SCM: Structural Contexts Model for Improving Compression in Semistructured Text Databases*

Joaquín Adiego[1], Gonzalo Navarro[2], and Pablo de la Fuente[1]

[1]Departamento de Informática, Universidad de Valladolid, Valladolid, España.
{jadiego, pfuente}@infor.uva.es
[2]Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile. gnavarro@dcc.uchile.cl

Abstract We describe a compression model for semistructured documents, called *Structural Contexts Model*, which takes advantage of the context information usually implicit in the structure of the text. The idea is to use a separate semiadaptive model to compress the text that lies inside each different structure type (e.g., different XML tag). The intuition behind the idea is that the distribution of all the texts that belong to a given structure type should be similar, and different from that of other structure types. We test our idea using a word-based Huffman coding, which is the standard for compressing large natural language textual databases, and show that our compression method obtains significant improvements in compression ratios. We also analyze the possibility that storing separate models may not pay off if the distribution of different structure types is not different enough, and present a heuristic to *merge* models with the aim of minimizing the total size of the compressed database. This technique gives an additional improvement over the plain technique. The comparison against existing prototypes shows that our method is a competitive choice for compressed text databases. Finally, we show how to apply SCM over text chunks, which allows one to adjust the different word frequencies as they change across the text collection.

Keywords: Text Compression, Compression Model, Semistructured Documents.

1 Introduction

Compression of large document collections not only reduces the amount of disk space occupied by the data, but it also decreases the overall query processing time in text retrieval systems. Improvements in processing times are achieved thanks to the reduced disk transfer times necessary to access the text in compressed form. Since in the last decades processor speeds have increased much faster than disk transfer speeds, trading disk transfer times by processor decompression times has become a better and better choice. Moreover, recent research

* This work was partially supported by CYTED VII.19 RIBIDI project (all authors) and Fondecyt Project 1-020831 (second author).

on "direct" compressed text searching, i.e., searching a compressed text without decompressing it, has led to a win-win situation where the compressed text takes less space and is searched faster than the plain text [WMB99,ZMNBY00].

Compressed text databases pose some requirements that outrule some compression methods. The most definitive is the need for random access to the text without the possibility of decompressing it from the beginning. This rules out most adaptive compression methods such as Ziv-Lempel compression and arithmetic coding. On the other hand, semiadaptive models such as Huffman [Huf52] yield poor compression. In the case of compressing natural language texts, it has been shown that an excellent choice is to consider the words, not the characters, as the source symbols [Mof89]. Thanks to the biased distribution of words, the use of this model joined to a Huffman coder gives compression ratios close to 25%, much better than those usually obtained with the best adaptive methods. These results are barely affected if one switches to byte-oriented Huffman coding, where each source symbol is coded as a sequence of bytes instead of bits. Although compression ratios raise to 30% (which is still competitive), we have in exchange much faster decoding and searching, which are essential features for compressed text databases. Finally, the fact that the alphabet and the vocabulary of the text collections coincide permits efficient and highly sophisticated searching, both in the form of sequential searching and in the form of compressed inverted indexes over the text [WMB99,ZMNBY00,NMN+00,MNZB00].

Although the area of natural language compressed text databases has gone a long way since the end of the eighties, it is interesting that little has been done about considering the structure of the text in this picture. Thanks to the widespread acceptance of SGML, HTML and XML as the standards for storing, exchanging and presenting documents, semistructured text databases are becoming the standard. Some techniques to exploit the text structure have been proposed, such as *XMill* [LS00] and *XMLPPM* [Che01]. However, these are not designed to permit searching the text. Others, like *XGrind* [TH02], permit searching but do not take advantage of the structure (they just allow it).

Our goal in this paper is to explore the possibility of considering the text structure in the context of a compressed text database. We aim at taking advantage of the structure, while still retaining all the desirable features of a word-based Huffman compression over a semiadaptive model. An idea like that of XMLPPM, where the context given by the path in the structure tree is used to model the text in the subtree, is based on the intuition that the text under similar structural elements (i.e., XML tags) should follow a similar distribution. (In fact XMLPPM uses the full path, which is more powerful.) Although this compression is adaptive and does not fit our search purposes, a simplification where only the last element in the path is considered can be joined to a semiadaptive model, which is suitable for searching. The idea is then to use separate semiadaptive models to compress the text that lies inside different tags. For example, in an email archive, a different model would be used for each of the fields `From:`, `Subject:`, `Date:`, `Body:`, etc.).

While the possible gain due to this idea is clear, the price is that we have to store several models instead of just one. This may or may not pay off. In our example, coding the dates separately is probably a good idea, but coding the subjects separate from the bodies is probably not worth the extra space of storing two models (e.g., two Huffman trees). Hence we also design a technique to *merge* the models if we can predict that this is convenient in terms of compressed file length. Although the problem of finding the optimal merging seems a hard combinatorial problem, we design a heuristic to automatically obtain a reasonably good merging of an initially separate set of models, one per tag. Other related techniques can be found in [BCC+00]

In a text collection, some words can be common in some parts (with high frequency) and rather uncommon in others (low frequency). This is typical in news archives, for example, where some subjects are hot issues today and fade out in a few weeks. Considering this fact, another possiblility is to apply SCM over different text chunks. This idea allows us to adjust word frecuencies as they change across the text, improving compression rates.

This model, which we call *Structural Contexts Model*, is general and does not depend on the coder. We plug it to a word-based Huffman coder to test it. Our experimental results show significant gains over the methods that are insensitive to the structure and over the current methods that consider the structure. At the same time, we retain all the features of the original model that makes it suitable for compressed text databases.

2 Related Work

With regard to compressing natural language texts in order to permit efficient retrieval from the collection, the most successful techniques are based on models where the text words are taken as the source symbols [Mof89], as opposed to the traditional models where the characters are the source symbols.

On the one hand, words reflect much better than characters the true entropy of the text [TCB90]. For example, a semiadaptive Huffman coder over the model that considers characters as symbols typically obtains a compressed file whose size is around 60% of the original size, on natural language. A Huffman coder when words are the symbols obtains 25% [ZMNBY00]. Another example is the WLZW algorithm (Ziv-Lempel on words) [BSTW86,DPS99].

On the other hand, most information retrieval systems use words as the main information atoms, so a word-based compression eases the integration with an information retrieval system. Some examples of successful integration are [WMB99,NMN+00,MW01].

The text in natural language is not only made up of words. There are also punctuation, separators, and other special characters. The sequence of characters between every pair of consecutive words will be called a *separator*. In [BSTW86] they propose to create two alphabets of disjoint symbols: one for coding words and another for separators. Encoders that use this model consider texts as a strict alternation of two independent data sources and encode each one independently.

Once we know that the text starts with a word or a separator, we know that after a word has been coded we can expect a separator and vice versa. This idea is known as the *separate alphabets model*.

A fact that the separate alphabets model does not consider is that in most cases a word is followed by a single blank space as a separator. Since at least the 70% of separators in text are single blanks [Mof89], they propose in [MNZB00] a new data model which uses a single alphabet for both words and separators, and represents the blank space implicitly. This model is known as *spaceless model*. Hence, after each word is decoded, we assume a single blank follows unless the next decoded symbol is a separator.

On the one hand we have to use a larger coding alphabet and then code lengths grow. On the other hand we do not need to code about 35% of the source symbols. It is shown in [MNZB00] that compression improves a bit using this method, although the improvement is not much.

A compression method that considers the document structure is *XMill* [LS00], developed in AT&T Labs. *XMill* is an XML-specific compressor designed to exchange and store XML documents, and its compression approach is not intended for directly supporting querying or updating of the compressed document. *XMill* is based on the *zlib* library, which combines Ziv-Lempel compression (LZ77 [ZL77]) with a variant of Huffman.

Another XML compressor is *XGrind* [TH02], which directly supports queries over the compressed files. An XML document compressed with XGrind retains the structure of the original document, permitting reuse of the standard XML techniques for processing the compressed document. It does not, however, take full advantage of the structure.

Other approaches to compress XML data exist, based on the use of a PPM-like coder, where the context is given by the path from the root to the tree node that contains the current text. One example is *XMLPPM* [Che01], which is an adaptive compressor pased on PPM, where the context is given by the structure.

3 Structural Contexts Model

Let us, for this paper, focus on a semiadaptive Huffman coder, as it has given the best results on natural language texts. Our ideas, however, can be adapted to other encoders. Let us call *dictionary* the set of source symbols together with their assigned codes.

An encoder based on the separate alphabets model (see Section 2) must use two source symbol dictionaries: one for all the separators and the other for all the words in the texts. This idea is still suitable when we handle semistructured documents —like SGML or XML documents—, but in fact we can extend the mechanism to do better.

In most cases, natural language texts are structured in a semantically meaningful manner. This means that we can expect that, at least for some tags, the distribution of the text that appears inside a given tag differs from that of another tag. In our example of Section 1, where the tags correspond to the fields of

an email archive, we can expect that the `From:` field contains names and email addresses, the `Date:` field contains dates, and the `Subject:` and `Body:` fields contain free text.

In cases where the words under one tag have little intersection with words under another tag, or their distribution is very different, the use of separate alphabets to code the different tags is likely to improve the compression ratio. On the other hand, there is a cost in the case of semiadaptive models, as we have to store several dictionaries instead of just one. In this section we assume that each tag should use a separate dictionary, and will address in the next section the way to group tags under a single dictionary.

3.1 Compressing the Text

We compress the text with a word-based Huffman [Huf52,BSTW86]. The text is seen as an alternating sequence of words and separators, where a word is a maximal sequence of alphanumeric characters and a separator is a maximal sequence of non-alphanumeric characters.

Besides, we will take into account a special case of words: *tags*. A tag is a code embedded in the text which represents the structure, format or style of the data. A tag is recognized from surrounding text by the use of delimiter characters. A common delimiter character for an XML or SGML tag are the symbols '<' and '>'. Usually two types of tags exist: *start-tags*, which are the first part of a container element, '`<...>`'; and *end-tags*, which are the markup that ends a container element, '`</...>`'.

Tags will be wholly considered (that is, including their delimiter characters) as words, and will be used to determine when to switch dictionaries at compression and decompression time.

3.2 Model Description

The structural contexts model (as the separate alphabets model) uses one dictionary to store all the separators in the texts, independently of their location. Also, it assumes that words and separators alternate, otherwise, it must insert either an empty word or an empty separator. There must be at least one word dictionary, called the *default dictionary*. The default dictionary is the one in use at the beginning of the encoding process. If only the default dictionary exists for words then the model is equivalent to the separate alphabets model.

We can have a different dictionary for each tag, or we can have separate dictionaries for some tags and use the default for the others, or in general we can have any grouping of tags under dictionaries. As explained, we will assume for now that each tag has its own dictionary and that the default is used for the text that is not under any tag.

The compression algorithm written below makes two passes over the text. In the first pass, the text is modeled and separate dictionaries are built for each tag and for the default and separators dictionary. These are based on the statistics of

words under each tag, under no tag, and separators, respectively. In the second pass, the texts are compressed according to the model obtained.

At the begining of the modeling process, words are stored in the default dictionary. When a start-structure tag appears we push the current dictionary in a stack and switch to the appropriate dictionary. When an end-structure tag is found we must return to the previous dictionary stored in the stack. Both start-structure and end-structure tags are stored and coded using the current dictionary and then we switch dictionaries. Likewise, the encoding and decoding processes use the same dictionary switching technique.

The following code describes the dictionary switching used for modeling, coding and decoding.

Algorithm 1 (Dictionary Switching)

```
current_dictionary ← default_dictionary
while there are more symbols do
        word ← get_symbol()
        if (word is separator)
           then store/code/decode(word, separators_dictionary)
            else store/code/decode(word, current_dictionary)
                 if (word is a start-structure tag)
                    then push(current_dictionary)
                         current_dictionary ← dictionary(word)
                     else if (word is an end-structure tag)
                             then current_dictionary ← pop()
```

3.3 Considering Text Chunks

In addition to tags, we may decide to separate the text collection into a sequence of *chunks*. There will be a different dictionary for each different tag appearing in each chunk. This permits the method to adapt to word frequencies as they change across the text collections.

For each chunk we have a separate default dictionary, but still there is a unique separators dictionary for the whole collection.

There is a tradeoff regarding chunk size. Too small chunks will create too many dictionaries which will require a larger header table to find the right dictionary. Even if many dictionaries are finally merged (Section 4) and shared by many of these headers, the header table may get too large. Also, merging may become too expensive. On the other hand, too large chunks will not permit adapting fast enough to changes in text distribution.

3.4 Entropy Estimation

The entropy of a source is a number that only depends on its model, and is usually measured in *bits/symbol*. It is also seen as a function of the probability distribution of the source (under the model), and refers to the average amount

of information of a source symbol. The entropy gives a lower bound on the size of the compressed file if the given model is used.

Definition 1 (Raw frequency) *Let n be the total number of terms that appear in the text. The raw frequency f_i of term i is given by*

$$f_i = \frac{\mathrm{occ}_i}{n} \tag{1}$$

where occ_i is the number of occurrences of vocabulary term i in the text. The raw frequency is also called occurrence probability of term i.

The fundamental theorem of Shannon [Sha48] establishes that the entropy of a probability distribution $\{p_i\}$ is $\sum_i p_i \log_2(1/p_i)$ bits. That is, the optimum way to code symbol i is to use $\log_2(1/p_i)$ bits. In a zero-order model, the probability of a symbol is defined independently of surrounding symbols. Usually one does not know the real symbol probabilities, but rather estimate them using the raw frequencies seen in the text.

Definition 2 (Zero-order entropy estimation) *Let T_v be the number of vocabulary terms. Bearing in mind Shannon's theorem and assuming that a single dictionary is used to encode symbols, we estimate the zero-order entropy $\mathcal{H}$ of a text*

$$\mathcal{H} = \sum_{i=1}^{T_v} f_i \log_2 \frac{1}{f_i} \tag{2}$$

This definition lets us estimate the entropy when we have only one dictionary. If we want to estimate the entropy value when our model includes multiple dictionaries, we have to combine the entropies of each dictionary.

Definition 3 (Zero-order entropy estimation for a dictionary) *Let n^d be the total number of text terms in dictionary d. Let T_v^d be the total number of distinct terms in dictionary d. Let f_i^d be raw frequency of term i in dictionary d given by*

$$f_i^d = \frac{\mathrm{occ}_i^d}{n^d} \tag{3}$$

where occ_i^d is the number of occurrences of vocabulary term i of dictionary d in the texts. We can reformulate equation 2 to get the entropy for terms in dictionary d:

$$\mathcal{H}^d = \sum_{i=1}^{T_v^d} f_i^d \log_2 \frac{1}{f_i^d} \tag{4}$$

Definition 4 (Zero-order entropy estimation with multiple dictionaries) *Let N be the total number of dictionaries. The zero-order entropy for all dictionaries, $\mathcal{H}$, is computed as the weighted average of zero-order entropies contributed by each dictionary ($\mathcal{H}^d, d \in 1 \ldots N$):*

$$\mathcal{H} = \frac{\sum_{d=1}^{N} n^d \ \mathcal{H}^d}{n} \tag{5}$$

4 Merging Dictionaries

Up to now we have assumed that each different tag and chunk uses its own dictionary. However, this may not be optimal because of the overhead to store the dictionaries in the compressed file. In particular, if two dictionaries happen to share many terms and to have similar probability distributions, then merging both tags under a single dictionary is likely to improve the compression ratio.

In this section we develop a general method to obtain a good grouping of tags/chunks under dictionaries. For efficiency reasons we will use the entropy as the estimation of the size of the text compressed using a dictionary, instead of actually running the Huffman algorithm and computing the exact size.

Definition 5 (Estimated size contribution of a dictionary) *Let $\mathcal{V}^d$ be the size, in bits, of the vocabulary that constitutes dictionary d, and $\mathcal{H}^d$ its estimated zero-order entropy. Then the estimated size contribution of dictionary d is given by*

$$\mathcal{T}^d = \mathcal{V}^d + n^d \mathcal{H}^d \tag{6}$$

Considering the last definition we determine to merge dictionaries i and j when the sum of their contributions is larger than the contribution of their union. In other words, when

$$\mathcal{T}^i + \mathcal{T}^j > \mathcal{T}^{i \cup j} \tag{7}$$

To compute $\mathcal{T}^{i \cup j}$ we have to compute the union of the vocabularies and the entropy of that union. This can be done in time linear in the vocabulary sizes.

Definition 6 (Estimated saving of a merge) *Let $\mathcal{A}^{i \cup j}$ be the estimated saving of merging dictionaries i and j. Then*

$$\mathcal{A}^{i \cup j} = \mathcal{T}^i + \mathcal{T}^j - \mathcal{T}^{i \cup j} \tag{8}$$

Our optimization algorithm works as follows. We start with one separate dictionary per tag/chunk, plus the default dictionary for each chunk (the separators dictionary is not considered in this process). Then, we progressively merge pairs of dictionaries until no further merging promises to be advantageous. Obtaining the optimal division into groups looks as a hard combinatorial problem, but we use a heuristic which produces good results and is reasonably fast.

We start by computing $\mathcal{T}^i$ for every dictionary i, as well as $\mathcal{T}^{i \cup j}$ for all pairs i, j of dictionaries. With that we compute the savings $\mathcal{A}^{i \cup j}$ for all pairs. Then, we merge the pair of dictionaries i and j that maximizes $\mathcal{A}^{i \cup j}$, if this is positive. Then, we erase i and j and introduce $i \cup j$ in the set. This process is repeated until all the $\mathcal{A}^{i \cup j}$ values are negative.

The algorithm is depicted next. We have hidden the details on when the $\mathcal{T}$ values are precomputed and updated. Its cost is $O(VN^3)$ when there are N dictionaries and the vocabulary size is V. This can be reduced to $O(VN^2 \log N)$ by simple tricks such as recomputing savings only for the newly merged dictionaries and keeping dictionary pairs in a priority queue sorted by gain.

Algorithm 2 (Merging Dictionaries)

do $best\_saving \leftarrow 0$
  for $1 \leq i < j \leq N$ **do**
    $current\_saving \leftarrow \mathcal{T}^i + \mathcal{T}^j - \mathcal{T}^{i \cup j}$
    if $(current\_saving > best\_saving)$
      then $best\_saving \leftarrow current\_saving$
        $bi \leftarrow i \quad , \quad bj \leftarrow j$
  if $(best\_saving > 0)$
    then $d_{bi} \leftarrow merge\_dictionaries(d_{bi}, d_{bj})$
      $d_{bj} \leftarrow d_N$
      $N \leftarrow N - 1$
while $(best\_saving > 0)$

5 Evaluation of the Model

We have developed a prototype implementing the Structural Contexts Model with a word-oriented Huffman coding, and used it to empirically analyze our model and evaluate its performance. Dictionaries are compressed using arithmetic character-based adaptive coding. Tests were carried out on the Linux Red Hat 7.2 operating system, running on a computer with a Pentium III processor at 500 MHz and 128 Mbytes of RAM.

For the experiments we selected different size collections of WSJ, ZIFF and AP, from TREC-3 [Har95]. Several characteristics of the collections are shown in Table 1. We concatenated files so as to obtain approximately similar subcollection sizes from the three collections, so the size in Mbytes is approximate.

The structuring of the collections is similar: they have only one level of structuring, with the tag `<DOC>` indicating documents, and inside each document tags indicating document identifier, date, title, author, source, content, keywords, etc.

Size	TREC-WSJ			TREC-ZIFF			TREC-AP		
(Mb)	#T.W.	#V.W.	Ratio	#T.W.	#V.W.	Ratio	#T.W.	#V.W.	Ratio
1	193899	18380	9.479%	161900	12924	7.982%	195915	19103	9.750%
5	874586	38750	4.430%	992067	35555	3.583%	956340	41263	4.314%
10	1669506	52218	3.127%	1821015	51094	2.805%	1721137	54058	3.140%
20	3370544	71832	2.131%	3489650	71136	2.038%	3486098	73820	2.117%
40	6690067	97190	1.452%	6970106	102737	1.473%	6985763	101480	1.452%
60	10015765	116221	1.160%	10272649	125326	1.219%	10411824	122340	1.175%
100	16672690	144701	0.867%	17289782	165113	0.954%	17252119	157376	0.912%

Table 1. Collection characteristics. For each collection we show the total number of words (#T.W.), the total number of vocabulary words (#V.W.) and the ratio between the two (Ratio).

When text chunks are not used, the average speed to compress all collections is around 128 Kbytes/sec. In this value we include the time needed to model, merge dictionaries and compress. Time for merging dictionaries ranges from 4.37 seconds for 1 Mb to 40.27 seconds for 100 Mb. Its impact is large for the smallest collection (about 50% of the total time), but it becomes much less significant for the largest collection (about 5%). The reason is that merging time is linear in the vocabulary size, which grows sublinearly with the collection size [Hea78], typically close to $O(\sqrt{n})$. Although merging time also depends quadratically on the number of different tags, this number is usually small and does not grow with the collection size but depends on the DTD/schema.

In Table 2 we show original sizes, compressed sizes and compression ratios for each collection. It can be seen that compression ratios improve for larger collections, as the impact of the vocabulary is reduced [Hea78].

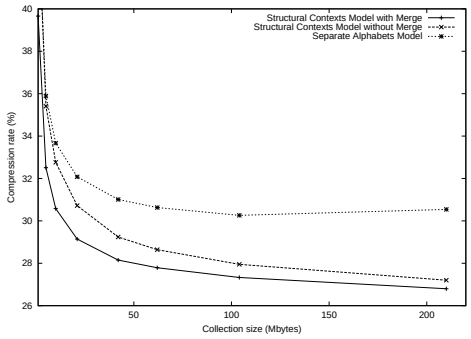
TREC-WSJ			TREC-ZIFF			TREC-AP		
Original	Compr.	Ratio	Original	Compr.	Ratio	Original	Compr.	Ratio
1221659	484575	39.66%	1021882	376180	36.81%	1185968	492832	41.55%
5516592	1793950	32.51%	6083389	1956195	32.15%	5805776	1952979	33.63%
10510481	3214613	30.58%	11164171	3480842	31.17%	10469592	3315087	31.66%
21235547	6190051	29.14%	21306059	6414762	30.10%	21219693	6371426	30.02%
42113697	11858566	28.15%	42659558	12452756	29.19%	42523572	12307072	28.94%
62963963	17498136	27.79%	62966279	18131869	28.79%	63343648	18054387	28.50%
104942941	28681879	27.33%	105709264	29972861	28.35%	105018927	29479824	28.07%

Table 2. Sizes and compression ratios for each collection.

In Figure 1 we can see a comparison, for WSJ (using up to 200 Mb this time), of the compression performance using the plain separate alphabets model (SAM) and the structural context model (SCM) with and without merging dictionaries. For short texts, the vocabulary size is significant with respect to the text size, so SCM without merging pays a high price for the separate dictionaries and does not improve upon SAM. As the text collection grows, the impact of the dictionaries gets reduced and we obtain nearly 10% additional compression. The SCM with merging obtains similar results for large collections (12.25% additional compression), but its performance is much better on small texts, where it starts obtaining 11% even for 1 Mbyte of text.

Table 3 shows the number of dictionaries merged. Column "Initial" tells how many dictionaries are in the beginning: The default and separators dictionary plus one per tag, except for `<DOC>`, which marks the start of a document and uses the default dictionary. Column "Final" tells how many different dictionaries are left after the merge.

For example, for small WSJ subsets, the tags `<DOCNO>` and `<DOCID>`, both of which contain numbers and internal references, were merged. The other group that was merged was formed by the tags `<HL>`, `<LP>` and `<TEXT>`, all of which



Size	SCM+merge	SCM	SAM
1221659	39.66%	44.64%	44.52%
5516592	32.51%	35.41%	35.89%
10510481	30.58%	32.76%	33.67%
21235547	29.14%	30.72%	32.08%
42113697	28.15%	29.24%	31.01%
62963963	27.79%	28.64%	30.63%
104942941	27.33%	27.95%	30.26%
210009482	26.80%	27.20%	30.54%

Figure 1. Compression ratios using different models, for WSJ.

contain the text of the news (headlines, summary for teletypes, and body). On the larger WSJ subsets, only the last group of three tags was merged. This shows that our intuition that similar-content tags would be merged is correct. The larger the collection, the less the impact of storing more vocabularies, and hence the fewer merges will occur.

Aprox.	TREC-WSJ		TREC-ZIFF		TREC-AP	
Size(Mb)	Initial	Final	Initial	Final	Initial	Final
1	11	8	10	4	9	5
5	11	8	10	4	9	5
10	11	8	10	4	9	7
20	11	9	10	6	9	7
40	11	9	10	6	9	7
60	11	9	10	6	9	7
100	11	9	10	7	9	7

Table 3. Number of dictionaries used.

The method to predict the size of the merged dictionaries from the vocabulary distributions was quite accurate: our prediction was usually 98%–99% of the final value.

Let us now consider the use of text chunks. In Table 4 we can see a comparison of the compression performance using differents chunks sizes over the same collection sizes for WSJ. The best gain obtained is around 0.03%, not really significant. This can be due to the characteristics of WSJ: all the texts are very uniform, with similar distributions of words. In fact, all dictionaries in different chunks of tags `<HL>`, `<LP>` and `<TEXT>` were merged. On the other hand, the time for generating and merging dictionaries grows fast as the number of dictio-

Aprox.	Chunk size (Mbytes)				
Size(Mb)	0	2	4	8	16
1	39.66%	39.66%	39.66%	39.66%	39.66%
5	32.51%	32.51%	32.51%	32.51%	32.51%
10	30.58%	30.57%	30.57%	30.58%	30.58%
20	29.14%	29.13%	29.13%	29.13%	29.14%
40	28.15%	28.13%	28.13%	28.14%	28.14%
60	27.79%	27.76%	27.76%	27.76%	27.77%
100	27.33%	27.28%	27.28%	27.28%	27.29%

Table 4. Compression ratios using differents chunk sizes in Mbytes. Zero size shows compression ratio without using chunks.

naries grows. With these results, we can conclude that the use of chunks is not profitable in this case.

Finally, we compared our prototype (using merging) against other compression systems: the *MG* system, *XMill*, and *XMLPPM*. The MG system [WMB99] is a public domain software, versatile and of general purpose, which handles text and images. MG compresses structured documents by handling tags as words, and uses a variant of word-based Huffman compression called *Huffword*. On the other hand, *XMill* [LS00] is an XML-specific compressor based on Ziv-Lempel and Huffman, able to handle the document structure. XMLPPM [Che01] is also specific of XML and based on adaptive PPM over the structural context.

We compressed all the collections with the four systems[1] and averaged compression rates for each collection size. Average compression rates are shown in Figure 2. *XGrind* was not included because we could not find public code for it. *CGrep* [MNZB00] was not included because it is byte-oriented and the comparison would be unfair against it.

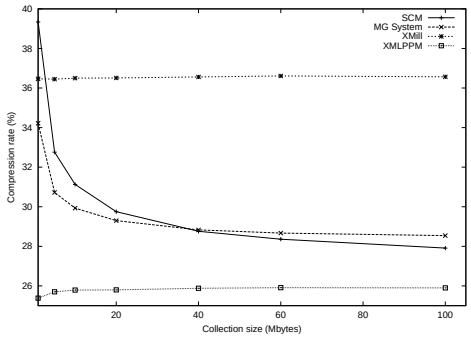
XMill obtains an average compression ratio roughly constant in all cases because it uses *zlib* as its main compression machinery. The compression ratio obtained is not competitive in this experiment.

XMLPPM, on the other hand, obtains the best compression. This shows that the idea of using the structural context to compress is good. The problem of XMLPPM is that its compression is adaptive, and hence it is not suitable for direct access on large compressed text databases.

Our prototype is better than MG for medium and large collection sizes, but not for small sizes. This can be due to our penalty in storing more than one dictionary. SCM starts to be better from 40 Mbytes, and for 100 Mbytes it improves over MG by 2.2%.

Note also that the difference between XMLPPM and our prototype is rather small for large collection sizes. In any case, the penalty is a rather small price for permitting direct access to the text.

[1] XMLPPM required several changes to the sources in order to run properly, but these did not affect the compressibility of the collection.



Size	SCM	MG	XMill	XMLPPM
1	39.34%	34.22%	36.46%	25.38%
5	32.76%	30.72%	36.44%	25.70%
10	31.13%	29.93%	36.49%	25.79%
20	29.75%	29.30%	36.51%	25.80%
40	28.76%	28.83%	36.55%	25.88%
60	28.36%	28.67%	36.61%	25.91%
100	27.91%	28.54%	36.56%	25.90%

Figure 2. Comparison between SCM and other systems over WSJ, using default settings for all. The ratios shown in the table are average values for each collection size, over the different collections tested.

6 Conclusions and Future Work

We have proposed a new model for compressing semistructured documents based on the idea that texts under the same tags should have similar distributions. This is enriched with a heuristic that determines a good grouping of tags so as to code each group with a separate model. On the other hand, the impact of the model on the retrieval performance is insignificant, in fact it is similar to the retrieval performance over compressed documents.

We have shown that the idea actually improves compression ratios by more than 10% with respect to the basic technique. We have compared our prototype against state-of-the-art compression systems, showing that our prototype obtains the best compression for medium and large collections (more than 40 Mbytes) among techniques that permit direct access to the text, which is essential for compressed text databases. On very large texts, the difference with the best prototype, which however does not permit direct text access, is no more than 7.2%. These text sizes are the most interesting for compressed text databases.

The prototype is a basic implementation and we are working on several improvements, which will make it even more competitive. We can tune our method to predict the outcome of merging dictionaries: Since we know that usually our prediction is 1%–2% off, we could add a mean value to our prediction. Also, we can try the *spaceless model* [MNZB00], which should give a small additional gain. However, the need to include the separators in all the dictionaries may make this approach unsuitable for our case.

Use of text chunks did not appear to be promising, but we plan to work on defining them more cleverly. We still have to test their effect on other collections. With respect to the study of the method itself, we have to investigate more in depth the relationship between the type and density of the structuring and the improvements obtained with our method, since its success is based on a semantic

assumption and it would be interesting to see how this works on other text collections.

References

[BCC+00] A. L. Buchsbaum, D. F. Caldwell, K. Ward Church, G. S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *Symposium on Discrete Algorithms*, pages 175–184, 2000.

[BSTW86] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.

[Che01] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proc. Data Compression Conference (DCC 2001)*, pages 163–, 2001.

[DPS99] J. Dvorský, J. Pokorný, and V. Snásel. Word-based compression methods and indexing for text retrieval systems. In *ADBIS'99*, LNCS 1691, pages 75–84. Springer, 1999.

[Har95] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[Hea78] H. S. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, 1978.

[Huf52] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Engineers*, 40(9):1098–1101, 1952.

[LS00] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proc. ACM SIGMOD 2000*, pages 153–164, 2000.

[MNZB00] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[Mof89] A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.

[MW01] A. Moffat and R. Wan. RE-store: A system for compressing, browsing and searching large documents. In *Proc. 8th Intl. Symp. on String Processing and Information Retrieval (SPIRE 2001)*, pages 162–174, 2001.

[NMN+00] G. Navarro, E. Silva de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[Sha48] C. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27:398–403, July 1948.

[TCB90] Ian H. Witten Timothy C. Bell, John G. Cleary. *Text Compression*. Prentice Hall, Englewood Cliffs, N.J., 1990.

[TH02] P. Tolani and J.R. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, 2002. citeseer.nj.nec.com/503319.html.

[WMB99] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, Inc., second edition, 1999.

[ZL77] J. Ziv and A. Lempel. An universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–343, 1977.

[ZMNBY00] N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, November 2000.