Philipp Dopichaj

Space-Efficient Indexing of XML Documents for Content-Only Retrieval

More and more documents are stored in semistructured formats like XML. In contrast to traditional information retrieval, the documents can become quite large, and it is often desirable to retrieve not complete documents, but isolated elements that satisfy an information need. To make this possible, the index structures from traditional information retrieval must be adapted to semistructured documents (specifically XML) so that term occurrences can be pinpointed to specific elements inside the documents. This paper explores several enhancements of the index structures and evaluates the advantages and drawbacks of the different versions with respect to index size and retrieval time.

1 Information retrieval in digital libraries

XML has become a standard file format for storing semistructured information; in particular, digital libraries make use of this format for storing their data. As with the web, a large collection of books requires good retrieval capabilities: The best information is useless if it cannot be found. In the digital libraries, there is not only a large number of books, but the books themselves are several hundred pages long. When searching such a collection, users should expect to get focused results – the hint that the sought information is somewhere in a 500-page book is of limited value. Thus, the search engine must be able to search and index *parts* of documents like chapters or sections. Due to its hierarchical nature, XML is a suitable format for this.

1.1 XML Retrieval

XML documents are not atomic entities, but consist of nested elements that provide both the macro and the micro structure of the documents. For example, elements are used to demarcate the sections in an article and to mark up phrases as "to be printed in italics". The logical structure of the documents is of particular interest for element retrieval: If a single section satisfies the user's information need, it should be retrieved in favor of the complete article.

The most simplistic approach would be to index all elements as "document"' with a traditional information retrieval search engine; this alone is not, however, a viable approach for two reasons: First, the index is inflated dramatically, because the number of elements is much larger than the number of documents. Second, the retrieval results should not simply be displayed in a flat list of (more or less) relevant elements: On the one hand, the user might not want to look at repeated information – if, for example, the result list contains a chapter and a section from that chapter –, or it might make sense to group the results by document.

1.2 Problem statement

We aim at providing good support for simple keyword queries like they are used for web search engines.

The authors of current full-text index structures for XML pay little attention to storage overhead, their main focus is on retrieval quality. As a result, the indexes that are used for retrieval can be several times the size of the original document collection, which is acceptable for research prototypes, but may be impractical for real-world uses, like digital libraries.

In digital libraries, the document collection is mostly static: Modifications of existing documents are rare (once a book is published, it may be replaced by a revised edition, but the original edition is not modified), but new documents are added frequently.

Two main forms of digital libraries are common: online [Dopichaj 2006] and libraries that are sold on read-only media like CDs and DVDs. In both cases, the users of these libraries will want to search them and obtain clearly focused results, but do not need (and cannot) update the contents of the collections. Although excessive usage of disk space may not be a problem in the first scenario, it definitely is important in the second one: The end user does not want to use twice the storage space for the library, and the computer may not be state of the art, so efficient processing is a must.

For obvious reasons, it is infeasible to do a full-text scan of all documents whenever a query is posed to the retrieval system; to get acceptable performance, we need index structures that facilitate quick access to all relevant documents.

1.3 Contribution

In this paper, we describe and analyze various optimizations of XML full-text indexes based on previous work. In particular, we describe difference-based storage of inverted lists for the XML documents and suitable storage of metadata about the elements; the main idea is to omit data that can easily be derived from other data that is stored in the index.

- We support retrieval of *all* elements in the collection, even very short ones that are frequently omitted from indexes. Our aim is to reduce the size of the index significantly without negatively affecting retrieval time or retrieval quality.
- Our main focus in this paper is not retrieval quality, but index size and retrieval speed, so our evaluation is strongly biased towards the latter two. Our index structures are flexible enough to support a variety of retrieval models and similarity measures; our implementation currently uses a variant of Okapi BM25 [Spärck Jones, Walker, and Robertson 1998], but it could be adapted to other approaches like language modeling easily.
- We provide a detailed analysis of the implications of the space-saving index structures and the various trade-offs between space and time.

Although it is an interesting problem in its own right, we do not address the combination of full-text with structural retrieval (so-called *content-and-structure* retrieval). In this form of retrieval, the user also specifies structural constraints

like "the following keywords should appear in the bibliography". The indexes hold all information that is needed to evaluate such queries, but direct access to elements with specific structural properties is not possible, so a straightforward extension would probably lead to bad performance.

2 Index Structures

Our index structures shall support simple keyword queries, that is, queries that comprise a set of words. The aim of the retrieval engine is to retrieve all elements in the collection that contain at least one of the query words and then rank the elements based on a similarity function.

For determining the candidate elements and determining the similarity to the query, we need the following information for effective and efficient retrieval (this is mostly based on standard work in information retrieval [Witten, Moffat, and Bell 1999]):

- Inverted lists, containing references to the elements along with the corresponding term frequencies.
- Metadata, for example information about the positions of the elements in their documents, typically represented by XPaths.
- The lexicon, containing information about the terms, including document frequencies and the pointers to the inverted list.

Furthermore, the similarity function may need additional information; the Okapi BM25 function we use also needs the length of an element to calculate its score; this, too, will have to be stored in the metadata.

Using these index structures, the search engine can efficiently answer a query. We assume that the query q is composed of a set of query terms $t_1, \ldots, t_n$. For each query term t_i, the search engine must perform the following steps:

- Retrieve the lexicon entry for t_i. This provides us with the term's weight and the entry point in the inverted list file.
- Retrieve the inverted list, starting at the entry point from the lexicon.

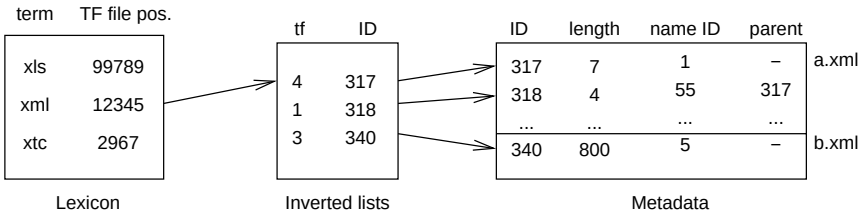


Figure 1: Index structures. For each query term, the lexicon is consulted to find the corresponding entries in the inverted list file, which is then read to determine the candidate elements and their term frequencies. Using the metadata, the elements are grouped by document, and the final similarities are calculated. The name ID in the metadata represents the element name, which is stored in another table.

- Determine the tree structure of the document from the inverted list entries and the metadata.
- Update the term frequencies based on the tree structure.

After this, we can use the standard process for calculating the similarity of each element and apply XML-specific post-processing operations to improve the ranking; this is outside the scope of this paper. Figure 1 illustrates the index structures we need to support the retrieval process.

The remainder of this section will describe these aspects of XML indexing.

2.1 Tokenization

Before we look at the index structures themselves, we discuss tokenization, because it has important implications for the index structures. In order to index a text, the indexer must first break it into *tokens*, typically based on words. These words are the entry points to the inverted lists.

For tokenization, we consider every sequence of Unicode letters and digits as a term. Additionally, all tags in the input are considered token boundaries; for example, the XML fragment <fn>John</fn><ln>Doe</ln> is parsed as the two tokens John and Doe, *not* as the single token JohnDoe.

As we will see later, this is important for difference-based storage, because it ensures that all ancestors of an element e include at least as many instances of the terms contained in e as e itself. Formally, for all elements e with the respective parent $p(e)$ and all terms t, with $tf(t, e))$ the number of occurrences of t in e, the following equation holds:

$$tf(t, p(e)) \geq tf(t, e)$$

Because of mixed content (that is, sub-elements embedded within text), the term frequencies of the parent may be greater than the sum of the term frequencies of all its children:

$$tf(t, e) \geq \left(\sum_{f \in c(e))} tf(t, f) \right)$$

Although this assumption does not always model the intention of the document author – for example, it fails if inline markup is used inside a word, like highlight –, it works in the majority of cases and is frequently made in XML retrieval research.

Furthermore, we remove stop words and apply the Porter stemming algorithm as implemented by the Snowball project[1].

2.2 Lexicon

The lexicon provides the entry point from the query terms to the inverted list. For each term that occurs in the document collection, it stores the document frequency of this term and a pointer to the position in the inverted list file where this term's inverted list starts.

While indexing, there is little choice but to store the lexicon in main memory: Documents typically contain thousands of words, and for each term, the indexer must determine whether the term is already in the lexicon (in this case, it uses the old ID for the inverted list). Otherwise, it assigns the next ID to the term and add it to the lexicon. Even with an ef-

1. http://snowball.tartarus.org/

ficient index structure such as a B-tree, this will require two to three seeks, which would increase indexing time.

While searching, the time to access the lexicon is not critical, because most queries only refer to a less than ten words. Thus, it is possible to perform the search on a machine with less main memory than the machine used for indexing.

2.3 Inverted Lists

A large part of the index is stored in the inverted lists; compared to traditional information retrieval, the size of a document in the index is much larger: It also needs to record information about *where* (that is, in which element) inside a document a given term occurs. A straightforward approach is to simply index the textual content of each element separately, but the cost is prohibitive (because of nesting, the size is several times the size of the document-level index).

Many search engines for XML retrieval do not index small elements up to a given length in tokens, but this does not alleviate the problem much; the size is still about six times as large as the document-level index. One should note that the authors of these search engines typically do not give index size as an argument in favor of omitting these elements. Instead, they argue that these elements are never good retrieval results that the users want. Although this may be the case for typical element retrieval scenarios, this alone is not a sufficient reason, as it is always possible to filter these elements from the results.

We should consider that some retrieval approaches use the small elements to improve the quality of the retrieval results, even if they are not eturned to the user [Ramírez, Westerveld, and de Vries 2006, Dopichaj 2007]. In addition, if we broaden our scope just a little to include content-and-structure queries, the small elements may be vital to achieve acceptable retrieval results: Content-and-structure queries often reference metadata elements like authors' names or titles. These metadata elements are by their nature short, and if they are not included in the index, it becomes impossible to answer such queries.

[1]<section>
[2]<title>Inverted lists</title>
[3] Inverted lists are an
[4]index structure.
</section>

Indexed document; the superscripted numbers at the start tags are the element IDs.

Term	Element ID	tf	stf
index	1	1	0
index	3	1	0
index	4	1	1
inverted	1	2	0
inverted	2	1	1
inverted	3	1	1
...			

Inverted lists for this document. Observe that the section element's (ID 1) entries can all be omitted.

Figure 2: Example of indexing.

Thus, there are good reasons for including *all* elements in the index, if we can make sure that this does not increase its size too much. The *difference-based* method we describe in this section is still significantly larger than a document-level index, but this is unavoidable; at least it tries to minimize redundant storage of information.

The basic idea is to omit data that can be derived by using the tree-structure of the XML documents. Thus, for an element e, we only store the term frequencies that result from the text nodes directly below e, *excluding* text contained in child elements. Thus, for elements with child elements, the stored term frequency $stf(t, e)$ of term t in element e is:

$$stf(t, e) = tf(t, e) - \sum_{f \in c(e)} tf(t, f)$$

For elements without children, the stored term frequency is the real term frequency: $stf(t, e) = tf(t, e)$. Figure 2 illustrates this.

It may happen that a term does not occur in text node children of e, but only in element children; in this case, the stored term frequency is zero, and no inverted list entry for this element is stored in the index. In the extreme case, if the element does not have text nodes as children, it will not occur in the inverted lists at all. This happens for higher-level structural elements, like chapters.

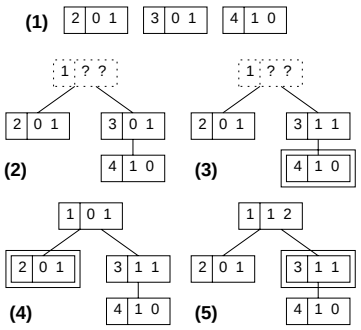


Figure 3: Reconstruction of term frequencies. Each box contains the element ID and the frequencies for "index" and "inverted". First, the stfs are retrieved as a flat list (1), next, the document structure is reconstructed from the metadata, including missing nodes (2). Finally, the tree is traversed bottom-up; for each node, the tfs are added to the parent's tfs (3,4,5).

During retrieval, a post-processing step is needed to reconstruct the original term frequencies from the stored term frequencies, as Figure 3 shows

Using this storage technique leads to significant savings in the number of entries in the inverted lists and, consequently, in the total size of the inverted lists. Although a post-processing step is required after retrieving the term frequencies from the inverted lists, the cost of this step is offset by the reduced time needed to read the entries from disk.

2.4 Building the inverted lists

The document collections to be searched can get large, too large to be indexed in main memory alone. Thus, it is necessary to keep large parts of the data on disk while indexing.

2.5 Metadata

The lexicon and the inverted list only contain information about term occurrences, but not about the structure of the XML document. This is a deliberate design decision; many search engines use index structures that store the structural information in the inverted lists, but this unnormalized form of storage increases the size of the index.

Instead, we normalize the schema and only store IDs for the elements in the in-

verted list. This alone is obviously not sufficient: At least, the search engine needs to get the name of the document and the position of the fragment inside the document – typically an XPath – to display the results.

Even before that point, it may be necessary to obtain more detailed information. For example, if the search engine must avoid presenting nested elements, like a chapter and a section from that chapter, to the user in its result list, it needs data about results' parent–child relations, and exploiting small elements, as mentioned earlier, also requires this information. Many similarity measures, for example Okapi BM25, use the length of an element's text to adjust its similarity. And, of course, if we store only the differences of the term frequencies for inner elements, we must know the structure of the documents.

Thus, we need to store element-specific information in the index in a space- and time-efficient format. The simplest approach is to simply store the information per fragment ID:

- The structural information can be reduced to the ID of an element's parent (encoded as the difference to the parent's ID to save space).
- The element's length can be stored as-is.
- If we need the XPath for identification purposes, we must also store the ID of the tag name.

The IDs of the elements themselves need not be stored, the metadata entries are simply stored in order, so that the ID can be derived from the position in the list. This is possible because we must read the complete metadata for a document anyway to obtain the structural information.

A minor optimization is to apply the same technique we used for the inverted lists and only store the difference to the child elements' lengths for a parent element. In this case, we still have to keep elements whose text is completely contained in children in the metadata index, because the link to the parent's parent and the tag ID must still be available.

2.6 Index Compression

On the level of actually storing integral values in files, we can save space if we know some characteristics of the numbers to be stored: If most of the numbers are small, it is advisable to use a coding method that stores small values in fewer bits at the expense of using more bits than necessary for larger values that occur infrequently.

We use *unary coding*, which stores $n-1$ of one-bits followed by one zero-bit to store a number $n \geq 1$, Υ *coding*, which breaks the number into an exponent part and a remainder: .

$$n = 2^{\lfloor \log n \rfloor} + (n - 2^{\lfloor \log n \rfloor})$$

The exponent is the stored as the unary code for $\lfloor \log n \rfloor + 1$, where $\lfloor x \rfloor$ is the floor function, followed by the value of $n - 2^{\lfloor \log n \rfloor}$ in binary, in as many bits as needed. The δ *coding* uses the same breakdown of the number, but stores the exponent in Υ coding instead of unary. See Bentley and Yao [1976] and Elias [1975] for details.

It is customary to store the *difference* of the document IDs in the inverted lists instead of the IDs themselves; for example, if a term occurs in documents 2, 6, 7, and 24, the following values would be stored in the index: 2, 4, 1, 17. Assuming that terms occur in clusters, we can expect that the numbers to be stored are small in magnitude. This assumption has proved to be accurate for traditional information retrieval, and it is even more relevant for XML retrieval.

Term occurrences are more localized in XML retrieval if we use a consecutive numbering of elements, because all elements from the same document – whose IDs are close to each other – are typically about the same topic, so they contain similar terms.

Furthermore, the term frequencies of elements that are deeper in the document tree tend to be extremely small in magnitude because the elements are quite short.

Depending on the expected distributions of values, different encoding functions can be used [Witten, Moffat, and Bell 1999]. In our scenario, we expect many term frequencies to be close to 1, so a unary encoding is most suitable. The differences of the document IDs are somewhat bigger, so the search engine uses a δ coding.

Much more data is read from the metadata index than from the inverted lists, so the decoding overhead of the bit-based unary, δ, and Υ codings is more noticeable. These encodings are expensive because they require expensive shift and mask operations. Thus, we also evaluate a byte-based encoding as an alternative: A sequence of bytes with the high bit set followed by a byte with the high bit clear forms the value; the lower seven bits of each byte are concatenated to obtain the value.

Adaptive coding methods like arithmetic coding cannot be used for our purposes: When decompressing, it assumes that *all* data is read sequentially; in our application, however, we must be able to read isolated runs from the inverted files without also reading all the preceding runs. Thus, it would only be possible to use arithmetic coding within a single run, but – because the runs are relatively short – it is unlikely to achieve a good compression ratio then.

Huffman coding [Huffman 1952] is more realistic for our scenario, because it is based on a static model of the frequencies, so the same decoding table is used for all runs in the inverted file. The frequency information can be collected while writing the final merged run file, and then another pass is needed to copy this run file to the final file with the Huffman coding.

Experiments show that for the high-frequency values up to five, the Huffman encoding takes the same number of bits for storage. On the other hand, there are large gaps between extant term frequency values in the higher ranges; thus, using a fixed coding table or function, the encodings will use more bits than is necessary.

Table 1 shows that Huffman coding is indeed the most effective encoding for the term frequencies, as was to be expected. The differences can be significant for the baseline – 20 percent for IEEE and 5 percent for Wikipedia –, but for the difference-based encoding, it is virtually indistinguishable from the unary encoding. This behavior can be explained by the extreme bias towards low term frequency values in the difference-based index.

Table 1: Storage sizes of the term frequencies in the inverted lists using different encodings, in megabits. Note that the numbers do not include padding between the runs; the Huffman figure excludes the size for the code table.

	IEEE		Wikipedia	
	Baseline	Diff	Baseline	Diff
unary	248	49	690	159
γ	215	53	699	167
Huffman	208	49	662	159

Thus, there is little benefit in using Huffman for encoding the term frequencies, but the additional overhead while indexing suggests the use of the simpler unary encoding.

Using Huffman coding for the element ID deltas is infeasible because there is a large range of possible values; this leads to an enormous code tree that has to be stored in main memory.

The encodings described above strongly favor small numbers, but the element name IDs are assigned in the order they occur in the document collection. Depending on the collection, the numbers can get quite large; the INEX Wikipedia collection, for example, has 1,257 different element names. This will be even more pronounced if different collections with diverse schemas are stored in the same index.

The first thing that comes to mind is to assign the element name IDs in decreasing order of frequency in the collection, instead of in sequence. This requires another pass over the metadata index at indexing time, but does not affect retrieval time. Although the global distribution may not be optimal in the general case – different documents may use different tags, depending on the author or the schema –, it should work for our test collections, both of which use a single schema with a limited vocabulary.

2.7 Related work

Many of the optimizations that were proposed for atomic-document retrieval are not applicable for element retrieval without modification, because they assume that each document is independent of each other document.

Most of the researchers working on XML retrieval pay little attention to space or time savings; a commonly used approach is to store the indexes in an SQL database [Geva 2006, Theobald 2006].

The GPX retrieval engine [Geva 2006] only stores nodes with text children in the index, but does not attempt to obtain the correct term frequencies for their ancestors. Instead, it calculates their retrieval score based on a completely different formula than that for the leaves, by simply summing the children's scores and applying a dampening factor. Although evaluation results indicate that this model works well in the tested scenarios, it does not support the retrieval of small elements at all (elements of types like italics are simply not included in the index), so if a query demands that some text occurs in a title element, GPX's index structures cannot answer this query.

Most of the research on index structures optimized for semistructured data goes back to the early days of XML or even SGML. The idea of calculating the term frequency values of inner nodes from the term frequency values of their children is not new. Shin, Jang, and Jin [1998] use this approach (but they do not discuss mixed-content elements). Their storage structure for the inverted lists is not particularly space-efficient: For each entry in the inverted list, they store the document ID, a unique element ID that also encodes the position in the document, the element's level in the document's DOM tree, and an ID of the element type. This storage format leads to a lot of redundancy, as it is repeated for each term in a given element.

Furthermore, although the idea of a simple element ID that can be used to calculate the parent's ID using a simple formula is nice, their implementation is simply not practical for arbitrary document collections. They assume a k-ary tree structure – that is, each element can have only up to k children. A fixed limit for the number of children is problematic, because it has to be rather large in order to accommodate all conceivable documents. If we use a single value for all documents, it would have to be large: For the INEX test collections we used for evaluation, k would have to be 1023 (IEEE collection) respectively 8503 (Wikipedia collection). Even if k is determined separately for each document, new problems arise: Because k has to be known to determine the parent's ID, it has to be stored somewhere for each document, either redundantly in the inverted list or in a separate index structure.

This implies that the element IDs get large, and they do not lend themselves to delta encoding, so they take up much space in the index. The element type is obviously redundant, and although the authors state that they use a length-based normalization for scores, they do not mention where the length is stored; either it too is part of the index, in which case it is another redundant number in the index, or it is stored separately in an unspecified place.

Lee, Yoo, Yoon, and Berra [1996] present a space-saving full-text index structure for structured documents. The basic idea is to reduce storage requirements by not storing term occurrences that can be derived from other index nodes. Unfortunately, their index structures only support boolean queries, so they do not provide for term frequencies. Furthermore, they do not take into account that non-leaf nodes may also contain text in mixed content, so their index structures are not applicable in all circumstances.

Myaeng, Jang, Kim, and Zhoo [1998], too, ignore mixed-content elements for their index structures. Furthermore, they store the complete element type information for each term occurrence; this is a significant waste of space. Jang, Kim, and Shin [1999] and Shin, Jang, and Jin [1998] also describe difference-based indexing, but they store element-level metadata redundantly in the inverted list.

3 Evaluation

To show that our index structures are indeed useful, we evaluate its properties on standard test collections. As we have mentioned before, our main focus is on

index size and retrieval time. We made sure that all versions of the search engine yield exactly the same results.

3.1 Implementation and test environment

We implemented the retrieval system in Java. The tests were executed on a 3.2 GHz Pentium 4 system with 1 GB RAM on an ext3 file system on two SATA hard drives in a RAID 0 under Ubuntu Linux 6.10.

As the baseline, we use an index with all elements stored in the inverted lists, with the inverted lists compressed using bit-based compression and the digest compressed using byte-based compression. We compare that baseline to the difference-based indexing scheme with two variants of the compression of the metadata, bit-based and byte-based.

To make our comparison fair, most of the code is shared in the implementations. One notable difference is the propagation of term frequencies: In the baseline version, it is not needed at all, so we made sure that the corresponding code is not executed at all, in order to avoid the (slight) penalty it incurs.

To smooth out random effects on retrieval time, all topics were executed in sequence five times and the mean time was used for our evaluation. (Note that the executions of the same topics were not adjacent, but all the other topics were in between; this avoids possible caching effects.)

3.2 INEX

For a scientific approach to XML retrieval, new test collections were needed because the existing ones (notably TREC) did not support element retrieval. Thus, the Initiative for the Evaluation of XML Retrieval (INEX) was started in 2002 [Fuhr, Gövert, Kazai, and Lalmas 2002] in order to establish a testbed for XML retrieval methods, along the lines of TREC.

Each year, a new set of test data is created and used for evaluation the participants' retrieval engines:

- A collection of documents that is searched; until 2005, this was a collection of IEEE journal articles [Fuhr, Lalmas, Malik, and Szlávik 2005], from 2006 on, a conversion of Wikipedia[2] to XML format is used [Denoyer and Gallinari 2006].
- A set of topics comprising a description of an information need and corresponding queries.
- A set of relevance assessments, where the authors of the topics assign a level of relevance to (a subset of) the elements in the collection.

We use the test collections from INEX 2005 and INEX 2006. Table 2 gives an overview of the properties of these collections. The two collections differ in several important qualities:

- The Wikipedia collection has about forty times as many documents as the IEEE collection, but the average length of a document is only a tenth.
- The Wikipedia collection has about eight times as many distinct index terms.
- Although the Wikipedia collection has many more distinct element names, the average number that is used in each document is smaller.

Table 2: Test collections statistics. The IEEE collection is the collection that was used for INEX 2005, the Wikipedia collection is the collection that was used for INEX 2006. The token count excludes stop words.

	Number of ...				Mean length of
	documents	elements	distinct terms	dist. el. names	a document
IEEE	16,819	11,411,134	280,980	178	2,917
Wikipedia	659,388	52,562,497	2,337,819	1,257	241

3.3 Index size

It is clear that less storage is required if less data is stored in the index. Figure 4 shows that the savings are significant: The index size is reduced to 56 percent of the baseline index size for the Wikipedia collection (from 871 megabytes to 485 megabytes) and to 50 percent for the IEEE collection (from 202 megabytes to 99 megabytes).

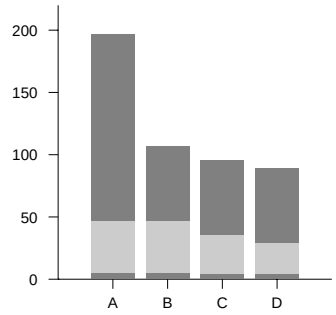
The lexicon and the metadata are not different between the baseline and the difference-based index. The size of the lexicon is negligible compared to the other parts of the index, but the metadata has a size comparable to that of the inverted list. Using a bit-based encoding for the metadata file reduces the size of it by about 25 to 30 percent but increases retrieval time by about 10 to 15 percent.

Re-ordering the element names by decreasing frequency of occurrence in

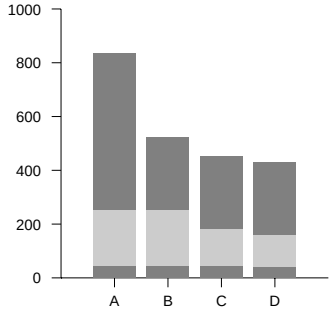
2.http://wikipedia.org

A: Baseline
B: Difference-based inverted list, byte-compressed metadata
C: Difference-based inverted list, bit-compressed metadata
D: Difference-based inverted list, bit-compressed metadata, sorted element name IDs

The boxes are, from bottom to top, the lexicon, the metadata index, and the inverted list.

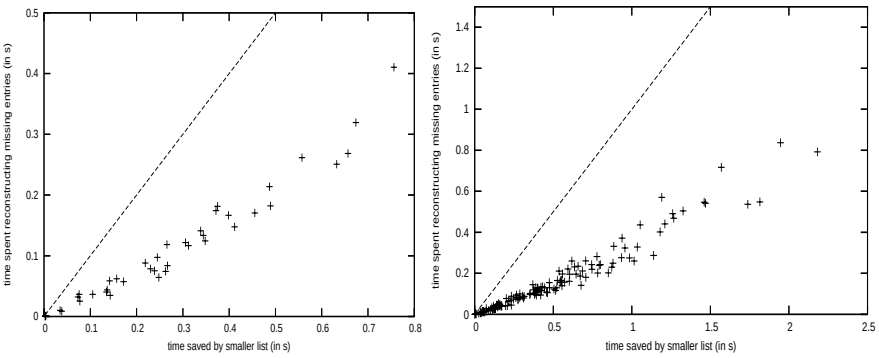


INEX 2005 data (IEEE)



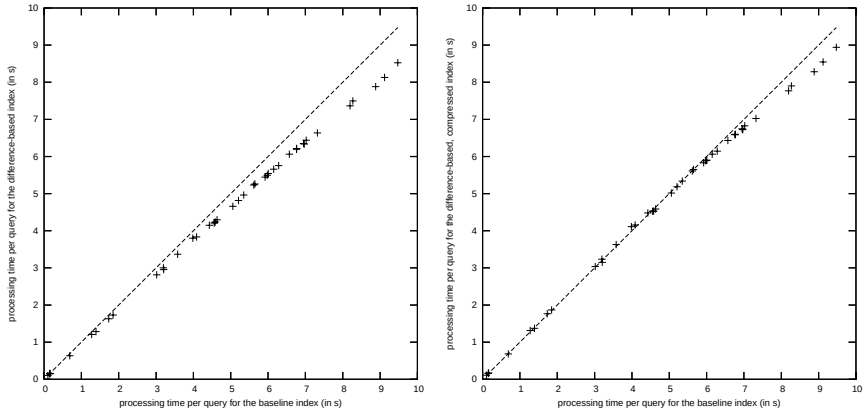
INEX 2006 data (Wikipedia)

Figure 4: Index sizes, in relation to the size of the original XML files. For example, the smallest Wikipedia index (D) is about 8 percent the size of the original XML files.

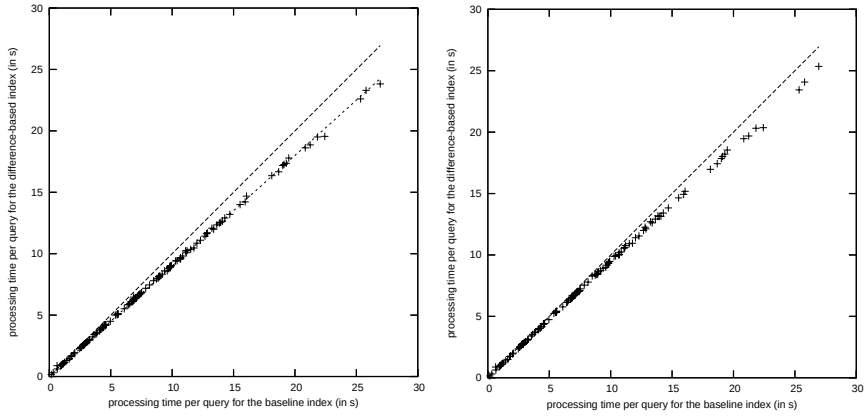


INEX 2005 *INEX 2006*

Figure 5: Explanation why the retrieval time does not change much: The time needed to read the inverted list entries is reduced (*x* axis), but this is offset by the time needed to reconstruct the nodes that are not stored in the index (*y* axis). Points below the dashed line indicate a net saving for that query, points above the line indicate a net loss.



INEX 2005 test data (IEEE)



INEX 2006 test data (Wikipedia)

Figure 6: Retrieval time: baseline versus difference-based. Each point represents the time needed to process one particular query; for points that lie below the dashed line, retrieval on the difference-based index is faster. The diagrams on the left compare the baseline to the difference-based index with byte-compressed metadata, the diagrams on the right compare the baseline to the difference-based index with bit-compressed metadata.

the collection only leads to minor savings of about 0.2 percent of the metadata index size for IEEE and less than 0.001 percent for Wikipedia with byte-based encoding. This is no big surprise for the IEEE collection, which only has 178 distinct element names, but for Wikipedia, a better compression ratio could have been expected. However, of the Wikipedia collection's 1,257 element names, 1,052 (more than 80 percent) occur at most three times each, because of peculiarities of the conversion program (it regarded all text occurring in angle brackets as element names, leading to element names like stdio.h). Because the vocabulary of the original markup is quite limited, the most-used element names have low numbers anyway, so the savings that can be expected are rather low. For the bit-based encoding, the savings are more significant: about 12 percent (Wikipedia) and 16 percent (IEEE).

3.4 Retrieval time

For timing, we use the official query sets from 2005 and 2006, using the title field of the topics (that is, the keyword-based query without structural constraints, see [Trotman and Sigurbjörnsson 2004]).

We cannot expect retrieval time to drop, because the reduced storage requirements and shorter read times are offset by the reconstruction of the missing elements. Retrieval time should not, however, increase compared to the baseline. This is confirmed by our measurements, as Figure 6 shows: The retrieval time does not increase noticeably for any topic, in fact, we get a minor reduction of retrieval time (about 5 percent). Figure 5 shows that indeed the time needed to read the inverted list entries from the index is reduced, but extra time is required to reconstruct the entries.

One important observation is that the search engine spends a significant portion, from 55 up to 85 percent, of the total time it needs to process a query in obtaining metadata.

As a side note, the time needed to create the index is reduced by about 40 percent for difference-based indexes, because the inverted lists are significantly shorter, so less data has to be sorted on

disk. Indexing the complete Wikipedia collection takes about 130 minutes, indexing the IEEE collection takes about 25 minutes.

3.5 Comparison to traditional information retrieval

Witten, Moffat, and Bell [1999] give figures for the index size in relation to the size of the original documents for traditional information retrieval. The size of the index structures – inverted files and auxiliary files – is about 10 to 25 percent of the size of the documents for four collections with different characteristics. Compared to the uncompressed XML files, XML retrieval indexes are in the same range, at about 10 to 15 percent, although they store considerably more data. The main reason is, of course, that XML is a very verbose format; the markup takes up a large fraction of the file, and whitespace is often used to make the XML file more attractive.

3.6 Conclusions and future work

We have shown that it is possible to significantly reduce the space overhead of XML full-text indexes without affecting the results or increasing retrieval time – in fact, retrieval time is slightly reduced compared to the baseline. Our index structures support retrieval of *all* element types, including very short elements. This enables our search engine to support queries that target these small elements.

In the future, we will need to further reduce the retrieval time; our experiments clearly show that for our index structures, obtaining the metadata is the most costly operation, so applying top-k methods to avoid getting the metadata for all results seems promising. The rationale is that we are hardly ever interested in *all* retrieval results, but only in the best k; for typical interactive retrieval scenarios, k will typically be just large enough to cover a few result pages.

Furthermore, as we have mentioned, the index structures are not update-friendly; if the usage scenario differs from our expected scenario of unchanging documents, the index structures will have to be adapted to deal with this.

4 References

[Bentley and Yao 1976] *Bentley, J. L.; Yao, A. C.-C.:* An almost optimal algorithm for unbounded searching. Information Processing Letters, 5(3):82–87, 1976.

[Denoyer and Gallinari 2006] *Denoyer, L.; Gallinari, P.:* The Wikipedia XML corpus. SIGIR Forum, 40(1):64–69, 2006.

[Dopichaj 2006] *Dopichaj, P.:* The University of Kaiserslautern at INEX 2005. In [Fuhr, Lalmas, Malik, and Kazai 2006].

[Dopichaj 2007] *Dopichaj, P.:* Improving content-oriented XML retrieval by applying structural patterns. In ICEIS 2007 – Proceedings of the Ninth International Conference on Enterprise Information Systems. 2007.

[Elias 1975] *Elias, P.:* Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, 21(2):194–203, 1975.

[Fuhr, Gövert, Kazai, and Lalmas 2002] *Fuhr, N.; Gövert, N.; Kazai, G.; Lalmas, M. (eds.).:* Proceedings of the 1st INEX Workshop. ERCIM, 2002.

[Fuhr, Lalmas, Malik, and Kazai 2006] *Fuhr, N.; Lalmas, M.; Malik, S.; Kazai, G. (eds.).:* Advances in XML Information Retrieval and Evaluation: Fourth Workshop of the INitiative for the Evaluation of XML Retrieval (INEX 2005). Springer, 2006.

[Fuhr, Lalmas, Malik, and Szlávik 2005] *Fuhr, N.; Lalmas, M.; Malik, S.; Szlávik, Z. (eds.).:* Advances in XML Information Retrieval: Third International Workshop of the Initiative for the Evaluation of XML Retrieval (INEX 2004). Springer, 2005.

[Geva 2006] *Geva, S.:* GPX – gardens point XML IR at INEX 2005. In [Fuhr, Lalmas, Malik, and Kazai 2006], pp. 240–253.

[Huffman 1952] *Huffman, D.:* A method for the construction of minimum-redundancy codes. In Proc. of the I.R.E., 40(9):1098–1101

[Jang, Kim, and Shin 1999] *Jang, H.; Kim, Y.; Shin, D.:* An effective mechanism for index update in structured documents. In CIKM 1999 proceedings. 1999.

[Lee, Yoo, Yoon, and Berra 1996] *Lee, Y. K.; Yoo, S.-J.; Yoon, K.; Berra, P. B.:* Index structures for structured documents. In Proc. DL 1996, pp. 91–99. 1996.

[Myaeng, Jang, Kim, and Zhoo 1998] *Myaeng, S. H.; Jang, D.-H.; Kim, M.-S.; Zhoo, Z.-C.:* A flexible model for retrieval of SGML documents. In W. B. Croft; A. Moffat; C. J. van Rijsbergen; R. Wilkinson; J. Zobel (eds.), SIGIR 1998 proceedings, pp. 138–145. ACM Press, 1998.

[Ramírez, Westerveld, and de Vries 2006] *Ramírez, G.; Westerveld, T.; de Vries, A. P:.* Using small XML elements to support relevance. In E. N. Efthimiadis; S. T. Dumais; D. Hawking; K. Järvelin (eds.), SIGIR 2006 proceedings. ACM, 2006.

[Shin, Jang, and Jin 1998] *Shin, D.; Jang, H.; Jin, H.:* BUS: an effective indexing and retrieval scheme in structured documents. In DL 1998 proceedings. 1998.

[Spärck Jones, Walker, and Robertson 1998] *Spärck Jones, K.; Walker, S.; Robertson, S. E.:* A probabilistic model of information retrieval: development and status. Technical report, Computer Laboratory, University of Cambridge, 1998.

[Theobald 2006] *Theobald, M.:* TopX – Efficient and Versatile Top-*k* Query Processing for Text, Structured, and Semistructured Data. Ph.D. thesis, Universität des Saarlandes, 2006.

[Trotman and Sigurbjörnsson 2005] *Trotman, A.; Sigurbjörnsson, B.:* Narrowed extended XPath I (NEXI). In [Fuhr, Lalmas, Malik, and Szlávik 2005].

[Witten, Moffat, and Bell 1999] *Witten, I. H.; Moffat, A.; Bell, T. C.:* Managing Gigabytes. Morgan Kaufmann, 1999.

Philipp Dopichaj is a scientific staff member of the database and information systems group (DBIS) at the University of Kaiserslautern. He is currently finishing his Ph. D. thesis on content-oriented XML retrieval.

Philipp Dopichaj
Universität Kaiserslautern
Fachbereich Informatik
Gottlieb-Daimler-Str.
67663 Kaiserslautern
dopichaj@informatik.uni-kl.de
http://wwwlgis.informatik.uni-kl.de