Inf Retrieval (2012) 15:527–557
DOI 10.1007/s10791-012-9184-1

Implicit indexing of natural language text by reorganizing bytecodes

Nieves R. Brisaboa · Antonio Fariña · Susana Ladra · Gonzalo Navarro

Received: 6 June 2011 / Accepted: 14 January 2012 / Published online: 4 February 2012

Abstract Word-based byte-oriented compression has succeeded on large natural language text databases, by providing competitive compression ratios, fast random access, and direct sequential searching. We show that by just rearranging the target symbols of the compressed text into a tree-shaped structure, and using negligible additional space, we obtain a new *implicitly indexed* representation of the compressed text, where search times are drastically improved. The occurrences of a word can be listed directly, without any text scanning, and in general any inverted-index-like capability, such as efficient phrase searches, can be emulated without storing any inverted list information. We experimentally show that our proposal performs not only much more efficiently than sequential searches over compressed text, but also than explicit inverted indexes and other types of indexes, when using little extra space. Our representation is especially successful when searching for single words and short phrases.

Keywords Word-based compression · Searching compressed text · Compressed indexing

A preliminary partial version of this paper appeared in Proc. SIGIR 2008, pp. 139–146.

N. R. Brisaboa · A. Fariña · S. Ladra (✉)
Database Laboratory, University of A Coruña, Campus de Elviña s/n, 15071 A Coruña, Spain
e-mail: sladra@udc.es

N. R. Brisaboa
e-mail: brisaboa@udc.es

A. Fariña
e-mail: antonio.farina@udc.es

G. Navarro
Department of Computer Science, University of Chile, Blanco Encalada, 2120 Santiago, Chile
e-mail: gnavarro@dcc.uchile.cl

1 Introduction

Text compression is not only useful to save disk space, but more importantly, to save processing, transmission, and disk transfer time. In text databases, reducing the compression ratio (that is, the size of the compressed file as a percentage of its uncompressed size) is only one of the criteria to choose a compressor. It is also particularly important to decompress quickly, to access the collection at arbitrary points for display purposes, and to support fast pattern searches.

Compression techniques designed for natural language text databases have obtained good compression ratios (around 25–30%) while allowing one to decompress the text collection from any point (Turpin and Moffat 1997; Moura et al. 2000). By using slightly more space they also offer very fast sequential searches on the compressed text, 3–8 times faster than on the plain text (Moura et al. 2000; Brisaboa et al. 2007).

In order to offer indexed searches (that is, search times that do not scale up linearly with the database size), an *inverted index* is usually added on top of the collection. A *word-addressing* inverted index records the positions of each word in the collection, and it can easily double the space of the compressed text (Baeza-Yates and Ribeiro-Neto 1999; Witten et al. 1999). *Block addressing* (Navarro et al. 2000) is used to reduce space: the text is cut into blocks and the index records only the blocks where each word appears. Then the indexed searches must be complemented with some sequential scanning on compressed text blocks, giving a space/time tradeoff related to the block size.

Most of the compressors that have succeeded in natural language text databases are statistical methods that use a word-based model (Bentley et al. 1986; Moffat 1989), where words, not characters, are taken as the source symbols[1]. Words exhibit a more biased distribution of frequencies than characters. Thus, regarded as a sequence of words, the text is highly compressible with a zero-order modeling. The use of words captures high-order entropy statistics, while ensuring that the model is not too large [as the vocabulary grows sublinearly with the size of the text collection (Heaps 1978)]. With the optimal binary Huffman (1952) coding, compression ratios can be as low as 25%.

Although somewhat inferior to binary Huffman codes in compression effectiveness, different coding methods, such as Plain Huffman Codes (Huffman 1952; Moura et al. 2000) or Restricted Prefix Byte Codes (Culpepper and Moffat 2005) encode the source symbols as sequences of bytes instead of bits. This enables much faster decompression.

Other coding methods, such as Tagged Huffman Codes (Moura et al. 2000), End-Tagged Dense Codes, and (s, c)-Dense Codes (Brisaboa et al. 2007), worsen the compression ratios a bit more in exchange for *self-synchronization*. This means that codeword boundaries can be distinguished starting from anywhere in the encoded sequence. Self-synchronization enables random access to the compressed text, as well as very fast Boyer-Moore-like direct searches on the compressed text (Boyer and Moore 1977).

In this paper we show that self-synchronized byte-encodings and block-addressing inverted indexes may be unnecessary. We propose a rearrangement of the bytes of the compressed text codewords into a tree-shaped data structure that we call *Wavelet Trees on Bytecodes (WTBC)* for its resemblance to wavelet trees (Grossi et al. 2003). This reordering by itself brings self-synchronization to any byte-encoding scheme. For example, even using Plain Huffman Codes, the reordered compressed text can be directly accessed at any word offset. This encourages the use of the most space-efficient byte-encodings, on which direct access is achieved.

[1] The strings separating words are called "separators" and handled like words too.

What is even more striking is that the rearranged text turns out to offer *implicit indexing* properties. That is, it can list the text positions of any word directly, just as with a word-addressing inverted index. Moreover, it can in general simulate any functionality of such an index, for example carrying out efficient phrase searches on the text. In addition, it efficiently supports some operations that are hard to compute with inverted indexes, such as counting word frequencies along ranges of the text.

We implemented block-addressing inverted indexes on top of different word-based compressors and using the most efficient list compression and list intersection techniques, in order to compare to WTBCs. Our results demonstrate that, within the same space usage, it is more convenient to use WTBC than those space-efficient inverted indexes. Only if one is willing to degrade compression ratios over some point, inverted indexes may take over in some queries.

We remark that we are focusing on the so-called "full-text retrieval", that is, on the problem of retrieving the occurrences of a query in the text, and therefore we compare to word-addressing inverted indexes. These must be converted into block-addressing indexes in order to compete in space with our structure. A different problem is "document retrieval", that is, retrieving the documents where a query appears. This is handled with a document-addressing inverted index. We briefly discuss in the Conclusions about derivatives of our present work that handle this type of search.

We also compare our proposal with other word-based compressed indexes in the literature. Those directly based on wavelet trees (Claude and Navarro 2008) achieve non-competitive compression ratios. Those based on word-based suffix arrays (Brisaboa et al. 2008b), on the other hand, can use slightly less space than WTBC and are faster when searching for phrases of 3 or more words. Yet, they are slower on the more common word and 2-word queries[2], and particularly slow to display portions of the text. They are also unable to efficiently restrict the search to an area of the database.

Our technique is applicable in main memory due to its random access pattern. There has been much recent interest on inverted indexes that operate in main memory (Sanders and Transier 2007; Transier and Sanders 2010; Strohman and Croft 2007; Culpepper and Moffat 2007, 2010), 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 and energy) would be required. In small handheld devices, secondary memory may even be absent, and using less space may make the difference between being able or not to handle a collection.

The paper is organized as follows. The next section describes the byte-coding schemes we build on. Section 3 describes wavelet trees and how they can be used as compressed indexes. Sections 4 and 5 present our WTBC technique, first the structure and then the access algorithms. Section 6 presents our experimental results. We conclude in Section 7 and give future work directions.

2 Bytewise encoders

We cover the most representative byte-oriented coding methods: Huffman-based ones (Huffman 1952; Moura et al. 2000), Dense Codes (Brisaboa et al. 2007), and Restricted Prefix Bytes codes (Culpepper and Moffat 2005).

[2] According to up-to-date studies, these comprise 65–90% of the queries posed to Web search engines in various countries, see http://www.keyworddiscovery.com/keyword-stats.html.

The byte-oriented variant of the binary Huffman code, called *Plain Huffman Code* (*PH*) (Huffman 1952; Moura et al. 2000), is just a Huffman code with arity 256, so its target symbols are bytes instead of bits. This worsens the compression ratio by around 5 percentage points over that obtained by binary Huffman coding on natural language and using words as source symbols (Moffat 1989). In exchange, decompression and searching are much faster on PH because no bit manipulations are needed. Tagged Huffman codes (*TH*) (Moura et al. 2000) are similar to PH, but they use a flag bit to obtain synchronism at the expense of another 5 percentage point loss in compression ratio.

End-Tagged Dense Code (ETDC) is the simplest and fastest member of the family of (s, c)-Dense Codes (*SCDC*) (Brisaboa et al. 2007). It reserves the first bit of each byte to flag the last byte of the codewords. Such flag bit is enough to ensure that the code is a prefix code regardless of the content of the other 7 bits, so all the 128 possible combinations are used. ETDC is better than TH in almost every aspect. In the more general SCDC codes, values from 0 to $s - 1$ are final codeword bytes, and the other $c = 256 - s$ values denote that the codeword continues. SCDC codes get very close to PH in compression ratio and are almost as efficient as ETDC, which is the particular case $s = c = 128$.

TH, ETDC, and SCDC are self-synchronizing codes, that is, one can start decompression at any point in the compressed sequence, even from the middle of a code, in any direction. They are also amenable to direct searching: a search pattern can be encoded and searched for in the compressed text with any string matching algorithm, even those skipping characters (Boyer and Moore 1977), without fear of false positives.

In *Restricted Prefix Byte Codes* (*RPBC*) (Culpepper and Moffat 2005) the first byte of each codeword completely specifies its length. The encoding scheme is determined by a 4-tuple[3] (v_1, v_2, v_3, v_4) satisfying $v_1 + v_2 + v_3 + v_4 \leq R$, where the radix R is typically 256. 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 that $v_1 + v_2R + v_3R^2 + v_4R^3$ is not less than the vocabulary size. This method improves the compression ratio of ETDC as it adds more flexibility to the codeword lengths. It maintains efficiency with simple encode and decode procedures, but it loses the self-synchronization property. It is possible to run Boyer-Moore-like searches over this encoding, but this is slower than searching text compressed with ETDC.

In this paper we will use PH, ETDC, and RPBC as the techniques to illustrate our rearrangement strategy. Since we will obtain self-synchronization on any code and indexed searches, there will be a strong reason to prefer PH over the other codes, as it achieves minimum space and the advantages of the other codes will be blurred.

3 Wavelet trees

The wavelet tree is a data structure proposed by Grossi et al. (2003) for representing a sequence $S[1, n]$ in compressed form. 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. The root handles the whole sequence and each child handles recursively the subsequence with the symbols assigned to it. The leaves correspond to a single symbol and are not represented. On an alphabet of size

[3] Considering codewords composed of up to 4 bytes, the codeword assignment in *RPBC* can be easily accomplished by using a simple brute force calculation. It can be extended to handle longer codeword lengths, allowing for codewords of five or more bytes if required. Yet, in our experiments the longest codeword used at most 4 bytes.

σ, the wavelet tree has $\lceil \log_2 \sigma \rceil$ levels, storing n bits overall per level. This makes up a total of $n\lceil \log_2 \sigma \rceil$ bits, that is, the same as a plain representation of S.

To extract any $S[i]$, the wavelet tree starts at the root bitmap $B[1, n]$. If $B[i] = 0$, then $S[i]$ belongs to the left half of the alphabet (i.e., $S[i] < \sigma/2$) and we go to the left child, otherwise we go to the right child. Now, on the left child, the symbol $S[i]$ has been mapped to position $i' \leq i$, which is the number of 0s in $B[1, i]$. This is called $rank_0(B, i)$. Similarly, we move to $i' = rank_1(B, i)$ when going to the right child.

Operation *rank* on bitmaps $B[1, n]$ can be solved in constant time using $o(n)$ bits on top of B (Jacobson 1989). If we create *rank* structures for all the bitmaps, we can compute any $S[i]$ in time $O(\log\sigma)$ and using $n\log\sigma + o(n\log\sigma)$ bits of space. A similar algorithm computes $rank_c(S, i)$, the number of occurrences of symbol c in $S[1, i]$, in time $O(\log\sigma)$.

The operation complementary to *rank* is $select_c(S, j)$, which gives the position of the j-th occurrence of c in S. On binary sequences, *select* can be solved also in constant time with $o(n)$ extra bits (Clark 1996; Munro 1996). If we give *select* support to the bitmaps, we can also compute $select_c(S, j)$ in time $O(\log\sigma)$, by an upwards traversal from the leaf that corresponds to symbol c, to the root.

The space required by the wavelet tree can be reduced to the zero-order entropy of S in two ways. One is changing the balanced tree by a Huffman-shaped tree (Grossi et al. 2003), according to the frequencies of the symbols in S. Another is to use a compressed bitmap representation that also gives constant-time *rank* and *select* (Raman et al. 2002).

Multi-ary wavelet trees were introduced by Ferragina et al (2007). As the tree is not binary, it stores sequences over small alphabets, rather than bitmaps, on the nodes. In theory the space is the same but the time can be divided by $O(\log\log n)$. No practical implementation of this idea exists.

Claude and Navarro (2008) explored the idea of S being the sequence of word identifiers of a text database. Then the wavelet tree represents S within its zero-order entropy (plus some overhead) and allows accessing S at any position. Furthermore, the inverted list of the positions of any word w is obtained with $select_w(S, 1)$, $select_w(S, 2)$, and so on. Arbitrary positions of the list can also be obtained in order to simulate various list intersection algorithms. The best space/time performance was obtained by combining Huffman shape with compressed bitmap representations. Still, the compression ratio obtained when applied to English texts was around 50%.

4 Wavelet trees on bytecodes

4.1 Conceptual description

Our proposal, called *Wavelet trees on bytecodes*, can be applied to any prefix-free byte-oriented encoding technique (such as all those mentioned in Sect. 2). Basically the idea is to reorganize the different bytes of each codeword, placing them in different nodes of a wavelet-like tree (wavelet tree from now on, for shortness). 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 composed of more than one byte. 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 the 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 as $\langle w_1, w_2 \ldots w_n \rangle$. Let us call cw_i the codeword representing word w_i. Note 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 $\langle c_i^1 \ldots c_i^m \rangle$ were m is the size in bytes of codeword cw_i. The root node of the tree is formed by the sequence of bytes $\langle c_1^1, c_2^1, c_3^1 \ldots c_n^1 \rangle$. 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 with more than one byte. Assume there are r words in the source text encoded by codewords (longer than 1 byte) starting with the byte $x : cw_{i_1} \ldots cw_{i_r}$. Then the node x will store the sequence $\langle c_{i_1}^2, c_{i_2}^2, c_{i_3}^2 \ldots c_{i_r}^2 \rangle$. 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 $\langle c_{j_1}^3, c_{j_2}^3, c_{j_3}^3 \ldots c_{j_k}^3 \rangle$ of all the third bytes of codewords $cw_{j_1} \ldots cw_{j_k}$ 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 is equal to the number of bytes of the longest codewords assigned by the encoding scheme.

Figure 1 shows a small example where we built a WTBC[4] from the text "`LONG TIME AGO IN A GALAXY FAR FAR AWAY`", and the source alphabet of words $\Sigma = \{\texttt{A}, \texttt{AGO}, \texttt{AWAY}, \texttt{FAR}, \texttt{GALAXY}, \texttt{IN}, \texttt{LONG}, \texttt{TIME}\}$. After obtaining the codewords for all the words in the text, using a known encoding technique, we reorganize their bytes in the WTBC data structure following the arrangement explained. The first byte of each codeword is placed 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 B_2, because it is the third word in the root node having b_2 as first byte. Its third byte is in node B_2B_4 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 Fig. 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 B_4, which is the child node of the root where the second bytes of all the codewords starting with byte b_4 are stored. Using a byte-wise *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 second byte in node B_4. In the next level, $B_4[2] = b_5$, 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 $rank_{b_5}(B_4, 1) = 1$ tells us that the third byte of that word will be in node B_4B_5 at position 1. One level down, we obtain $B_4B_5[1] = b_2$, and now the obtained sequence $b_4b_5b_2$ is a complete codeword according to the encoding scheme. It corresponds to "`GALAXY`", which therefore is the 6-th word in the source text.

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

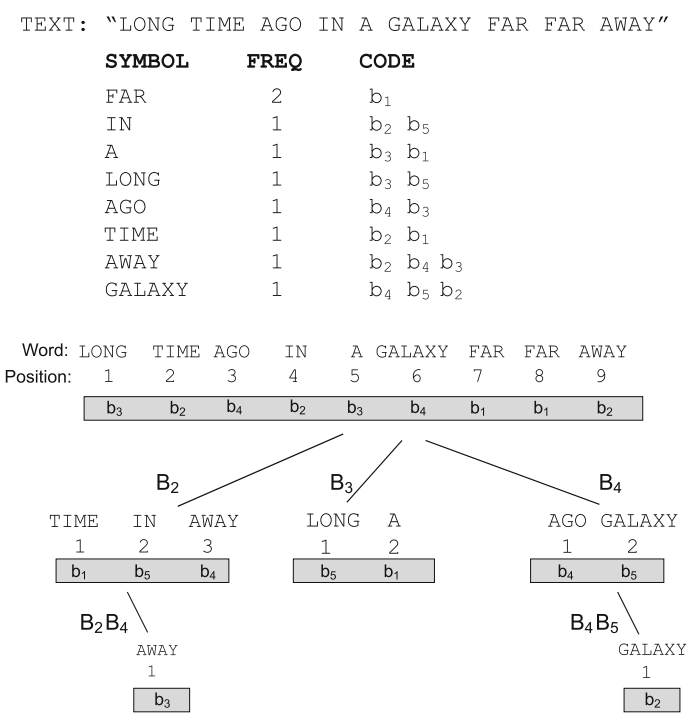

Fig. 1 Example of WTBC data structure for a short text

This process can be used to recover any word. Note that this mechanism gives direct access and random decompression capabilities to any encoding method, including those that do not mark the codeword boundaries. 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 Fig. 1, we start by finding out its codeword, which is $b_2b_4b_3$. Therefore, the search will start at the node B_2B_4, which holds all the codewords starting with b_2b_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 $select_{b_3}(B_2B_4, 1) = 1$ tells us that the first occurrence of our codeword is the first of all codewords starting with b_2b_4, thus in the node B_2 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 the first byte b_4 occurs in the node B_2, we perform $select_{b_4}(B_2, 1) = 3$. Now we know that, in the root node, the third byte b_2 will be the one corresponding to the first byte of our codeword. To know where that third byte b_2 is in the root node, we compute $select_{b_2}(root, 3) = 9$. Finally, the result is that the word "AWAY" appears for the first time as the 9-th word of the text. Note 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.

The sum of the space needed for the byte sequences stored in all the nodes of the tree is exactly the same as the size of the compressed text obtained by compressing the text with a word-based compressor using the same encoding technique as that used to build the WTBC data structure. Just a rearrangement has taken place. Yet, a negligible (as little as 0.05%, as we shall demonstrate shortly) amount of extra space is required to store a few pointers that

permit us to keep information of the tree shape. Actually, the shape of the tree is determined by the compression technique, so in many cases it is not necessary to store those pointers, but only the length of the sequence at each node. For example, if we use a canonical PH, it is not necessary to store pointers to maintain the shape of the tree and determine the *i*-th child of a given node in constant time. In the same way, the wavelet trees built using ETDC or RPBC can be navigated without the need of extra pointers due to the dense assignment of codewords, which causes that all the nodes with children are contiguously located in the wavelet tree. If an arbitrary code is used, the use of pointers or bitmaps may be required to determine which node is the *i*-th child of a given node.

In addition, some extra space can be used to support fast *rank* and *select* operations over the byte sequences.

4.2 Implementation of bytewise *rank* and *select*

We explored different alternatives to implement *rank* and *select* operations over byte sequences, due to their importance on the efficiency of the final structure.

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 searches to be carried out more efficiently than in classically compressed files. In both cases we perform sequential searches, but with WTBC these searches process a reduced portion of the file. Likewise, it is possible to access the text at random, even using non-synchronized codes such as PH and RPBC, much faster than scanning the file from the beginning.

Furthermore, it is possible to drastically improve the performance of *rank* and *select* operations at a very moderate cost in extra space, by adapting well-known techniques (Jacobson 1989). Given a sequence of bytes $B[1, n]$, we use a two-level directory structure, dividing the sequence into blocks of size b and superblocks of size sb. The first level stores the number of occurrences of each byte[5] 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, and hence can be represented with fewer bits. We use integers for superblock values and short integers for block values.

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 the values stored in the corresponding block and superblock for byte b_i. Instead of $O(n)$, this structure answers *rank* in time $O(b)$.

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 values stored in the superblocks, then those in the blocks inside the right superblock, and finally complete the search with a sequential scan in the right block. The time is $O(b + \log n)$.

There is a space/time tradeoff associated to parameter b. The shorter the blocks, the faster the sequential counting of occurrences of byte b_i. In addition, we can speed up *select* operations by storing the result obtained for the last query. Since it is common to perform several $select_{b_i}(B,j)$ operations for the same byte value b_i and consecutive j values, for instance when finding all the occurrences of a word, this stored value can be used when the previous occurrence of the byte value is located in the same block than the one sought.

[5] Actually, only for bytes that appear in the sequence.

Hence, instead of searching from the first position of the block, we can start the sequential search from the position of the previous occurrence. This improved performance in practice. Another related improvement we tried was exponential instead of binary searches, but this did not have much effect.

With this solution we obtain better overall performance in practice than using other alternatives to compute *rank* and *select* over arbitrary sequences, as shown by Ladra (2011). We remark that this is not the general case, but it holds for our particular application, due to the frequency distribution of words and the higher relevance of *access* and *select* operations compared to *rank* in the operations that emulate an inverted index.

4.3 Construction algorithm

The construction algorithm performs two passes over the source text. In the first pass we obtain the vocabulary and the model (frequencies), and then assign codewords using any prefix-free 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 in 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 precompute how many nodes will form the tree and the length of the sequence of each node before the second pass starts, as it is determined by the encoding scheme and the frequencies of the words of the vocabulary. Then, the nodes can be allocated according to these sizes 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 obtain the WTBC representation as the concatenation of the sequences of all the nodes in the wavelet tree, and we add a header with the assignment between the words of the vocabulary and their codewords, determined by the encoding technique employed. In addition, WTBC data structures include the length of the sequence for all the nodes of the tree and some extra information, if needed, of the shape of the tree. This information depends on the encoding method used; if ETDC is the chosen technique, then there is no extra information to maintain, whereas if we reorganize the compressed text of PH, then a few extra bytes representing the canonical Huffman tree are needed.

Algorithm 1 shows the pseudocode of this procedure, where the input is the source text we want to represent and the output is the WTBC data structure generated.

5 Access and search algorithms

In the previous section we have described the new data structure WTBC. We showed how it is navigated using a small example. In this section we detail the general algorithms for accessing to any position of the text and extracting the word located at that position, as well as those for searching for patterns in the text represented by the data structure.

5.1 Random extraction

Operation *extract* is vital for a structure that replaces the text, as the latter is not available otherwise. This operation allows one to decompress portions of the text, starting at any word offset, or to recover the whole original text.

Algorithm 1 Construction algorithm of WTBC

```
Input: T, source text
Output: WTBC representing T
voc ← first-pass(t)
sortByFrequency(voc)
totalNodes ← calculateNumberNodes()
forall the node ∈ totalNodes do
    length[node] ← calculateSeqLength(node)
    wt[node] ← allocate(length[node])
    marker[node] ← 1
end
forall the word ∈ T do
    cw ← code(word) (= c^1 c^2 ...)
    currentnode ← rootnode
    for i ← 1 to |cw| do
        j ← marker[currentnode]
        wt[currentnode][j] ← c^i
        marker[currentnode] ← j + 1
        currentnode ← child(currentnode, c^i)
    end
end
return concatenation of node sequences, vocabulary, and length of node sequences
plus some extra information for the compression technique if needed
```

We first explain how a single word is extracted using the WTBC data structure, and in the next section we generalize the algorithm such that longer sequences of the text can be extracted.

To extract a random text word j, we access the j-th byte of the root node sequence to obtain the first byte of its 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 B_i, the one corresponding to words with b_i as first byte. All the codewords starting with that byte b_i store their second byte in B_i, so we count the number of occurrences of byte b_i in the root node before position j by using a *rank* operation, $rank_{b_i}(root, j) = k$. Thus k is the position in the child node B_i of the second byte of the codeword. We repeat this procedure as many times as the length of the codeword, as we show in Algorithm 2 (which also defines operation *fullaccess*(x) as the one returning the codeword at position x).

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

We can also decompress backward or forward from a given position. For instance, if we need to return a snippet consisting of r words around the occurrence of a word at position

Algorithm 2 Extract x (*Fullaccess* if it returns *cw* instead of *w*)

```
Input: x, position in the text
Output: w, word at position x in the text
currentnode ← rootnode
c ← wt[currentnode][x]
cw ← ⟨c⟩
while cw is not completed do
    x ← rank_c(currentnode, x)
    currentnode ← child(currentnode, c)
    c ← wt[currentnode][x]
    cw ← cw||c
end
w ← decode(cw)
return w
```

p we can follow the same algorithm starting with the entries at positions $[p - r, p + r]$ in the root node.

5.2 Full text decompression

Since WTBC represents the text, we must be able to recover the original text from its data structures. After loading the vocabulary and the whole structure of the WTBC, a full recovery of the text consists in decoding sequentially each entry of the root.

Instead of extracting each word individually, which would require $(\ell - 1)$ *rank* operations for each word (ℓ being the length of its codeword), we follow a faster procedure that avoids all those *rank* operations. Since all the nodes of the tree will be processed sequentially, we can gain efficiency if we maintain pointers to the current first unprocessed entry of each node, similarly to the markers used at construction time (Sect. 4.3). Once we obtain the child node where the codeword of the current word continues, we can avoid unnecessary *rank* operations because the next byte of the codeword will be the next byte to be processed in the corresponding node. Except for this improvement, the procedure is the same as the one explained in Sect. 5.1. Its pseudocode is given in Algorithm 3.

5.2.1 Starting the decompression at a random position

It is also possible to *extract* a portion of the text, starting from a random position different from the first position of the text. The algorithm is the same as the one described in Algorithm 3, which retrieves the whole original text, except for the initialization of the markers. If we do not start the decompression of the text from the beginning, we cannot initialize the markers with the value 1 for each node, they must be initialized with their corresponding values, that are at first unknown. Hence, we start the algorithm with all the markers uninitialized. During the top-down traversal of the tree performed to obtain the codeword of each word, the marker of a node might not contain the value of the next byte to be read. Thus, if the marker is uninitialized, a *rank* operation is performed to establish that value. If the marker is already initialized, the *rank* operation is avoided and the value contained in the marker is used. At most t *rank* operations are performed, being t the total number of nodes of WTBC data structure.

Algorithm 3 Full text decompression

```
Output: T, original text represented by the WTBC data structure
forall the node ∈ totalNodes do
  | marker[node] ← 1
end
T ← ε
for pos ← 1 to length[rootnode] do
  | currentnode ← rootnode
  | c ← wt[currentnode][pos]
  | cw ← ⟨c⟩
  | while cw is not completed do
  |   | currentnode ← child(currentnode, c)
  |   | x ← marker[currentnode]
  |   | c ← wt[currentnode][x]
  |   | marker[currentnode] ← x + 1
  |   | cw ← cw||c
  | end
  | T ← T || decode(cw)
end
return T
```

5.3 Searching

As already mentioned, WTBC data structure provides some implicit indexing properties to the compressed text. Hence, it enables some search operations in a more efficient way than over the text compressed with a regular compressor.

5.3.1 Counting word occurrences

If we want to *count* the occurrences of a given word, we can just 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.

For instance, if we want to count how many times the word "TIME" occurs in the text of the example in Fig. 1, we first notice that its codeword is b_2b_1. Then, we just count the number of times its last byte b_1 appears at node B_2 (since the first byte of its codeword is b_2). Analogously, to count the occurrences of the word "GALAXY", we obtain its codeword $b_4b_5b_2$, and count the number of times its last byte b_2 appears at node B_4B_5 (since the first bytes of its codeword are b_4b_5). The pseudocode is presented in Algorithm 4.

The main advantage of this procedure is that we count the number of times that a byte appears within a node, instead of processing the whole text. Generally, leaf nodes are not large and the procedure is much faster than searching the regular compressed text, while using essentially the same space. In addition the cost in time is drastically reduced if we include structures to support efficient *rank* operations on the bytes stored at the node.

An extension to the *count* operation consists in counting the number of times a word appears in a *range* within the text collection. This is relevant for handling hierarchical, versioned, or temporal databases, for example. To count the number of occurrences of word w between text words i and j, we use operation $fullrank(cw, i)$, which maps position i towards the leaf of $cw = code(w)$, that is, it counts the number of occurrences of codeword cw in $T[1, i]$ (just as symbol *rank* operation on sequences). Then counting in range $[i, j]$ is efficiently implemented as $fullrank(cw, j) - fullrank(cw, i-1)$. Algorithm 5 gives the pseudocode for *fullrank*.

Algorithm 4 Operation *count*

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

Algorithm 5 Operation *fullrank*

Input: x, a position in the text
Input: cw $(= c^1c^2\ldots)$, a codeword
Output: y, the number of occurrences of cw up to position x in the compressed text
$currentnode \leftarrow rootnode$
$y \leftarrow rank_{c^1}(currentnode, x)$
for $i \leftarrow 1$ **to** $|cw| - 1$ **do**
  $currentnode \leftarrow child(currentnode, c^i)$
  $y \leftarrow rank_{c^{i+1}}(currentnode, y)$
end
return y

5.3.2 Locating individual words

As explained in the example of Sect. 4, to *locate* all the occurrences of a given word, we start by 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, $c^{|cw|}$, occurs at position j in the leaf node, then the previous byte $c^{|cw|-1}$ of that codeword will be the j-th one occurring in the parent node. We proceed in the same way up in the tree until reaching the position x of the first byte c^1 in the root node. Thus x is the position of the first occurrence of the word searched for. The basic procedure, also called *fullselect* when receiving the codeword instead of the word, is shown in Algorithm 6.

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, as explained in Sect. 4.2. Furthermore, to find all the occurrences of a word in the text range $[i, j]$, we use *fullrank* to find the range of occurrences of the word in that range, and then *locate* only those occurrences.

5.3.3 Counting and locating phrase patterns

It is also possible to search for a *phrase pattern*, that is, a pattern composed of several words. We locate all the occurrences of the least frequent word in the root node, and then check if all the first bytes of each codeword of the pattern match with the previous and next entries at the root node. If all the first bytes of the codewords of the pattern match, we verify their complete codewords around the candidate occurrence by performing the corresponding top-down traversal over the tree, until either a byte fails to match the search pattern or we find the complete phrase pattern.

This algorithm describes both the procedure for counting and locating the occurrences of a given phrase pattern, so both operations are equally time-costly. Its pseudocode is detailed in Algorithm 7.

In addition to this *native method* for searching for phrase patterns over the WTBC, it is interesting to remark that WTBC also supports *list intersection* algorithms to search for phrases over the compressed text. Inverted indexes search for phrase patterns by obtaining the lists associated to the words that compose the pattern, and then intersecting those lists. The efficiency of the list intersection is crucial for search engines, and it continues to be an open research problem, where new list intersection algorithms are constantly being proposed (Sanders and Transier 2007; Transier and Sanders 2010; Culpepper and Moffat, 2007, 2010; Barbay et al. 2009). These algorithms can be applied over WTBC by noticing that we can generate any arbitrary entry of the posting list associated to any word on the fly.

Algorithm 6 Locate the j-th occurrence of word w (*Fullselect* if it receives cw instead of w)

```
Input: w, word
Input: j, integer
Output: position of the j-th occurrence of w in the root node
cw ← code(w) (= c^1 c^2 ...)
Let cw = cw'||c, being c the last byte
currentnode ←node corresponding to cw'
for i ← |cw| downto 1 do
    j ← select_{c^i}(currentnode, j)
    currentnode ← parent(currentnode)
end
return j
```

Algorithm 7 Locate all occurrences of phrase $w_1 w_2 \ldots w_p$

```
Input: w_1 w_2 ... w_p, phrase pattern
Output: positions of all the occurrences of phrase w_1 w_2 ... w_p in the root node
for i ← 1 to p do
    cw_i ← code(w_i) (= c_i^1 c_i^2 ...)
end
Let w_j be the least frequent word of the phrase pattern and minpos ← j
cw ← cw_j, (= c^1 c^2 ...)
m ← count(w_j)
positions ← ∅
for z ← 1 to m do
    j ← fullselect(cw, z)
    i ← 1
    while i ≤ p do
        if rootnode[j − minpos + i] ≠ c_i^1 then break
        i ← i + 1
    end
    if i > p then
        i ← 1
        while i ≤ p do
            if fullaccess(j − minpos + i) ≠ cw_i then break
            i ← i + 1
        end
        if i > p then positions ← positions ∪ {j − minpos + 1}
    end
end
return positions
```

As an example, the pseudocode of a *set-vs-set*-type intersection algorithm implemented over WTBC is shown in Algorithm 8. Note that the *native* method we explained first, however, has been especially adapted to take advantage of WTBC data structures. For instance, it will not be necessary to make complete top-down traversals over the tree to check an occurrence in the longest list if we detect a mismatch at an upper level of the tree on the first codeword bytes of some word. In the next section we experimentally show that our *native* method outperforms the *set-vs-set*-type list intersection algorithm when searching for phrases over a real text.

6 Experimental evaluation

This section presents the experimental performance of the new method proposed, WTBC. We first show that WTBC is much more efficient than the sequential representation of the compressed text when search functionality is required. This is due to the implicit indexing properties that WTBC provides.

We also compare our WTBC data structure with explicit inverted indexes, when using the same amount of space. More concretely, we use block-addressing compressed inverted indexes (Navarro et al. 2000; Zobel et al. 1998), since they are the best choice, as far as we know, when little space is available. Our results demonstrate that using WTBC is more convenient than trying to use very space-efficient inverted indexes. In addition to this comparison, we compare the performance of WTBC with some recent compressed indexes of the literature.

Section 6.1 describes the collections and the machines used in the experiments. Section 6.2 compares the new technique with the original compression methods. Section 6.3 compares our proposal with indexing structures, that is, inverted indexes and other compressed indexes.

Algorithm 8 List intersection

```
Input: w_1, word
Input: w_2, word
Output: positions of the occurrences of the pattern w_1 w_2
cw_1 ← code(w_1)
cw_2 ← code(w_2)
x_1 ← fullselect(cw_1, 1)
x_2 ← fullselect(cw_2, 1)
while max{x_1, x_2} ≤ n do
    if x_1 + 1 = x_2 then
        report occurrence
        x_1 ← fullselect(cw_1, fullrank(cw_1, x_1) + 1)
        x_2 ← fullselect(cw_2, fullrank(cw_2, x_2) + 1)
    else
        if x_1 + 1 < x_2 then
            x_1 ← fullselect(cw_1, fullrank(cw_1, x_2 − 2) + 1)
        if x_1 + 1 > x_2 then
            x_2 ← fullselect(cw_2, fullrank(cw_2, x_1 − 1) + 1)
end
return j
```

6.1 Experimental framework

We used a large corpus (ALL), with around 1 GB, created by aggregating the following text collections: AP Newswire 1988 and Ziff Data 1989–1990 (ZIFF) from TREC-2, Congressional Record 1993 (CR) and Financial Times 1991–1994 from TREC-4[6], in addition to the small Calgary corpus[7]. We also used CR and ZIFF corpora individually to have smaller corpora to experiment with. Table 1 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 the number of words that compose the corpus, and the fourth the number of different words in the text.

To create our vocabulary, we split the text into words (a maximal sequence of alphanumerical characters) and separators (a sequence of non-alphabetical characters between two contiguous words). Then, both words and separators were encoded. We used the spaceless word model (Moura et al. 2000) to model the separators. That is, if a word is followed by a single space, we just encode the word, otherwise both the word and the separator are encoded. As a result, the vocabulary is formed by all the different words and all the different separators, excluding the single white space. We did not perform any additional pre-processing of the text. Therefore, operations such as case-folding, stemming, etc. were not considered.

Two different machines have been used for the experiments. In Sect. 6.2 we used an isolated Intel®Pentium®-IV 3.00 GHz system (16Kb L1 + 1024Kb L2 cache), with 4 GB dual-channel DDR-400Mhz RAM. It ran Debian GNU/Linux (kernel version 2.4.27). The compiler used was gcc version 3.3.5 and -O9 -m32 compiler optimizations were set. In Sect. 6.3 we used an isolated Intel®Xeon®-E5520@2.26GHz with 72GB-DDR3@800MHz

[6] http://trec.nist.gov.

[7] We concatenated in a single file a subset of the files from the Calgary collection that includes only the text files: book1-2, bib, news, and paper1-6. It is available at ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus.

Table 1 Description of the corpora used

Corpus	Size (bytes)	No. of 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

RAM. It ran Ubuntu 9.10 (kernel 2.6.31-19-server), using gcc version 4.4.1 with `-O9 -m32` options. Time results refer to CPU user time.

In Sect. 6.3 we analyze the search performance of our technique over the ALL corpus. We use 8 sets of 100 test patterns. The first four sets are composed of single-word patterns with different frequency ranges: W_a, W_b, W_c, and W_d with words occurring respectively [1, 100], [101, 1000], [1001, 10000], and $[10001, \infty]$ times. Those words were chosen at random from the vocabulary following the model by Moura et al. (2000) were each word is sought with uniform probability. The overall number of occurrences for such sets are 5,679; 30,664; 258,098; and 2,273,565 respectively. The other four sets, P_2, P_3, P_4, and P_6, consist of phrase-patterns composed of 2, 3, 4, and 6 words respectively that were randomly chosen from the text. We ensured that phrases consisting only of stopwords[8] were not included in the sets P_i. The number of occurrences of such sets are 201,956; 31,964; 4,415; and 144 respectively.

6.2 Comparison with regular text compressors

As already explained, WTBC can be built over different byte-oriented encoding schemes. The new proposed structure rearranges the bytes of the codewords that conform the compressed text in a tree-shaped data structure. In this section, we build the WTBC structure over PH, ETDC, and RPBC (see Sect. 2) obtaining respectively what we call *WTPH, WTDC*, and *WTRPBC*.

We measure how the reorganization of the codeword bytes induced by our proposal affects the main compression parameters, such as compression ratio and both compression and decompression times. We also show the searching capabilities of the new WTBC-based structures, including results for *count* and *locate* operations.

Table 2 shows that compression ratio is essentially not affected, as expected. There is a very slight loss of compression (close to 0.01%), due to the storage of the tree shape. In this experiment addressing just compression, no blocks and superblocks are built on WTPH, WTDC, and WTRPBC.

Tables 3 and 4 show the compression and decompression times obtained using the WTBC data structure. The absolute differences in time are similar both at compression and decompression: WTBC worsens the time by around 0.1 s for CR corpus, 0.4 s for ZIFF corpus and 3.5 s for ALL corpus. This is because with WTBC strategy, compression and decompression operate with data that is not sequentially stored in main memory. For each word of the text, a top-down traversal is carried out on the tree, so the benefits of cache and spatial locality are reduced. This is more noticeable at decompression than at compression, since in the latter the overhead of parsing the source text blurs those time differences. Therefore, compression time is almost the same (2–4% worse) as for the sequential

[8] We used a list of stopwords (prepositions, articles, etc.) available at http://vios.dc.fi.udc.es/indexing/wsi/download.html.

Table 2 Compression ratio (in %) of WTBC built using PH, ETDC and RPBC versus their regular counterparts for three different natural language texts

	PH	ETDC	RPBC	WTPH	WTDC	WTRPBC
CR	31.057	31.941	31.062	31.060	31.948	31.065
ZIFF	32.876	33.770	32.883	32.878	33.774	32.885
ALL	32.833	33.659	32.845	32.835	33.662	32.847

Table 3 Compression time (s)

	PH	ETDC	RPBC	WTPH	WTDC	WTRPBC
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

Table 4 Decompression time (s)

	PH	ETDC	RPBC	WTPH	WTDC	WTRPBC
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

compression techniques. That is, almost the same time is required to build the WTBC from the text than just to compress it. In decompression, those gaps increase and WTBC structures become around 20–25% slower than the regular counterparts.

We now compare the search results obtained by WTBC with those obtained when performing searches over text compressed with PH, ETDC, and RPBC.[9] We focus in two main search operations: we measure the user time required to *count* all the occurrences of a pattern (in milliseconds) and to *locate* all those occurrences (in seconds). We run our experiments over the largest corpus, ALL, and show the average time to search for 100 distinct words randomly chosen from the vocabulary (we removed stopwords, since it makes no sense to search for them). We present the results obtained by the compression methods PH, ETDC, and RPBC; and by the WTBC data structure implemented without blocks and superblocks (WTPH, WTDC, and WTRPBC). We also include alternatives WTPH+, WTDC+, and WTRPBC+, which correspond to wasting 1% of extra space in the WTBC (i.e., 1% of the size of the original collection T) on block and superblock structures to speed up the operations.

To adjust WTBC to a desired extra space, we proceed as follows. Firstly, being N the number of bytes of the indexed sequence, the overall size of the *rank/ select* structures (E) is roughly estimated as $E = (K_s \times 256)\, N/(s \times b) + (K_b \times 256)\, N/b$, where K_s is the byte size of the superblock counters (in our case 4, the size of an unsigned int) and K_b is the byte size of the block counters (in our case 2, the size of an unsigned short int). Therefore, we

[9] We used our own implementations to search within compressed text. For PH the searcher marks the searched pattern in the vocabulary and then simulates decompression (Moura et al. 2000). For ETDC we used a Horspool-based searcher (Horspool 1980) available at http://vios.dc.fi.udc.es/codes. Finally, for RPBC we implemented the Horspool-based algorithm from RPBC's authors (Culpepper 2007, p. 100).

Table 5 Search performance for the ALL corpus

	Memory usage (%)	Count (ms)	Locate (s)
PH	35.128	2605.600	2.648
ETDC	35.955	1027.400	0.940
RPBC	35.140	1996.300	2.009
WTPH	35.129	238.500	0.754
WTDC	35.957	221.900	0.762
WTRPBC	35.141	238.700	0.773
WTPH+	36.113	0.015	0.123
WTDC+	36.953	0.015	0.129
WTRPBC+	36.086	0.015	0.125

obtain $b = N\,[256 \times (K_s/s + K_b)]/E$. By fixing s (to a small value) and the expected extra space E, we obtain a first approximation for the value of b. Finally, we manually fine-tune b until we reach the expected 1% extra space. In our case, we obtained $b = 21{,}000$ bytes and superblocks of $s = 10$ blocks.

Table 5 shows time results for *count* and *locate* for each method and also the amount of memory they need in order to solve those queries. To have a fairer comparison, all the compared alternatives maintain the vocabulary of words using a hash table with identical parameters and data structures. Its space requirements are also included within the values in Table 5.

We observe that, even when no extra space is used for the block and superblock structures, the use of WTBC data structure improves search performance by an order of magnitude compared to scanning regular compressed text, especially for counting the number of occurrences. By using just 1% of extra space for *rank* and *select* support, searching times improve much more.

On the other hand, the time performance of the different realizations of WTBC and WTBC+ is very similar.

6.3 Comparison with other indexes

As explained, the reorganization carried out by the WTBC data structure brings some (implicit) indexed search capabilities into the compressed file. It improves searches in such a way that it becomes competitive with other indexing structures. In this section we compare the search performance of WTPH+ with two block-addressing compressed inverted indexes (Navarro et al. 2000), a bit-oriented Huffman-shaped wavelet tree as described in Sect. 3 (Grossi et al. 2003; Claude and Navarro 2008) and a word-based compressed index based on suffix arrays (Brisaboa et al. 2008b), working in main memory.

The inverted indexes used are block-grained: they assume that the indexed text is partitioned into blocks of size b, and for each term they keep a list of occurrences that stores all the block-ids in which that term occurs.

The first compressed inverted index, *II-scdc*, is built over text compressed with SCDC, whereas the second index, *II-huff*, is built over text compressed with binary Huffman. We use SCDC for one of the inverted indexes due to its efficiency at decompression and searches, while achieving a good compression ratio (33.02% for the ALL corpus). For the other inverted index we use *Huffword*, which consists in the well-known bit-oriented

Huffman coupled with a word-based modeler (Witten et al. 1999). It obtains better compression ratios than SCDC (29.22% for ALL corpus), but it is much slower at both decompression and searches. For our two alternatives, II-scdc and II-huff, we built several indexes where we varied the block size, which brings an interesting space/time tradeoff. If we use the slower Huffman coding, we can exchange the space gain by a denser sampling, so that shorter blocks will be scanned. If we use the faster SCDC, scanning will be faster but it will be performed on longer blocks.

To reduce the size of the index, the lists of occurrences were compacted using Rice codes (Witten et al. 1999) for the shorter lists and bitmaps for the longer ones. We follow a list compression strategy (Moffat and Culpepper 2007; Culpepper and Moffat 2010) where the list L of a given word is stored as a bitmap if $|L| > u/8$, being u the number of blocks. No sampling is used. As the posting lists are compressed with variable-length codes, intersection of lists is performed using a *merge*-type algorithm along with the decoding of such lists (that is, the lists are intersected as they are sequentially decoded). We have tried other strategies to deal with the inverted lists, including sampling for direct access (Sanders and Transier 2007; Transier and Sanders 2010; Culpepper and Moffat 2007, 2010) or codes that are only slightly less space-efficient but faster to decode (Ding et al. 2010; Anh and Moffat 2005; Zukowski et al. 2006; Yan et al. 2009). Yet, search times were practically unaffected as they depend mainly on the block size. The reason is that most of the time is spent in scanning blocks and not on traversing lists. Adding sampling or a less space-efficient code wastes some space that is much better used in a denser sampling with reduced block size.

In addition, we compare WTBC with other compressed indexes that support fast searches for words or phrases and occupy space comparable to our WTBC. We will not compare our proposal with classical full-text compressed indexes that can search for any pattern (not only words). This comparison is unfair because these indexes offer stronger functionality, and require much more space: around 40–60% for natural language text (Ferragina et al. 2009). Instead, we first compare WTPH+ with a binary Huffman-shaped wavelet tree (Grossi et al. 2003; Claude and Navarro 2008) representing the sequence of words of the text, denoted *WTbitHuff*, and also with a word-based version of a classical compressed index such as the word-based Compressed Suffix Array (*WCSA*) (Brisaboa et al. 2008b).

For the comparison, we create several Huffman-shaped wavelet trees with different sizes, varying the size for the extra structure used to compute fast binary *rank* and *select* operations. We used the implementations of WTbitHuff available at the Compact Data Structures Library (libcds)[10]. For WCSA, we create several indexes with different sizes, varying construction parameters such as the sample periods t_A, t_A^{-1} and t_Ψ for A, A^{-1} and Ψ, which also gives an interesting space/time tradeoff.

To illustrate the behavior of WTBC, we compute search times for the variant built over PH (WTPH+), since it obtains the best space/time results.

Note that, for the experiments of this section and the following ones, the vocabulary is not stored using a hash table, as in the previous section. We store the vocabulary in alphabetic order, so that we can obtain the codeword assigned to a word with a binary search over this structure. This solution is lighter than using a hash table, and the WTBC data structure built over the compressed text of the ALL corpus using PH requires just 33.32% of the original text to solve any query (without any *rank* and *select* extra structure). Our method cannot use less than that memory to represent the ALL corpus in an indexed

[10] http://libcds.recoded.cl/.

way, whereas other indexes, such as WCSA or the inverted index using Huffman coding (II-huff) can still go beyond our lower bound.

We built several configurations for WTPH+ using different sizes for the *rank* and *select* structure, so that we can show the space/time tradeoff obtained by the representation. We compare WTPH+ with the other indexes over the corpus ALL, using the sets of patterns W_a, W_b, W_c, W_d, P_2, P_3, P_4, and P_6 described in Sect. 6.1. We measure the amount of main memory occupied by the indexes, and the time to perform the following search operations:

- *locate*: we measure the time to locate all the occurrences of a pattern.
- *extract*: we measure the time to extract some portions of text of different lengths.
- *display*: we measure the time to display a snippet around all the occurrences of a pattern, which includes the time to locate its occurrences and to extract snippets containing 20 words, starting at an offset 10 words before each occurrence.

Results for both *locate* and *display* operations refer to average time per occurrence (in msec/occurrence). We do not measure *counting* time since it could be solved trivially for word patterns by including the number of occurrences for each word along with the vocabulary (worsening compression ratio by around 0.75 percentage points). WTBC counting times for phrase patterns are similar to locating them; hence, those counting times can be obtained from the figures for *locate* operation. Results for *extract* are measured in time per character extracted (in μs/char).

6.3.1 Locating times

Figure 2 shows the performance of the indexes for locating individual words for scenarios W_a (top left),W_b (top right),W_c (bottom left), and W_d (bottom right). We can observe that WTPH+ obtains the best results, regardless of the frequency of the word sought, when little space is used to index the compressed text.

Compared with inverted indexes, WTPH+ is faster since it directly jumps to the next occurrence, while inverted indexes have to scan the text. When little memory is used, the inverted indexes obtain poor results, since a sequential scan must be performed over large blocks. The worst scenario for WTPH+ is locating low-frequency words, since it must perform a bottom-up traversal of the tree from the deepest leaves, and thus several *select* operations must be carried out. For this scenario W_a, inverted indexes overcome WTPH+ when the index occupies more than 39% of the original text size. This scenario is particularly advantageous for II-scdc inverted index: we are searching for low-frequency words, which have long codewords assigned, over short blocks of SCDC compressed text. Moreover, SCDC enables Boyer-Moore-type searching, which skips bytes during the search, and since the codewords sought are long, the Boyer-Moore algorithm can skip more bytes. For scenarios W_b, W_c, and W_d WTPH+ obtains better times than the inverted indexes, even when using much space.

WTPH+ also outperforms the binary Huffman-shaped wavelet tree (WTbitHuff). Since the alphabet is so large (around 885,000 words) the wavelet tree requires several levels, and thus accessing, counting, and locating the symbols of the sequence become slow. In addition, the tree has a large number of nodes, which require many pointers to maintain the tree shape. Therefore, WTbitHuff uses significantly more space than the zero-order entropy of the text (note that the compression ratio obtained by binary Huffman code over ALL corpus is 28.55%).

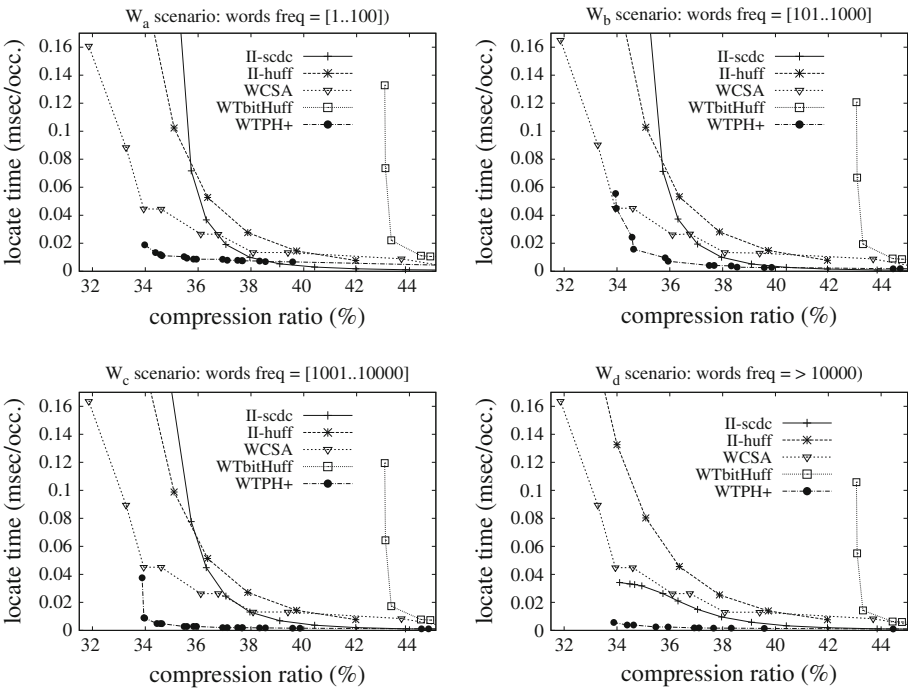

Fig. 2 Time/space tradeoff for *locating* individual words with WTBC strategy over PH against other searching structures, varying the frequency of the word sought

Compared with WCSA, WTPH+ is significantly faster at locating the occurrences of individual words. However, WCSA can achieve lower spaces than WTPH+. WTPH built over ALL corpus occupies 33.32% of the text, when no *rank* or *select* structures are used. In the figures, we illustrate the behavior of several configurations of WTPH+ using a structure for *rank* and *select* operations with varying sample period. When very little space is used for *rank* and *select* structures, the compression ratio obtained gets close to that of WTPH, but WTPH+ becomes very inefficient due to the sparseness in the samples of the *rank* and *select* directory of blocks and superblocks. The efficiency of WCSA also decreases when we use less space, but it can index the same text using less than 33% of space.

Figure 3 shows the performance when locating phrase patterns for scenarios P_2 (top left), P_3 (top right), P_4 (bottom left), and P_6 (bottom right). From the experimental results we can observe that WTPH+ can efficiently locate short phrase patterns (of length 2) but its efficiency decreases for longer patterns. Note that the average time for *locate* is measured in milliseconds per occurrence. Since long phrase patterns are less frequent than the short ones, this average time is worse for long phrase patterns. In addition, when the phrases are long, verifications require to perform ℓ top-down traversals over the tree, being ℓ the length of the phrase. Even if some more false matchings are detected at the root level, those extra *rank* operations worsen the average locating time. Inverted indexes become a better choice to search for long phrase patterns for compression ratios above 37%, as it happened when searching for less frequent patterns: when searching for long phrases, we can skip more bytes during the sequential scan of the blocks. However, WTPH+ is always the preferred solution when little space is used.

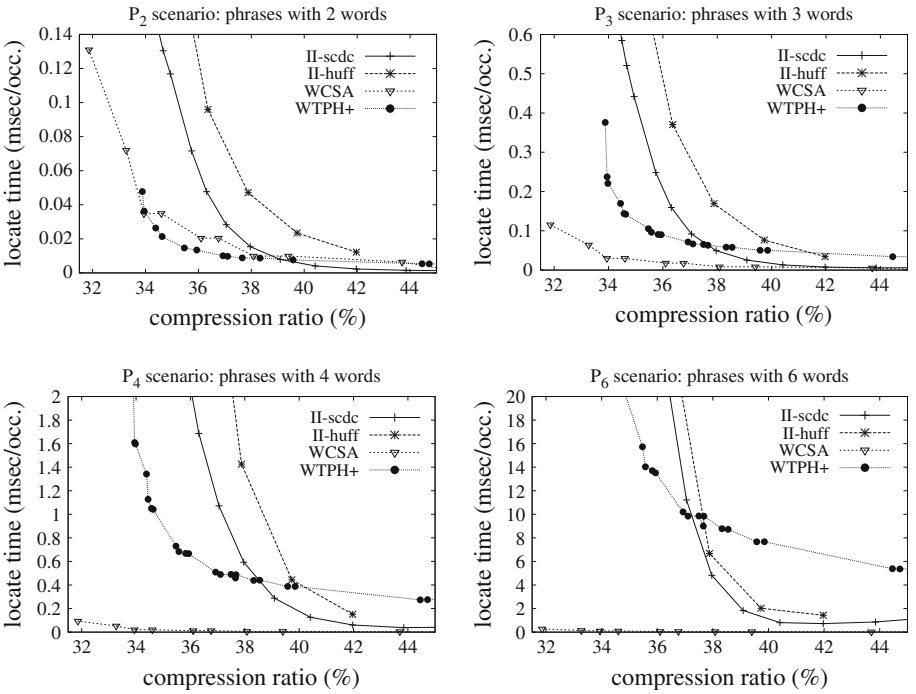

Fig. 3 Time/space tradeoff for *locating* phrases with WTBC strategy over PH against other searching structures, varying the length of the phrase sought

Word-based compressed suffix array clearly outperforms WTPH+ when searching for long phrase patterns. This is an expected result since suffix arrays were designed to efficiently *count* and *locate* all the occurrences of substrings of the text. WCSA is a word-based compressed index based on suffix arrays, hence, it easily recovers all the occurrences of the word phrases of the text. However, WTPH+ still obtains better results than WCSA when searching for phrases composed of two words.

6.3.2 Locating phrase patterns versus list intersection

Recall that, apart from the *native* algorithm presented in Sect. 5.3.3 for locating the occurrences of phrase patterns, other list intersection algorithms could be used. We now compare the performance of the *native* algorithm with the implementation of the *set-vs-set*-type intersection method in the WTBC.

We run our experiments over the ALL corpus and show the average time to search for two different sets of phrase-patterns composed of 2 words. The first set (S_1) contains 100 distinct 2-words phrases randomly chosen from the text, where the most frequent word of each phrase occurs less than 100,000 times in the text. The second test set (S_2) contains 100 distinct phrases composed of two words that were randomly chosen from the vocabulary among all the words of frequency f, such that $1{,}000 \leq f \leq 50{,}000$. Note that the artificially generated phrases in S_2 do not necessarily exist in the text. We present the results obtained for both techniques by WTBC built over PH (WTPH+) using blocks of 21,000 bytes and superblocks of 10 blocks, which waste 1% of extra space, to speed up *rank* and *select* operations.

Table 6 Average times (in msec/pattern) to locate 2-words phrases from the sets S_1 and S_2, for WTPH+ using two different intersection algorithms

Searching technique	S_1	S_2
Native phrase searching algorithm	86.07	28.89
Set-vs-set-like list intersection algorithm	411.30	100.15

In Table 6, we can observe that the best results are obtained by the *native* algorithm when searching for phrases in the WTBC. Remember that this algorithm consists in searching for the occurrences of the least frequent word and then checking the surrounding positions to know whether there is an occurrence of the phrase or not. This can be very efficiently checked by just comparing the first bytes of the codeword in the first level of the WTBC, which permits fast detection of false matchings. If the first bytes match, then we check the bytes at the second level. Only if all the bytes at each level of the tree coincide, we reach the leaf level of the WTBC and check if there is an occurrence of the phrase-pattern. On the other hand, the *set-vs-set*-type list intersection algorithm performs complete top-down traversals of the WTBC, which may be unnecessary.

Note that the *set-vs-set*-type algorithm for list intersection may be faster than the *native* method if we search for a phrase composed of two words, where each word occurs more frequently in one portion of the text. Thus, we will avoid checking all the occurrences of the least frequent word, as the algorithm may skip several occurrences of the word that appears in one portion of the document by jumping to another portion of the document. However, these list intersection algorithms (Barbay et al. 2009) were designed for intersecting lists of documents on bag-of-word queries, where the described situation is more plausible. It seems less likely that this arises in words that are searched for as a phrase. Our experiments show that the effect is far from relevant when we choose random pairs of words.

6.3.3 Range-restricted locating

As mentioned in Sect. 5.3, our WTBC strategy efficiently supports counting and locating the occurrences of patterns within a certain range (for phrases the only way to count the occurrences is to locate them). We now compare its performance with inverted indexes and WCSA. Inverted indexes can find, for each of the involved lists, the first entry that falls within the range, and continue the intersection until leaving the range. This way they support efficient locating in a range, yet they cannot directly count, even for simple word queries. For the WCSA the problem is even harder, as the positions are delivered out of order, so the only way to query within a range is to carry out the full query and then restrict the positions.

We generated random intervals of width n, $n/10$, $n/100$, $n/1{,}000$, and $n/10{,}000$, being $n = 228{,}707{,}250$ the number of words of the text ALL. We configured the indexes to obtain a compression around 36%. For WTPH+ we use blocks of 5,000 bytes and superblocks of 8 blocks, obtaining a compression ratio of 35.94%. II-scdc is tuned to obtain a compression ratio of 36.31%, II-huff obtains 36.36%, and WCSA obtains 36.10%.

We measured the time to *count* and *locate* all the occurrences of 100 distinct patterns from three different sets: W_c, P_3 and P_6, averaging over 20, 1,000, and 5,000 random ranges of each size, respectively. We measured *counting* and *locating* average times per pattern (in msec/pattern).

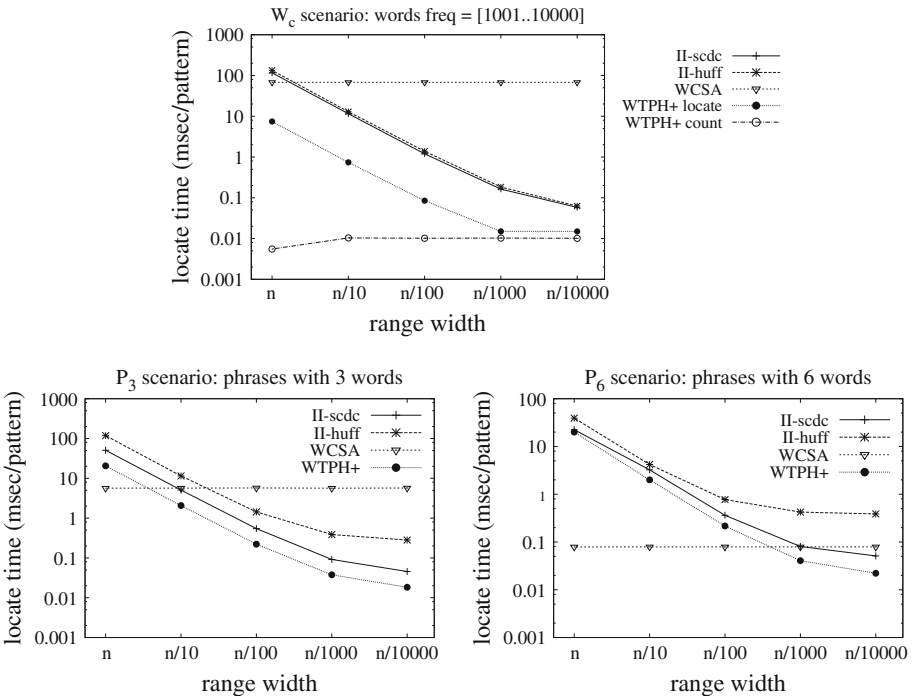

Fig. 4 Time/space tradeoff for *locating* patterns in a range with WTBC strategy over PH against other structures, varying the width of the range considered

Figure 4 shows space/time results for scenarios W_c (top), P_3 (bottom left) and P_6 (bottom right). For W_c we represent separately the times for counting and locating the occurrences of the query word with WTPH+ whereas for phrase-pattern scenarios we only show locating times, as there is no independent algorithm for counting.

We can observe that WTPH+ becomes the most efficient technique to locate patterns in a range as its width decreases, that is, as the query becomes more selective. Even for the most disadvantageous scenario, that is, when searching for the occurrences of long phrase patterns (P_6), WTPH+ outperforms WCSA when restricting to shorter ranges.

Moreover, WTPH+ can efficiently count the number of occurrences of an individual word between two positions of the text by performing just two byte-wise *rank* operations, whereas the other indexing structures must obtain the occurrences and count them.

6.3.4 Extraction times

In this section we study the efficiency of WTBC at extracting portions of text, comparing it with the other two compressed indexes, that is, WCSA and WTbitHuff. We do not compare WTBC with inverted indexes, since these carry out sequential searches on the blocks, and therefore they can display any snippet around occurrences found without any extra time penalty.

We created three sets of intervals $[i, i + w - 1]$, where i is a random position on the sequence of N words that compose the text ($1 \leq i \leq N - w$), and w is the interval width. We tried three different values for w (10, 100, and 1,000 words), so that we start the extraction of text from the i-th word and recover a substring containing the following

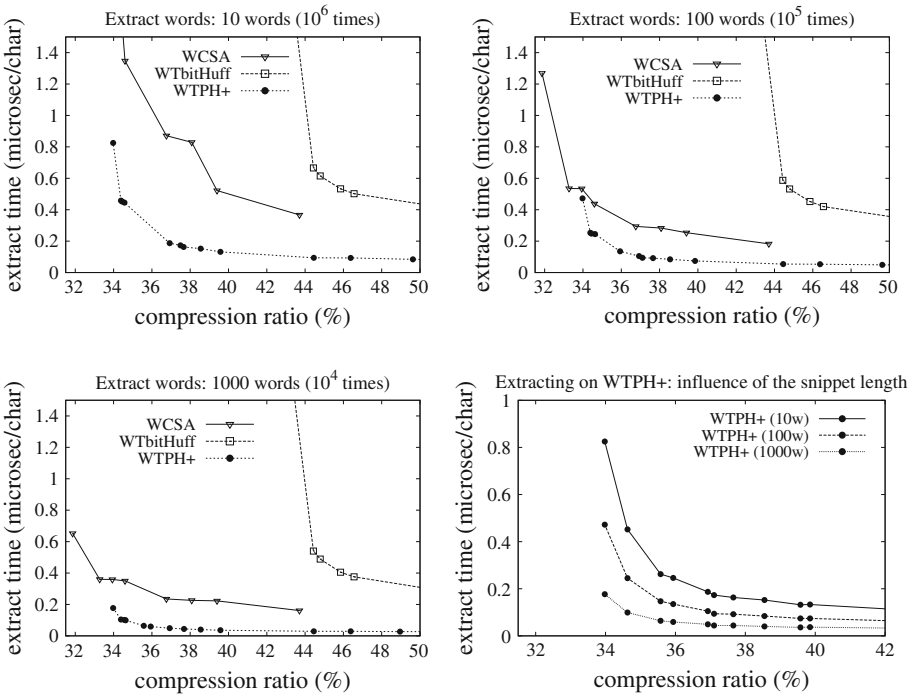

Fig. 5 Time/space tradeoff for *extracting* operation. WTBC strategy over PH is compared with the other compressed indexes, varying the length of the extracted snippet

w words. We will refer to such sets as $10w$, $100w$, and $1000w$, and they contain respectively 10^6, 10^5, and 10^4 intervals. Therefore, we will extract 10^6 substrings consisting of 10 words, 10^5 of 100 words, and 10^4 substrings with 1,000 words respectively. Results for extract are given in microseconds per extracted character (μsec/char).

Figure 5 shows the results for the set $10w$ (top left), $100w$ (top right), and $1000w$ (bottom left). As we can observe, WTBC outperforms the other indexes when extracting snippets. This shows in particular a weak side of the WCSA, which is slow at this task.

Figure 5 (bottom right) shows how the performance of the *extract* operation improves as the length of the snippet increases. Remember from Sect. 5.2.1 that we use one pointer per node to avoid *rank* operations over the tree. These pointers are uninitialized at first, and one *rank* operation is required to set their value for a node if needed. Once its value is established, no more *rank* operations are performed to access a position of that same node. The longer the snippet, the lower the amortized cost per traversed byte. In addition, we can see in the figure that the time depends on the size of the structure that supports *rank* operations. As expected, we obtain better time results if we spend more extra space to speed up this bytewise operation.

6.3.5 Display times

We now show the results for operation *display*, which are analogous to the results obtained for *locate*. Note that the *display* operation consists in first locating the occurrences and then extracting some portion of text around those occurrences. Therefore, again, as long as we set the indexes to use less space, WTPH+ becomes the preferred choice.

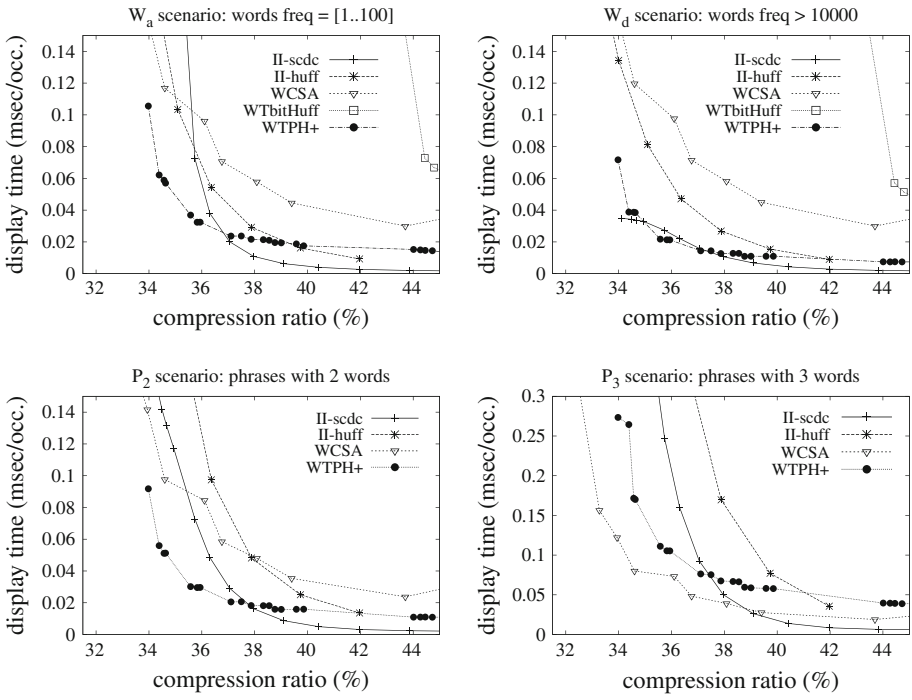

Fig. 6 Time/space tradeoff for *displaying* 20-words snippets. WTBC strategy over PH is compared with the other indexes, for individual words (*top*) and phrase patterns (*bottom*)

Figure 6 shows only the results for some of the scenarios for individual words and phrase patterns, since the results can be obtained by adding locating plus extracting times. More concretely, we show space/times results for scenarios W_a (top left) and W_d (top right), as well as for scenarios P_2 (bottom left) and P_3 (bottom right).

Differences in time between WTBC and inverted indexes are larger when comparing *locate* times than when we compare snippet extraction times. Note that those gaps in snippet extraction time tend to reduce since decompression is faster in inverted indexes than in WTPH+, especially for the inverted index built over SCDC. However, WTBC still obtains better time results when displaying all the occurrences of a word, especially for not very frequent words, where there are fewer snippets to extract.

Compared with the other compressed indexes, the results of WTBC for *display* are slightly better than in the case of the *locate* operation, as the extraction of the text is more efficient in our WTBC strategy than for WCSA or WTbitHuff. For instance, when searching for either single-word patterns or short phrases, we can observe how WTPH+ always outperforms WCSA at *displaying*, whereas their performance is more similar for *locating*. WCSA is again the best choice to *display* some portions of the text around the occurrences of long phrase patterns, but WTBC dominates the space/time tradeoff for the rest of the scenarios: displaying individual words and short phrases.

We remark that our good results compared with inverted indexes 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 many) blocks. As more space is allowed to those structures, both improve in time but the inverted indexes eventually take over WTPH+ (this occurs when both use around 37% of the text size).

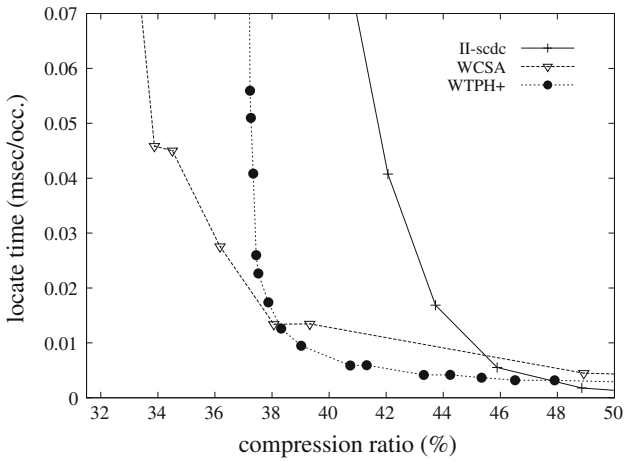

Fig. 7 Time/space tradeoff for *locate* comparing WTBC strategy over PH with other indexes on the INEX corpus

Of course, if sufficient space were given, the inverted indexes could directly point to occurrences instead of blocks and that scanning could be avoided. Yet, as explained in the Introduction, using little space is very relevant for the current trend of maintaining the index distributed among the main memory of several processors. What our experiments show is that WTBC makes better use of the available space when there is not much to spend.

6.4 Scalability

We present now new experiments on a larger text collection: the INEX 2009 Wikipedia Dataset[11]. It consists of a dump of the English Wikipedia created on October 8, 2008 and contains 2,666,190 articles that make up 50.7GiB of XML data (Schenkel et al. 2007). We removed all the XML tags and retained only the text content, obtaining a corpus containing 8.76GiB of plain text (1.8×10^9 words), and a vocabulary of 14.88 million words (a rather heterogeneous collection). We include experiments for *locate* operation searching for a set of 429 queries[12] (including both phrases and single words) extracted from the topics in the INEX 2009-2010 Adhoc Track[13]. Figure 7 shows the results.

We use the inverted index based on SCDC encoding, which gave us the best results. Its space is lower bounded by the compression ratio obtained by SCDC compression, which is around 36.1%. Even when we set the block size to 16MiB, the inverted index uses around 39.2% space. WTPH+, with a sparse sampling configuration, is able to obtain compression ratios under 37%. WCSA takes advantage of higher order compression and almost reaches 30% compression.

Focusing on operation *locate*, we observe similar results as those in Fig. 3 for P_2 scenario. That is, WTPH+ obtains the best space/time trade-off in the wide range of 38–47% compression ratios. Only when we want to obtain compression ratios below 38%,

[11] http://www.mpi-inf.mpg.de/departments/d5/software/inex.

[12] Available at http://vios.dc.fi.udc.es/indexing/wsi/download.html.

[13] https://inex.mmci.uni-saarland.de/data/documentcollection.jsp.

the WCSA becomes the best choice due to its better compression. At that compression level, we force WTPH+ to use a very sparse sampling whereas WCSA uses still a rather dense setup. Both techniques clearly overcome block-addressing inverted indexes.

7 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 (Moura et al. 2000). However, the more space-efficient compressors such as Plain Huffman (Moura et al. 2000) and Restricted Prefix Byte Codes (Culpepper and Moffat 2005; Culpepper 2007) are not that fast at searching or random decompression, because they are not self-synchronizing techniques. In this paper we have shown that a simple reorganization of the bytes of the codewords obtained when a text is being compressed, marks clear codeword boundaries for those compressors. Our proposal, *Wavelet Trees on Bytecodes* (WTBC), gives better search and random access capabilities than all the byte-oriented compressors, even those that exchange some compression degradation by marking codeword boundaries [Tagged Huffman (Moura et al. 2000), End-Tagged Dense Codes (Brisaboa et al. 2007)].

As our reorganization permits carrying out all those operations efficiently over Plain Huffman, 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 WTBC over Plain Huffman (WTPH) will do better in almost all aspects.

This reorganization has also surprising consequences related to implicit indexing of the compressed text. Block-addressing inverted indexes over compressed text have been long considered as the best low-space structure to index a text for efficient word and phrase searches (Navarro et al. 2000). 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 WTBC, 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.

Other compressed indexes like word-based compressed suffix arrays (Brisaboa et al. 2008b) perform better than WTBC when searching for phrases of 3 words or more, and may achieve less space. They are, however, slower at other operations like searching for words and short phrases, searching on text ranges, and displaying portions of the text.

An interesting challenge for this representation is to support dynamism, so as to remove documents from the collection or add others at the end. This requires traversing all the nodes and remove part of the sequences, or append more data at the end of the sequences. The total amount of work is similar to that of updating an inverted index, where each list must be edited. There is the issue, however, of maintaining the encoding up to date with changing frequencies. Recent work on dynamic (s, c)-Dense Codes (Brisaboa et al. 2010) may prove this code better suited for this task, as it can maintain optimality within a moderate number of changes in the encoding.

Since its conference publication (Brisaboa et al. 2008a), WTBC has been successfully extended in various ways. A very interesting extension has been its adaptation to emulate a document-addressing inverted index (Arroyuelo et al. 2010), so as to natively support document retrieval operations. In particular, they target at answering conjunctive queries. Their solution is more efficient than inverted indexes in some scenarios. Another extension

has been the use of WTBC to index XML documents (Brisaboa et al. 2009), supporting XPath queries in efficient time and space proportional to the compressed XML document. We believe that many other extensions will come.

Finally, we wish to remark that our general idea could have wider applications. Grossi et al. (2003) initially defined *balanced* wavelet trees, which are built on the *source* symbols of a sequence. They also introduced the idea of *skewed* wavelet trees, by giving them Huffman shape. It is not hard to see that both wavelet trees on Huffman codes and our wavelet trees on byte codes are just two representatives of a more general idea: Given an encoder $C : \Sigma \rightarrow \tau^*$, and a sequence $S[1, n]$ over Σ, create a $|\tau|$-ary tree with one root-to-leaf path spelling each element in the image of $C(\Sigma)$. Store at the root a sequence over τ with the first symbol of the code of each $S[i]$, and continue recursively with the t-th child of the root with the subsequence of S formed by the symbols whose target code starts with $t \in \tau$. Give *rank* capabilities to the sequences. Then the total space of the tree is essentially the same as that of S encoded with C, and we have *direct access* to any $S[i]$ in the time required to compute $|C(S[i])|$ *rank* queries. This is an interesting alternative to the usual sampling schemes. Moreover, we can find all the occurrences of a given symbol $c \in \Sigma$ in S by going upwards from the leaf corresponding to $C(c)$ and using *select*. In this paper we have shown how relevant this idea can be to Information Retrieval, but we believe it may find applications in many other areas as well.

Acknowledgments Funded by MICINN grants TIN2009-14560-C03-02 and TIN2010-21246-C02-01, Ministerio de Ciencia e Innovación grant CDTI CEN-20091048, and Xunta de Galicia grant 2010/17 (for the Spanish group); and for the fourth author by Fondecyt grant 1-110066.

References

Anh, V., & Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval, 8*(1), 151–166.

Arroyuelo, D., González, S., & Oyarzún, M. (2010). Compressed self-indices supporting conjunctive queries on document collections. In *Proceedings of the 17th international symposium on string processing and information retrieval (SPIRE), LNCS 6393*, (pp. 43–54).

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern information retrieval*. Boston, MA: Addison-Wesley Longman.

Barbay, J., López-Ortiz, A., Lu, T., & Salinger, A. (2009). An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics (JEA), 14*(7), 3, 24 pp.

Bentley, J., Sleator, D., Tarjan, R., & Wei, V. (1986). A locally adaptive data compression scheme. *Communications of the ACM (CACM), 29*(4), 320–330.

Boyer, R., & Moore, J. (1977). A fast string searching algorithm. *Communications of the ACM (CACM), 20*(10), 762–772.

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

Brisaboa, N., Fariña, A., Ladra, S., & Navarro, G. (2008a). Reorganizing compressed text. In *Proceedings of the 31th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR)*, (pp. 139–146).

Brisaboa, N., Fariña, A., Navarro, G., Places, A., & Rodríguez, E. (2008b). Self-indexing natural language. In *Proceedings of the 15th international symposium on string processing and information retrieval (SPIRE), LNCS 5280*, (pp. 121–132).

Brisaboa, N., Cerdeira, A., & Navarro, G. (2009). A compressed self-indexed representation of XML documents. In *Proceeding of the 13th European conference on digital libraries (ECDL), LNCS 5714*, (pp. 273–284).

Brisaboa, N., Fariña, A., Navarro, G., & Paramá, J. (2010). Dynamic lightweight text compression. *ACM Transactions on Information Systems (TOIS), 28*(3), 10, 32 pp.

Clark, D. (1996). *Compact pat trees. PhD thesis*. Canada: University of Waterloo.
Claude, F., & Navarro, G. (2008). Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th international symposium on string processing and information retrieval (SPIRE), LNCS 5280*, (pp. 176–187).
Culpepper, S. (2007). *Efficient data representations for information retrieval. PhD thesis*. Australia: Department of Computer Science and Software Engineering, University of Melbourne.
Culpepper, S., & Moffat, A. (2005). Enhanced byte codes with restricted prefix properties. In *Proceedings of the 12th international symposium on string processing and information retrieval (SPIRE), LNCS 3772*, (pp. 1–12).
Culpepper, S., & Moffat, A. (2007). Compact set representation for information retrieval. In *Proceedings of the 14th international symposium on string processing and information retrieval (SPIRE), LNCS 4726*, (pp. 137–148).
Culpepper, S., & Moffat, A. (2010). Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS), 29*(1), 1, 25 pp.
Ding, S., Attenberg, J., & Suel, T. (2010). Scalable techniques for document identifier assignment in inverted indexes. In *Proceedings of the 19th international conference on world wide web (WWW)*, (pp. 311–320).
Ferragina, P., Manzini, G., Mäkinen, V., & Navarro, G. (2007). Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG), 3*(2), 20, 24 pp.
Ferragina, P., González, R., Navarro, G., & Venturini, R. (2009). Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA), 13*, 12, 31 pp.
Grossi, R., Gupta, A., & Vitter, J. (2003). High-order entropy-compressed text indexes. In *Proceedings of 14th annual ACM-SIAM symposium on discrete algorithms (SODA)*, (pp. 841–850).
Heaps, H. (1978). *Information retrieval—computational and theoretical aspects*. New York, NY: Academic Press.
Horspool, R. (1980). Practical fast searching in strings. *Software: Practice and Experience (SPE), 10*(6), 501–506.
Huffman, D. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers (IRE), 40*(9), 1098–1101.
Jacobson, G. (1989). Space-efficient static trees and graphs. In *Proceedings of 30th IEEE symposium on foundations of computer science (FOCS)*, (pp. 549–554).
Ladra, S. (2011). *Algorithms and compressed data structures for information retrieval. PhD thesis*. Spain: Department of Computer Science, University of A Coruña.
Moffat, A. (1989). Word-based text compression. *Software: Practice and Experience (SPE), 19*(2), 185–198.
Moffat, A., & Culpepper, S. (2007). Hybrid bitvector index compression. In *Proceedings of the 12th Australasian document computing symposium (ADCS)*, (pp. 25–31).
Moura, E., Navarro, G., Ziviani, N., & Baeza-Yates, R. (2000). Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS), 18*(2), 113–139.
Munro, I. (1996). Tables. In *Proceedings of the 16th conference on foundations of software technology and theoretical computer science (FSTTCS), LNCS 1180*, (pp. 37–42).
Navarro, G., Moura, E., Neubert, M., Ziviani, N., & Baeza-Yates, R. (2000). Adding compression to block addressing inverted indexes. *Information Retrieval, 3*(1), 49–77.
Raman, R., Raman, V., & Rao, S. (2002). Succinct indexable dictionaries with applications to encoding *k*-ary trees and multisets. In *Proceedings of the 13th annual ACM-SIAM symposium on discrete algorithms (SODA)*, (pp. 233–242).
Sanders, P., & Transier, F. (2007) Intersection in integer inverted indices. In *Proceeding of the 9th workshop on algorithm engineering and experiments (ALENEX)*, (pp. 71–83).
Schenkel, R., Suchanek, F., & Kasneci, G. (2007) Yawn: A semantically annotated wikipedia xml corpus. In *12th GI conference on databases in business, technology and web (BTW)*, (pp. 277–291).
Strohman, T., & Croft, B. (2007). Efficient document retrieval in main memory. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR)*, (pp. 175–182).
Transier, F., & Sanders, P. (2010). Engineering basic algorithms of an in-memory text search engine. *ACM Transactions on Information Systems (TOIS) 29*(1), 2, 37 pp.
Turpin, A., & Moffat, A. (1997). Fast file search using text compression. In *Proceedings of the 20th Australasian Computer Science Conference (ACSC)*, (pp. 1–8).
Witten, I., Moffat, A., & Bell, T. (1999). *Managing gigabytes: Compressing and indexing documents and images, 2nd edn*. San Francisco, CA: Morgan Kaufmann Publishers.

Yan, H., Ding, S., & Suel, T. (2009) Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on world wide web (WWW)*, (pp. 401–410).

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

Zukowski, M., Heman, S., Nes, N., & Boncz, P. (2006). Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd international conference on data engineering (ICDE)*, (p. 59).