Reorganizing Compressed Text [*]

Nieves R. Brisaboa
Fac. de Informática,
Univ. da Coruña,
A Coruña, Spain.
brisaboa@udc.es

Antonio Fariña
Fac. de Informática,
Univ. da Coruña,
A Coruña, Spain.
fari@udc.es

Susana Ladra
Fac. de Informática,
Univ. da Coruña,
A Coruña, Spain.
sladra@udc.es

Gonzalo Navarro
Dept. of Computer Science,
Univ. of Chile,
Santiago, Chile.
gnavarro@dcc.uchile.cl

ABSTRACT

Recent research has demonstrated beyond doubts the benefits of compressing natural language texts using word-based statistical semistatic compression. Not only it achieves extremely competitive compression rates, but also direct search on the compressed text can be carried out faster than on the original text; indexing based on inverted lists benefits from compression as well.

Such compression methods assign a variable-length codeword to each different text word. Some coding methods (Plain Huffman and Restricted Prefix Byte Codes) do not clearly mark codeword boundaries, and hence cannot be accessed at random positions nor searched with the fastest text search algorithms. Other coding methods (Tagged Huffman, End-Tagged Dense Code, or (s, c)-Dense Code) do mark codeword boundaries, achieving a self-synchronization property that enables fast search and random access, in exchange for some loss in compression effectiveness.

In this paper, we show that by just performing a simple reordering of the target symbols in the compressed text (more precisely, reorganizing the bytes into a wavelet-tree-like shape) and using little additional space, searching capabilities are greatly improved without a drastic impact in compression and decompression times. With this approach, all the codes achieve synchronism and can be searched fast and accessed at arbitrary points. Moreover, the reordered compressed text becomes an *implicitly indexed* representation of the text, which can be searched for words in time independent of the text length. That is, we achieve not only fast sequential search time, but indexed search time, for almost no extra space cost.

We experiment with three well-known word-based compression techniques with different characteristics (Plain Huffman, End-Tagged Dense Code and Restricted Prefix Byte Codes), and show the searching capabilities achieved by reordering the compressed representation on several corpora. We show that the reordered versions are not only much more efficient than their classical counterparts, but also more efficient than explicit inverted indexes built on the collection, when using the same amount of space.

[*]Funded in part (for the Spanish group) by MEC grant (TIN2006-15071-C03-03) and AECI grant (A/8065/07), and (for the fourth author) by Yahoo! Research grant "Compact Data Structures" and also AECI grant (A/8065/07).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGIR'08, July 20–24, 2008, Singapore.
Copyright 2008 ACM 978-1-60558-164-4/08/07 ...$5.00.

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: Data Compaction and Compression; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms

Algorithms

Keywords

Word-based compression, searching compressed text, compressed indexing.

1. INTRODUCTION

Text compression is useful not only to save disk space, but more importantly, to save processing, transmission and disk transfer time. Compression techniques especially designed for natural language texts permit searching the compressed text much faster (up to 8 times) than the original text [17, 10], in addition to their proven effectiveness (with compression ratios around 25%-35%).

Those ratios are obtained using a word-based model [9], where words are encoded instead of characters. Words present a more biased distribution of frequencies than characters, following a Zipf Law [18, 1]. Thus the text (regarded as a sequence of words) is highly compressible with a zero-order encoder such as Huffman code [8]. With the optimal Huffman coding, compression ratios approach 25%.

Although necessarily inferior to Huffman code in compression effectiveness, different coding methods, such as Plain Huffman [11] or Restricted Prefix Byte Codes [4], try to approach the performance of classical Huffman while encoding the source symbols as sequences of bytes instead of bits. This degrades compression ratios to around 30%, yet allows much faster decompression.

Still other encoding methods, such as Tagged Huffman codes [11], End-Tagged Dense Codes, and (s, c)-Dense Codes [3], worsen the compression ratios a bit more (up to 35%) in exchange for being *self-synchronized*. This means that codeword boundaries can be distinguished starting from anywhere in the encoded sequence, which enables random access to the compressed text, as well as very fast Boyer-Moore-like direct search of the compressed text.

In this paper, we propose a reordering of the bytes in the codewords of the compressed text following a wavelet-tree-like strategy. We show that this simple variation obtains a compressed text that is *always* self-synchronized, despite building on encodings which are not. That is, the reorganized compressed text can be accessed at any point, even if Plain Huffman coding is used, for example. This encourages using the most efficient bytewise encodings with no penalty.

What is even more striking is that the reorganized text turns out to have some *implicit indexing* properties. That is, with very little extra space, it is possible to search it in time that is not proportional to the text length (as any sequential search method) but logarithmic on it (as typical indexed techniques). Indeed, we compare our reorganized codes against the original techniques armed with an *explicit* inverted index, and show that the former are more efficient when using the same amount of space. Within that little allowed space, block-addressing compressed inverted indexes are the best choice as far as we know [14, 19]. We implement such a compressed block-addressing index following the most recent algorithms for list intersections [5]. Our results demonstrate that it is more convenient to use reorganized codes than trying to use very space-efficient inverted indexes; only if one is willing to pay a significant extra space do inverted indexes pay off.

We note that our technique is tailored to main memory due to its random access pattern. Therefore, it can only compete with inverted indexes in main memory. There has been a lot of recent interest on inverted indexes that operate in main memory [15, 16, 5], mainly motivated by the possibility of distributing a large collection among the main memories of several interconnected processors. By using less space for those in-memory indexes (as our technique allows) more text could be cached in the main memory of each processor and fewer processors (and less communication) would be required.

The paper is organized as follows. The next section describes the coding schemes used as the basis for our research. Section 3 describes wavelet trees and how they can be used. Section 4 presents our reorganizing strategy in detail. Finally, Sections 5 and 6 present our empirical results, conclusions and future work.

2. BYTEWISE ENCODERS

We cover the byte-oriented encoding methods we will use in this paper; many others exist, e.g. [11, 3].

The basic byte-oriented variant of the original Huffman code is called *Plain Huffman* (PH) [11]. Plain Huffman does not modify the basic Huffman code except by using bytes as the symbols of the target alphabet. This worsens the compression ratios to 30%, compared to the 25% achieved by the original Huffman coding on natural language and using words as symbols [9]. In exchange, decompression and searching are much faster with Plain Huffman code because no bit manipulations are necessary.

End-Tagged Dense Code (ETDC) [3] is also a word-based byte-oriented compression technique where the first bit of each byte is reserved to flag whether the byte is the last one of its codeword. The flag bit is enough to ensure that the code is a prefix code regardless of the content of the other 7 bits of each byte, so there is no need at all to use Huffman coding in order to maintain a prefix code. Therefore, all possible combinations are used over the remaining 7 bits of each byte, producing a *dense* encoding. ETDC is easier to build and faster in both compression and decompression. While searching Plain Huffman compressed text requires inspecting all its bytes from the beginning, the tag bit in ETDC permits Boyer-Moore-type searching [2] (that is, skipping bytes) by simply compressing the pattern and then running the string matching algorithm. On Plain Huffman this does not work, as the pattern could occur in the text not aligned to any codeword [11]. Moreover, it is possible to start decompression at any point of the compressed text, because the 7th bit gives ETDC the self-synchronization property: one can easily determine the codeword boundaries.

In general, ETDC can be defined over symbols of b bits, although in this paper we focus on the byte-oriented version where $b = 8$. Given source symbols with decreasing probabilities $\{p_i\}_{0 \le i < n}$ the corresponding codeword using the ETDC is formed by a sequence of symbols of b bits, all of them representing digits in base 2^{b-1} (that is, from 0 to $2^{b-1} - 1$), except the last one which has a value between 2^{b-1} and $2^b - 1$, and the assignment is done sequentially.

Note that the code depends on the rank of the words, not on their actual frequency. As a result, only the sorted vocabulary must be stored with the compressed text for the decompressor to rebuild the model. Therefore, the vocabulary will be slightly smaller than in the case of Huffman codes, where some information about the shape of the Huffman tree must be stored (even for canonical Huffman trees).

As it can be seen, the computation of codes is extremely simple: It is only necessary to sort the source symbols by decreasing frequency and then sequentially assign the codewords. But not only the sequential procedure is available to assign codewords to the words. There are simple encode and decode procedures that can be efficiently implemented, because the codeword corresponding to symbol i is obtained as the number x written in base 2^{b-1}, where $x = i - \frac{2^{(b-1)k} - 2^{b-1}}{2^{b-1}-1}$, and adding 2^{b-1} to the last digit.

In *Restricted Prefix Byte Codes* (RPBC) [4] the first byte of each codeword completely specifies its length. The encoding scheme is determined by a 4-tuple (v_1, v_2, v_3, v_4) satisfying $v_1 + v_2 + v_3 + v_4 \le R$. The code has v_1 one-byte codewords, Rv_2 two-byte codewords, R^2v_3 three-byte codewords and R^3v_4 four-byte ones. They require $v_1 + v_2R + v_3R^2 + v_4R^3 \ge n$ where R is the radix, typically 256. This method improves the compression ratio of ETDC as it adds more flexibility to the codeword lengths, it maintains the efficiency with simple encode and decode procedures (it is also a dense code) but it loses the self-synchronization property. If we seek to a random position in the text, it is not possible to determine the beginning of the current codeword. It is possible to adapt Boyer-Moore searching over text compressed with this technique, but it is slower than searching over text compressed with ETDC.

3. WAVELET TREES

A wavelet tree is a succint data structure. It was proposed in [7] for solving *rank* and *select* queries over sequences on large alphabets. Given a sequence of symbols B, $rank_b(B, i) = y$ if the symbol b appears y times in the prefix $B_{1,i}$, and $select_b(B, j) = x$ if the j^{th} occurrence of the symbol b in the sequence B appears at position x.

The original wavelet tree is a balanced binary tree that divides the alphabet into two halves at each node, and stores bitmaps in the nodes to mark which side was chosen by each symbol in the sequence. Each child handles recursively the part of the sequence formed by its symbols. Solving rank and select queries over bit sequences in constant time is well-known [12, 13]. The wavelet tree reduces rank and select operations on a sequence S to rank and select operations over the bitmaps stored at the nodes. For rank, the tree is traversed top-down, and bottom-up for select.

Multi-ary wavelet trees are introduced in [6], where symbol rank and select operations are needed within the nodes. Huffman shaped wavelet trees have also been considered [7, 13]. Our wavelet trees in this paper are in some sense inspired by these.

4. REORGANIZATION OF CODEWORDS

Our method can be applied to any word-based, byte-oriented semistatic statistical prefix-free compression technique (as all those mentioned in Section 2). Basically the idea is to reorganize the different bytes of each codeword, placing them in different nodes of a tree that we call wavelet tree for its similarity with the wavelet trees used in [7]. That is, instead of representing the compressed text as a concatenated sequence of codewords (composed of one or more bytes), each one replacing the original word at that position in the text, we represent the compressed text as a wavelet tree where the different bytes of each codeword are placed at different nodes.

The root of the wavelet tree contains the first byte of all the codewords, following the same order as the words in the original text. That is, at position i in the root we place the first byte of the codeword that encodes the i^{th} word in the source text. The root has as many children as different bytes can be the first byte of a codeword. For instance, in ETDC the root has always 128 children and in RPBC it will typically have $256 - v_1$. The node x in the second level (taking the root as the first level) stores the second byte of those codewords whose first byte is x. Hence each node handles a subset of the text words, in the same order they have in the original text. That is, the byte at position i in node x is the second byte of the i^{th} text codeword that starts with byte x. The same arrangement is done to create the lower levels of the tree. That is, node x has as many children as different second bytes exist in codewords with more than 2 bytes having x as their first byte.

Formally, let us represent the text words[1] as $\langle w_1, w_2 \dots w_n \rangle$. Lets call cw_i the codeword representing word w_i. Notice that two codewords cw_i and cw_j can be the same if the i^{th} and j^{th} words in the text coincide. The bytes of codeword cw_i are denoted as $\langle c_i^1 ... c_i^m \rangle$ were m is the size of codeword cw_i. The root node of the tree is formed by the following sequence of bytes $\langle c_1^1, c_2^1, c_3^1 ... c_n^1 \rangle$. Notice that the root has as many bytes as words has the text. As explained, the root has a child for each byte value that can be the first in a codeword. Assume there are r words in the source text encoded by codewords (longer than 1 byte) starting with the byte x: $cw_{i_1} ... cw_{i_r}$. Then the node x will store the sequence $\langle c_{i_1}^2, c_{i_2}^2, c_{i_3}^2 ... c_{i_r}^2 \rangle$. Some of those will be the last byte of their codeword, yet others would correspond to codewords with more than two bytes.

Therefore, node x would have in turn children as explained before. Assume node xy is a child of node x. It stores the byte sequence $\langle c_{j_1}^3, c_{j_2}^3, c_{j_3}^3 ... c_{j_k}^3 \rangle$ of all the third bytes of codewords $cw_{j_1} ... cw_{j_k}$ starting with xy, in their original text order. Our wavelet tree is not balanced because some codewords are longer than others. The number of levels will be equal to the number of bytes of the longer codewords.

Figure 1 shows an example of a wavelet tree[2], built from the text `LONG TIME AGO IN A GALAXY FAR FAR AWAY`, and the alphabet $\Sigma = \{$`A`, `AGO`, `AWAY`, `FAR`, `GALAXY`, `IN`, `LONG`, `TIME`$\}$. After obtaining the codewords for all the words in the text, using a known compressor, we reorganize their bytes in the wavelet tree following the arrangement explained. The first byte of each codeword is in the root node. The next bytes are contained in the corresponding child nodes. For example, the second byte of the word `'AWAY'` is the third byte of node $B2$, because it is the third word in the root node having b_2 as first byte. Its third byte is in node $B2B4$ as its two first codeword bytes are b_2 and b_4.

Assume we want to know which is the 6^{th} word in the text. Starting at the root node in Figure 1, we read the byte at position 6 of the root node: $Root[6] = b_4$. The encoding scheme indicates that the codeword is not complete yet, so we move to the second level of the tree. The second byte is contained in the node $B4$, which is the child node of the root where the second bytes of all codewords starting by byte b_4 are stored. Using a byte *rank* operation we obtain $rank_{b_4}(Root, 6) = 2$. This means that the second byte of the codeword starting in the byte at position 6 in the root node will be the 2^{nd} byte in the node $B4$. In the next level, $B4[2] = b_5$, therefore b_5 is the second byte of the codeword we are looking for. Again the encoding scheme indicates that the codeword is still not complete, and $rank_{b_5}(B4, 1) = 1$ tells us that the 3^{rd} byte of that word will be in the node $B4B5$ at position 1. One level down, we obtain $B4B5[1] = b_2$, and now the obtained sequence $b_4 b_5 b_2$ is a complete codeword according to the encoding scheme. It corresponds to `'GALAXY'`, which therefore is the 6^{th} word in the source text.

This process can be used to recover any word. Notice that this mechanism gives direct access and random decompression capabilities to encoding methods that do not mark boundaries in the codewords. With the proposed arrangement, those boundaries become automatically defined (each byte in the root corresponds to a new codeword).

If we want to search for the first occurrence of `'AWAY'` in the example of Figure 1, we start by finding out its codeword, which is $b_2 b_4 b_3$. Therefore the search will start at the node $B2B4$, which holds all the codewords starting with $b_2 b_4$. In this leaf node we find out where the first byte b_3 occurs, because b_3 is the third byte of the codeword sought. Operation $select_{b_3}(B2B4, 1) = 1$ tell us that the first occurrence of our codeword is the first of all codewords starting with $b_2 b_4$, thus in the node $B2$ the first occurrence of byte b_4 is the one encoding the first occurrence of the word `'AWAY'` in the text. Again, to know where in the node $B2$ is the first byte b_4 we perform $select_{b_4}(B2, 1) = 3$. Now we know that in the root node the 3^{rd} byte b_2 will be the one corresponding to the first byte of our codeword. To know where in the root node is that 3^{rd} byte b_2 we compute $select_{b_2}(Root, 3) = 9$.

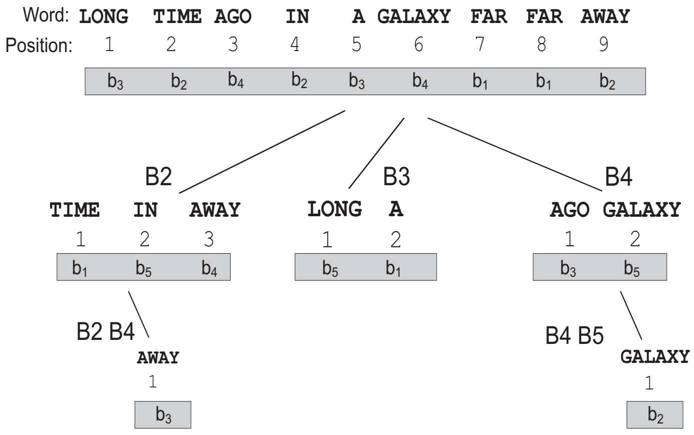
[1]We speak of words to simplify the discussion. In practice both words and separators are encoded as atomic entities in word-based compression.

[2]Note that only the shaded byte sequences are stored in the nodes; the text is shown only for clarity.

Figure 1: Example

TEXT: "LONG TIME AGO IN A GALAXY FAR FAR AWAY"

SYMBOL	FREQ	CODE
FAR	2	b_1
IN	1	b_2 b_5
A	1	b_3 b_1
LONG	1	b_3 b_5
AGO	1	b_4 b_3
TIME	1	b_2 b_1
AWAY	1	b_2 b_4 b_3
GALAXY	1	b_4 b_5 b_2


Word: LONG TIME AGO IN A GALAXY FAR FAR AWAY
Position: 1 2 3 4 5 6 7 8 9
b3 b2 b4 b2 b3 b4 b1 b1 b2
B2
TIME IN AWAY
1 2 3
b1 b5 b4
B3
LONG A
1 2
b5 b1
B4
AGO GALAXY
1 2
b3 b5
B2 B4
AWAY
1
b3
B4 B5
GALAXY
1
b2

Finally the result is that the word **'AWAY'** appears for the first time as the 9th word of the text. Notice that it would be easy to obtain a snippet of an arbitrary number of words around this occurrence, just by using the explained decompression mechanism, on any encoding.

The sum of the space needed for the byte sequences stored at all nodes of the tree is exactly the same as the size of the compressed text. Just a reordering has taken place. Yet, a minimum of extra space is necessary in order to maintain the tree shape information with a few pointers. Actually, the shape of the tree is determined by the compression technique, so it is not necessary to store those pointers, but only the length of the sequence at each node.

4.1 Algorithms

We now detail the algorithms for compression, decompression, and searching.

4.1.1 Compression

The compression algorithm makes two passes on the source text. In the first pass we obtain the vocabulary and the model (frequencies), and then assign codewords using any prefix-free semistatic encoding scheme. In the second pass the source text is processed again and each word is translated into its codeword. Instead of storing those codewords sequentially, as a classical compressor, the codeword bytes are spread along the different nodes in the wavelet tree. The node where a byte of a codeword is stored depends on the previous bytes of that codeword, as explained.

It is possible to precalculate how many nodes will form the tree and the sizes of each node before the second pass starts, so they can be allocated and filled with the codeword bytes as the second pass takes place. We maintain an array of markers that point to the current writing position at each node, so that they can be filled sequentially following the order of the words in the text.

Finally, we generate the compressed text as the concatenation of the sequences of all the nodes in the wavelet tree, and add a header with the words ↔ codewords assignment, plus the length of the sequence at each tree node.

4.1.2 Random decompression

To decompress from a random text word j, we access the j-th byte of the root node sequence to obtain the first byte of the codeword. If the codeword has just one byte, we finish at this point. If the byte read b_i is not the last one of a codeword, we have to go down in the tree to obtain the rest of the bytes. As explained, the next byte of the codeword is stored in the child node Bi, the one reached from the first byte b_i. All the codewords starting with that byte b_i are stored in Bi, so we have to count the number of occurrences of the byte b_i in the root node before position j by using the rank operation, $rank_{b_i}(root, j) = k$. Thus k is the position in the child node Bi of the second byte of the codeword. We repeat this procedure as many times as the length of the codeword.

If we need to decompress the previous or the next word we follow the same algorithm starting with the previous or the next entry of the root node.

The complexity of this algorithm is $(l-1)$ times the complexity of rank operation, where l is the length of the codeword. Therefore, its performance depends on the implemen-

Algorithm 1 Construction of WTDC

```
//input: $t$, source text
//output: compressed text with shape of wavelet tree
$voc \leftarrow first\text{-}pass(t)$
$sort(voc)$
$totalNodes \leftarrow calculateNumberNodes()$
for all $node \in totalNodes$ do
    $length[node] \leftarrow calculateSeqLength(node)$
    $wt[node] \leftarrow allocate(length[node])$
    $marker[node] \leftarrow 0$
end for
for all $word \in t$ do
    $cw \leftarrow code(word)$
    $currentnode \leftarrow rootnode$
    for $i \leftarrow 1$ to $|cw|$ do
        $j \leftarrow marker[currentnode]$
        $wt[currentnode][j] \leftarrow cw^i$
        $marker[currentnode] \leftarrow j + 1$
        $currentnode \leftarrow child(currentnode, cw^i)$
    end for
end for
return concatenation of node sequences, vocabulary, and length of node sequences
```

Algorithm 2 *Display x*

```
//input: $x$, position in the compressed text
//output: $p$, word at position $x$ in the compressed text
$currentnode \leftarrow rootnode$
$c \leftarrow wt[currentnode][x]$
$cw \leftarrow [c]$
while $cw$ is not completed do
    $x \leftarrow rank_c(currentnode, x)$
    $currentnode \leftarrow child(currentnode, c)$
    $c \leftarrow wt[currentnode][x]$
    $cw \leftarrow cw||c$
end while
$p \leftarrow decode(cw)$
return p
```

tation of the rank operation.

4.1.3 Full decompression

After loading the vocabulary and rebuilding the wavelet tree, the full decompression of the compressed text consists of decoding sequentially each entry of the root. All the nodes of the tree will be also processed sequentially, so to gain efficiency we maintain pointers to the current first unprocessed entry of each node. Once we obtain the child node where the codeword of the current word continues, we can avoid unnecessary rank operations because that byte will be the next one to process in the corresponding node. Except for this improvement, the algorithm is the same as that explained in Section 4.1.2.

4.1.4 Searching

To **count** the occurrences of a given word, we compute how many times the last byte of the codeword assigned to that word appears in the corresponding leaf node. That leaf node is the one identified by all the bytes of the codeword except the last one. The pseudocode is presented in Algorithm 3.

Algorithm 3 *Count* operation

```
//input: $w$, a word
//output: $n$, number of occurrences of $w$
$cw \leftarrow code(w)$
Let $cw = cw'||c$, being $c$ the last byte
$currentnode \leftarrow node\ corresponding\ to\ code\ cw'$
$n \leftarrow rank_c(currentnode, length[currentnode])$
return $n$
```

To **locate** all the occurrences of a given word, we start looking for the last byte of the corresponding codeword cw in the associated leaf node using operation *select*. If the last symbol of the codeword, $cw^{|cw|}$, occurs at position j in the leaf node, then the previous byte $cw^{|cw|-1}$ of that codeword will be the j^{th} one occurring in the parent node. We proceeded in the same way up in the tree until reaching the position x of the first byte cw^1 in the root. Thus x is the position of the first occurrence of the word searched for. To find all the occurrences of a word we proceed in the same way, yet we can use pointers to the already found positions in the nodes to speed up the select operations (this might be relevant depending on the *select* algorithm used).

It is also possible to search a *phrase pattern*. We locate all the occurrences of the least frequent word in the root

Algorithm 4 *Locate j^{th} occurrence of word w* operation

```
//input: $w$, word
//input: $j$, integer
//output: position of the $j$-th occurrence of $w$
$cw \leftarrow code(w)$
Let $cw = cw'||c$, being $c$ the last byte
$currentnode \leftarrow node\ corresponding\ to\ code\ cw'$
for $i \leftarrow |cw|$ to 1 do
    $j \leftarrow select_{cw^i}(currentnode, j)$
    $currentnode \leftarrow parent(currentnode)$
end for
return $j$
```

node, and then check if all the first bytes of each codeword of the pattern match with the previous and next entries of the root node. If those first bytes match, we verify their complete codewords around the candidate occurrence found. As shown in the experiments, this is a very efficient method in practice.

4.2 Rank and select over bytes

As it was mentioned before, the efficiency of the search and random decompression algorithms depends on the implementation of *rank* and *select* operations.

A baseline solution is to carry out those operations by brute force, that is, by sequentially counting all the occurrences of the byte we are interested in, from the beginning of the node sequence. This simple option does not require any extra structure. Interestingly enough, it already allows that operations count and locate are carried out more efficiently than in classically compressed files. In both cases we do sequential searches, but in the reorganized version these searches are done over a reduced portion of the file. Likewise, it is possible to access the compressed text at random, even using non-synchronized codes such as PH and RPBC, faster than scanning the file from the beginning.

However, it is possible to drastically improve the performance of rank and select operations at a very moderate extra space cost, by adapting well-known theoretical techniques [6]. Given a sequence of bytes $B[1,n]$, we use a two-level directory structure, dividing the sequence into sb superblocks and each superblock into b blocks of size $n/(sb*b)$. The first level stores the number of occurrences of each byte from the beginning of the sequence to the start of each superblock. The second level stores the number of occurrences of each byte up to the start of each block from the beginning of the superblock it belongs to. The second-level values cannot be larger than $sb*b$, and hence can be represented with fewer bits.

With this approach, $rank_{b_i}(B,j)$ is obtained by counting the number of occurrences of b_i from the beginning of the last block before j up to the position j, and adding to that the values stored in the corresponding block and superblock for byte b_i. Instead of $O(n)$, this structure answers *rank* in time $O(n/(sb*b))$. To compute $select_{b_i}(B,j)$ we binary search for the first value x such that $rank_{b_i}(B,x) = j$. We first binary search the stored values in the superblocks, then those in the blocks inside the right superblock, and finally complete the search with a sequential scanning in the right block. The time is $O(\log sb + \log b + n/(sb*b))$.

An interesting property is that this structure is parameterizable. That is, there is a space/time tradeoff associated

Table 1: **Description of the corpora used.**

CORPUS	size (bytes)	num words	voc. size
CR	51,085,545	10,113,143	117,713
ZIFF	185,220,211	40,627,131	237,622
ALL	1,080,720,303	228,707,250	885,630

Table 2: **Compression ratio (in %).**

	PH	ETDC	RPBC	WPH	WTDC	WRPBC
CR	31.06	31.94	31.06	31.06	31.95	31.07
ZIFF	32.88	33.77	32.88	32.88	33.77	32.89
ALL	32.83	33.66	32.85	32.83	33.66	32.85

to parameters *sb* and *b*. The shorter the blocks, the faster the sequential counting of occurrences of byte b_i.

5. EXPERIMENTAL RESULTS

We used some large text collections from TREC-2: AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF), as well as TREC-4, namely Congressional Record 1993 (CR) and Financial Times 1991 to 1994 (FT91 to FT94) to create a large corpora (ALL) by aggregating them all. We also used CR and ZIFF corpus individually. Table 5 presents the main characteristics of the corpora used. The first column indicates the name of the corpus, the second its size (in bytes). The third column in that table indicates the number of words that compose the corpus and finally the fourth column shows the number of different words in the text.

We used the spaceless word model [10] to create the vocabulary, that is, if a word was followed by a space, we just encoded the word, otherwise both the word and the separator were encoded.

An isolated Intel®Pentium®-IV 3.00 GHz system (16Kb L1 + 1024Kb L2 cache), with 4 GB dual-channel DDR-400Mhz RAM was used in our tests. It ran Debian GNU/Linux (kernel version 2.4.27). The compiler used was gcc version 3.3.5 and `-O9` compiler optimizations were set. Time results measure CPU user time in seconds.

5.1 Compression properties

We measure how our reorganization of codeword bytes affects the main compression parameters, such as compression ratio and compression and decompression times.

We use our reorganization method over the compressed texts obtained using three well-known compression techniques explained in Section 2. We call WPH, WTDC, and WRPBC to the wavelet-tree reorganization applied over Plain Huffman, End-Tagged Dense Code, and Restricted Prefix Byte Codes, respectively.

Table 2 shows that compression ratio is essentially not affected. There is a very slight loss of compression (close to 0.01%), due to the storage of the tree shape.

Tables 3 and 4 show the compression and decompression time obtained using the wavelet trees. Whereas the compression time is almost the same (2%-4% worse), there are larger differences in decompression time (20%-25% slower). With the wavelet tree, decompression is not just a sequential process. For each word of the text, a top-down traversal is carried out on the tree, so the benefits of cache and spatial locality disappear. This is more noticeable than at compression, where the overhead of parsing the source text

Table 3: **Compression time.**

	PH	ETDC	RPBC	WPH	WTDC	WRPBC
CR	2.886	2.870	2.905	3.025	2.954	2.985
ZIFF	11.033	10.968	11.020	11.469	11.197	11.387
ALL	71.317	71.452	71.614	74.631	73.392	74.811

blurs those time differences.

Table 4: **Decompression time.**

	PH	ETDC	RPBC	WPH	WTDC	WRPBC
CR	0.574	0.582	0.583	0.692	0.697	0.702
ZIFF	2.309	2.254	2.289	2.661	2.692	2.840
ALL	14.191	13.943	14.131	16.978	17.484	17.576

5.2 Searching and displaying

We show now the efficiency achieved by the reordering technique for pattern searching and random decompression.

Table 6 summarizes the performance, measuring user time, of the main search operations: *count* all the occurrences of a pattern (in milliseconds), locate the position of the *first* occurrence (in milliseconds), *locate all* (in seconds) and extract *all the snippets* around the occurrences of a pattern (in seconds). We run our experiments over the largest corpus, ALL, and show the average time of searching for 100 randomly chosen words of low and medium frequency (to avoid the stopwords). We present the result obtained by the compression methods PH, ETDC, and RPBC; by the wavelet trees implemented without blocks and superblocks (WPH, WTDC, and WRPBC); and also wavelet trees using those structures (covering 21,000 bytes per block, and 10 blocks per superblock) with a waste of 1% of extra space, to speed up rank and select operations (WPH+, WTDC+, and WRPBC+). Table 5 shows the loading time, so that the compressed text becomes ready for querying, and the internal memory usage to solve queries needed for each method. In the case of *rank* structures, it takes more than 1 second to create the two-level directory. This time is not as important as search times, because this loading is performed only once.

Even without using the extra space for the blocks and superblock structures, wavelet trees improve all searching capabilities except for extracting all the snippets, as shown in Table 6. This is because snippets require decompressing several codewords around each occurrence, and random decompression is very slow in the wavelet trees if one has no extra support for the rank operations that track random codewords down.

By just spending 1% extra space in block and superblock data structures, *rank* operations are dramatically improved, including extracting all the snippets. Only the self-synchronized ETDC is still faster than its corresponding wavelet tree (WTDC+) for extracting snippets. This is because extracting a snippet around a word in a non self-synchronized code implies extra operations to permit the decompression of the previous words, while ETDC can easily move backwards in the compressed text.

By raising the extra space allocated to blocks and superblocks to 5%, WTDC+ finally takes over ETDC in extracting snippets as well. It is important to remark that our proposal improves all searching capabilities when a compres-

Table 5: Load time (in seconds) and memory usage (% of corpus size).

	PH	ETDC	RPBC	WPH	WTDC	WRPBC	WTPH+	WTDC+	WRPBC+
Load time (s)	0.37	0.39	0.36	0.38	0.35	0.37	1.57	1.6	1.57
Memory usage	35.13%	35.95%	35.14%	35.13%	35.96%	35.14%	36.11%	36.95%	36.09%

Table 6: Searching capabilities

	Count (ms)	First (ms)	Locate (s)	snippet (s)
PH	2605.600	48.861	2.648	7.955
ETDC	1027.400	22.933	0.940	1.144
RPBC	1996.300	41.660	2.009	7.283
WPH	238.500	17.173	0.754	72.068
WTDC	221.900	17.882	0.762	77.845
WRPBC	238.700	17.143	0.773	75.435
WPH+	0.015	0.017	0.123	5.339
WTDC+	0.015	0.014	0.129	6.130
WRPBC+	0.015	0.018	0.125	5.036

sion technique is not self-synchronized.

5.3 Implicit indexing versus classical indexes

As explained, our reorganization brings some (implicit) indexed search capabilities into the compressed file. In this section we compare the search performance of WTDC+ against a block-addressing compressed inverted index (in the style of [14]) over text compressed with ETDC (II), which works completely in main memory. Basically, II is a block-grained inverted index: It assumes that the indexed text is partitioned into blocks of size b, and for each term it keeps a *list of occurrences* that stores all the block-ids in which that term occurs. To reduce its size, the lists of occurrences were compacted using the *index+bc* strategy in [5]; that is, an absolute sample is kept every s values, and the remaining values are kept with a d-gap strategy combined with byte-codes (bc). Moreover, the text is compressed with ETDC. Therefore, the list of occurrences of a term t points actually to the blocks that contain the beginning of the codeword $ETDC(t)$ associated by ETDC to t. Searches for a word t are done by obtaining the *block-ids* of the blocks in which t appears and then searching for $ETDC(t)$ (using Horspool's algorithm) in the pointed blocks. Searches for a phrase $t_1 \dots t_m$ imply intersecting the lists of occurrences of all the terms $t_1 \dots t_m$ and finally applying Horspool's algorithm to search for the sequence of their codewords $p = ETDC(t_1) \dots ETDC(t_m)$ in such blocks. The intersection of lists is done using *set-vs-set*[3] approach, as recommended in [5], combined with binary search on the largest set.

II is parameterizable by two parameters: the block size (b) measured in kilobytes, and the sample value (s), such that given a posting list of size n, $\lceil n/s \rceil$ absolute samples will be chosen at regular intervals of size s. Different space-time trade-offs are obtained depending on such parameters. To make a fair comparison between II and WTDC+ we chose two different configurations for II ($II_{s8,b16}$ and $II_{s32,b256}$) by setting $b = 16$, $s = 32$ and $b = 256$, $s = 32$, respectively. Then, we chose two settings of WTDC+ (WT_1 and WT_2) that required almost the same amount of memory. More precisely, WT_1 uses one block per each 2,000 bytes in the compressed text, and 1 superblock per each 8 blocks. In WT_2, we used 1 superblock per each 20 blocks and 1 block per 7,000 bytes. The sizes of the resulting four structures, as well as their compression ratios, are shown in Table 7.

[3]Set-vs-set begins with the shortest set as a pivot and intersects it against the others in increasing order of size.

Table 7: Sizes of the compared WTDC+ and II structures.

Index	Wavelet trees		Block-add Inv Indexes	
type	WT_1	WT_2	$II_{s8,b16}$	$II_{s32,b256}$
size(MB)	457.32	397.97	469.38	402.66
C. ratio(%)	44.37	38.61	45.54	39.07

Table 8 shows the time (in seconds) needed to locate all the text occurrences of 100 randomly-chosen single-word patterns and to extract all the snippets around such occurrences. We show results for 4 groups of words depending on their number of frequencies (f): *i)* $f \leq 100$, *ii)* $100 < f \leq 1000$, *iii)* $1000 < f \leq 10000$, and *iv)* $f > 10000$. The snippets were obtained by decompressing 20 words, starting in an offset 10 words before an occurrence. Both locate and the extraction of snippets are faster in WTDC+ than in II. Only when we are extracting the snippets of very frequent words can $II_{s8,b16}$ beat WT_1.

Table 8: Searching for words: WTDC+ Vs Block-addressing Inverted Index. Times in seconds.

	Freq.	WT_1	$II_{s8,b16}$	WT_2	$II_{s32,b256}$
Locate	1-100	0.005	0.020	0.008	0.270
	101-1000	0.134	0.580	1.343	7.260
	1001-10000	0.478	5.820	1.715	42.130
	>10000	3.702	31.240	6.748	66.450
Snippet	1-100	0.028	0.030	0.064	0.280
	101-1000	0.771	0.640	2.845	7.300
	1001-10000	5.456	6.130	13.251	42.440
	>10000	44.115	33.700	102.722	68.870

Table 9 shows the time (in seconds) needed for performing locate and extract-snippet operations over 100 phrase patterns randomly chosen from the text. Results are given depending on the number of words $(2, 4, 6, 8)$ in the phrase pattern. In practice, when we aim at using the smallest index configuration, WTDC+ behaves much better than II and WT_2 clearly overcomes the results of $II_{s32,b256}$. Only when II is allowed to use more memory, $II_{s8,b16}$, it can compete with WT_1 in snippet-extraction time. Gaps between both approaches are reduced because the extraction process is slower in WT than in II once an occurrence has been located. However, the locate operation is still much faster on WT_1.

We remark that our good results essentially owe to the fact that we are not sequentially scanning any significant portion of the file, whereas a block addressing inverted index must sequentially scan (sometimes a significant number of)

Table 9: Searching for phrases: WTDC+ Vs Block-addressing Inverted Index. Times in seconds.

	#words	WT_1	$II_{s8,b16}$	WT_2	$II_{s32,b256}$
Locate	2	1.920	5.570	4.250	22.880
	4	2.020	4.980	4.090	16.500
	6	1.300	2.020	3.050	10.990
	8	0.940	1.250	2.630	8.470
Snippet	2	5.070	5.750	11.730	23.050
	4	2.100	4.990	4.230	16.500
	6	1.300	2.020	3.060	11.000
	8	0.950	1.250	2.630	8.460

blocks. Inverted indexes that directly point to occurrences instead of blocks require much more space and hence are not competitive for this comparison.

6. CONCLUSIONS AND FUTURE WORK

It has been long established that semistatic word-based byte-oriented compressors such as those considered in this paper are useful not only to save space and time, but also to speed up sequential search for words and phrases. However, the more efficient compressors such as PH and RPBC are not that fast at searching or random decompression, because they are not self-synchronizing. In this paper we have shown how a simple reorganization of the bytes of the codewords obtained when a text is being compressed, can produce clear codewords boundaries for those compressors. This gives better search capabilities and random access than all the byte-oriented compressors, even those that pay some compression degradation to mark codeword boundaries (Tagged Huffman, ETDC).

As our reorganization permits carrying out all those operations efficiently over PH, the most space-efficient byte-oriented compressor, the usefulness of looking for coding variants that sacrifice compression ratio for search or decoding performance is questioned: A reorganized Plain Huffman (WPH) will do better in almost all aspects.

This reorganization has also surprising consequences related to implicit indexing of the compressed text. Block-addressing indexes over compressed text have been long considered the best low-space structure to index a text for efficient word and phrase searches. They can trade space for speed by varying the block size. We have shown that the reorganized codewords provide a powerful alternative to these inverted indexes. By adding a small extra structure to the wavelet trees, the search operations are speeded up so sharply that the structure competes successfully with block-addressing inverted indexes that take the same space on top of the compressed text. Especially, our structure is superior when little extra space on top of the compressed text is permitted. More experiments are required to compare more exhaustively our wavelet trees against not only inverted indexes but also other reduced-space structures.

7. REFERENCES

[1] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman, May 1999.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, Oct. 1977.

[3] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.

[4] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In M. P. Consens and G. Navarro, editors, *SPIRE*, volume 3772 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[5] J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *SPIRE*, volume 4726 of *Lecture Notes in Computer Science*, pages 137–148. Springer, 2007.

[6] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.

[7] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 03)*, pages 841–850, 2003.

[8] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Inst. Radio Eng.*, pages 1098–1101, Sept. 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.

[9] A. Moffat. Word-based text compression. *Softw. Pract. Exper.*, 19(2):185–198, 1989.

[10] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In B. Croft, A. Moffat, C. Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proceedings of the 21th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'98)*, pages 298–306. York Press, 1998.

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

[12] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS v. 1180, pages 37–42, 1996.

[13] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

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

[15] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[16] T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 175–182. ACM Press, 2007.

[17] A. Turpin and A. Moffat. Fast file search using text compression. In *Proc. 20^{th} Australian Comp. Sci. Conf*, pages 1–8, 1997.

[18] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley (Reading MA), 1949.

[19] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)*, 23(4):453–490, 1998.