ nodes / document	96	1947
# elements in DTD	44	172
# attributes in DTD	44	35
# distinct terms	310 919	96 358
∅ collection frequency	215.65	34.72
∅ node frequency	337.92	118.25
# postings in inverted lists	105 064 625	11 394 494
∅ path length	3.91	5.97

Table 1: Statistics of the two test collections used

For our experiments, we used two fairly large XML collections with quite different characteristics. The statistics of these collections are shown in Table 1.

The first collection is the Reuters Corpus[5] which contains about 810 000 Reuters, English language news stories. 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[6] (IEEE-CS). In terms of the number of documents (5 063 documents), this collection is quite small compared to the Reuters Corpus. But the IEEE-CS documents are typical journal

---

[4]`http://ls6-www.cs.uni-dortmund.de/ir/projects/hyrex/`
[5]`http://about.reuters.com/researchandstandards/corpus/`
[6]This collection can be ordered as CD set from `http://www.computer.org/cspress/catalog/cs-96.htm`.

papers, so their average size is much higher than that of the Reuters documents. The parameters describing the DTD, the average path length, and the average number of XML nodes per document show that the structure of the IEEE-CS documents is rather complex and heterogeneous, compared to the structure of the Reuters documents.

For compression of the two document collections, we used the `bzip2`[7] routine here, since [Liefke & Suciu 00] showed that this simple method achieves compression rates that can hardly be surpassed by XML-specific compression methods. For indexing these collections, we eliminated stopwords and performed stemming on the remaining terms. Furthermore, we consider only one term occurrence per leaf element of the XML documents.

5.2 Experiments

In order to measure the efficiency in terms of compression rate and retrieval time of the different compression methods described in Sections 3 and 4 we conducted various experiments for both approaches. Table 2 gives an overview on the different experiments.

PIL_{noc}	no compression of the paths in inverted lists
PIL_{uc}	universal codes, global Huffman code
PIL_{hc}	context Huffman codes, element prediction
PIL_{prefix}	common path prefixes
XS_{noc}	no compression of the XS trees
XS_{uc}	universal codes, global Huffman code
XS_{hc}	context Huffman codes, simple element prediction
$XS_{predict}$	enhanced element prediction
$Scan$	scan documents with given path handle
NS	no indexing of structure at all

Table 2: Overview on variants of the *paths in inverted lists (PIL)* and the *XS tree* approaches

For the *paths in inverted list (PIL)* approach, our baseline is the case where paths are not compressed at all (PIL_{noc}). Here the various path parameters are represented by fixed length integers: path lengths are represented by one byte each, element names and indexes are represented by two bytes each. Within the PIL_{uc} experiment we applied universal codes in order to compress path lengths and indexes and a global Huffman code for compression of the element names. In addition, for a given sequence index of a path, we only encoded the difference to its respective element index. Regarding the distributions for path lengths, element indexes and sequence index gaps, we found that unary coding gives best compression rates for the path lengths, while γ encoding gives best results for encoding the indexes. The PIL_{hc} extends the PIL_{uc} experiment in that the compression of the element names is refined by a local coding scheme: Instead of using only one global Huffman code for all element names, we computed one Huffman code for each content model within the respective DTD. In addition, a simple form of element name prediction has been applied. In case a content model allows a single child only, we do not encode its name in the inverted list but predict it at retrieval time from the DTD. Finally, the PIL_{prefix} experiment extends PIL_{hc} in that also common path prefixes are regarded.

With the XS tree approach path handles have to be stored within the inverted lists, which point into the external XS tree data structure. For these handles, we always used the same compression technique: The run lengths of the path handles for a given document are encoded with the δ universal code. Depending on the collection, this might not be the optimum solution, but with regard to our test collections δ codes gave the best compression. As for the PIL approach, we calculated a baseline, where compression of the XS trees has been omitted (XS_{noc}). Within the XS_{uc} experiment we applied a global Huffman code for compressing the element names and unary coding for the level number gaps (as described in Section 4). The XS_{hc} experiment improves the compression of the element names in that the global Huffman code has been replaced by context specific Huffman codes for each content model of the DTD. In addition, the same element name prediction mode as in the PIL_{hc} experiment has been used. The $XS_{predict}$ experiment is a further refinement; full element name prediction as described in Section 4 has been applied here.

As an alternative to XS trees, we also considered scanning of the original documents (again, the inverted lists contain path handles only) in order to get the required structural information.

Where applicable, we compare our methods for indexing structure with an index where indexing of structure was omitted (NS).

---

[7] `http://sources.redhat.com/bzip2/`

5.3 Indexing

	Reuters		IEEE-CS	
	time	space	time	space
PIL_{noc}	10:11	2573.14	1:11	406.08
PIL_{uc}	10:44	474.24	1:12	81.90
PIL_{hc}	10:54	338.75	1:17	56.82
PIL_{prefix}	11:26	319.71	1:21	45.06
XS_{noc}	7:50	414.35	0:50	42.26
XS_{uc}	7:55	260.94	0:50	34.36
XS_{hc}	8:18	225.85	0:52	25.10
$XS_{predict}$	8:25	220.04	0:53	24.74
NS	7:05	81.34	0:45	3.53
$Scan$	7:34	189.68	0:50	20.04

Table 3: Indexing time and index sizes. Times are given in hh:mm format, sizes are in megabytes.

Table 3 gives an overview on the index sizes resulting from the various experiments and the time needed to create the indexes. The index sizes are including administrative overhead, such as a table for document identifiers and the dictionary. Considering the times, one sees that adding a compression technique requires additional indexing time. However the difference for the variants of the XS tree is not as big as for the various PIL approaches. In general, indexing with the XS tree approach needs less time than indexing with the PIL approach.

Reuters	PIL_{noc}	PIL_{uc}	PIL_{hc}	PIL_{prefix}	XS	NS
$\sum$ size of ILs (MB)	2566.77	467.87	332.38	313.34	183.31	74.97
$\varnothing$ bytes / IL	8656.45	1577.89	1120.94	1056.74	618.21	252.85
$\varnothing$ bits / posting	204.93	37.36	26.54	25.01	14.64	9.38

IEEE-CS	PIL_{noc}	PIL_{uc}	PIL_{hc}	PIL_{prefix}	XS	NS
$\sum$ size of ILs (MB)	405.06	80.88	55.80	44.04	19.02	2.51
$\varnothing$ bytes / IL	4416.17	880.14	607.31	479.25	206.99	27.33
$\varnothing$ bits / posting	298.77	59.54	41.09	32.42	14.00	6.29

Table 4: Inverted lists (IL) statistics.

Table 4 displays statistics regarding the lengths of the inverted lists in the different experiments. Since the XS tree has no variations w. r. t. to the encoding of the inverted lists in the different experiments, we have one column for all XS tree experiments here. The XS inverted lists clearly beat the best PIL variant w. r. t. disk space needed.

Reuters	XS_{noc}	XS_{uc}	XS_{hc}	$XS_{predict}$
$\sum$ size of XS trees (MB)	224.67	71.26	36.17	30.36
$\varnothing$ bytes / XS tree	292	92	47	39

IEEE-CS	XS_{noc}	XS_{uc}	XS_{hc}	$XS_{predict}$
$\sum$ size of XS trees (MB)	28.22	14.32	5.06	4.70
$\varnothing$ bytes / XS tree	5845	2965	1049	969.47

Table 5: XS tree statistics

The sizes of the XS tree representations are shown in Table 5 (including a four byte pointer per XS tree, which is needed for managing them in main memory). The most elaborated variant ($XS_{predict}$) requires little more than 1 % of the space occupied by the original data in order to represent the structure of the documents. With scanning, the XS trees can be omitted completely. However, this is feasible only if documents are stored in uncompressed[8] form, since decompression takes even longer than scanning (see below). Thus, the total space requirements of scanning are similar to that of PIL_{noc}.

---

[8]For all other methods, documents have to be decompressed only when the user wants to view a document.

5.4 Retrieval

For the access structures considered here, retrieval time is the sum of the times needed for the three subsequent phases in retrieval:

1. Reading the inverted list from disk. Here the disk seek time and rotational latency (about 10 ms for fast hard disks) plays the crucial role, therefore this adds an almost constant factor to the retrieval time for the different approaches presented below[9].
2. Decompressing the inverted list entries.
3. For the XS tree, path handles need to be converted into paths; in order to do this the respective XS trees are to be decoded.

	Reuters		IEEE-CS	
	$\varnothing$ ms / IL	$\varnothing$ ms / path	$\varnothing$ ms / IL	$\varnothing$ ms / path
PIL_{noc}	10.03	0.030	3.41	0.029
PIL_{uc}	11.82	0.035	4.10	0.035
PIL_{hc}	12.29	0.036	5.53	0.047
PIL_{prefix}	16.42	0.049	6.52	0.065
$XS\,/\,Scan$	3.19	0.009	0.63	0.005
NS	1.60	0.005	0.27	0.008

Table 6: Time for decompressing an average-sized inverted list (IL, in milliseconds).

In terms of retrieval efficiency we first measured the time for decompressing an average-sized inverted list. The times depicted in Table 6 reflect the times for arriving at a list of paths corresponding to the postings in the inverted list. The XS row is an exception; here additional effort is needed to resolve the path handles (resulting from the inverted lists) into paths (see below). The *path* column shows the average time spent to decode one path (or posting, respectively). As can be seen from Table 6, adding compression methods (for the PIL variants) increases the decompression time by roughly 6–10 %. Retrieval times given here were measured on inverted lists without any weighting information. Additional experiments have shown that term weighting information just adds a constant factor (≈ 0.120 ms for the IEEE-CS collection) per inverted list for any of the PIL and XS variants.

	XS_{noc}	XS_{uc}	XS_{hc}	$XS_{predict}$	Scan	
Reuters	0.440	0.345	0.460	0.500	2.075	(+ 3)
IEEE-CS	9.120	7.222	9.491	10.231	30.509	(+ 80)

Table 7: Times for resolving a path handle (in milliseconds). For the *Scan* method approximate additional time for decompressing a document is given in parentheses.

The time for resolving path handles for the XS tree variants is depicted in Table 7. There is a noticeable large gap in time for resolving path handles between the XS_{uc} and XS_{hc} variants. This results from the fact that XS_{hc} uses context-specific Huffman codes, thus the respective XML trees must be constructed at decompression time in order to apply the correct Huffman codes for decompression.

The last row displays the time needed to resolve path handles by scanning the document; here I/O times (10–15 ms per document) are not considered yet — for Reuters, this would be the dominant factor. Considering also the decompression times (given in parentheses), it becomes apparent that in any case, scanning is slower than resolving path handles with XS trees.

Table 8 cumulates the times of the three retrieval phases denoted above for an average-sized inverted list.

Apparently, PIL retrieval is much faster than XS-based approaches—especially for the IEEE-CS collection; thus, we have the standard space-time tradeoff. However, the difference reduces significantly when we have several term matches per document. In this case, XS retrieval times increase only marginally, whereas the PIL times grow linearly with the number of matches. For XS, the dominant factor is the decoding of the tree structure. Given a path handle, decoding stops, when the corresponding position in the document is reached. For a single handle, we have to decode half of the document on average. For resolving k handles, the fraction to be decoded is $k/(k+1)$ [Cooper 68]. Measuring the actual decoding times for $k = 1\ldots 5$ matches, we found out that there is an additional constant factor t_{XTI} that is independent of the fraction to be decoded. Let t_{XTD}

---

[9]With a disk transfer rate of 20 MB/s, even a large inverted list of 100 KB could be read in 5 ms.

	Reuters		**IEEE-CS**	
	$\varnothing$ ms / IL	$\varnothing$ ms / path	$\varnothing$ ms / IL	$\varnothing$ ms / path
PIL_{noc}	20.03	0.059	13.41	0.386
PIL_{uc}	21.82	0.065	14.10	0.406
PIL_{hc}	22.29	0.066	15.53	0.447
PIL_{prefix}	26.42	0.078	16.52	0.476
XS_{noc}	108.15	0.320	410.95	11.836
XS_{uc}	87.66	0.260	340.12	9.796
XS_{hc}	112.46	0.333	510.60	14.706
$XS_{predict}$	121.09	0.359	539.42	15.536
$Scan$	460.73	1.365	1 290.06	37.156

Table 8: Cumulated retrieval times for one term queries (in milliseconds).

denote the remaining time for decoding a complete document. In addition, we have the time t_{XIL} for decoding an entry of the inverted list (see Table 6). Thus, the complete time for decoding a document with k matches is

$$t_{XSR} = k \cdot t_{XIL} + t_{XTI} + \frac{k}{k+1} t_{XTD}$$

In contrast, the PIL decoding times are proportional to the number of matches. As an example, for the XS_{uc} variant for Reuters, we have $t_{XTI} \approx 0.05\,\mathrm{ms}$ and $t_{XTD} \approx 0.60$; thus, resolving a document with 5 matches requires $0.56\,\mathrm{ms}$, which is about four times slower than the (fastest) PIL_{noc} variant ($0.15\,\mathrm{ms}$); in contrast, for single term queries, the ratio is about 12:1 ($0.35\,\mathrm{ms}$ vs. $0.03\,\mathrm{ms}$).

5.5 Discussion of results

The results described above show that we have reached our goal of developing a compact representation of the structure of XML documents; depending on the compression parameters, the XS tree requires about 1–3 % of the space of the original data. The overall size of the (compressed) PIL indexes is 2–3 times higher than for the combination of XS tree and corresponding inverted lists. Thus, when index space matters (like in PDAs or E-book devices), the XS tree offers significant advantages. Comparing the index size with that of compressed XML (see Table 1), we see that only the highly compressed indexes require less space than the document collection itself.
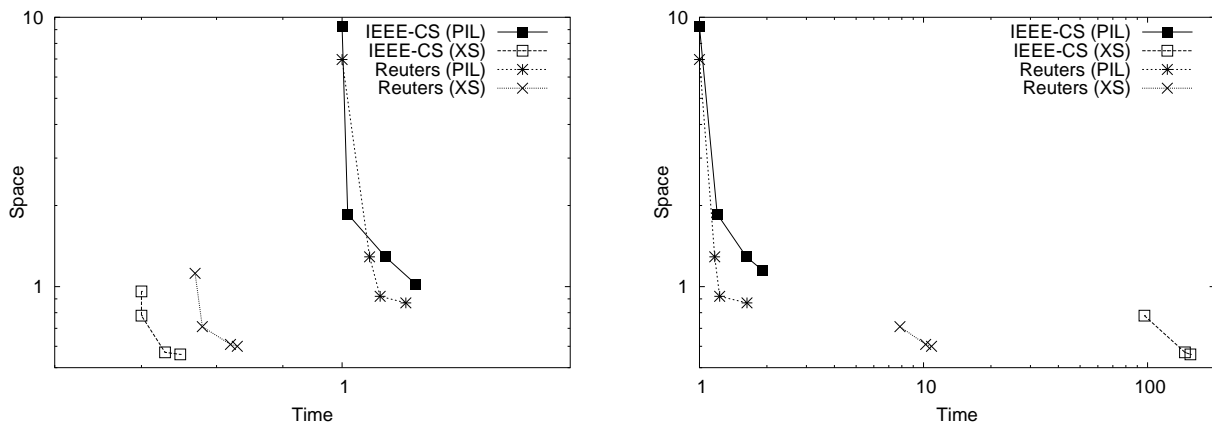

Figure 3: Space vs. time for indexing (left-hand side) and retrieval (right-hand side)

In Figures 3, 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_{noc} variant.) 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 (the outlier XS_{noc} is omitted in the graphics). 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 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_{noc}, decoding processes about 1 MB/s (and decompression needs more time per path), whereas I/O rates are about 20 MB/s. Whereas in [Moffat & Zobel 96], decompression and I/O time were in the same order of magnitude, the complex structure of XML documents leads to this imbalance[10]. Thus, also additional mechanisms (like the skip lists in [Moffat & Zobel 96]) 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

With the increasing availability of XML documents, there is a growing need for IR systems supporting appropriate retrieval methods. So far, there has been little literature on index structures for XML documents, and no experimentation with realistic XML collections. In this paper, we have investigated two different XML index structures, and we have performed experiments using two large collections with quite different characteristics.

With the XS tree, we have developed a new access structure for XML retrieval that yields a very compact index. For specific applications where size matters (like in PDAs and E-book devices), the XS tree is the optimum solution.

The experimental investigation of the XS tree along with the PIL approach published before has shown that the complex structure of XML documents leads to high decompression times, which are higher than the I/O time for the uncompressed inverted lists. Since the XS tree encodes the complete structure of an XML document, its decompression time is significantly higher than that of the PIL inverted lists.

For all but one (XS_{uc}) of the variants investigated, compression of indexes increases retrieval time. Thus, in almost all cases retrieval efficiency can be optimized by using no compression at all. However, this solution leads to a storage overhead, which is huge in comparison to compressed XML documents.

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.

References

Callan, J. P. (1994). Passage-Level Evidence in Document Retrieval. In [Croft & Rijsbergen 94], pages 302–310.

Chamberlin, D.; Florescu, D.; Robie, J.; Siméon, J.; Stefanescu, M. (2001). *XQuery: A Query Language for XML.* Technical report, W3C. http://www.w3.org/TR/xquery/.

Cooper, W. (1968). Expected Search Length: A Single Measure of Retrieval Effectiveness Based on Weak Ordering Action of Retrieval Systems. *Journal of the American Society for Information Science 19*, pages 30–41.

---

[10]These findings are only marginally affected by the fact that most of our system—except the compute-intensive parts—is implemented in Perl; we estimate that implementing it completely in C would reduce retrieval times by not more than 50 %.

Croft, B. W.; van Rijsbergen, C. J. (eds.) (1994). *Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, London, et al. Springer-Verlag.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory 21(2)*, pages 194–202.

Fuhr, N.; Großjohann, K. (2001). XIRQL: A Query Language for Information Retrieval in XML Documents. In: Croft, W.; Harper, D.; Kraft, D.; Zobel, J. (eds.): *Proceedings of the 24th Annual International Conference on Research and development in Information Retrieval*, pages 172–180. ACM, New York.

Goldman, R.; Widom, J. (1997). DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: *Proceedings of the 23rd International Conference on VLDB*, pages 436–445. `http://www-db.stanford.edu/lore/pubs/dataguide_vldb97.pdf`.

Harman, D.; Fox, E.; Baeza-Yates, R.; Lee, W. (1992). Inverted Files. In: *Information Retrieval. Data Structures & Algorithms*, pages 28–43. Prentice Hall, Englewood Cliffs.

Hirschberg, D. S.; Lelewer, D. A. (1990). Efficient decoding of prefix codes. *Communications of the ACM 33(4)*, pages 449–459.

Liefke, H.; Suciu, D. (2000). XMill: an efficient compressor for XML data. In: Chen, W.; Naughton, J.; Bernstein, P. A. (eds.): *Proceedings of the 2000 ACM SIGMOD. International Conference on Management of Data*, pages 153–164. ACM Special Interest Group on Management of Data, ACM, New York. `http://www.acm.org/sigmod/sigmod00/eproceedings/`.

Moffat, A.; Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems 14(4)*, pages 349–379.

Thom, J. A.; Zobel, J.; Grima, B. (1995). *Design of indexes for structured document databases*. Technical Report TR-95-8, Collaborative Information Technology Research Institute, Melbourne, Australia.

Wilkinson, R. (1994). Effective Retrieval of Structured Documents. In [Croft & Rijsbergen 94], pages 311–317.

Witten, I. H.; Moffat, A.; Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 2nd edition.

Yoon, J. P.; Raghavan, V.; Chakilam, V. (2001). Bitmap Indexing-based Clustering and Retrieval of XML Documents. In: Dominich, S. (ed.): *ACM SIGIR '01 Workshop on Mathematical / Formal Methods in IR; Proceedings*, pages 60–68. `http://www.ucs.louisiana.edu/~vmc0583/sigIR_clustering_submitted.pdf`.