Using Structural Contexts to Compress Semistructured Text Collections *†

Joaquín Adiego ‡§ Gonzalo Navarro ¶ Pablo de la Fuente ‖

Abstract

We describe a compression model for semistructured documents, called *Structural Contexts Model (*SCM*)*, which takes advantage of the context information usually implicit in the structure of the text. The idea is to use a separate model to compress the text that lies inside each different structure type (e.g., different XML tag). The intuition behind *SCM* 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 mainly focus on semistatic models, and test our idea using a word-based Huffman method. This is the standard for compressing large natural language text databases, because random access, partial decompression, and direct search of the compressed collection is possible. This variant, dubbed *SCMHuff*, retains those features and improves Huffman's compression ratios.

We consider 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 gives an additional improvement over the plain technique. The comparison against existing prototypes shows that, among the methods that permit random access to the collection, *SCMHuff* achieves the best compression ratios, 2–4% better than the closest alternative.

From a purely compression-aimed perspective, we combine *SCM* with PPM modeling. A separate PPM model is used to compress the text that lies inside each different structure type. The result, *SCMPPM*, does not permit random access nor direct search in the compressed text, but it gives 2–5% better compression ratios than other techniques for texts longer than 5 megabytes.

Keywords: Text compression, Semistructured documents, Compressed text databases.

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.

*This work was partially supported by TIC2003-09268 project, MCyT, Spain (first and third authors); and Fondecyt Grant 1-050493, Chile (second author).

†Preliminary versions of this article appeared in *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, pages 153–167, 2003; and *Proc. 14th Data Compression Conference (DCC'04)*, page 522, 2004 (poster).

‡Depto. de Informática, Universidad de Valladolid, Valladolid, Spain. jadiego@infor.uva.es

§Corresponding author.

¶Depto. de Ciencias de la Computación, Universidad de Chile, Santiago, Chile. gnavarro@dcc.uchile.cl

‖Depto. de Informática, Universidad de Valladolid, Valladolid, Spain. pfuente@infor.uva.es

Improvements in processing times are achieved thanks to the reduced disk transfers 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 an attractive choice. Moreover, recent research on "direct" compressed text searching, that is, 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 (Witten et al., 1999; Ziviani et al., 2000).

Compressed text databases pose some requirements to the eligible compression methods. The most crucial is the need of random access to the text without decompressing it from the beginning. This rules out adaptive compression methods such as Ziv-Lempel or PPM compression. On the other hand, many semistatic methods yield poor compression. In the case of compressing natural language texts, however, it has been shown that an excellent choice is a semistatic model that considers the words, not the characters, as the source symbols (Moffat, 1989). Thanks to the biased distribution of words (which follows a Zipf law (Zipf, 1949)), the use of this model coupled with a Huffman coder (Huffman, 1952) gives compression ratios[1] below 30%, much better than those usually obtained with popular adaptive methods. It is even possible to switch to byte-oriented Huffman coding, where each source symbol is coded as a sequence of bytes instead of bits. Although compression ratios raise to 35% (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 source alphabet and the vocabulary of the text collections coincide permits efficient and highly sophisticated searching, both in the form of sequential searching and of compressed inverted indexes over the text (Witten et al., 1999; Ziviani et al., 2000; Navarro et al., 2000; Moura et al., 2000; Brisaboa et al., 2003).

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 compression techniques to exploit the text structure have been proposed, such as *XMill* (Liefke and Suciu, 2000) and *XMLPPM* (Cheney, 2001). However, these are not designed to permit searching the text. Others, like *XGrind* (Tolani and Haritsa, 2002), permit searching but do not take much advantage of the structure.

Our goal in this paper is to consider 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 semistatic word-based Huffman compression. Our essential idea is to use separate semistatic 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. This relies on the intuition that the text under similar structural elements (i.e., XML tags) should follow a similar distribution, different from other texts. We call our general approach *SCM* (for Structural Contexts Model), and call *SCMHuff* the specific compressor obtained with a word-based byte-oriented Huffman coder.

The idea of using different models depending on the structural context has been anticipated in most of the alternative structure-aware compressors mentioned above. Yet, those usually distinguish among *types* of XML tags. For example, *XMLPPM* has four separate models, namely for elements, attributes, characters, and others. Yet, our idea of using a different model for each different tag

[1] That is, the compressed text size as a percentage of the uncompressed size.

name has not been explored before as far as we know. Actually, *XMLPPM* does use this information up to some extent. When it starts compressing the text under a new structural element, it feeds the character model with a fake character identifying the tag name that contains the text. Note that this is not the same as using a separate model for each tag name, as the fake character is "forgotten" after a few characters [2].

Having to store all those different models 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. As finding the optimal merging seems to be a hard combinatorial problem, we design a heuristic to obtain a reasonably good merging from an initially separate set of models, one per tag.

The *SCM* approach can be coupled with an adaptive model as well. In this case the only goal is to use the contexts to improve compression, as the technique cannot be used for compressed text databases. We implement this idea by using a different PPM model for each different structural element (XML tag name). We call the resulting compressor *SCMPPM*.

Our experimental results show that *SCMHuff* compresses 2–4% better than other methods (including plain word-based Huffman without structure, the best alternative semistatic method we know of). Therefore our method is the best among those that permit random access to the text. From a pure compression point of view, *SCMPPM* compresses 2–5% more than any other compressor we tried, including the base PPM and structure-aware compressors like *XMill* and *XMLPPM*, soon after the collection exceeds 5 megabytes. As explained, those space savings translate into I/O time reductions.

2 Related Work

2.1 Standard Text Compression

In general, classic text compression methods (Bell et al., 1990; Moffat and Turpin, 2002) do not take into account the structure of the documents they compress. Our aim is not to cover the whole area but just to focus on three families of compressors that are relevant for this paper.

Text compression is usually divided into two kinds. *Statistical* compression is based on estimating source symbol probabilities and assigning them codes according to the probabilities. *Dictionary* methods consist in replacing text substrings by identifiers, so as to exploit repetitions in the text. Statistical compression is conceptually divided into two tasks. *Modeling* regards the text as a sequence of *source symbols* and assigns probabilities to them, possibly depending on their surrounding symbols. *Zero-order* modeling assigns probabilities to the symbols regarding them in isolated form, while *k-th order* modeling assigns their probabilities as a function of the k symbols preceding them. *Coding* assigns to each source symbol a sequence of *target symbols* (its *code*), based on the probabilities given by the model. The output of the compressor is the sequence of target symbols given by the coder. Compression is *semi-static* when a single model is obtained for the whole text before coding starts, so that all the occurrences of the same source symbol (in the same context) are assigned the same code. *Adaptive* compression interleaves the modeling and coding tasks, so that the model is built and updated as coding progresses. In adaptive compression, each new symbol

[2] From a personal communication with James Cheney.

is encoded using the current model and therefore different occurrences of the same source symbol may be assigned different codes.

Semi-static compression requires two passes over the text, as well as storing the model together with the compressed file. On the other hand, adaptive compression cannot start decompression at arbitrary file positions, because all the previous text must be processed so as to learn the model that permits decompressing the text that follows.

Lempel-Ziv. Lempel-Ziv compression is a dictionary method based on replacing text substrings by previous occurrences thereof. The two most famous algorithms of this family are called LZ77 (Ziv and Lempel, 1977) and LZ78 (Ziv and Lempel, 1978). A well-known variant of the latter is called LZW (Welch, 1984).

LZ77 maintains a window of the last N processed characters. In each step, it reads the longest possible string s from the input that also appears in the window. If s is of length ℓ, it is followed by character a in the input, and it was found at window position p (counting right to left), then the compressor outputs the triple (p, ℓ, a). Thus input string sa is replaced by the triple, and compression is obtained if the triple needs less bits than the string itself. Once this is done, the window is shifted forward by $\ell+1$ positions and the algorithm resumes the scanning just past string sa.

In principle the use of a longer window improves compression because it makes more likely to find longer strings for replacement. However, the representation of position p requires $\log_2 N$ bits, which worsens as N grows. In practice the most convenient window size is not very long (for example, 64 kilobytes). This considers not only the achievable compression but also the time and space cost of searching the window for strings.

Decompression of LZ77 compressed files is extremely fast and simple. The compressed text is basically a sequence of triples (p, ℓ, a). For each such triple we must copy ℓ characters starting p positions behind the current output position, and then output a. Well-known representatives of LZ77 compression are Info-ZIP's *zip* and GNU's *gzip*.

Other variants, such as LZ78 and LZW, restrict somehow which previous strings can be referenced. This is done for efficiency reasons of different types, for example to improve compression time or to improve the compression ratio. A well-known representative of LZW is Unix's *compress*.

The Lempel-Ziv family is the most popular to compress text because it combines compression ratios around 35% on plain English text with fast compression and decompression. However, Lempel-Ziv compressed text cannot be decompressed at random positions, because one must process all the text from the beginning in order to learn the window that is used to decompress the desired portion.

Huffman. Huffman coding (Huffman, 1952) is designed for statistical compression. It assigns a variable-length code to each source symbol, trying to give shorter codes to more probable symbols. Huffman algorithm guarantees that the code assignment minimizes the length of the compressed file under the probabilities given by the model.

A common usage of Huffman coding is to couple it with semi-static zero-order modeling, taking text characters as the source symbols and bits as the target symbols. That is, on a first pass over the text, character frequencies are collected, then Huffman codes (variable-length bit sequences) are assigned to the characters, and finally each character occurrence is replaced by its codeword in a second pass over the text. This combination, that we call "Huffman compression" for shortness,

reaches the zero-order entropy of the text up to one extra bit per symbol. Being semi-static, Huffman compression permits easy decompression of the text starting at any position.

Huffman compression is not very popular on natural language text because it achieves poor compression ratios compared to other techniques. However, the situation changes drastically when one uses the text *words*, rather than the characters, as the source symbols (Moffat, 1989). The distribution of words is much more skewed than that of symbols, and this permits obtaining much better compression ratios than character-based Huffman compressors. On English text, character-based Huffman obtains around 60% compression ratio, while word-based Huffman is around 25% (Ziviani et al., 2000). Actually, similar compression ratios can be obtained by using Lempel-Ziv on words (Bentley et al., 1986; Horspool and Cormack, 1992; Dvorský et al., 1999).

However, the text in natural language is not only made up of words. There are also punctuation, separator, and other special characters. The sequence of characters between every pair of consecutive words is called a *separator*. Separators must also be considered to be symbols of the source alphabet. In (Moffat, 1989) they use the so-called *separate alphabets model*, where words and separators are modeled separately. As every word is followed by a separator and vice-versa, once it is known whether the text starts with a word or a separator, no further information is necessary to decode the stream of codes from the two different alphabets.

Word-based Huffman compression has other advantages. Not only the text can be compressed and decompressed efficiently, as a whole or in parts, but it is also possible to search it without decompressing, *faster* than when searching the uncompressed text (Ziviani et al., 2000). Also, this type of compression integrates very well with information retrieval systems, because the source alphabet is equivalent to the vocabulary of the inverted index (Witten et al., 1999; Navarro et al., 2000; Moffat and Wan, 2001). One of the best known systems in the public domain relying on word-based Huffman is the MG system (Witten et al., 1999).

K-th order models. These models assign a probability to each source symbol as a function of the *context* of k source symbols that precede it. They are used to build very effective compressors such as Prediction by Partial Matching (PPM) and those based on the Burrows-Wheeler Transform (BWT).

PPM (Cleary and Witten, 1984) is a statistical compressor that models the character frequencies according to the context given by the k characters preceding it in the text, and codes the characters according to those frequencies using arithmetic coding (Witten et al., 1987). PPM is adaptive, so the statistics are updated as compression progresses. The larger k, the more accurate is the statistical model and the better the compression, but more memory and time is necessary to compress and decompress.

More precisely, PPM uses $k+1$ models, of order 0 to k, in parallel. It usually compresses using the k-th order model, unless the character to compress has never been seen in that model. In this cases it switches to a lower-order model until the character is found.

The BWT (Burrows and Wheeler, 1994) is a reversible permutation of the text that puts together characters having the same k-th order context (for any k). Local optimization over the permuted text obtain results similar to k-th order compression (for example, by applying move-to-front followed by Huffman or arithmetic coding).

PPM and BWT usually achieve better compression ratios than other families (around 20% on English text), yet they are much slower to compress and decompress, and cannot decompress arbitrary portions of the text collection. Well known representatives of this family are Seward's

bzip2, based on the BWT, and Shkarin/Cheney's *ppmdi* (Shkarin, 2002) and Bloom/Tarhio's *ppmz*, two PPM-based techniques.

2.2 Structured Text Compression

There exist a few approaches specifically designed to compress structured text, taking advantage of its structure.

XMill (Liefke and Suciu, 2000). Developed at AT&T Labs, *XMill* is an XML-specific compressor designed to exchange and store XML documents. Its compression approach is not intended for directly supporting querying or updating the compressed documents. *XMill* is based on the *zlib* library, which combines Lempel-Ziv compression with a variant of Huffman. Its main idea is to split the file into three components: elements and attributes, text, and structure. Each component is compressed separately. Another Lempel-Ziv based compressor, cutting the structure at some depth and using plain Lempel-Ziv compression for the subtrees, is commercial *XMLZip* (`http://www.xmls.com`).

XMLPPM (Cheney, 2001). This is a PPM-like compressor, where the context is given by the path from the root to the tree node that contains the current text. *XMLPPM* is an adaptive compressor that does not permit random access to individual documents. The idea is an evolution over *XMill*, as different compressors are used for each component, and the XML hierarchy information is used to improve compression.

XCQ (Levene and Wood, 2002) and Exalt (Toman, 2004). These are compression methods based on separating structure from data, and then using grammar-based compression for the structure. In *XCQ*, the tree shape is compressed using the DTD information, while the text is compressed using a standard Lempel-Ziv software such as *gzip*. In *Exalt*, both elements are compressed using grammar-based methods. In particular, zero-order prediction depending on the structural context, plus arithmetic coding, is used for the tags. Other grammar-based techniques can be found in (Tarhio, 2001), as well as in *XML-Xpress*, a commercial software (`http://www.ictcompress.com`) that compresses well when the DTD is known.

XGrind (Tolani and Haritsa, 2002). This compressor is interesting because it 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. Structure tags are represented in numeric form, while the text is compressed using character-oriented Huffman. A similar idea is explored in *XMillau* (Girardot and Sundaresan, 2000).

LZCS (Adiego et al., 2004; Adiego et al., 2006) This compressor uses an idea similar to LZ77, restricted to replacing whole subtrees. This permits the compressed text being accessed, searched, and navigated without decompressing it. Yet, the technique is oriented to highly structured documents (such as e-commerce exchanges, for example) and it does not perform well on general semistructured data.

3 Structural Contexts with a Semistatic Model

Our first contribution is a structure-aware compressor that permits random access and direct searching on the compressed text. For this sake, we build on a semistatic 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, but 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 the Introduction, 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 semistatic models, as we have to store several dictionaries instead of just one. In this section each tag uses a separate dictionary. Section 4 considers the way to group tags under a single dictionary.

3.1 Compressing the Text

We compress the text with word-based Huffman (Huffman, 1952; Bentley et al., 1986). 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 mark the first part of a container element, '<...>'; and *end-tags*, which mark the end of 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.

Figure 1 shows the general SCM scheme, and Algorithm 1 shows the generic pseudocode for a modeling, coding, or decoding pass over the text. For compression we make a modeling and a

coding pass 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 encoded according to the model obtained. Decompression reads the model and decodes the text in a single pass.

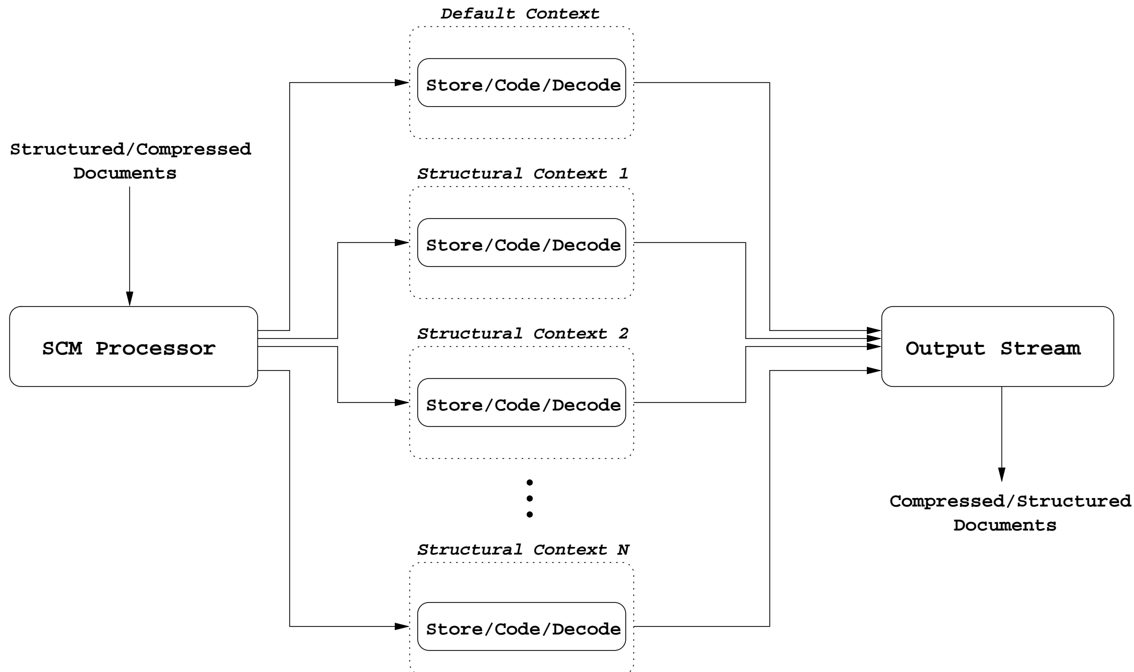

Figure 1: General SCM scheme.

Algorithm 1 (Dictionary Switching)

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

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 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.

4 Merging Dictionaries

Up to now we have assumed that each different tag 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 them 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 under dictionaries. A key part of the method is, given two dictionaries, determine whether or not their union will produce a shorter overall text. The exact way to do this is to compute the union of the vocabularies, sum the frequencies of common words, and run Huffman algorithm to obtain the exact length of the text when dictionaries are joined. As this is costly, we estimate the size of the Huffman-compressed text without running Huffman algorithm. For this sake, we use the fact that Huffman compression performance is very close to the zero-order entropy of the text. The following definitions give the conceptual framework for our final algorithm.

4.1 Entropy Estimation

Assume we have a text T of n terms partitioned into N texts $T_1 \ldots T_N$, so that T_d has n_d terms. The idea is that each T_d corresponds to the text under a given tag and will be encoded using its own dictionary. We define the raw frequency relative to a given dictionary d.

Definition 1 (Raw frequency) *The raw frequency $f_d(i)$ of vocabulary term i is given by*

$$f_d(i) = \frac{\mathrm{occ}_d(i)}{n_d},$$

where $\mathrm{occ}_d(i)$ is the number of occurrences of vocabulary term i in T_d. The raw frequency is also called occurrence probability of term i.

Definition 2 (Zero-order entropy estimation) *Let V_d be the number of vocabulary terms for text T_d. The zero-order entropy $\mathcal{H}_d$ of text T_d is estimated as*

$$\mathcal{H}_d = \sum_{i=1}^{V_d} f_d(i) \log_2 \frac{1}{f_d(i)}. \qquad (1)$$

We can now define the overall entropy of a text T partitioned into multiple texts. This is a lower bound to the average codeword length obtained by applying any zero-order compressor to the text under each dictionary.

Definition 3 (Zero-order entropy estimation with multiple dictionaries) *The zero-order entropy $\mathcal{H}$ for text $T = \{T_1, \ldots, T_N\}$ is computed as the weighted average of zero-order entropies $\mathcal{H}_d$ contributed by each text T_d:*

$$\mathcal{H} = \frac{\sum_{d=1}^{N} n_d \, \mathcal{H}_d}{n}. \qquad (2)$$

A Huffman coding over text T_d will assign $\ell_d(i)$ bits to symbol i, so that the average code length, $L_d = \sum_i f_d(i)\ \ell_d(i)$, is minimized. The Noiseless Coding Theorem (Shannon, 1948) establishes that $\mathcal{H}_d \leq L_d < \mathcal{H}_d + 1$ (in practice L is much closer to $\mathcal{H}$ than to $\mathcal{H}+1$).

If we apply Huffman encoding to each T_d, the resulting compressed file length is $\sum_{i=1}^{N} n_d L_d$ bits, and therefore the average codeword length is

$$L = \frac{\sum_{i=1}^{N} n_d L_d}{n}.$$

As $L_d < \mathcal{H}_d + 1$ for all d, $L < \sum_{i=1}^{N} \frac{n_d}{n}(\mathcal{H}_d + 1) = \mathcal{H} + 1$. Therefore, we have the bounds $\mathcal{H} \leq L < \mathcal{H} + 1$ with the partitioned text as well. For this reason, we will use $\mathcal{H}$ as an estimation of L.

Definition 4 (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 (considering vocabulary and text) is given by*

$$\mathcal{T}_d = \mathcal{V}_d + n_d \mathcal{H}_d \quad (3)$$

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 $\mathcal{T}_i + \mathcal{T}_j > \mathcal{T}_{i \cup j}$.

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.

The last definition gives the tool we use for the optimization algorithm.

Definition 5 (Estimated saving of a merge) *Let $\mathcal{A}_{i,j}$ be the estimated saving of merging dictionaries i and j. Then*

$$\mathcal{A}_{i,j} = \mathcal{T}_i + \mathcal{T}_j - \mathcal{T}_{i \cup j}.$$

4.2 Optimization Algorithm

Our optimization algorithm works as follows. We start with one separate dictionary per tag, plus the default dictionary (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 seems to be a hard combinatorial problem, so 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. We then compute the savings $\mathcal{A}_{i,j}$ for all $i < j$. Then, we merge the pair of dictionaries i and j that maximizes $\mathcal{A}_{i,j}$, if this is positive. If it is, we erase i and j and introduce $i \cup j$ in the set, computing savings $\mathcal{A}_{s,k}$ for all s versus the new element k that corresponds to $i \cup j$. This process is repeated until all the $\mathcal{A}_{i,j}$ values are negative.

Algorithm 2 depicts the process at a high level. H is a max-priority queue storing triples $(i, j, \mathcal{A}_{i,j})$ ordered by $\mathcal{A}_{i,j}$ (note that $\mathcal{A}_{i,j}$ is not explicitly stored). Each occurrence of $\mathcal{T}_{i \cup j}$ implies the $O(V)$ time computation of the union, but the result is not stored unless we explicitly indicate that the dictionary is created. The algorithm costs $O(N^2(V + \log N))$ time and $O(N(N + V))$ space, when there are N dictionaries and the vocabulary sizes are V.

Algorithm 2 (Merging Dictionaries)

$H \leftarrow \emptyset$
for $1 \leq i \leq N$ **do** *create dictionary* i *and compute* $\mathcal{T}_i$
for $1 \leq i < j \leq N$ **do** $H \leftarrow H \cup (i, j, \mathcal{T}_i + \mathcal{T}_j - \mathcal{T}_{i \cup j})$
$k \leftarrow N + 1$
$(i, j, savings) \leftarrow extract_max\ (H)$
while $(savings \geq 0)$ **do**
    $\mathcal{T}_k \leftarrow \mathcal{T}_i + \mathcal{T}_j - savings$ *(that is,* $\mathcal{T}_{i \cup j}$*)*
    create new dictionary k *as the union of dictionaries* i *and* j
    remove dictionaries i *and* j
    for $1 \leq s \leq N$ **do**
        if $(s \neq i)$ **then** $H \leftarrow H - (\min(i, s), \max(i, s), *)$
        if $(s \neq j)$ **then** $H \leftarrow H - (\min(j, s), \max(j, s), *)$
        $H \leftarrow H \cup (s, k, \mathcal{T}_s + \mathcal{T}_k - \mathcal{T}_{s \cup k})$
    $k \leftarrow k + 1$
    $(i, j, savings) \leftarrow extract_max\ (H)$

4.3 Example

Figure 2 shows an example XML file, from the personal repository of papers of John Smith. Let us first consider compressing all the words using a single model. The text has $V_{\texttt{all}} = 18$ different words and $n_{\texttt{all}} = 39$ total words. The words and their frequencies follow (excluding stopwords and separators).

```
John(4), Smith(4), Susan(1), Tate(1), Ashton(2), Albers(2), Jacob(1), Ziv(2),
Compression(4), Algorithms(4), Applications(3), Lempel(1), 2001(1), 2003(1),
2005(2), Journal(3), ACM(2), Communications(1)
```

According to Eq. (1) the entropy of this text is $\mathcal{H}_{\texttt{all}} = 3.965$ bits per word. To account for the cost of encoding the dictionary, let us assume that we need 8 bits per different dictionary word, thus $\mathcal{V}_{\texttt{all}} = 8 \cdot V_{\texttt{all}}$. Following Eq. (3), the overall number of bits to represent the text is $\mathcal{T}_{\texttt{all}} = \mathcal{V}_{\texttt{all}} + n_{\texttt{all}}\mathcal{H}_{\texttt{all}} = 8 \cdot 18 + 39 \cdot 3.965 = 299$ bits.

Let us now separate the tags and compute the raw frequencies of the words within each tag.

```
<author>: John(4), Smith(4), Susan(1), Tate(1), Ashton(2), Albers(2), Jacob(1),
          Ziv (1)
<title>: Compression(3), Algorithms(3), Applications(3), Ziv(1), Lempel(1)
<year>: 2001(1), 2003(1), 2005(2)
<journal>: Journal(3), ACM(2), Algorithms(1), Compression(1), Communications(1)
```

The entropies for each tag are computed using Eqs. (1) and (3).

Tag	Entropy (bits/word)	Number of words	Different words	Total bits (Eq. 3)
author	$\mathcal{H}_{\texttt{author}} = 2.750$	$n_{\texttt{author}} = 16$	$V_{\texttt{author}} = 8$	$\mathcal{T}_{\texttt{author}} = 108$
title	$\mathcal{H}_{\texttt{title}} = 2.163$	$n_{\texttt{title}} = 11$	$V_{\texttt{title}} = 5$	$\mathcal{T}_{\texttt{title}} = 64$
year	$\mathcal{H}_{\texttt{year}} = 1.500$	$n_{\texttt{year}} = 4$	$V_{\texttt{year}} = 3$	$\mathcal{T}_{\texttt{year}} = 30$
journal	$\mathcal{H}_{\texttt{journal}} = 2.156$	$n_{\texttt{journal}} = 8$	$V_{\texttt{journal}} = 5$	$\mathcal{T}_{\texttt{journal}} = 58$

```
<paper>
   <author>John Smith</author>
   <author>Susan Tate</author>
   <title>Compression Algorithms and Applications</title>
   <year>2001</year>
   <journal>Journal of the ACM</journal>
</paper>

<paper>
   <author>John Smith</author>
   <title>Compression has no Applications</title>
   <year>2003</year>
   <journal>Journal of Algorithms</journal>
</paper>

<paper>
   <author>Ashton Albers</author>
   <author>John Smith</author>
   <title>Algorithms for Compression</title>
   <year>2005</year>
   <journal>Journal of Compression</journal>
</paper>

<paper>
   <author>Ashton Albers</author>
   <author>John Smith</author>
   <author>Jacob Ziv</author>
   <title>Ziv-Lempel Algorithms and Applications</title>
   <year>2005</year>
   <journal>Communications of the ACM</journal>
</paper>
```

Figure 2: Example XML file.

If we consider the tags in separate form we obtain, using Eq. (2), a total entropy of $\mathcal{H} = 2.334$ bits per word. This is much better (41% less) than if we consider all tags together. This is not surprising, as the entropy is always lower when we split a text. However, we must also consider the cost of maintaining separate vocabularies for each tag. In this case, even if we consider the vocabulary storage, we obtain a total of 260 bits when we sum the values in the last column of the previous table. This is still lower (13% less) than the 299 bits used when the tags are not separated.

Finally, let us consider the possibility of merging tags `title` and `journal`, as they share some words. The raw frequencies are now as follows:

```
<author>: John(4), Smith(4), Susan(1), Tate(1), Ashton(2), Albers(2), Jacob(1),
          Ziv(1)
<year>: 2001(1), 2003(1), 2005(2)
<title∪journal>: Compression(4), Algorithms(4), Applications(3), Ziv(1),
                 Lempel(1), Journal(3), ACM(2), Communications(1)
```

Now $n_{\texttt{title}\cup\texttt{journal}} = 19$, $V_{\texttt{title}\cup\texttt{journal}} = 8$, and $\mathcal{H}_{\texttt{title}\cup\texttt{journal}} = 2.800$. Thus, $\mathcal{T}_{\texttt{title}\cup\texttt{journal}} = 118$ bits. Added to the previous sizes obtained for `author` and `year` we get 256 bits. This is better than both previous extremes and exemplifies the benefits of merging, even on a small example.

5 Random Access and Searching

One of the most important characteristics of a semistatic method like word-based Huffman is that it permits local decompression of the text from random positions, as well as efficient direct search of the compressed text (Ziviani et al., 2000). In this section we show how those tasks can be carried out on *SCMHuff*.

5.1 Local Decompression

The main obstacle to start decompressing from a given position in *SCMHuff* compressed text is that we need knowledge of the content of the stack of dictionaries at that point of the compression process (recall Algorithm 1). Which is equivalent, we need to know the position of the text to access in the structure tree of the document collection, so that the stack of dictionaries corresponds to the root-to-leaf path that leads to the text position we wish to start decompression at.

In some browsing schemes, the system is already aware of the place in the tree that corresponds to the position where decompression must start, so let us focus in the case where there is no such knowledge.

We propose, essentially, to maintain an explicit structure tree. Each tree node corresponds to a tag or to a maximal space between tags, and it indicates its tag type and the position in the compressed document where the node starts. This way, once we wish to start decompression at some position of the compressed text, we binary search the children of the tree root until finding the one containing the position. If the node corresponds to a space between tags, then we reached a tree leaf and can start decompression, otherwise we push the dictionary corresponding to the tag in a stack and repeat the process in the child node, continuing recursively until we reach a leaf. When we start decompression, we can use Algorithm 1 from that point, as we have the correct stack for that point of the compressed text. The process of rebuilding the stack for position *pos* is depicted in Algorithm 3 and costs $O(h \log a)$, where h is the structure height and a is the maximum

arity. In general h is taken as a constant, while a, especially at the first levels, can be $O(n)$, so in practice the cost is $O(\log n)$.

Algorithm 3 (Rebuilding the Stack)

$t \leftarrow$ *tree root*
$S \leftarrow$ *empty stack*
while *t is not a leaf* **do**
    $c \leftarrow$ *binary search the rightmost child of t starting before pos*
    if *c corresponds to a tag*
        then *push the dictionary of c in S*
    $t \leftarrow c$
start decoding with stack S at position pos

An obvious question is how much space do we require for the structure tree. A first choice is not to represent it, but to reconstruct it from a linear pass over the database at system startup time. This might be unreasonable depending on the application, so let us consider the alternative of explicitly storing the tree.

In the text collections of Section 7, we found a mean of 4,000 start tags per megabyte, and at most 10 different tags per collection. A tree can be represented using 2 bits per node, as a sequence of opening and closing parentheses. Then the tree can be rebuilt in memory from a linear pass over the sequence. We need furthermore 4 bits per node to represent the sequence of tag identifiers (in preorder, so they can be read as the tree is reconstructed from the sequence of parentheses). Finally, we need to represent the sequence of lengths of the leaves in the compressed text, so as to reconstruct the initial text position of every tree node. We estimated that 9 bits per leaf are sufficient if we use a variable length coding (Witten et al., 1999). Overall, we have an overhead of 15 bits per tree node in our text collections, which translates into incrementing our compression ratios by 0.7 percentual points (that is, we should add 0.7 to the compression ratios achieved by *SCMHuff* to access it at random). On the other hand, in decompressed traversable form this tree requires 3.4% extra space, so for example we can manage a 1 gigabyte text collection using just 35 megabytes of main memory for the structure tree.

We remark that having the structure tree might be anyway a requirement of the structured text retrieval system, and this also permits new search and traversal capabilities for such a system.

5.2 Searching

With plain Huffman compression, it is possible to compress the pattern and search the text for it, as the code of the pattern is the same across all the compressed text. This is a bit more complicated in *SCMHuff*. The reason is that the pattern is coded differently inside each dictionary.

The most general way the search can be carried out is as follows (Turpin and Moffat, 1997; Moura et al., 2000). We first find and mark all the vocabulary words that match the search pattern (it can be just one if the pattern is a simple string). Then we traverse the text, (essentially) scanning the target symbols one by one, and traversing the Huffman tree accordingly. Each time we arrive at a leaf, we check whether the leaf is marked. If so, we report an occurrence of the pattern. In any case, we return to the tree root and resume the scanning.

In our case we must preprocess the Huffman tree of each dictionary. In addition to the pattern, we are interested in marking the Huffman tree leaves corresponding to new start-tags and to the end-tag corresponding to the current dictionary. This way we keep track not only of the occurrences of the search pattern in the current dictionary, but also of the dictionary switchings that occur as we traverse the compressed text. When we find a new start-tag, we push the current dictionary in a stack and start using the Huffman tree of the new context. When we find the end-tag, we pop and restore the previous dictionary.

Boyer-Moore type searching is also possible on Huffman-compressed text (Moura et al., 2000), especially when the target symbols are bytes rather than bits. In this case, instead of a single-pattern search for the compressed pattern, we enable a multipattern search for the compressed pattern and also the codewords of start-tags and end-tags, so as to switch models when the context changes. Alternatively, we can search for all the pattern codewords that arise under the different dictionaries, and upon a possible occurrence, check the structure tree to find out whether the occurrence we have found corresponds to the current context.

6 SCMPPM

If we disregard any attempt of direct access or searching, still the *SCM* concept is useful in terms of boosting compression. To prove that this is the case and to show that *SCM* is general enough to accommodate other compression methods, we combine it with PPM compression. The essential idea of *SCMPPM* is that each different structural element name will have its own PPM model to compress the text that lies under it.

A PPM coder usually works with characters as symbols, and we use it in this way. Still, we parse the text as a sequence of tokens so as to recognize tags. As PPM is adaptive, it does not need to store the model in the compressed file. As a result, there is no penalty for maintaining multiple models other than more compression time and space. Hence there is no point in merging similar models.

At the begining of the process, the default model is used to predict the symbol probabilities. When a start-structure tag appears, we push the current model in a stack and switch to the appropriate model. When an end-structure tag is found we return to the previous model stored in the stack. Both start-structure and end-structure tags are coded using the current model and then we switch models. The encoding and decoding processes use the same model switching technique.

Algorithm 4 shows the model switching used for encoding and decoding.

Algorithm 4 (Model Switching)

```
current_model ← default_model
while there are more token do
        token ← get_token()
        encode/decode(each_symbol(token), current_model)
        if (token is a start-structure tag)
           then push(current_model)
                current_model ← model(token)
           else if (token is an end-structure tag)
                then current_model ← pop()
```

7 Experimental Evaluation

In this section we empirically evaluate *SCMHuff* and *SCMPPM*. For the former we use word-oriented Huffman coding, compressing the dictionaries using arithmetic character-based adaptive coding. For the latter we use *ppmdi* coders for each model. Both prototypes are compared against state-of-art compressors. We remind that *SCMHuff* permits random access to individual documents, while *SCMPPM* does not.

The tests were carried out on the SuSE Linux 9.1 operating system, running on a computer with a Pentium IV processor at 1.2 GHz and 384 megabytes (MB) of RAM. We used `g++` compiler with full optimization. For the experiments we selected different size subcollections (more precisely, prefixes) of WSJ, ZIFF and AP, from TREC-3 [3] (Harman, 1995). 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 MB is approximate.

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

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).

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. This structuring is very similar to INEX[4] structured collections, yet these have a few additional formatting tags.

7.1 Evaluation of the Merging Algorithm

In this section we focus on *SCMHuff*, and in particular in its algorithm to merge models (Section 4).

Let us first consider speed. The average speed to compress all collections is around 265 Kbytes/sec (variance is low and not dependent on the collection type). This value includes the time needed to build models, 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 (Heaps, 1978), typically around $O(\sqrt{n})$. Although merging time also depends 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.

[3] `http://trec.nist.gov/`

[4] INitiative for the Evaluation of XML Retrieval: `http://qmir.dcs.qmul.ac.uk/INEX`

We focus on the effectiveness of the algorithm now. Table 2 shows the number of dictionaries merged. Column “Initial” tells how many dictionaries are there 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 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. Also, the larger the collection, the less the impact of storing more vocabularies (Heaps, 1978), and hence the fewer merges will occur.

Aprox.	WSJ		ZIFF		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 2: Number of dictionaries used.

In Figure 3 we can see a comparison, for WSJ (in the other collections we have obtained similar results), of the compression performance with and without merging dictionaries. It can be seen that compression ratios improve for larger collections, as the impact of the vocabulary is reduced. Merging plays a sort of smoothing role. When the text is small and the overhead of storing vocabularies is significant, the method merges dictionaries more aggressively, obtaining almost 2% of additional compression on texts of 1 MB. As the text collection grows, the cost of storing (one or more) vocabularies becomes less significant, and thus the method merges fewer dictionaries. For large enough texts, merging would not occur and the method would store one different vocabulary per text tag. In column SAM we show the result of the basic separate alphabets model (that is, as if we merged all the dictionaries except that for separators). Note that, although SCM without merging can be worse then SAM for small collections, merging always finds an optimum point between the two extremes. We also included column MG, in principle similar to SAM, to show the differences due to different implementations of the same idea.

Another question is how good is our heuristic to choose which dictionaries to merge. On one hand, we use an entropy-based method to predict the size of the merged dictionaries from the vocabulary distributions. This is very accurate: our lower-bound prediction is usually 99%–99.5% of the final value.

To globally evaluate the heuristic, we have compared it against an exhaustive search for all the merging possibilities. We have taken 1 Mb from WSJ and limited the tags under consideration to reduced subsets of size 7 and 8. We have considered all the 666 and 2,284 merging possibilities, respectively, and have compressed the text using each of them. In both cases, our heuristic obtained

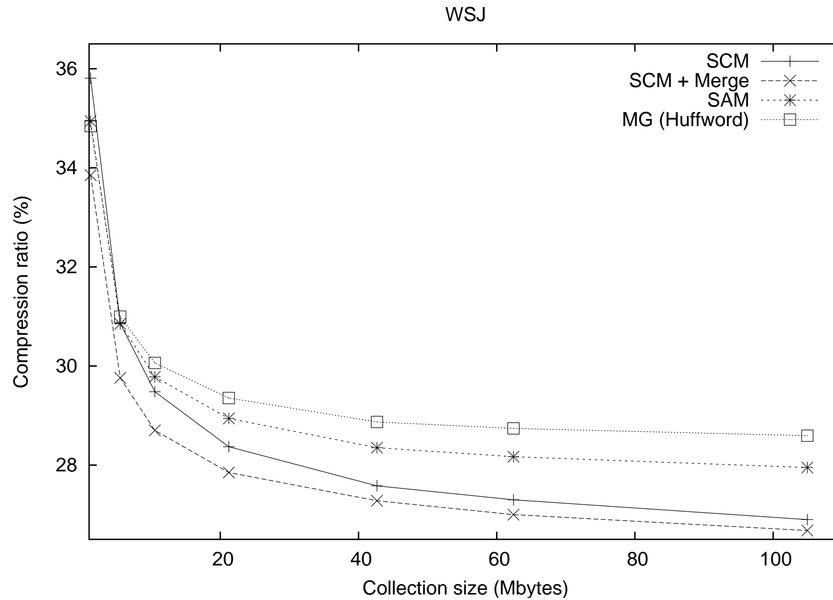

Figure 3: Compression ratios with and without merging, for WSJ.

the result that achieved best compression. Tables 3 and 4 show some detailed results.

Compression	Merged Tags
39.8091%	HL+LP+TEXT
39.8206%	DEFAULT+DATE and HL+LP+TEXT
39.8244%	DEFAULT+DOC and HL+LP+TEXT
39.8285%	DEFAULT+CO and HL+LP+TEXT
39.8452%	DEFAULT+HL+LP+TEXT
39.8582%	DOC+DATE and HL+LP+TEXT

Table 3: The 6 top merging alternatives and their compression ratios, when restricting the search to 1 Mb of collection WSJ and the 7 tags <DOC>, <CO>, <DATE>, <HL>, <LP>, and <TEXT>, apart from the default context (DEFAULT). Our heuristic chose the best merging.

We could produce a case where the heuristic failed to find the best combination, over a 10 Kb file with little structure. Yet, the difference with the best combination was just 0.01%.

7.2 Comparison against Classical Compressors

We now compare *SCMHuff* and *SCMPPM* against several classical compression systems: (1) *GNU gzip* v.1.3.5[5], which uses LZ77 plus a variant of Huffman algorithm (we also tried *zip* with almost identical results but slower processing); (2) *UNIX's compress* v.4.2.4, which implements LZW algorithm; (3)*bzip2* v.1.0.2[6], which uses the Burrows-Wheeler block sorting text compression algo-

[5] http://www.gnu.org

[6] http://www.bzip.org

Compression	Merged Tags
39.7517%	DOCNO+DOCID and HL+LP+TEXT
39.7633%	DEFAULT+DOCNO+DOCID and HL+LP+TEXT
39.7877%	DOCNO+DOCID and DEFAULT+HL+LP+TEXT
39.7902%	DOCNO+DOCID+DATE and HL+LP+TEXT
39.8056%	DEFAULT+DOCNO+DOCID+DATE and HL+LP+TEXT
39.8263%	DOCNO+DOCID+DATE and DEFAULT+HL+LP+TEXT

Table 4: The 6 top merging alternatives and their compression ratios, when restricting the search to 1 Mb of collection WSJ and the 8 tags <DOCNO>, <DOCID>, <DATE>, <CO>, <HL>, <LP>, and <TEXT>, apart from the default context (DEFAULT). Our heuristic chose the best merging.

rithm, plus Huffman coding; (4)*ppmdi* (extracted from *XMLPPM v.0.98.2*[7]), a PPM compressor; and (5)*MG System* v1.2.1[8] (Witten et al., 1999). The MG system is a public domain information retrieval system software, versatile and of general purpose, which handles compressed text, indexes and images. For texts it uses a word-based Huffman variant called *Huffword* and implements the separate alphabets model. We used standard options for all and also maximum compress option whenever possible, except for *bzip2* where maximum compression is the default. In this case, we also tried minimum compression option. Compression ratios for each collection are shown in Figure 4.

Let us first consider the standard compressors. *Compress* obtained the worst compression ratios, not competitive in this experiment. It is followed by *gzip*, which has almost no difference between its default and maximum compression options. The next is *bzip2*, with a wide difference between best and worst compression performance. Finally, the best standard compressor is *ppmdi*, which achieves 22%–23% compression ratios.

Our *SCMPPM* compresses better than *ppmdi* as soon as the text collections exceed 1 to 5 MB size (depending on the collection). This shows that *ppmdi* method is actually boosted by separating the models according to the text structure.

Let us now focus on methods that permit direct access to the text. In our experiment, those are the word-based methods *MG* and *SCMHuff*. Both improve as the text grows, because the impact of storing the vocabulary decreases. At some point the compression ratios of *MG* stabilize around 28%. Our *SCMHuff* compresses uniformly better than *MG*, reaching a stable improvement of 1.5 to 2 percentual points. After subtracting from this improvement the 0.7 points to have direct access to *SCMHuff* compressed text (Section 5), we still have 2% to 4% better compression ratios compared to *MG*.

7.3 Comparison against Structure-Aware Methods

We now compare *SCMHuff* and *SCMPPM* against other compression systems that exploit text structure: *XMill* v.0.8 [9] and *XMLPPM* v.98.2 [10]. *XMill* (Liefke and Suciu, 2000) is an XML-specific compressor based on Ziv-Lempel and Huffman, able to handle the document structure. On

[7] `http://sourgeforge.net/projects/xmlppm`
[8] `http://www.cs.mu.oz.au/mg`
[9] `http://sourceforge.net/projects/xmill`
[10] `http://sourceforge.net/projects/xmlppm`

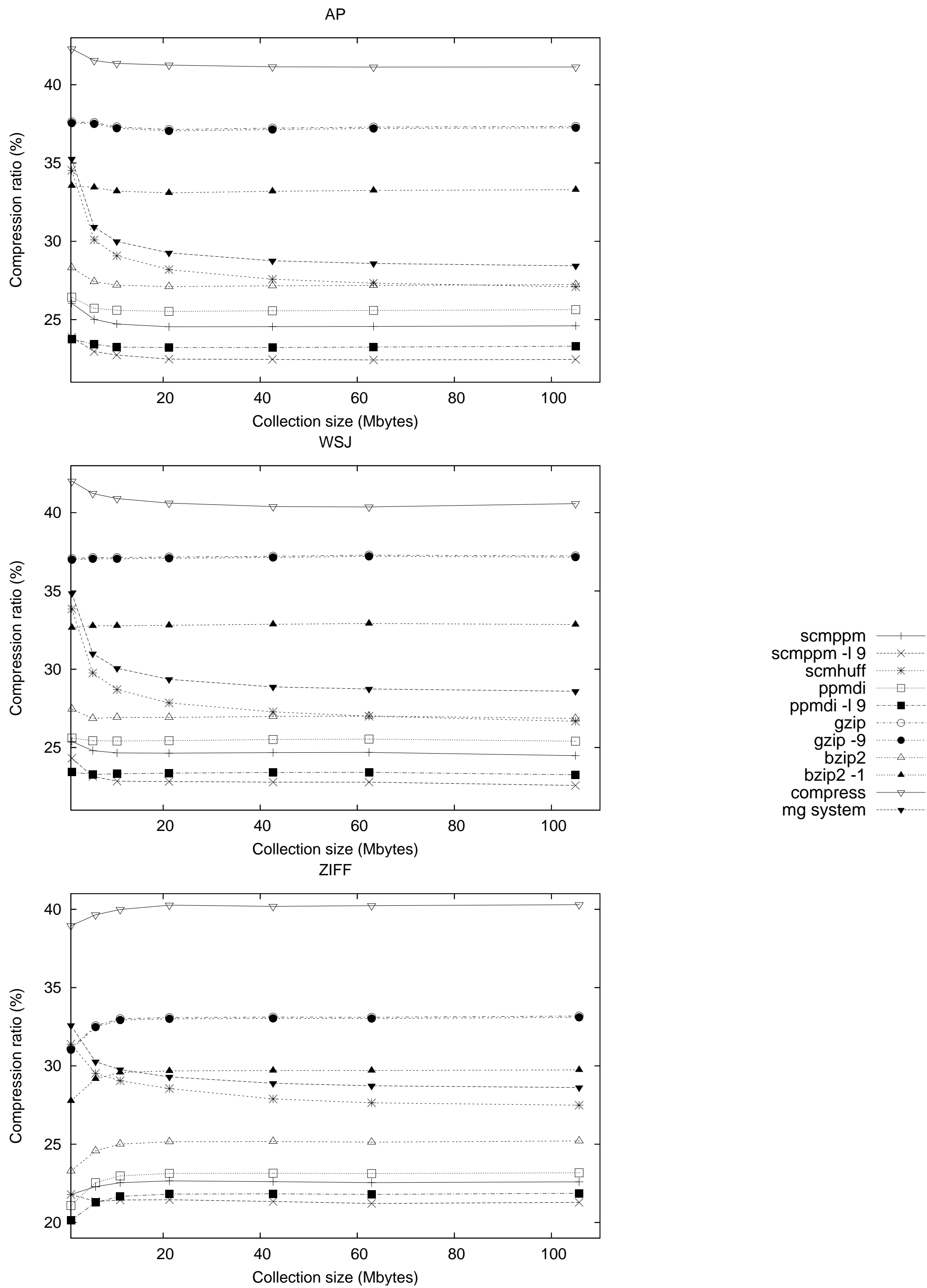

Figure 4: Comparison between *SCM* prototypes and classical compressors.

the other hand, XMLPPM (Cheney, 2001) is also specific of XML and based on adaptive PPM over the structural context.

XGrind [11] was excluded from this comparison because we could not make it work properly on our dataset. To be sure that this exclusion was not important, we altered our collection until producing 1 MB of text where *XGrind* finally worked. The resulting compression ratio was 57.28%, which is not competitive at all in this experiment. *XCQ* was also excluded because we could not find the code, yet results reported in (Lam et al., 2003) indicate that the compression ratios achieved are similar to those of *XMill*, which we will show to be not competitive in our experiments either. The same happens with *Exalt*, according to the results in (Toman, 2004).

Compression ratios are shown in Figure 5. We used standard options for all and also maximum compression option whenever possible.

XMill obtains an average compression ratio roughly constant in all cases because it uses *zlib* as its main compression machinery. The compression ratio obtained, 33%–35%, is not competitive in this experiment.

XMLPPM, on the other hand, is the most competitive alternative to *SCMPPM*. Although for 1 MB the compression ratios are similar, soon *SCMPPM* wins and for 100 MB its margin is around 3.3% in all cases. This shows that the idea of using the structural context to compress pays off.

SCMHuff is the only method permitting navigation and random access. It compresses better than *XMill* for collections of more than 1 MB, and for longer texts its margin over *XMill* grows up to 25%.

7.4 Speed and Memory Usage

Figure 6 shows the the overall average values for compression and decompression speed in relation to the compression ratio. We averaged values over all the collections because they did not significantly depend on the collection and variance was always low.

The fastest at decompression is *gzip* (based on LZ77), followed by *compress* (based on LZ78) and *XMill* (also based on LZ77). This is expected as this family of compressors is fast, especially at decompression. Much later come *bzip2* and *SCMHuff*. All the PPM-based compressors are slowest at decompression, as expected from this family. *MG* does not appear in the decompression plot, because it does not permit decompressing the whole collection. Decompression could be faster for *SCMHuff*, but this is currently a prototype that is not fully optimized.

For compression, the figures are similar except that *SCMHuff* is the slowest, possibly because of the need to parse the natural language text. *MG* is faster, as expected from a fine-tuned mature implementation, but it still takes the same time of the slow PPM compressors.

Figure 7 shows the overall average values for memory usage in relation to the compression ratio. Most compressors are tuned to use 10 Mb of memory to compress. The PPMDi variants use 20–30 Mb to achieve better compression ratio. Our *SCMPPM* uses more than 200 Mb, as it stores a PPMDi model per tag, yet it achieves improved compression ratio.

This raises the question of whether *SCMPPM* compresses more than *ppmdi* simply because it uses more memory. To show that memory can indeed be used better under the *SCM* approach, we have tuned *SCMPPM* to use the same memory given to *ppmdi -9*, 22 Mb. Table 5 shows that it is possible to compress better using the same memory under the *SCM* paradigm.

[11] `http://cvs.sourceforge.net/viewcvs.py/xmill/xmill/XGrind`

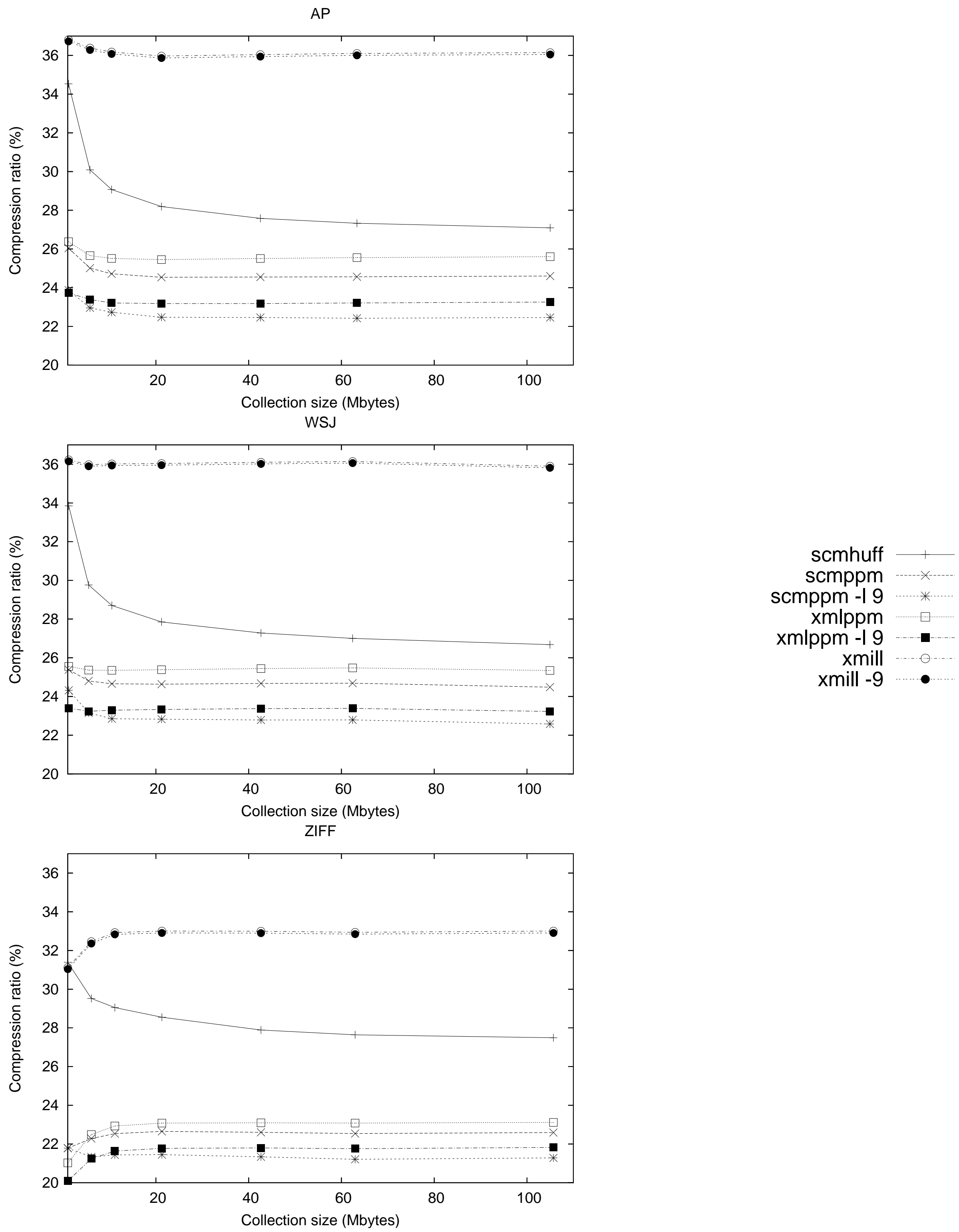

Figure 5: Comparison between *SCM* prototypes and other structure-aware methods.

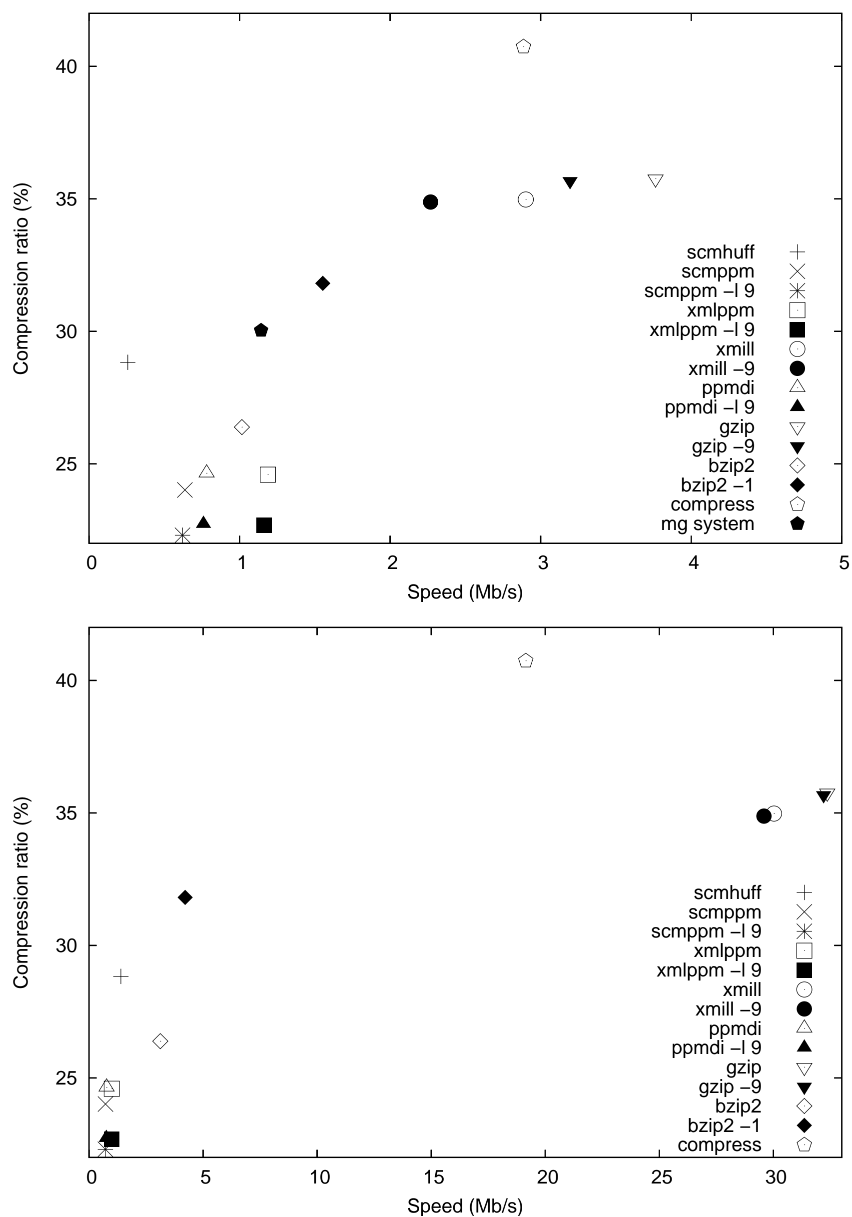

Figure 6: On the top, comparison between compression speed (MB/s) and compression ratios for all methods (overall average values are used). On the bottom, the same for decompression.

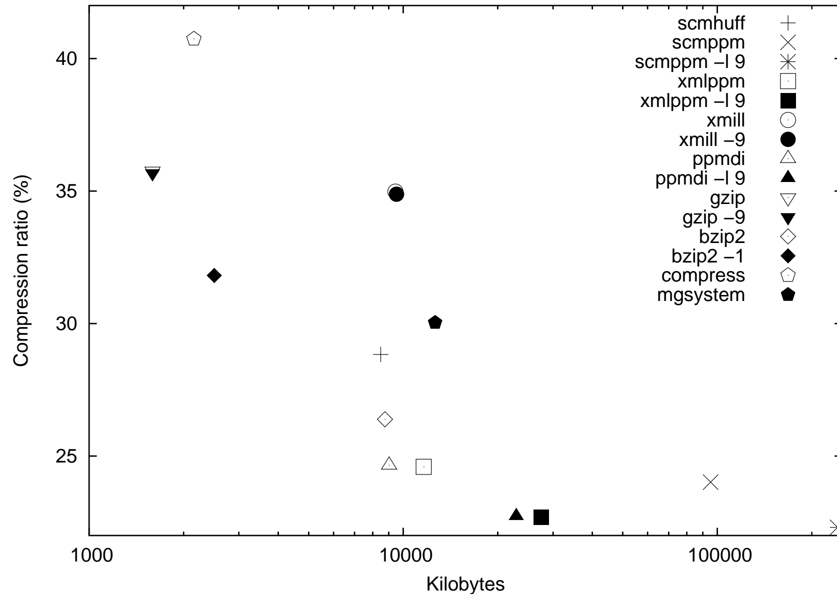

Figure 7: Comparison between memory usage (Kb) and compression ratios for all methods (overall average values are used).

Size	WSJ		AP		ZIFF	
	ppmdi -9	*scmppm -9*	*ppmdi -9*	*scmppm -9*	*ppmdi -9*	*scmppm -9*
1	23.426	23.356	23.755	23.724	20.126	20.034
5	23.281	23.394	23.424	23.376	21.280	21.265
10	23.324	23.305	23.244	23.236	21.665	21.660
20	23.365	23.280	23.217	23.166	21.809	21.801
40	23.407	23.346	23.217	23.199	21.830	21.795
60	23.416	23.354	23.247	23.210	21.794	21.755
100	23.259	23.176	23.300	23.279	21.860	21.817

Table 5: Comparison between compression ratios achieved by *ppmdi -9* and *scmppm -9* under similar memory requirements. File sizes are in Mb and approximate.

To achieve these results, however, we have tuned *SCMPPM*. The current prototype uses the same amount of memory for all the tags, which wastes a lot of memory on those tags that handle less text. A more refined implementation could increase the memory requirement as it processes more text, so that tags that handle little text would never require much memory. Moreover, we could use a two-pass technique which, based on the k-th order statistics, decided to merge models that yield small losses in compression due to the merge. As a proof of concept, we have manually tuned the amount of memory used by each tag and also merged the tags according to *SCMHuff* heuristic: `HL+LP+TEXT` and `DOCNO+DOCID` for WSJ, `HEAD+TEXT+NOTE` and `FIRST+SECOND` for AP, `TITLE+TEXT+ABSTRACT+SUMMARY+DESCRIPT` and `DOCNO+DOCID` for ZIFF. A more sound heuristic, better adapted to PPM modeling, could achieve better results and do it automatically.

8 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. The impact of the model on the retrieval performance should be negligible.

We have shown that the idea actually improves compression ratios. We have compared our prototype against state-of-the-art compression systems, showing that our word-based Huffman prototype obtains 2% to 4% better compression ratios than existing alternatives that permit random access to the compressed text. On the other hand, combining the *SCM* general concept with PPMDi does not permit random access anymore, but it yields unbeaten compression compared to any other compressor, by a margin of 2% to 5%.

Reducing space is not that important by itself, especially if we consider the low cost of massive storage devices. The point is that those massive devices are usually orders of magnitude slower than smaller memories. In recent years, the performance gaps in the memory hierarchy (registers, cache, RAM, disk, network) have only widened, making compression more and more appealing. A space reduction, even if it involves higher CPU processing time, is beneficial because it reduces transmission costs over the slowest devices (disk and network). Even random accesses of small pieces of the file over a disk benefit from compression, as the disk seek times depend on the number of tracks to traverse, and this a linear function of the file size.

Current research in compression usually strives to obtain gains that are below 1%. By taking the structure into account, we have obtained much more significant gains in compression ratios. It is likely that more improvements in compression ratios can be obtained by pursuing this line. In particular, we plan 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

Adiego, J., Navarro, G., and de la Fuente, P. (2004). Lempel-Ziv compression of structured text. In *Proc. 14th IEEE Data Compression Conference (DCC'04)*, pages 112–121.

Adiego, J., Navarro, G., and de la Fuente, P. (2006). Lempel-ziv compression of highly struc-

tured documents. *Journal of the American Society for Information Science and Technology (JASIST)*. To appear.

Bell, T., Cleary, J., and Witten, I. (1990). *Text Compression*. Prentice Hall, Englewood Cliffs, N.J.

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

Brisaboa, N., Fariña, A., Navarro, G., and Esteller, M. (2003). (s,c)-dense coding: An optimized compression code for natural language text databases. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, pages 122–136. Springer.

Burrows, M. and Wheeler, D. (1994). A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.

Cheney, J. (2001). Compressing XML with multiplexed hierarchical PPM models. In *Proc. 11th IEEE Data Compression Conference (DCC'01)*, pages 163–172.

Cleary, J. and Witten, I. (1984). Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communication*, 32:396–402.

Dvorský, J., Pokorný, J., and Snásel, V. (1999). Word-based compression methods and indexing for text retrieval systems. In *Proc. 2nd East European Symp. on Advances in Databases and Information Systems (ADBIS'99)*, LNCS 1691, pages 75–84. Springer.

Girardot, M. and Sundaresan, N. (2000). Millau: An encoding format for efficient representation and exchange of XML documents over the WWW. In *Proc. 9th Intl. World Wide Web Conf. on Computer Networks*, pages 747–765.

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

Heaps, H. (1978). *Information Retrieval - Computational and Theoretical Aspects*. Academic Press.

Horspool, R. and Cormack, G. (1992). Constructing word-based text compression algorithms. In *Proc. 2nd IEEE Data Compression Conference (DCC'92)*, pages 62–71.

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

Lam, W., Wood, P., and Levene, M. (2003). XCQ: XML compression and querying system. In *Proc. 12th Intl. Conf. on the World Wide Web (WWW'03)*. Poster.

Levene, M. and Wood, P. (2002). XML structure compression. In *Proc. 2nd Intl. Workshop on Web Dynamics*.

Liefke, H. and Suciu, D. (2000). XMill: an efficient compressor for XML data. In *Proc. Intl. ACM Conf. on Management of Data (SIGMOD'00)*, pages 153–164.

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

Moffat, A. and Turpin, A. (2002). *Compression and Coding Algorithms*. Kluwer Academic Publishers.

Moffat, A. and Wan, R. (2001). RE-store: A system for compressing, browsing and searching large documents. In *Proc. 8th Intl. Symp. on String Processing and Information Retrieval (SPIRE'01)*, pages 162–174. IEEE CS Press.

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

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

Shannon, C. (1948). A mathematical theory of communication. *Bell Syst. Tech. J.*, 27:398–403.

Shkarin, D. (2002). PPM: One step to practicality. In *Proc. 12th IEEE Data Compression Conference (DCC 2002)*, pages 202–211.

Tarhio, J. (2001). On compression of parse trees. In *Proc. 8th Intl. Symp. on String Processing and Information Retrieval (SPIRE'01)*, pages 205–211. IEEE Computer Society.

Tolani, P. and Haritsa, J. (2002). XGRIND: A query-friendly XML compressor. In *Proc. 18th Intl. Conf. of Data Engineering (ICDE'02)*, pages 225–234.

Toman, V. (2004). Syntactical compression of XML data. Presented at *16th Intl. Conf. on Advanced Information Systems Engineering (CAiSE'04)*, Riga, Latvia, June 7–11.

Turpin, A. and Moffat, A. (1997). Fast file search using text compression. In *Proceedings of the 20th Australian Computer Science Conference*, pages 1–8.

Welch, T. (1984). A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19.

Witten, I., Moffat, A., and Bell, T. (1999). *Managing Gigabytes*. Morgan Kaufmann Publishers, second edition.

Witten, I., Neal, R., and Cleary, J. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541.

Zipf, G. (1949). *Human Behaviour and the Principle of Least Effort*. Addison–Wesley.

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

Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, 24(5):530–536.

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