A

Word-based Self-Indexes for Natural Language Text

ANTONIO FARIÑA
NIEVES R. BRISABOA
University of A Coruña, Spain
and
GONZALO NAVARRO
University of Chile, Chile
and
FRANCISCO CLAUDE
University of Waterloo, Canada
and
ÁNGELES S. PLACES
EDUARDO RODRÍGUEZ
University of A Coruña, Spain

The inverted index supports efficient full-text searches on natural language text collections. It requires some extra space over the compressed text that can be traded for search speed. It is usually fast for single-word searches, yet phrase searches require more expensive intersections. In this article we introduce a different kind of index. It replaces the text using essentially the same space required by the compressed text alone (compression ratio around 35%). Within this space it supports not only decompression of arbitrary passages, but efficient word and phrase searches. Searches are orders of magnitude faster than those over inverted indexes when looking for phrases, and still faster on single-word searches when little space is available. Our new indexes are particularly fast at *counting* the occurrences of words or phrases. This is useful for computing relevance of words or phrases.

We adapt *self-indexes* that succeeded in indexing arbitrary strings within compressed space to deal with large alphabets. Natural language texts are then regarded as sequences of words, not characters, to achieve word-based self-indexes. We design an architecture that separates the *searchable* sequence from its *presentation* aspects. This permits applying case folding, stemming, removing stopwords, etc. as is usual on inverted indexes.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data Compaction and Compression; H.3.2 [**Information storage and retrieval**]: Information storage—*File organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Self-indexes, compressed data structures, inverted indexes

---

A preliminary partial version on this work appeared in *Proc. SPIRE'08* [Brisaboa et al. 2008].
Authors' address: Antonio Fariña, Nieves Brisaboa, Ángeles S. Places, Eduardo Rodríguez, Department of Computer Science, University of A Coruña, Facultade de Informática, Campus de Elviña, s/n 15071 A Coruña, Spain. {fari, brisaboa, asplaces, erodriguezl}@udc.es. Gonzalo Navarro, Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Francisco Claude, 200 University Avenue West, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada. fclaude@cs.uwaterloo.ca.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION AND RELATED WORK

Text indexing is the process of building a data structure in order to achieve reasonable search times over a large text collection. The most common text collections contain *natural language*: digital libraries, searchable online bookstores, news archives, jurisprudence databases and all sorts of public and private text information systems, are everywhere. The most prominent example is of course the Web, which contains terabytes of text data.

The term *natural language* hides several assumptions here. It is used to denote text that is composed of an alternating sequence of *words* and *separators*, which can be easily distinguished syntactically; where the set of different words follows some statistical laws such as growing sublinearly with the text size (Heaps' law [Heaps 1978]); and especially where only whole words and sequences thereof (called *phrases*) can be searched for in the text. Although this definition excludes many human languages (such as Chinese, Korean, other Far East languages, and even poses problems to agglutinating languages such as Finnish and German), it fits well many Western languages and it has been widely adopted.

For decades, the *inverted index* has been a simple and effective solution to accelerate searches on natural language text collections [Baeza-Yates and Ribeiro-Neto 2011; Witten et al. 1999]. It is behind all sorts of "search engines" that provide indexed access to text databases. Those search engines are able to solve various types of queries. These can be divided into two main classes: *document retrieval* queries take the document as the basic unit of retrieval and encompass boolean and ranked retrieval, whereas *full-text retrieval* queries aim at locating the precise occurrences of the query terms in the text collection. Information retrieval systems usually support some combination of those. In this article we focus on full-text retrieval queries, which are necessary, for example, for finding phrases, or for highlighting text passages where the query occurs in the text.

The full-text inverted index essentially consists of a *vocabulary* storing the different words in the text, and a *posting list* recording the text positions of each vocabulary word in increasing order. This simple data structure immediately answers single-word searches, and can handle phrase searches by essentially intersecting the corresponding posting lists. How to carry out those intersections is an active area of research [Demaine and Munro 2000; Barbay and Kenyon 2002; Baeza-Yates 2004; Baeza-Yates and Salinger 2005; Barbay et al. 2006; Sanders and Transier 2007; Culpepper and Moffat 2007; Barbay and Kenyon 2008; Barbay et al. 2009; Culpepper and Moffat 2010].

Inverted indexes can take much space if represented in plain form (40% to 80% on top of the space required by the raw text). This is a serious drawback to fit large text collections in main memory (RAM), or to reduce I/Os. Since early days, compression techniques have been applied to inverted indexes to reduce their space [Witten et al. 1999; Zobel and Moffat 2006]. The most common approach is to differentially encode each posting list (as its numbers are increasing) and encode those *gaps* with an encoding that favors small numbers (so that longer lists are compressed most). Absolute samples are inserted to allow for direct access, which is essential to carry out intersections efficiently. There is recent research on this topic [Culpepper and Moffat 2007; Sanders and Transier 2007], which interacts with the chosen intersection algorithms.

The text is usually compressed as well. The preferred choice is Huffman coding [Huffman 1952] where source symbols are words and target symbols are bits [Moffat 1989] (hence called "word-oriented bit-wise Huffman"). To further save space, the text can be divided into blocks, so that the postings point to the blocks where the word appears. This is called a *block-addressing* inverted index [Baeza-Yates and Navarro 2000; Navarro et al. 2000]. At search time, the resulting blocks must be sequentially scanned to find the exact occurrences. The block size provides an obvious space/time tradeoff. In this tradeoff, it is advantageous to opt for a text compression method that permits much faster searches than bitwise Huffman [Brisaboa et al. 2007; Culpepper and Moffat 2005; Navarro et al. 2000]. Nowadays, very

efficient indexed searching can be obtained by occupying, with the compressed text plus the compressed index, 30% to 40% of the original text size.

The situation, up to the last decade, was far less satisfactory with other types of sequences. Without a concept of word, it is necessary to provide searching for *any* text substring. This was accomplished with powerful data structures called suffix trees [Weiner 1973; Apostolico 1985] and suffix arrays [Manber and Myers 1993]. Those were able to locate the *occ* occurrences of a pattern of length m in as little as $O(m + occ)$ time, regardless of the text size. However, they require 10–20 (suffix trees) or at best 4 (suffix arrays) times the text size, plus the text, and this rendered them unsuitable for many applications.

This changed drastically with the rise of *compressed self-indexes*, which were able to represent the text in space proportional to its (zero or higher, depending on the index) empirical entropy [Manzini 2001], and within that space, offer indexed searching for any text substring [Navarro and Mäkinen 2007]. For example, the smallest compressed self-index [Ferragina et al. 2007] offers searching in $O(m\lceil\frac{\log\sigma}{\log\log n}\rceil + occ\cdot\log^{1+\epsilon} n)$ time, where n is the collection size, σ is the alphabet size, and ϵ is any positive constant. Other self-indexes, not always reaching the same theoretical bounds, are relevant for this work because they perform almost equally well (and sometimes better) in practice [Ferragina et al. 2009; Claude and Navarro 2008] and handle well large alphabets: Sadakane's Compressed Suffix Array (*CSA*) [Sadakane 2003], and the Succinct Suffix Array (*SSA*) [Mäkinen and Navarro 2005; Ferragina et al. 2007; Mäkinen and Navarro 2008; Claude and Navarro 2008].

For example, on natural language texts, these indexes take around 60% to 70% of the original text size (and replace it). This is remarkable compared with the 500% (including the text) needed by suffix arrays, yet not competitive with the 30% to 40% achieved by compressed inverted indexes over compressed text. However, the comparison is not fair, because self-indexes can search for any text substring, whereas inverted indexes search only for whole words and phrases[1].

In this article we explore the potential of self-indexing for natural language text collections. We apply a compressed self-index (as developed for general strings) over the sequence of *words* of a natural language text, that is, regarding the words as the basic symbols. This is promising because a self-index achieving high-order entropy should capture the dependence between consecutive words, which is known to be significant in natural language [Bell et al. 1990, Chapter 4]. Even the zero-order entropy (which is what some variants of the *SSA* achieve) allows one to reach compression ratios around 35% when words are the source symbols. With respect to time performance, even the theoretically slower *CSA* is able to locate the occurrences of a phrase of m *words* in time $O(m\log n + occ\cdot\log^{1+\epsilon} n)$ (and know the *number* of occurrences in just $O(m\log n)$ time). This compares favorably with inverted indexes, which need to carry out intersections when $m > 1$. For example, for a phrase of 2 words appearing occ_1 and occ_2 times, an inverted index can take as much as $\min(occ_1, occ_2)\log(\frac{\max(occ_1,occ_2)}{\min(occ_1,occ_2)} + 1)$ time, where both occ_1 and occ_2 are (possibly much) larger than *occ*, the actual number of occurrences of the phrase.

We note that self-indexes operate in main memory, and therefore require that the compressed text does not exceed the available RAM. Because of their access pattern, self-indexes are not promising on secondary memory, where inverted indexes perform well. Recently, however, there has been much interest in inverted indexes that operate in RAM [Strohman and Croft 2007; Sanders and Transier 2007], motivated by the large main memories available at reasonable prices (up to 4GiB is standard and 128GiB is not out of reach) and the common distributed architectures where the text collection resides in the RAM of several

[1]It is possible to search for more complex patterns, but the process is far more cumbersome [Baeza-Yates and Ribeiro-Neto 2011, Sec. 9.2.3]. Another path is to use q-gram indexes, which are inverted-index-like structures for general strings [Puglisi et al. 2006].

computers (then the problem is how to integrate the results of several indexes across the slow network). Therefore, main memory data structures are of interest nowadays, unlike what was assumed 10 years ago.

Applying a self-index to natural language words poses some challenges. A first one is that the alphabet is very large, and this rules out the theoretically best schemes [Grossi et al. 2003; Ferragina et al. 2007], which achieve k-th order entropy at the price of $\Omega(\sigma^k)$ extra space, where in our case σ is the vocabulary size. According to Heaps' law [Heaps 1978], a text of n words has a vocabulary of size $\sigma = \Theta(n^\beta)$, where β is close to 0.5 [Baeza-Yates and Ribeiro-Neto 2011]. Thus σ^k may become $\Omega(n)$ already for $k = 2$! However, other self-indexes such as the *CSA* [Sadakane 2003] and some variants of the *SSA* [Mäkinen and Navarro 2008] approach high-order entropy space without such a dependence on σ. Our first structures, the *word-based self-indexes* (*WSI*), result from regarding the text as a sequence of word and separator identifiers and representing it with a self-index adapted to deal with large alphabets. We present in Section 3 two new word-based self-indexes: the word-based *CSA* (*WCSA*) and the word-based *SSA* (*WSSA*).

A second challenge is that, in many applications, we wish to have more flexible searching. For example, inverted indexes often permit to find phrases regardless of whether the words are separated by a space, two spaces, a tab, a newline, etc. Moreover, it is customary to apply some *filtering* on the text words to be searched for [Baeza-Yates and Ribeiro-Neto 2011], that is, users normally want to regard `"preprocess"`, `"pre-process"`, `"Preprocess"` and `"PRE-PROCESS"` as occurrences of `"preprocess"`, and even also `"preprocessing"` and `"preprocessed"` (the latter is achieved by *stemming*, that is, indexing/searching the roots of the words). It is also usual to disregard *stopwords* (articles, prepositions, etc.) in the searches. This complicates the simple *WSI* model where the self-index can reproduce the original text and thus the latter can be discarded. We must store some information on the separators in order to be able of exactly recreating the original text. We introduce the concept of a *presentation layer*, where the text is filtered into the *searchable sequence* of (possibly stemmed, lowercase, stopwords removed) bare words, and the *presentation sequence* containing the separators and all extra information on the bare words that permits recreating the original sequence. The searchable sequence is self-indexed, while the presentation sequence is just compressed with a technique that permits fast direct access for displaying purposes. Both sequences are compressed by different means, thus the choice of what is searchable is not a space/time tradeoff but rather depends on the user's needs. We call *Flexible WSI (FWSI)* this word-based self-index that allows the user to determine what is searchable. Section 4 describes the general architecture we have developed to separate self-indexing from presentation aspects, and describes two particular *FWSI* realizations: the *flexible WCSA* (*FWCSA*) and the *flexible WSSA* (*FWSSA*).

Section 5 shows that the resulting self-indexes achieve interesting space/time tradeoffs in practice. They obtain very good compression results, close to many natural language text compressors that do not actually provide any indexing. Texts are usually compressed to around 35-40% of their original size, or even further down to around 30% at the expense of slower response time. As expected, using different definitions of what is the searchable sequence has a minimal impact on compression performance. On the other hand, the flexible variants are slower as they need to recover the presentation aspects.

As full word-addressing inverted indexes would require about twice the space of our word self-indexes, we developed new block-addressing inverted indexes *(IIs)* to compare with, following the ideas in previous work [Witten et al. 1999; Navarro et al. 2000; Culpepper and Moffat 2007; Moffat and Culppeper 2007; Sanders and Transier 2007]. These *II*s can be parameterized so that they use the same amount of space than our *WSI*. Experimental results show that the self-indexes are much faster at counting the number of occurrences of a pattern than the *II*; in particular the *WSSA* is the fastest. For locating and extracting operations the *WSSA* is always slower than the *WCSA*. Nevertheless, both self-indexes

excell when searching for phrase patterns. In particular, the *WCSA* clearly overcomes the *II* when searching for phrases, what the *WSSA* achieves only for phrases of 4 words or more. Consistently with previous work [Transier and Sanders 2010], the *II* behaves very well on searches for very frequent single-word patterns, but it only becomes clearly faster than the *WCSA* when a large amount of memory is allowed for the indexes (i.e., 40% space or more). With lower compression ratios the *WCSA* overcomes the *II* in most aspects.

From a practical point of view, we paid special attention at creating indexing structures that could be of interest for our research community. Indeed, our prototypes uncouple the self-indexing from the presentation layer, so that any self-index on integers implementing a well-defined software interface automatically generates a *FWSI*. Open-source implementations of our self-indexes will be available at `http://vios.dc.fi.udc.es/wsi/indexing`. This opens the door for practitioners to test and use our new indexing schemes in Information Retrieval applications.

2. SELF-INDEXES SUITABLE FOR LARGE ALPHABETS

Let $T[1,n]$ be a sequence over an alphabet Σ of size σ. A *self-index* is a data structure that represents T, hopefully within little space, so that the following operations are supported:

— $count(P)$, where $P[1,m]$ is a pattern over alphabet Σ, gives the number of occurrences of P in T.
— $locate(P)$ returns the list of positions where P occurs in T, in some order.
— $extract(s,e)$ returns $T[s,e]$.

Since $T = extract(1,n)$, the self-index replaces T, which can thus be discarded. In the sequel we describe two self-indexes that are practical and cope well with large alphabets.

2.1. Sadakane's Compressed Suffix Array (CSA)

The *suffix array* [Manber and Myers 1993] $A[1,n]$ of T is a permutation of $[1,n]$ of all the suffixes $T[i,n]$ so that $T[A[i],n] \prec T[A[i+1],n]$ for all $1 \leq i < n$, being $\prec$ the lexicographic ordering. Since every substring of T is the prefix of a suffix, and all suffixes prefixed by a search pattern $P[1,m]$ are contiguous in A, we can binary search A for the interval $A[sp,ep]$ of the pointers to all the occurrences of (i.e., suffixes starting with) P in T, in time $O(m \log n)$. Note that each step of the binary search needs to access $T[A[i],A[i]+m-1]$ for some i, in order to compare that string with $P[1,m]$.

Let us now define another permutation $\Psi[1,n]$ such that $\Psi(i) = A^{-1}[A[i]+1]$ (or $A^{-1}[1]$ if $A[i]=n$) [Grossi and Vitter 2000]. Hence $\Psi(i)$ tells where the pointer to the text position $T[A[i]+1]$ (i.e., the one that follows $T[A[i]]$) occurs in A. Assume one has computed $C[1,\sigma]$, so that $C[c]$ is the number of occurrences of symbols $\prec c$ in T. We show how one can obtain the successive letters of $T[A[i]...]$ (so as to carry out the binary search) with Ψ and C and without A and T. To extract the first letter, note that all the suffixes starting with c are in the area $A[C[c]+1, C[c+1]]$, and therefore a binary search on C for the c such that $C[c] < i \leq C[c+1]$ gives the desired first letter, $T[A[i]] = c$. To extract the next letter, we use the identity $T[A[i]+1] = T[A[\Psi(i)]]$, thus we simply have to move to $i' \leftarrow \Psi(i)$ and carry out the same process again to obtain $T[A[i']]$, and so on.

The search on C is implemented in constant time as follows. Set up a string $S[1,\sigma']$, $\sigma' \leq \sigma$, containing the distinct symbols that actually occur in T, in increasing lexicographic order. Set up a bitmap $D[1,n]$ with all zeros except $D[C[c]+1] = 1$ for all $c \in \Sigma$. Now, the c corresponding to an i value is $c = S[rank_1(D,i)]$, where $rank_1(D,i)$ is the number of 1s in $D[1,i]$. This can be (easily) computed in constant time using $o(n)$ further bits [Jacobson 1989; Munro 1996].

The description above is the essential idea of Sadakane's *CSA* [Sadakane 2003], where we have removed several possible optimizations that are not promising for our particular application (backward searching, compressed bitmaps, etc.). One important remaining point

is how to compress Ψ, as in principle it is as large as the suffix array A it replaces. Sadakane shows that, since Ψ is formed by σ increasing subsequences [Grossi and Vitter 2000], it can be compressed to around the zero-order entropy of T. Let H_k be the k-th order empirical entropy of T [Manzini 2001]. Then, the data structure requires $nH_0 + O(n \log\log \sigma)$ bits, by encoding its differential values with δ-codes (in a practical implementation he used γ-codes) [Witten et al. 1999]. Furthermore, as shown later [Navarro and Mäkinen 2007], Ψ can be partitioned into at most $nH_k + \sigma^k$ (for any k) *runs* of values, so that consecutive differences equal 1 within each run. Thus, by enriching the gap encoding with run-length compression of those runs one achieves higher-order compression, that is, one captures the statistical dependence of a word from its k previous words. Absolute Ψ values at regular intervals t_Ψ are retained to permit fast random access to Ψ (constant time can be achieved in theory).

Note that, since we do not have A anymore, determining the interval $A[sp, ep]$ is not sufficient to locate the occurrences, that is, to output the values $A[i]$ in the interval. For this sake, the text is sampled at regular intervals l, and the suffix array positions pointing to sampled text positions are recorded, in suffix array order, into an array $A_S[1, n/l]$. Those sampled positions in A are marked in a bitmap $B_A[1, n]$, thus if $B_A[i] = 1$ we know that $A[i] = A_S[rank_1(B_A, i)]$. Otherwise, we try $i \leftarrow \Psi(i)$ successively, as we are virtually moving forward in T by one position at each iteration. Hence, if we determine $A[i] = j$ after k applications of Ψ, then our original value was $j - k$. Due to the regular sampling in T we carry out at most l iterations until finding a sampled position in A.

Finally, in order to discard T, we need to be able to extract any substring $T[s, e]$. For the same sampled text positions $j \cdot l$ sampled above[2], we store $A^{-1}[j \cdot l]$ in text position order into an array $A_S^{-1}[1, n/l]$. Thus, we find the latest sampled position $j \cdot l$ preceding s, $j = \lfloor s/l \rfloor$, and know that $T[j \cdot l]$ is pointed from $A[i], i = A_S^{-1}[j]$. From that i we use the mechanism we have described to extract a string using C and Ψ, to find out the substring $T[j \cdot l, e]$ which covers the one of interest to us. (This is not the way Sadakane's theoretical description handles this [Sadakane 2003], but the way he implemented it in practice.)

In practice, on a large alphabet of size $\sigma = \Theta(n^\beta)$, reasonable sampling rates are $t_\Psi = l = O(\log n)$. Such a *CSA* requires $nH_0 + O(n \log\log n)$ bits of space, yet in practice the space is closer to $2nH_k + O(n)$ due to the use of run-length compression and γ-coding. Counting time is $O(m \log^2 n)$, yet only $O(m \log n)$ accesses are non-local[3]. Similarly, it solves *locate* in time $O(\log^2 n)$ per occurrence and *extract*(s, e) in time $O((e - s + \log n) \log n)$, where only one out of $O(\log n)$ accesses are non-local.

2.2. Succinct Suffix Array (SSA)

The Succinct Suffix Array *(SSA)* [Mäkinen and Navarro 2005; Ferragina et al. 2007; Mäkinen and Navarro 2008] is a self-index that builds on the Burrows-Wheeler transform (BWT) of a text [Burrows and Wheeler 1994]. It belongs to a class of self-indexes called *FM-indexes* [Ferragina and Manzini 2000; 2005]. Their main idea is to represent the BWT of text T (which reorders the positions of T) in compressed space and support searches by emulating a suffix array.

The BWT $T^{bwt}[1, n]$ of $T[1, n]$ is defined as $T^{bwt}[i] = T[A[i] - 1]$ (except $T^{bwt}[i] = T[n]$ if $A[i] = 1$), that is, at the position i it stores the text character that precedes the suffix pointed from $A[i]$. The reordered T exhibits more local regularities than T itself (as all the text characters followed by the same context are put together).

[2]In practice, different sample periods t_A and t_A^{-1} can be used for A and A^{-1} instead of using the same value l. Note that A_S^{-1} is only needed for *extracting* data, whereas A_S is more profusely used. Therefore, it is usually a good idea to use a sparser sampling on A^{-1}.

[3]We say that accesses to memory are local when they fall within a restricted region of the data, so that most of them do not produce page faults.

A fundamental operation on T^{bwt} is known as *LF-mapping*. As $T^{bwt}[i]$ can be thought of as aligned to $A[i]$, $j = LF(i)$ gives the position j that is aligned with $A[i] - 1$, that is, $LF(i) = A^{-1}[A[i] - 1]$ (except that $LF(i) = A^{-1}[n]$ if $A[i] = 1$). Thus LF is precisely the inverse of function Ψ of Section 2.1.

What makes FM-indexes special is that, unlike Ψ, LF does not need to be represented, but it can be computed from T^{bwt}. Given a string S, we define $rank_c(S, i)$ as the number of times the symbol c appears in $S[1, i]$. When $S = T^{bwt}$, it holds $LF(i) = C[c] + rank_c(S, i)$ [Ferragina and Manzini 2000], where $c = T^{bwt}[i]$ and C is as in Section 2.1 (but in this case it is better to store it as an array).

Instead of emulating a classical suffix array search, FM-indexes use a concept called *backward search*. The idea is to search for P backwards, maintaining all the time the interval $A[sp, ep]$ of suffixes starting with $P[i, m]$ (we start with $i = m + 1$, $sp = 1$ and $ep = n$), and at each step updating sp and ep so that the invariant is reestablished for $P[i - 1, m]$. When they reach $i = 1$, $A[sp, ep]$ is the result of the search. The formulas to move from i to $i - 1$ are similar to that of the LF-mapping: $sp \leftarrow C[c] + rank_c(S, sp - 1) + 1$ and $ep \leftarrow C[c] + rank_c(S, ep)$, where $c = P[i - 1]$.

Thus *count* is solved with $O(m)$ computations of *rank*. Operations *locate* and *extract* are solved with sampling, analogously to the *CSA*: just like Ψ moves forward in T, LF moves backwards, so the same techniques of sampling A and A^{-1} are used.

The different FM-indexes differ in how they represent S and how they implement operations $S[i]$ (called *access*) and $rank_c(S, i)$. The *SSA*, in particular, represents S using a data structure called a *wavelet tree* [Grossi et al. 2003], where in practice these operations are solved in $O(\log \sigma)$ time. Therefore counting takes time $O(m \log \sigma)$, *locate* with sampling step l takes time $O(l \log \sigma)$ per occurrence retrieved, and *extract*(e, s) requires $O((e - s + l) \log \sigma)$ time.

The wavelet tree reduces the general problems of accessing any $S[i]$ and computing $rank_c(S, i)$ to the problems of computing access and rank over bitmaps.

The main idea is to represent the sequence as a tree of bitmaps that correspond to a binary decomposition of the alphabet. For example, consider the root and a binary partition of the alphabet into two sets Σ_0 and Σ_1. The bitmap contained in the root contains a 0 at position i iff the symbol at position i is in Σ_0, in case the symbol is in Σ_1 we write a 1. Then, the left subtree continues recursively considering the subsequence of elements that belong to Σ_0 and the right subtree considers the subsequence of elements belonging to Σ_1.

The query $rank_c(S, i)$ is answered by moving across the tree considering the encoding of c, which corresponds to the bit used to represent c in each level. Let B be the bitmap stored at the wavelet tree root. Then, depending on whether $c \in \Sigma_0$ or Σ_1, we descend to the left or right child of the root, now with value $i \leftarrow rank_0(B, i)$ or $i \leftarrow rank_1(B, i)$. Then we continue recursively until reaching the leaves. Accessing $S[i]$ is similar, except that we descend left or right depending on whether $B[i] = 0$ or $B[i] = 1$. The leaf arrived at corresponds to the symbol $c = S[i]$.

If the binary decomposition of the alphabet consists just in splitting it into halves at every level, the space required by the wavelet tree is $n \log \sigma(1 + o(1))$ bits, and the query time for rank/access is $O(\log \sigma)$. We can further improve the space to be $nH_0 + o(n \log \sigma)$ if we give a Huffman shape [Huffman 1952] to the tree; in this case, the query time is $O(H_0)$ on average. Figure 1 shows an example of a wavelet tree.

In a practical extension of wavelet trees to larger alphabets, Claude and Navarro [2008] proposed a wavelet tree representation that reduces the number of pointers from $O(\sigma)$ to either $O(1)$ or $O(\log \sigma)$.

It was shown [Mäkinen and Navarro 2008] that by representing the bitmaps in the wavelet tree used by the *SSA* with the structure by Raman et al. [2002], one can achieve $nH_k + o(n \log \sigma)$ bits of space for any $k \leq \alpha \log_\sigma n - 1$ and constant $0 < \alpha < 1$. On a large

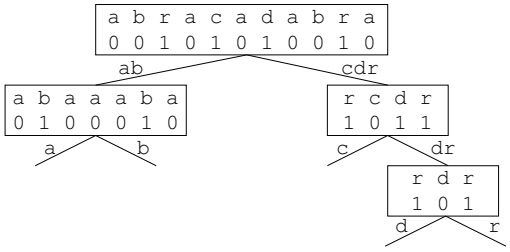

Fig. 1. Example of a wavelet tree for the string `abracadabra`.

alphabet of size $\sigma = \Theta(n^\beta)$, however, this is similar to the $nH_0 + O(n \log\log n)$ bits of the *CSA*. The counting operation takes $O(m \log \sigma)$ time on the *SSA*, which on a large alphabet becomes $O(m \log n)$ non-local memory accesses. Setting the sampling rates to a reasonable value $l = \log n$, operation *locate* takes $O(\log^2 n)$ per occurrence and *extract*(s, e) takes $O((e - s + \log n) \log n)$ non-local memory accesses on a large alphabet. This is in accordance with our later empirical findings that, on word identifiers, the *SSA* is faster for counting but slower for locating and extracting than the *CSA*.

3. WORD-BASED SELF-INDEXES

In this section we introduce the concept of a *word-based self-index (WSI)* and, based on the classical self-indexes described in Section 2, we define two particular cases: the *word-based CSA (WCSA)* and the *word-based SSA (WSSA)*.

The first task required to develop a *WSI* is to adapt a classical self-index to deal with very large alphabets, in order to handle the integers that will represent word identifiers. We will refer to the self-indexes that handle integer alphabets as *int-based self-indexes (ISI)*. By just coupling an *ISI* with a word-based parser that maps words into integer ids, we will obtain a *word-based self-index (WSI)*. We will adapt, from Section 2, the *CSA* to obtain the *int-based CSA(ICSA)*, and the *SSA* to obtain the *int-based SSA(ISSA)*. The respective word-based self-indexes will be called *word-based CSA (WCSA)* and *word-based SSA (WSSA)*, respectively.

In the rest of the section we give the general ideas behind our word-based self-indexes, and further details that arise from the adaptation of the original self-indexes to int-based self-indexes.

3.1. General structure and functionality of a word-based self-index

To create a *WSI*, we first parse the text T into a sequence of *words* (a user-defined concept, usually maximal concatenations of letters and symbols) and *separators* (maximal sequences between words). We use the *spaceless word model* [Moura et al. 2000], where separators consisting of a single blank space are not encoded but assumed by default when two consecutive words are encoded. We build a vocabulary V with the different *tokens* (words and separators). Then we alphabetically sort V and create an array Sid, by replacing each token w_{id} in T by the integer id such that $V[id] = w_{id}$. Finally, the sequence Sid is self-indexed using an *ISI*.

The resulting *WSI* is a two-layer structure: the top layer consists of V, and permits us to map any token w_{id} into its corresponding id, and the bottom layer is composed of an *ISI*. This *ISI* not only provides a compact representation of Sid to the *WSI*, but it also gives support for performing indexed searches. The *WSI* supports the following operations:

— *countWSI*(P) counts the number of occurrences of P, which is a token or a sequence thereof, in T. The occurrences must be exact, that is, the words and the separators of P must match exactly in T.
— *locateWSI*(P) gives the list of the positions of P in T. These positions are given in terms of token offsets, not character offsets.
— *extractWSI*(s, e) returns a substring of T formed by the sth to the eth tokens (words or separators).

Searching for a pattern of m tokens $P = \langle w_1 \dots w_m \rangle$ starts by looking up the tokens w_i in V (using binary search) to make up a new pattern $P' = \langle id_1 \dots id_m \rangle$, such that $w_i = V[id_i]$. After that, *countWSI(P)* reduces to *count*(P') on the *ISI* of Sid, and *locateWSI*(P) directly translates into *locate*(P'). The last operation, *extractWSI*(s, e), retrieves $Sid[s, e]$ using *extract*(s, e) on the *ISI*, and then accesses the vocabulary V at positions $Sid[s] \dots Sid[e]$ to obtain the substring of T. It also adds the implicit blank between any two consecutive words.

3.2. The word-based CSA (WCSA)

Developing the *WCSA* implies self-indexing the sequence Sid, and consequently building an *ICSA*, that is, adapting the *CSA* to work over integers.

The first step to build an *ICSA* over Sid is the construction of the suffix array A and the bitmap D. We used *qsufsort*[4] algorithm [Larsson and Sadakane 2007] for computing A in main memory. This algorithm gave us good results on large-alphabet texts. In the next step, permutations A^{-1} and Ψ are built in linear time from A. Then, samples from both A^{-1} and A are extracted, and arrays A_S^{-1} and A_S, as well as the bitmap B_A, are created. At this point A^{-1} and A are no longer needed and can be discarded. Finally, Ψ is also represented in a compact way. More details of this compact representation are given in Section 3.2.1.

If we assume that the maximum value in Sid is σ (i.e., there are σ different words), the vocabulary used by the *ICSA* consists of the numbers $\langle 1, 2, \dots, \sigma \rangle$. Therefore, by just keeping the value σ, the vocabulary remains implicit and there is no need to store it (nor $S[1, \sigma']$) as in the traditional *CSA*.

3.2.1. A compact representation of Ψ. We followed the ideas in Sadakane's implementation of Ψ [Sadakane 2003; Navarro and Mäkinen 2007], adapting them to a large alphabet. Absolute samples at positions $k \cdot t_\Psi$, $0 \le k \le \lfloor n/t_\Psi \rfloor$, are stored compactly in an array. For each sample we store the absolute Ψ value at that entry and a pointer to the compressed Ψ bitstream where the next cells are stored in compressed and differential form. Those integers use the maximum number of bits needed to represent all the stored values.

In Sadakane's approach, γ-codes are used for compressing the differences, and when a difference is 1, the next number encodes the length of its run, that is, the number of consecutive 1s that follow. We explore also other alternatives, such as δ-coding, Huffman coding, and some combinations.

An important difference with respect to a small alphabet is that, in the word-based approach, the differential sequence $\Psi(i) - \Psi(i-1)$ may contain a significant number of negative values, up to $\sigma - 1$. While this is 255 in a byte alphabet, it can be much larger on words. For example, our 1GiB text (corpus ALL described in Section 5) produced about 228 million words. Of those 228 million differences in Ψ, about half a million are negative values. Albeit not a very large percentage, encoding those values improperly does impact compression ratios.

When using δ–codes, negative values must be encoded somehow as exceptions. We also consider another folklore approach, where instead of representing a difference that may be

[4] `http://www.larsson.dogma.net/research.html`

positive or negative, we represent $x_i = \Psi(i)\ xor\ \Psi(i-1)$,[5] and this is the number δ-encoded. Note x_i is always positive, and if $x_1 = \Psi(1)$, then $\Psi(i)$ can be recovered as $x_i\ xor\ \Psi(i-1)$. We will call *δ-codes-gaps* the variant that represents differences and *δ-codes-xor* the one that represents *xor*. Analogously, we define *γ-codes-gaps* and *γ-codes-xor*.

Huffman coding seems in principle to be necessarily better than previous encodings in terms of space. However, as there are many different values to encode, the model size is a concern. In practice Huffman coding must be combined with some of the previous encodings. In addition, run-length encoding for the 1s can be introduced in different ways. Recall that run-length encoding is necessary to capture inter-word dependences in the compression.

We developed twelve different techniques to represent the non-sampled values of Ψ. These variants permitted us to study the effect of encoding 1-runs in Ψ, encoding differences versus encoding the *xor* of consecutive values, as well as the impact of using either Huffman-codes or δ-codes instead of γ-codes as in the original *CSA*.

— *Huffman-rle.* This strategy is based on combining Huffman coding with the encoding of runs in Ψ. We use two different Huffman codings: a main coding H^c to represent values from Ψ, and a secondary coding H^r to represent lenghts of runs that occur within Ψ. The *Huffman-rle* representation contains a Huffman code (from H^c) per each non-sampled value $\Psi(i)$ (except for adjacent positions in Ψ that are encoded with a 1-run). When representing those $\Psi(i)$ values, three different cases are dealt with depending on the value $d_i = \Psi(i) - \Psi(i-1)$: *i) Small gaps* are directly encoded by a Huffman codeword $H^c_{d_i}$. The number of *small values* is tuned via a parameter s: the gap values d_i such that $2 \leq d_i < s$ are encoded with their own Huffman codeword ($H^c_2 \dots H^c_{s-1}$). *ii) Negative and large gaps.* When either $d_i < 0$ or $d_i \geq s$, an escape codeword (H^c_0) is output, and followed by the number $\Psi(i)$ represented with $\lceil \log_2 n \rceil$ bits. *iii) Run-encoding.* If $d_i = 1$, a 1-run is detected in $\Psi(i)$. Assuming that its length is l we encode such run as follows: First, an escape codeword H^c_1 is output, and then l is represented. As l is usually very small ($1 \leq l < t_\Psi$, because runs are cut artificially if they reach the next sampled value), a secondary Huffman coding H^r is used to represent the l values.
Therefore, the Huffman coding H^c is obtained as follows: We consider a vocabulary that consists in the symbols $\{0 \dots s-1\}$. We compute the frequency of each symbol, so that $freq[i]$, $i \in 2 \dots s-1$ counts the number of times that gap value i occurrs. Similarly we sum in $freq[1]$ the number of runs that occur in Ψ and in $freq[0]$ the number of both large and negative values. Then we sort the symbols by frequency and apply Huffman algorithm to obtain the codeword H^c_i that corresponds to each symbol. Similarly, H^r is obtained by assuming a vocabulary composed of the symbols $l \in 1 \dots t_\Psi - 1$ and counting the number of times a run of length l occurs.
To check the effect of not encoding runs, we also developed a simplified version of *Huffman-rle* not using run-length encoding. Therefore, each occurrence of the gap value $+1$ is represented by a codeword H^c_1. We refer to it as *Huffman* variant.

— *Huffman-rle-opt.* Based on the previous approach, we created an improved format to compress Ψ that addresses its two main weaknesses: the need for using an escape codeword (H^c_1) to introduce runs; and, for both large and negative gaps, the need to represent $\Psi(i)$ with $\lceil \log_2 n \rceil$ bits after the escape codeword H^c_0. From the s symbols that can be encoded through Huffman code H^c we reserve $w+w$ symbols, where w is the machine-word size, to represent the length of the binary representation of both large positive gaps and negative gaps. In addition, t_Ψ symbols are reserved to encode the length of the 1-runs, and the remaining $s - 2w - t_\Psi$ symbols are used for small gaps.
Therefore, we have four different cases: *i)* Codewords $H^c_2 \dots H^c_{runbeg-1}$, where $runbeg = s - 2w - t_\Psi$, directly encode gaps $d_i = \Psi(i) - \Psi(i-1)$ such that $2 \leq d_i < runbeg$.

[5] *xor* refers to eXclusive OR, represented with '^' in `C/C++`.

ii) The next t_Ψ codewords, $H^c_{runbeg} \dots H^c_{posbeg-1}$, where $posbeg = runbeg + t_\Psi$, directly encode 1-runs. Precisely, a run of length l is encoded as $H^c_{runbeg+l}$. *iii)* The codewords $H^c_{posbeg} \dots H^c_{negbeg-1}$, being $negbeg = posbeg + w$, are used as escape codewords to introduce large d_i values, and implicitly represent the length of the binary representation of d_i. Assuming $l = \lceil \log_2 d_i \rceil$, $\Psi(i)$ is encoded by codeword $H^c_{posbeg+l}$ followed by d_i binary-represented with just l bits. *iv)* In a similar way, codewords $H^c_{negbeg} \dots H^c_{s-1}$ are reserved for negative values. Therefore, we implicitly keep the sign of d_i and proceed as for the large positive values, this time representing $-d_i$. Assuming $l = \lceil \log_2(-d_i) \rceil$, $\Psi(i)$ is encoded by codeword $H^c_{negbeg+l}$ followed by l bits representing $-d_i$ in binary.
As in the previous approach, we also developed a simplified counterpart of *Huffman-rle-opt* named *Huffman-opt* not including encoding of runs.

— *δ-codes-gaps and γ-codes-gaps.* As δ-codes can only represent positive numbers, we reserve some positive values (1 value for every K positive integers) to represent negative values. Parameter K is tuned to maximize compression. Basically, a negative gap d_i is represented as δ-code$(-Kd_i)$, whereas a positive value d_i is encoded as δ-code$((Kd_i - 1)/(K - 1))$. Assuming $K = 3$, 1 is encoded as δ-code(1), 2 as δ-code(2), 3 as δ-code(4), 4 as δ-code(5), 5 as δ-code(7), etc; whereas -1 is represented by δ-code(3), -2 as δ-code(6), and so on. Decoding a value C implies applying $v \leftarrow \delta$-decode(C). Then, if $v \ mod \ K = 0$ we return the value $-v/K$; otherwise, the value $v - (v/K)$ is obtained. γ-coding is analogous.
We have also created variants *δ-codes-gaps-rle and γ-codes-gaps-rle* that profit from encoding 1-runs of length l. Again, the idea is that when a 1-run is to be encoded the value $+1$ acts as an *escape* code that is followed by the encoding of l. That is, we represent that run as δ-code(1) δ-code(l) or γ-code(1) γ-code(l) respectively.

— *δ-codes-xor and γ-codes-xor.* Instead of encoding differences as in the previous representation, we represent the *xor* with the previous value of Ψ, as explained. Therefore, we δ-encode or γ-encode $x_i = \Psi(i)$ *xor* $\Psi(i-1)$. Decoding is simply $\Psi(i) \leftarrow x_i$ *xor* $\Psi(i-1)$. Including encoding of runs to obtain *δ-codes-xor-rle* requires, as in *δ-codes-gaps-rle*, to use the value $+1$ as an escape code, so that 1-runs of length l are represented as δ-code(1) δ-code(l). However, since value $+1$ must be reserved, encoding regular values $x_i = \Psi(i)$ *xor* $\Psi(i-1)$ requires an extra trick, as it could occur that $\Psi(i)$ *xor* $\Psi(i-1) = 1$ even when $\Psi(i) \neq \Psi(i-1) + 1$ (hence this is not a run). For example, if $\Psi_i = 8$ and $\Psi_{i-1} = 9$ we obtain that $\Psi_i - \Psi_{i-1} = -1$, but (in binary) 1000 *xor* 1001 = 0001. To overcome this issue we encode $x'_i = (\Psi(i)$ *xor* $\Psi(i-1)) + 1$ instead of $x_i = \Psi(i)$ *xor* $\Psi(i-1)$. Decoding is done as $\Psi(i) = (x'_i - 1)$ *xor* $\Psi(i-1)$. *γ-codes-xor-rle* are obtained similarly.

Figure 2 compares the different variants to compress Ψ from the 1GiB corpus, where a plain representation of Ψ would require around 872MiB. We show the size of the compressed Ψ representation, and the time needed to access all the values in Ψ, starting at position $i \leftarrow 0$, and then repetitively at $i \leftarrow \Psi(i)$ (that is, we traverse the text from left to right). For Huffman codes we use $s = 2^{14}$, which allows us to represent most of the interesting values with a very moderate-size model. We used powers of 2 for t_Ψ, that is, 1024, 512, 256, and so on. In the plots we show the case $t_\Psi = 16$, and then each point towards the left represents the next power of 2 (larger t_Ψ implies less space and more time).

Figure 2-top-left shows that *δ-codes-gaps* clearly overcomes *δ-codes-xor* and that *δ-codes-gaps-rle* is able to obtain a slight improvement over *δ-codes-gaps*. This also holds for the γ-coding-based counterparts, which are clearly overcome by those using δ-coding. The only exception is *γ-codes-xor*, which performs the best for very dense sampling values (obtaining a compression ratio worse than 50%). Figure 2-top-right shows that part of the plot in detail.

Figure 2-bottom compares the Huffman-based techniques with those using δ-coding. The bottom-right plot includes a zoomed area of the bottom-left plot. It can be seen that δ-coding

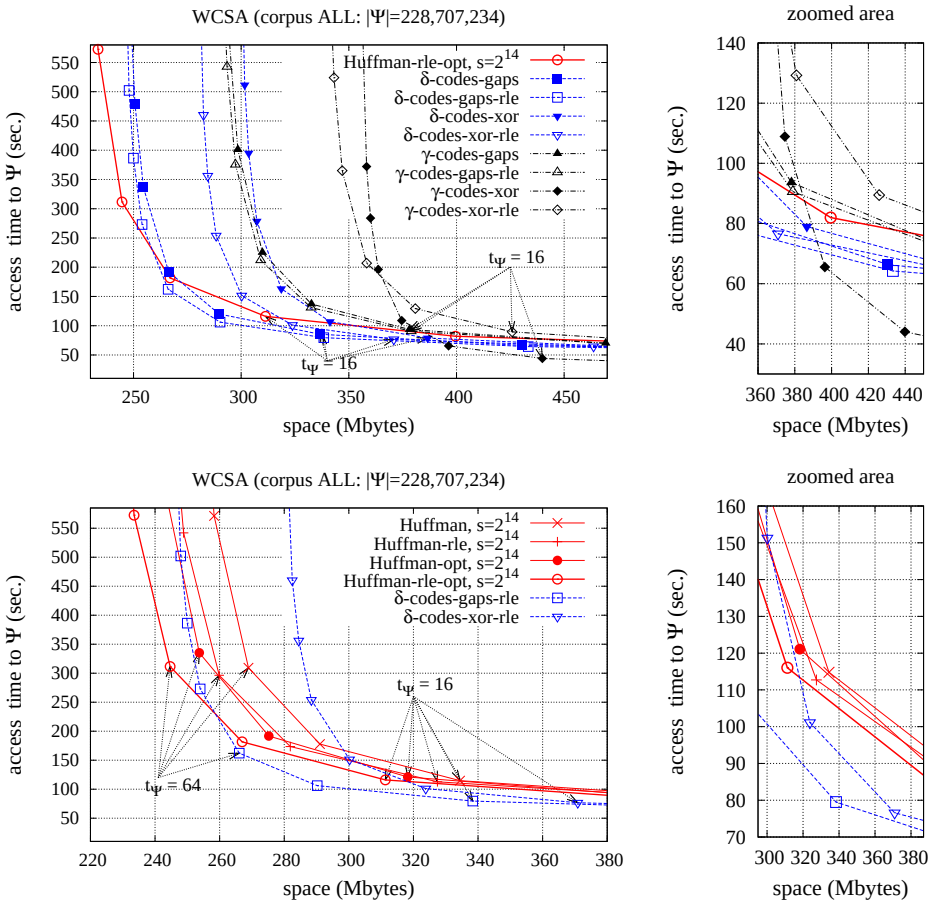

Fig. 2. Space/access-time tradeoff of the compressed representations of Ψ.

is very competitive when a dense sampling of Ψ is used ($t_\Psi \leq 32$), which leads to a heavy but fast index. For example, *δ-codes-gaps-rle* for $t_\Psi = 64$ gives a slightly better space/time tradeoff than that of *Huffman-rle-opt* with $t_\Psi = 32$. However, if a sparser sampling is used, Huffman-rle-based techniques are preferable. From now on, all our *WCSA*-based indexes will compress Ψ using our *Huffman-rle-opt* technique, setting $s = 2^{14}$.

3.3. The word-based SSA (WSSA)

The practical implementation considers the combinations shown in Table I for wavelet trees and their corresponding bitmap representations. The implementation of `RRR` is due to Claude and Navarro [2008] and the `BRW` implementation is due to González et al. [2005]. The wavelet tree implementations are the ones presented by Claude and Navarro [2008]; `ptrs` considers the regular pointer-based implementation and `noptrs` implements the wavelet tree using only $\log \sigma$ pointers, one per level. The `ptrs` wavelet tree can be shaped using Huffman codes, `huff`, or just as a balanced tree, denoted `balanced`.

Note that it is not possible to mix the `noptrs` variant with a Huffman shape, since `noptrs` uses the fact that there are exactly n bits per level. As explained in Section 2.2, using `RRR` gives space nH_k, while using Huffman gives nH_0, yet the k values we can achieve are rather

Table I. Space required for different combinations for representing the BWT in the SSA index.

Base combination	`balanced`	`huff`
`ptrs+BRW`	$n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$	$nH_0 + o(nH_0) + O(\sigma \log n)$
`ptrs+RRR`	$nH_k + o(n \log \sigma) + O(\sigma \log n)$	$nH_k + o(nH_0) + O(\sigma \log n)$
`noptrs+BRW`	$n \log \sigma + o(n \log \sigma)$	
`noptrs+RRR`	$nH_k + o(n \log \sigma)$	

low. Huffman shapes also reduce the redundancy from $o(n \log \sigma)$ to $o(nH_0)$. On the other hand, balanced trees reduce the pointer space from $O(\sigma \log n)$ to $O(\log \sigma \log n) = o(n \log \sigma)$.

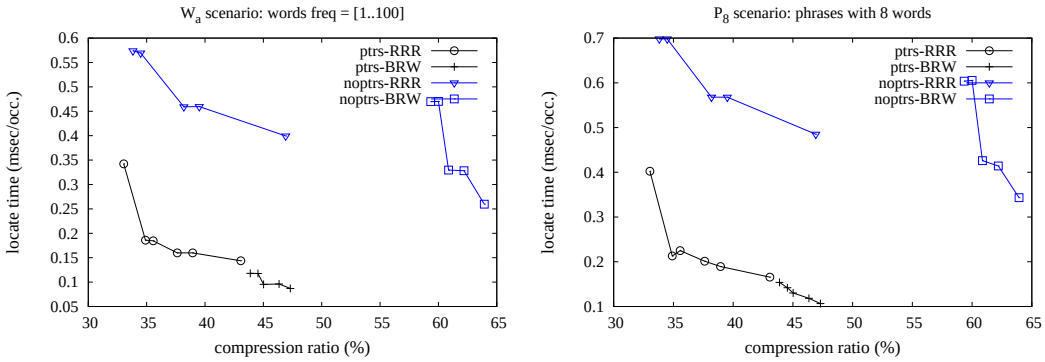

Fig. 3. Space/Locate-time tradeoff for different configurations of the *WSSA*.

Figure 3 shows the performance in *locate* for several configurations of the *WSSA* built over our 1GiB corpus. In this particular case, we only include the `ptrs` variant that uses Huffman shape (`huff`), since the `balanced` version is slower in practice, and the space required by the `huff` version tends to be smaller for our alphabet sizes [Claude and Navarro 2008]. The `ptrs` representation offers a better tradeoff than the `noptrs` counterpart.

The *WSSA* supports four tuning parameters, called t_{suff}, t_{pos}, t_{bit1}, and t_{bit2}. The t_{suff} and t_{pos} parameters determine the sampling density for the text and suffix array positions respectively, whereas t_{bit1} and t_{bit2} represent sampling parameters for the bitmaps. Both `RRR` and `BRW` support such parameters to adjust the space/time tradeoff. The t_{bit1} parameter determines the sampling for the bitmap B_A that marks the sampled suffixes (see Section 2.1). The second parameter, t_{bit2}, determines the sampling for the bitmaps in the wavelet tree.

In terms of compression, the `RRR` variant offers the best ratios. Given the space results obtained by the other indexes, we use the `ptrs` version, with Huffman shape and `RRR`, for the comparison with the other indexes.

4. FLEXIBLE WORD-BASED SELF-INDEXES

Our word-based self-indexes can be adapted so that they can deal with many typical requirements of natural language text searching: the possibility of using stemming, performing case-insensitive searches, disregarding stopwords and/or separators, etc. In this way, a search for "`compression of a SELF-index`" will be able to recover occurrences such as "`compressed self index`" or "`compressing self-indexes`". We call the resulting indexes *Flexible Word-Based Self-Indexes (FWSI)*.

Instead of indexing the original text, a *FWSI* uses a *normalized* version of it. Normalization is a user-defined function that maps any token (word or separator) from the original text to either a (normalized) word or a null word.

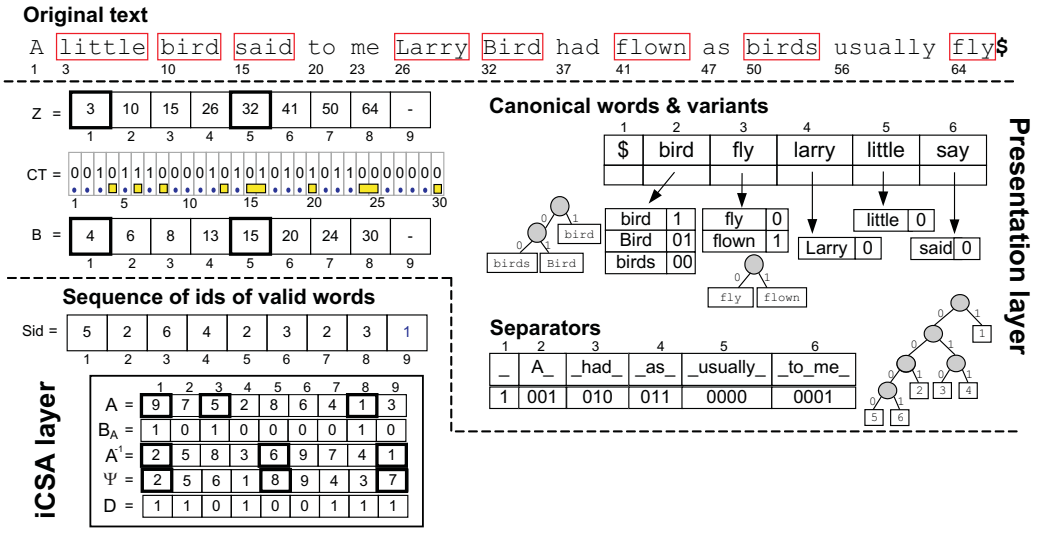

Fig. 4. General Structure of *FWCSA*.

Normalization gives the desired flexibility to *FWSI*s. One expects that separators are mapped to the null word, but this is not necessary; also, some words such as stopwords may be mapped to the null word. Normalization can be used for other purposes. For example, a text with many large numbers would have a huge vocabulary if we insisted in regarding each number as a word. Instead we may choose to regard each digit as a word and assume that there is an empty-string null word between any pair of digits. Similarly, if normalization wishes to treat some separators as valid words, we would also insert empty-string null words between normalized words. Thus any normalized word is followed by at least one null word.

To build a *FWSI* we process the source text, and for each token we obtain its normalized form (or the null word). Then, the resulting (non-null) normalized words are mapped to integer *ids*, and a sequence of ids (*Sid*) that represents the normalized version of the text is created, and an *ISI* is built over *Sid*, as in the *non-flexible* counterpart (Section 3). Note that, at this point, we would be able to perform searches over the normalized version of the source text. However, we could not actually recover any part of the original text (*extract*), as no information about the variants of a normalized word nor about the null words has been stored. Therefore, the *FWSI* must include additional information related to normalized words, variants and null words, on top of the *ISI* that represents *Sid*. All this extra information is called the *presentation layer*.

Different indexes can be obtained depending on the *ISI* chosen, and on how normalization is defined. In Figure 4 we show the general structure of a *FWSI*[6]. Apart from the *WSI*-like operations on the normalized text that are naturally inherited from its *ISI*, a *FWSI* supports the following character-based operations:

— *countFWSI*(P) counts the number of occurrences of the sequence of canonical words that normalization extracts from P. The separators between those canonical words are ignored.
— *locateFWSI*(P) gives the list of the positions of T, measured in byte offsets, where the canonical words of P match.
— *extractFWSI*(s, e) returns the substring $T[s, e]$.

[6]In this case we show the structures of a *Flexible WCSA*. In this example, normalization maps words to lower-case and use Porter's stemming algorithm ([Porter 1980]), and skips stopwords and separators.

4.1. Normalization and the Presentation Layer

A first pass over the original text is needed to gather some statistics. We split the text into "valid words" and "separators". A "valid word" is a token[7] that normalization does not map to the null word. On the other hand, we redefine "separator" in this section to mean all the text between valid words, that is, a maximal sequence of tokens that are mapped to the null word by normalization. Hence valid words and separators alternate strictly in the text[8]. A vocabulary of *canonical* (i.e., normalized) words is built, and kept sorted alphabetically. For each canonical word, a list with all the variants (i.e., tokens of the source text) that the normalization has mapped to it is stored (sorted by frequency). Similarly, a vocabulary containing all the separators in the source text is created and sorted by frequency.

A second pass over the original text permits to fill the structures from the presentation layer shown in Figure 4, as well as array *Sid*. Note that $Sid[5] = 2$ because the fifth valid word from the text, "`Bird`" is mapped via normalization to the second canonical word, "`bird`". As shown, once the presentation layer is built, the *ISI* is constructed over the sequence *Sid* (an *ICSA* in our example).

In the presentation layer, bitmap CT keeps a compressed representation of the presentation aspect of the text. Based on the alternation between valid words and separators, CT keeps the codeword of a variant of a canonical word, followed by the codeword of a separator, and so on. As an example, in Figure 4, we can observe that $CT[1 \dots 3] =$'001' is the codeword associated to the separator "`A `", $CT[4] =$'0' is the codeword of the variant "`little`" of the canonical word "`little`", $CT[5] =$'1' refers to a single *blank*, and $CT[6] =$'1' to the variant "`bird`" of the canonical word "`bird`", and so on.

Those codewords are obtained as follows. On the one hand, the Huffman algorithm [Huffman 1952] is run over the set of separators, based on their frequency of occurrence, and thus a codeword is assigned to each separator. Storing the shape of the Huffman tree requires little space overhead by using canonical Huffman [Moffat and Turpin 1997]. On the other hand, the set of variants of each canonical word is also encoded using Huffman codewords. Therefore, along with the variants of each canonical word, the shape of the Huffman tree used to encode the variants of that canonical word has also to be maintained for decoding. When a canonical word has a unique variant it is not actually encoded in CT (however, in the example in Figure 4 we used 1 bit for clarity). Section 4.2 discuses how to keep the information related to canonical words, variants and separators in a compact way. Together with the information related to the canonical words provided by *Sid* (which is not explicitly stored but obtained via the *ISI*), we can recreate the original text from the beginning, as $Sid[i]$ indicates which Huffman tree must be used when decoding the ith valid word from CT.

To enable decoding from any random word position in the text we provide synchronism in CT via a vector B. Given a position i in *Sid*, $B[i] = p$ tells the offset in CT from which the corresponding variant of the canonical word $j = Sid[i]$ can be decoded (using the Huffman tree associated to the jth canonical word). After decoding one symbol from that point p in CT, we will find the beginning of the codeword of a separator, and after it the codeword of the variant of the canonical word in $Sid[i + 1]$, and so on. In our example, we can see that $B[5] = 15$ is the beginning in CT of the codeword '01' that corresponds to a variant of the second ($Sid[5] = 2$) canonical word ('01' $\rightarrow$ variant "`Bird`"). Then, $CT[17 \cdots 19] =$'010' is the codeword of the separator "` had `".

To support the character-based operations *locateFWSI* and *extractFWSI*, a second array (Z) is needed. It maps any position i from *Sid* to its actual byte offset in the original text T: $Z[i] = j$ means that $T[j]$ is the first character of the token represented by $Sid[i]$.

[7]As explained, what is a token can also be user-defined.

[8]Since words and separators alternate strictly, we do not use the spaceless word model in our flexible self-indexes, but rather use separate dictionaries for words and separators.

To save space, both B and Z are sampled at regular positions $i \cdot t_b$ and $i \cdot t_z$, respectively, and only those positions are actually stored. A non-sampled value p from B ($i \cdot t_b < p < i \cdot (t_b+1)$) is obtained by just moving to the position $B[i \cdot t_b]$ in CT and then decoding alternatively $p - i \cdot t_b$ words and separators[9]. The number of decoded bits from CT added to the value $B[i \cdot t_b]$ tells us the value of $B[p]$. A non-sampled value p from Z is obtained similarly by adding to the previous sampled value $Z[i \cdot t_z]$ the number of characters decoded after processing $p - i \cdot t_z$ words and $p - i \cdot t_z$ separators. In this case, decoding should start at position $B[i \cdot t_z]$ in CT. Therefore it is wise that t_b divides t_z.

4.2. Detailed Structures of the Presentation Layer

We present the details of the structures that keep the words and separators of the presentation layer, that is, canonical words, variants, and separators.

4.2.1. Array of canonical words. Each time we need to find the *id* associated to a canonical word, a binary search (comparing strings) has to be performed. Figure 5.a) illustrates a simple setting where those canonical words are kept in plain form in a vector *bufferCanonicals*, and $O(1)$ time access to them is obtained with an array of pointers (*canonicals*). Each cell of the array occupies $\lceil \log_2 |bufferCanonicals| \rceil$ bits. The length of the ith word is obtained as $len(i) = canonicals[i+1] - canonicals[i]$. Note that since there are only 5 canonical words, $canonicals[6]$ is a virtual pointer to the end of *bufferCanonicals* just to permit us to easily compute $len(5) = canonicals[6] - canonicals[5]$.

We also tried a more sophisticated representation of the canonical words applying a *Front-Coding Hu-Tucker* technique [Brisaboa et al. 2011] that was recently used to compress string dictionaries. Basically our (sorted) dictionary of canonical words is partitioned into buckets of k words. By following a *Front Coding* approach [Witten et al. 1999], the first word within each bucket is explicitly stored and the rest are differentially encoded (both the size of the common prefix and the remaining suffix are stored for each entry). In the *Front-coding Hu-Tucker* version, the whole bucket[10] is compressed with Hu-Tucker coding. Searches for a given canonical word involve two phases: *i)* a binary search over the first entry of the buckets to find a candidate bucket; and *ii)* a sequential scan of the candidate bucket that rebuilds words within that bucket and performs the comparison with the searched word on the fly. The bucket size yields a time/space tradeoff. We performed some experiments to adjust that parameter and found that the best choice is to set bucket size to $k = 4$. In our 1GiB text, this reduced the size of the vocabulary from around 8MiB to around 3MiB while obtaining similar search times in general.

4.2.2. Variants of each canonical word. The variants of the canonical words are only needed when the text is decoded (*extractFWSI*) or for the *locateFWSI* operation. Recall from Section 4.1 that the variants of each canonical word are sorted by frequency so that word-oriented Huffman can be run on them. Yet, those variants are still compressible. We keep them compressed with a character-oriented Huffman. Figure 5.c) shows the three-level representation that gives constant-time access to the codeword associated to the first character of any variant. The third level stores the compressed variants. The second level (*subDir*) keeps both absolute values (for the first) and relative jumps (for the others) for addressing the codeword of the variants of a given canonical word. Finally, the first level, $j = Dir[i]$, points to the first position j in *subDir* where the data related to the ith canonical word is kept. Note that all the integers in the figure are encoded bitwise to reduce space. Assuming that an absolute sample in *SubDir* takes α bits ($\alpha = \lceil \log_2 max\_abs \rceil$) and a jump is encoded with γ bits ($\gamma = \lceil \log_2 max\_jump \rceil$), accessing to the beginning of the codeword of the jth

[9]Recall that canonical words with one variant have no counterpart in CT, but these are spotted as we extract the canonical words from the *ISI* and check their set of variants.
[10]The length of the common prefixes is also compressed with Vbyte [Williams and Zobel 1999].

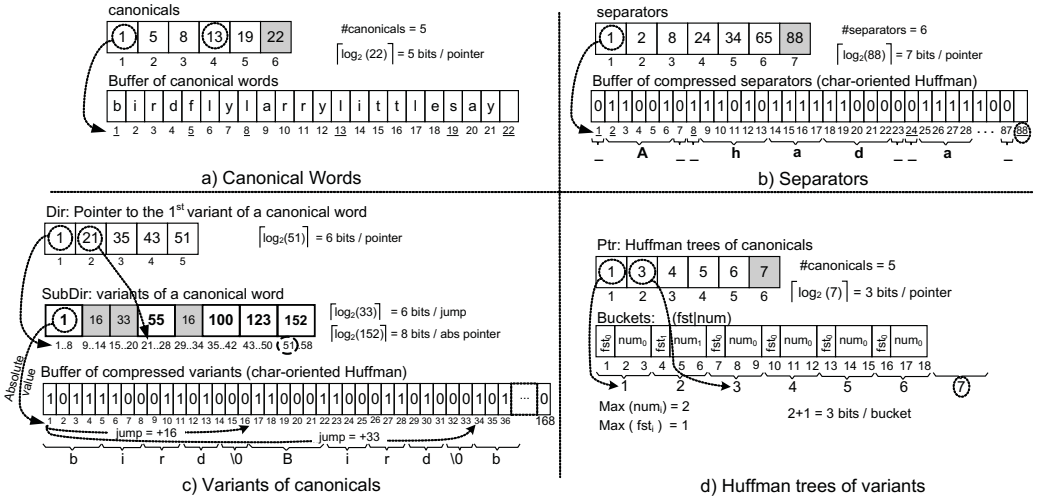

Fig. 5. Detailed structures used in *FWCSA* presentation layer.

variant of the ith canonical word involves: *i)* reading the absolute value abs from $SubDir$ at position $p \leftarrow Dir[i]$; *ii)* if $j > 1$, reading a $jump$ value at position $Dir[i] + \alpha + (j-2)\gamma$; *and iii)* computing $offset \leftarrow abs + jump$.

4.2.3. Huffman trees of variants. Storing the shape of a canonical Huffman tree only requires to keep its height H, and two H-element vectors known as num (number of elements in each level) and *first* (first codeword in each level) [Moffat and Turpin 1997]. This is usually a very small overhead. However, we must store the shape of the canonical tree of each canonical word (the tree used to encode its variants). Figure 5.d) shows a compact way to keep all those Huffman trees that allows constant-time access to any num or *first* value for any canonical word.

As an example, in Figure 5.d), $Ptr[2] - Ptr[1] = 2$ indicates that the height of the Huffman tree of the variants of the first canonical word is $3 - 1 = 2$. By accessing $bucket[Ptr[1]]$ and $bucket[Ptr[1] + 1]$ we obtain the values *first* and num for the two levels of such Huffman tree. As in Section 4.2.1, $Ptr[6]$ holds a virtual element that simplifies computing the height of the Huffman tree of variants for the 5th canonical word: $height(5) = Ptr[6] - Ptr[5]$.

Note that num and *first* hold very small values in practice, as the maximum number of variants of a given canonical word is usually small. In our experiments, 4 and 7 bits per entry were used for *first* and num, respectively.

4.2.4. Separators. Separators are stored in a similar structure to that of the plain representation of the canonical words, but compressed with character-based Huffman. That structure is shown in Figure 5.b).

4.3. Search Operations in the FWSI

When searching for a pattern $P = \langle w_1 \dots w_M \rangle$, we apply the same normalization used when indexing the text, and obtain a new filtered pattern $P^f = \langle w_1 \dots w_m \rangle$ $(m \leq M)$ composed of normalized words (the null words are discarded). Then, the words in P^f are binary searched for in the vocabulary of canonical words (to obtain their id), and a new pattern $P' = \langle id_1 \dots id_m \rangle$ is obtained.

In a *FWSI*, the searches typically require accessing Sid (through *countISI*, *locateISI*, and *extractISI*) in synchronization with array CT. The different search operations in a *FWSI* are discussed below.

— *countFWSI(P)* is converted to *countISI(P′)*, and answered in time $O(m \log n)$.
— *locateFWSI*(P) returns the char-offsets in the original text T where pattern P occurs. In this case, *locateISI*(P') gives the word offsets $i_1, \ldots, i_k$ of P' in *Sid*. Then, the byte offsets in the original text are obtained as $Z[i_1] \ldots Z[i_k]$.
— *extractFWSI*(s, e) recovers the substring $T[s, e]$ of the original text. An initial search within Z (first a binary search on the sampled values, completed with a sequential search) gives the last $Z[i] \leq s$. Then we synchronize the compressed text in CT by moving to position $B[i]$. From there on, we decompress using CT until the number of decompressed characters is greater than $e - Z[i]$. Characters before s and after e are skipped to form the answer.

We can also support *locateWSI* and *extractWSI*, that is, the word-based operations. The only difference with the *WSI* is that we have to normalize P first, and that in the case of *extractWSI* we have to consider the presentation layer in order to obtain the original text.

Other more complex searches, such as wildcards and approximate matching, could be supported through scanning the vocabulary. See Moura et al. [2000] for details.

5. EXPERIMENTAL RESULTS

We used for our experiments a text collection of around 1GiB, obtained by aggregating Calgary corpus[11] and several corpora from TREC-2[12]: AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF), as well as from TREC-4: Congressional Record 1993 (CR) and Financial Times 1991 to 1994.

An isolated Intel®Xeon®-E5520@2.26GHz with 72GiB-DDR3@800MHz RAM was used. 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. All of our experiments and indexes run entirely in RAM.

We present three main comparisons in this section. First, we compare the effects of either using or skipping the presentation layer in our self-indexes, by comparing *WCSA* with its flexible counterpart *FWCSA*. These results are shown in Section 5.1. In Section 5.2 we compare different configurations for a block-addressing inverted index (*II*) and choose the best setup depending on its space/time tradeoff. This best setup, that will be referred to as *II* stands for the best state-of-art competitor against our *WSI* variants. Finally, we show the comparison of the best *II* with our word-based self-indexes in Section 5.3.

Comparisons are focused on the in-memory size of the indexes, and in three main operations: *countWSI*, *locateWSI*, and *extractWSI* (or their flexible counterparts). For searches we used 8 sets of 100 random test patterns. Four sets were composed of single-word patterns with different frequency ranges: W_a, W_b, W_c, and W_d, and other four were formed by phrases of lengths $2, 4, 6$ and 8, and named P_2, P_4, P_6, and P_8 respectively. Table II describes them. Words were chosen uniformly at random from the vocabulary obtained by our *WSI*s. Although real-life queries follow a power-law distribution, this is independent of their distribution of frequencies in the text [Baeza-Yates and Navarro 2004]. This independence implies that a uniform sampling of queries yields the same average performance as real-life distributed ones. We discarded patterns consisting in either a separator or a stopword. To obtain the phrase patterns for our sets P_i, we accessed the source file at random positions and parsed i words (or separators) from there on, ensuring that each pattern contained at least one bare word. That is, the sum of words and separators in P_6 is 6, and at least one of them is a bare word. The normalization process in the flexible self-indexes has a pernicious effect on phrase searches, as some phrases containing more than one stopword could become single-word patterns. This could lead to an unfair comparison between flexible and

[11] `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`
[12] `http://trec.nist.gov/`

Table II. Description of our search patterns.

Set	Description	total # of occurrences		
		on *WSI*s	on *FWSI*s lowercase	on *FWSI*s Porter's
W_a	100 Random words with freqs in $[1, 100]$	1,135	14,789	170,840
W_b	100 Random words with freqs in $[101, 1000]$	30,664	151,470	668,021
W_c	100 Random words with freqs in $[1001, 10000]$	258,098	389,967	1,919,131
W_d	100 Random words with freqs in $[10001, \infty]$	2,273,565	3,220,383	5,382,130
P_2	100 Random phrases of 2 words	201,956	371,465	568,019
P_4	100 Random phrases of 4 words	4,415	677,748	964,047
P_6	100 Random phrases of 6 words	144	123,043	257,205
P_8	100 Random phrases of 8 words	169	2,032	6,000
P'_2	100 Random phrases of 2 words	32,359	40,114	58,189
P'_4	100 Random phrases of 4 words	4,057	8,139	11,442
P'_6	100 Random phrases of 6 words	131	319	665
P'_8	100 Random phrases of 8 words	108	303	362

non-flexible variants and explains why P_6 occurs 123,043 times on *FWSI*s and only 144 times on *WSI*s. To overcome this issue, we created four new sets P'_2, P'_4, P'_6, and P'_8 and we manually removed phrases that would become 1-word patterns after normalization. That is, P'_i sets contain at least two bare words. Our experiments in Section 5.1 use P'_i instead of P_i sets.

For *extractWSI* experiments, we created three sets of intervals $[i, i + w - 1]$, where i is a random word-rank on the sequence of words Sid ($0 \leq i \leq |Sid| - w$), and w is the interval width. We tried three different values for w (10, 100, and 1000 words), so that we start the extraction of text from the ith word and recover a substring containing the following 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 1000 words respectively. In all cases, we recover around 45MiB of plain text.

Finally, in Section 5.4 we include experiments on construction time and memory usage.

5.1. Flexible and non-flexible word-based self-indexes

In this section we compare *WSI*s with *FWSI*s. We focus the comparison on the *CSA*-based structures. In particular, we compare the *WCSA* with two versions of our *FWCSA* that use different normalization functions. The first variant, denoted $FWCSA^s$, uses a simple normalization that consists in removing stopwords and mapping to lower-case. The second, called $FWCSA^p$, also removes stopwords, but then it performs stemming using Porter's algorithm[13] [Porter 1980]. Both map all separators to the null word.

We used three different setups for each self-index by setting the parameters $\langle t_A, t_{A^{-1}}, t_\Psi \rangle$ to $\langle 16, 64, 16 \rangle$, $\langle 32, 32, 32 \rangle$, and $\langle 128, 256, 128 \rangle$. Parameters $\langle t_Z, t_B \rangle$ were set to $\langle 512, 32 \rangle$ for the *FWCSA* variants. Table III gives a breakdown of the space.

By indexing only normalized words, the flexible versions have to deal with less than 49% of the words ($|Sid|$) indexed by the *WCSA*. In addition, the normalization performed in the flexible indexes reduces the size of the vocabulary ($|V|$) from 885,637 words (and separators) in the *WCSA* to 745,512 and 657,826 canonical words in $FWCSA^s$ and $FWCSA^p$, respectively. Note that for $FWCSA^s$ and $FWCSA^p$ we show the size of the array of canonical words assuming that it is either kept in plain form (as in *WCSA*) or compressed using the *Front-Coding Hu-Tucker* approach. The smaller vocabulary in $FWCSA^p$ leads to a heavier presentation layer than that of $FWCSA^s$. Since $FWCSA^p$ obtains fewer different canonical words than $FWCSA^s$, $FWCSA^p$ produces more variants for each normalized word, and therefore such variants are encoded with longer codewords. This is the reason why array

[13] `http://tartarus.org/~martin/PorterStemmer/`

Table III. Space breakdown of our *CSA*-based self-indexes: $FWCSA^s$, $FWCSA^p$, and $WCSA$.

		$FWCSA^s$			$FWCSA^p$			$WCSA$		
Params	t_A	16	32	128	16	32	128	16	32	128
	$t_{A^{-1}}$	64	32	256	64	32	256	64	64	256
	t_Ψ	16	32	128	16	32	128	16	32	128
	t_Z	512	512	512	512	512	512	–	–	–
	t_B	32	32	32	32	32	32	–	–	–
$\|Sid\|$		111,885,753			111,885,753			228,707,234		
$\|V\|$		745,512			657,826			885,637		
ratio(%)		40.326	37.747	34.191	40.210	37.610	34.038	45.027	38.079	31.860
Presentation layer (size in KiB)										
CT		110,767	110,767	110,767	116,922	116,922	116,922	–	–	–
B		13,658	13,658	13,658	13,658	13,658	13,658	–	–	–
Z		854	854	854	854	854	854	–	–	–
variants		9,897	9,897	9,897	9,546	9,546	9,546	–	–	–
Huff var		2,662	2,662	2,662	2,872	2,872	2,872	–	–	–
separators		19,328	19,328	19,328	19,328	19,328	19,328	–	–	–
canonicals		3,422	3,422	3,422	2,967	2,967	2,967	9,882	9,882	9,882
Hu-Tucker		*3,422*	*3,422*	*3,422*	*2,967*	*2,967*	*2,967*	–	–	–
plain		*8,122*	*8,122*	*8,122*	*6,979*	*6,979*	*6,979*	*9,882*	*9,882*	*9,882*
total		160,588	160,588	160,588	166,147	166,147	166,147	9,882	9,882	9,882
ICSA **layer** (size in KiB)										
Ψ		193,309	172,920	157,582	186,519	165,910	150,404	318,757	273,352	239,119
D		18,780	18,780	18,780	18,780	18,780	18,780	38,388	38,388	38,388
A_S		27,316	13,658	3,414	27,316	13,658	3,414	55,837	27,918	6,980
B_A		18,780	18,780	18,780	18,780	18,780	18,780	38,388	38,388	38,388
A_S^{-1}		6,829	13,658	1,707	6,829	13,658	1,707	13,959	13,959	3,490
total		265,013	237,795	200,263	258,223	230,785	193,085	465,328	392,005	326,364

CT occupies around 5MiB more than in $FWCSA^s$. This loss of compression is partially compensated by storing data for fewer canonical words (variants and Huffman trees of variants). A different situation arises at the *ICSA* layer. Both $FWCSA^s$ and $FWCSA^p$ are built over a sequence of *ids* (Sid) of the same length. However, since $|V|$ is smaller in the $FWCSA^p$ (fewer different canonical words), it obtains a more compressible Sid sequence. Finally, we can see that the overall compression is slightly better in $FWCSA^p$ than in $FWCSA^s$ in all cases.

Comparing the similar compression ratio of the $WCSA$ (which self-indexes 228 million words) with that of its flexible counterparts (that index only 111 million words), we can conclude that the information that is not implicitly self-indexed is instead kept compressed explicitly in the presentation layer. Also we can see that $WCSA$ can be tuned to obtain a compression ratio around 30% whereas its flexible counterparts are hardly able to reach the 33% limit. The reason behind that is the redundancy introduced by compressors like Huffman.

Although $WCSA$ and $FWCSA$ offer different search capabilities, we also performed some experiments to compare their performance on searches. A first difference is due to normalization, as the average number of bare words in the sets of patterns P'_2, P'_4, P'_6, and P'_8 are respectively 2.00, 2.56, 3.42, and 3.81. Therefore, $FWCSA$ searches for shorter patterns than $WCSA$ in practice.

Figure 6 compares average *count* time for our self-indexes. Note that we have obtained additional dots in our plots by trying more parameter values. In addition, to have a fairer comparison with the $WCSA$, we include the regular $FWCSA$ versions and two additional variants that do not compress the array of canonical words. We refer to them as $FWCSA^{s|p(plain)}$. Compressing the canonical words saves some space (around 0.3 percentage points in compression ratio) but almost doubles *count* times for single-word patterns. Yet, it has a negligible influence in phrase queries.

As expected, the performance at counting for single words (with an uncompressed vocabulary) is similar in both the regular and the flexible self-indexes; this is shown in Figure 6-top-left where we include average count times when searching for all the patterns in the sets W_a to W_d. However, the flexible versions are a bit faster at *count* than the *WCSA* when searching for phrases due to the reduced vocabulary and because they search for fewer bare words. The only exception occurs for compression ratios $\leq 34.5\%$ where *FWCSA* must use a much sparser sampling than that of *WCSA*.

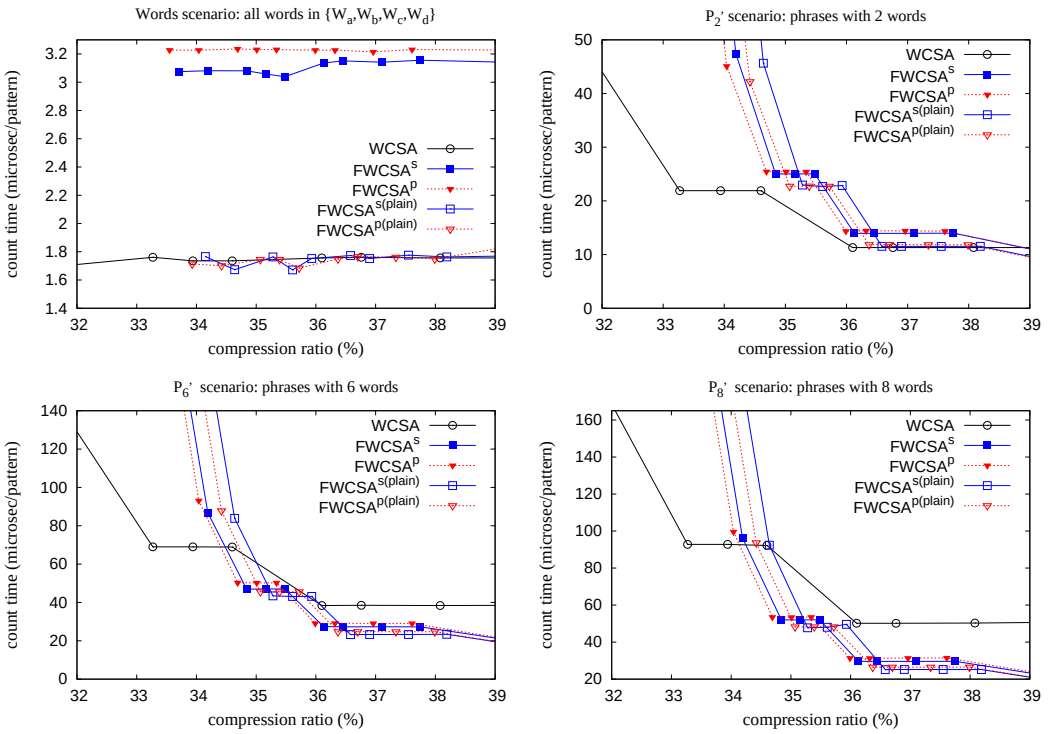

Fig. 6. Count space/time tradeoff for *WCSA*, $FWCSA^s$, and $FWCSA^p$.

Figures 7 and 8 show our experiments for *locate* and *extract* operations. We do not include here $FWCSA^{s|p(plain)}$ variants as they obtained the same *locate* times as the regular *FWCSA* counterparts. Recall also that the array of canonical words is not used during *extract*. In the case of *extract* the sets of intervals *10w*, *100w*, and *1000w* contain respectively: *i)* 10, 100, and 1000 words for *WCSA* (as explained before), and *ii)* 6, 51, and 501 bare words for *FWCSA*. In all cases, the sum of the lengths of the snippets extracted range from 45 to 50MiB.

Now, the flexible self-indexes have to pay for the synchronization required between the presentation layer (position in *CT*) and the *ICSA* layer. Therefore, at locating, *WCSA* obtains the best results except when a very dense sampling is used (ratio $\leq$38%). At *extract*, that costly synchronization can be partially compensated if long substrings of text are extracted (synchronizations occur only at the beginning). This is the reason why *FWCSA* performs best in the *1000w* scenario, and why *WCSA* performs much better as we decrease the number of words in the extracted text. Of course, it is not a matter of choosing one of them, since they fit different scenarios.

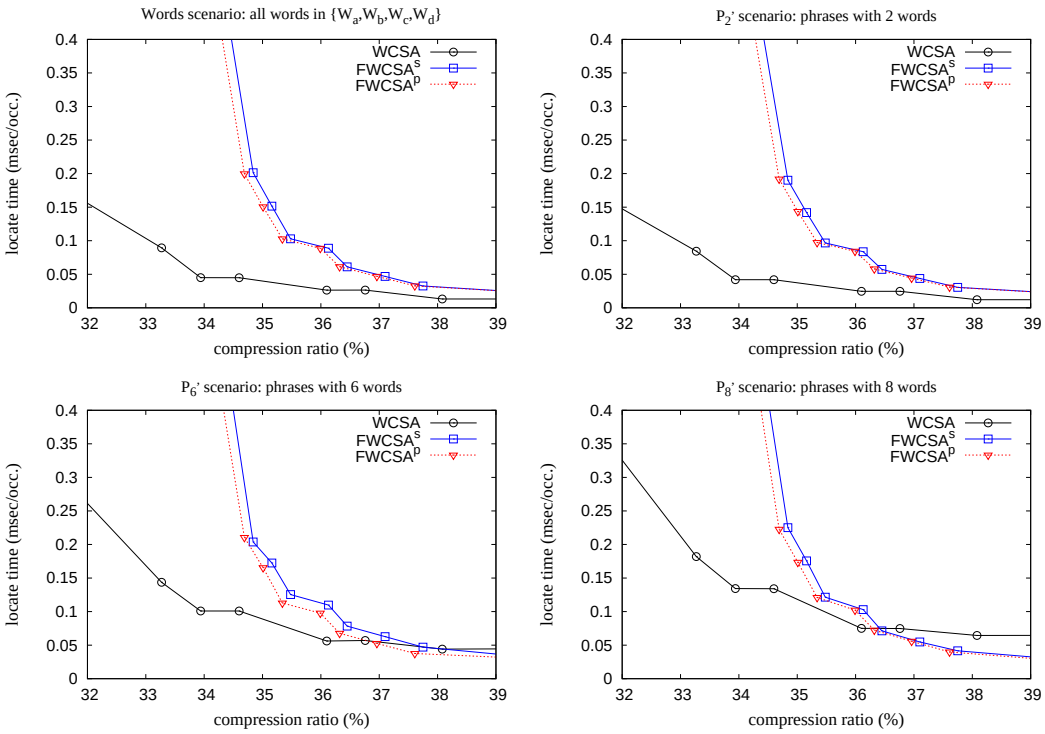

Fig. 7. Locate space/time tradeoff for $WCSA$, $FWCSA^s$, and $FWCSA^p$.

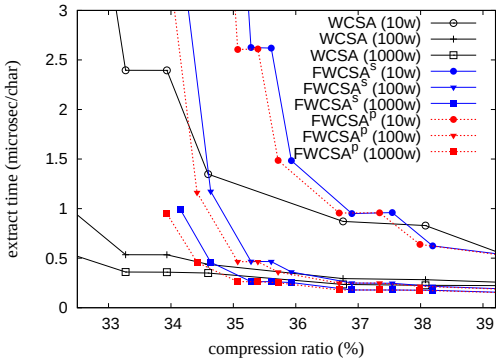

Fig. 8. Extract time/space tradeoff for $WCSA$, $FWCSA^s$, and $FWCSA^p$.

5.2. Implementing a good baseline block-addressing inverted index (II)

The main goal of our experimental section is to compare our word-based self-indexes with an inverted index with similar features and space requirements. For such reason, we opted for comparing with a good main-memory block-addressing inverted index built over compressed text [Navarro et al. 2000]. To save space, instead of keeping the exact position where each word occurs, the text is logically partitioned into fixed-length blocks and the block-addressing inverted indexes point to just blocks, Thus, all the occurrences of a given word inside the same block are stored only once in the corresponding posting list. The cost to pay

is that the posting lists are no longer enough to perform most search operations, and some scanning of the text is needed. Searches proceed as follows: First, the posting lists are traversed to identify candidate blocks that could contain a given pattern. When searching for a single word, this step requires fetching the block numbers from the posting list. Searches for phrases imply performing the intersection of the posting lists of the words in the phrase to obtain a list of candidate blocks. Finally, a further scanning of those candidate blocks is required to complete the search for the pattern.

The block size yields a space/time tradeoff. On the one hand, using small blocks leads to large indexes, but the inverted index is able to discard many blocks, and little data must typically be scanned during searches. On the other hand, using large blocks leads to smaller and slower inverted indexes. Note that the occurrences of phrases spanning through block boundaries could be skipped by the index as all the words in the phrase must occur within the same candidate block. This issue can be overcome by just adding some overlap between blocks. For example, an N-words overlap would ensure that all the occurrences of phrases including up to $N+1$ words would be found. Of course, this overlap would lead to some loss of compression and search time (the block is a bit larger), which would be more noticeable when using a small block-size. In our experiments we are giving some advantage to *II*s over their competitors and do not use overlapping blocks. Therefore, we permit *II* s to miss some occurrences.

Apart from the effect of the block size, the space/time tradeoff of the inverted indexes potentially depends on two additional parameters: the compression of the posting lists, and the compression technique applied to the source text.

— *Compressing and intersecting posting lists.* Most of the list compression algorithms rely on the fact that the posting lists are increasing, and that the differences between consecutive entries are smaller on the longer lists. Thus, a scheme that represents those differences with encodings that favor small numbers works well [Witten et al. 1999]. In this case, we opted for using *rice-codes* and *vbyte-codes* [Williams and Zobel 1999] to encode such differences. The problem of compressing the posting lists with variable-length codes is that the intersection of lists can only be carried out using a *merge*-type algorithm. This is known to be a good choice only when the intersected lists have similar lengths, typically if $|N| \leq 20|M|$, being M the shortest list, and N the longest one [Baeza-Yates 2004]. If $|N| > 20|M|$ a *set-versus-set* algorithm (*svs*) is preferred. The *svs* approach is based on searching the longest list for the elements of the shortest (typically using either binary or exponential search). However, this requires direct access to the longest list.
Different schemes to represent the posting lists have been devised to allow *svs*-type intersection over compressed lists [Culpepper and Moffat 2007; Sanders and Transier 2007; Moffat and Culppeper 2007], hence avoiding the decompression of the whole lists. These schemes store some sampled absolute values (so that direct access is provided to the sampled positions), and only partial decompression from the previous sample is required to access any position. We will refer to the scheme by Culpepper and Moffat [2007] as *cm-svs*, and to that by Sanders and Transier [2007] as *lookup*. Moffat and Culppeper [2007] proposed a hybrid representation for the lists, so that the lists of the most frequent words are represented with bitmaps (the ith bit is set if that word occurs in block i), and the remaining lists are represented following the scheme by Culpepper and Moffat [2007]. We will refer to this scheme as *cm-hybrid*.
We have implemented five different alternatives to compress the posting lists in our *II*, and combined all of them with both *rice-codes* and *vbyte-codes*. The resulting variants are: *i)* Using no-sampling and no-bitmap approaches, denoted *rice* and *vbyte* respectively; *ii) rice-bitmap* and *vbyte-bitmap*, where either rice or vbyte codes are used for the shorter lists and bitmaps for the longer ones. No sampling is used. As proposed by Moffat and Culppeper [2007] and assuming that u is the number of blocks, the list L of a given word

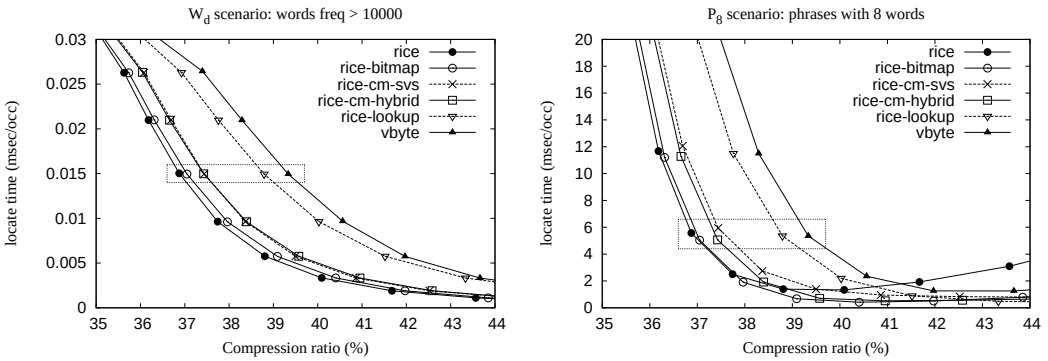

Fig. 9. Locate time/space tradeoff for the different strategies to compress the posting lists. Left: W_d (frequent words). Right: P_8 (long phrases).

is stored as a bitmap if $|L| > u/8$; *iii) rice-cm-svs* and *vbyte-cm-svs* using exponential search over the samples; *iv) rice-lookup* and *vbyte-lookup*; and finally, *v) rice-cm-hybrid* and *vbyte-cm-hybrid* approaches.

Figure 9 compares the rice-based variants and the *vbyte* technique. We measured the average locate time per occurrence of the *II* for the sets of patterns W_d and P_8. Different points were obtained by varying the block-size parameter. The source text was compressed with (s, c)-dense codes [Brisaboa et al. 2007], as we discuss soon. The results show that either the fetch time or the intersection time are a negligible part of the overall search time. Note that all the values within the highlighted rectangles in the plots are obtained by setting the block-size to 16KiB. Consequently, the gaps in locate time within those rectangles depend only on the fetch/intersection time, as searches scan exactly the same candidate blocks. Therefore, since the gaps in locate time among the different strategies are negligible, it is a good idea to choose the list compression technique yielding the best compression. This permits us to save some space that can be used to reduce the block size, that greatly affects the search time. The only exception is when we have many (small) blocks and we search for long phrases containing both short and long lists that are intersected in a *merge*-type fashion. This is the case of the *rice* variant (see Figure 9-right). In general, using vbyte-based compression of the lists is not promising in our inverted indexes due to its poor compression ratio. Among the *rice*-based variants, we can see that it is not worth to use sampling (except when we permit compression ratios over 42-43% and block size under 2KiB), as variants without sampling can typically trade that sampling space by almost halving the block size. For example, the *II* using *rice* with block size of 16KiB obtains a compression ratio around 36.8%, whereas *rice-cm-svs* obtains 36.7% with a block size of 32KiB and 37.4% with a block size of 16KiB, respectively. To sum up, *rice* and *rice-bitmap* obtain the best space/time tradeoff. However, *rice-bitmap* behaves much better on phrase-based searches. Thus we opt for *rice-bitmap* as the best choice for the *II*.

— *Compressing the source text.* As shown, searches in a block addressing inverted index require scanning some blocks. Such scanning can be boosted if direct searches are performed over the compressed text. It is well-known that performing searches over compressed text can be much faster than searching uncompressed text [Navarro et al. 2000; Brisaboa et al. 2007] if the text is compressed with a byte-oriented word-based compressor. Among these compressors, Tagged-Huffman code [Navarro et al. 2000] and the Dense Codes [Brisaboa et al. 2007] are the most successful techniques since they enable very fast Boyer-Moore-type direct searches over the compressed text (that is, the pattern is compressed and directly searched for over the compressed text, hence decompression is not needed for

searching). We choose (s, c)-Dense Code (*SCDC*) due to is excellent space/time tradeoff [Brisaboa et al. 2007]: Compression ratio around 30-35% (33.02% in our 1GiB text) and very fast at decompression and searches.
In addition, we also are interested in checking the viability of using a more powerful compressor such as the well-known bit-oriented Huffman coupled with a word-based modeler (*Huffword*) [Witten et al. 1999]. By using a bit-oriented encoding, *Huffword* compresses more than *SCDC* (compression ratio is 29.22% in our 1GiB corpus). However, it is much slower at decompression and does not permit Boyer-Moore type direct searching. Indeed, searches over text compressed with *Huffword* require simulating decompression one codeword at a time and checking if the decoded codeword belongs to the searched pattern. Aiming at performing decompression as fast as possible, we used canonical Huffman in our *Huffword* implementation and created an optimized decoder based on previous work [Moffat and Turpin 1997].
Figure 10 compares the resulting block-addressing inverted indexes built over both *SCDC* and *Huffword*. We show the average time per occurrence needed for locating. Space usage for *II* refers to the memory usage of the inverted index plus the size of the compressed text, that is, the amount of memory required to perform searches.
As expected, *SCDC* is always the best choice if we permit the index to use over 36% of the original text size. With a compression ratio around 36-34% the *SCDC*-based *II* is still able to overcome the *Huffword*-based counterpart in most cases (with the exception of the low/medium frequency single-word patterns). In this case, the *II* with *SCDC* must use a block size in the range 32KiB-16MiB, whereas the *Huffword*-based variant uses very small blocks of only 2-8KiB. For compression ratios around 34% searches are typically slow, being the *SCDC* variant close to its lower bound in space usage.
We can also see that when searching for phrase-patterns, the *II* built over text compressed with *SCDC* clearly benefits of scanning blocks in a Boyer-Moore-fashion, which is much faster than that over blocks compressed with *Huffword*.

To sum up, we have shown that the best block-addressing inverted index is generally obtained when we compress the text with *SCDC* and the posting lists with the *rice-bitmap* approach.

5.3. Word based self-indexes vs block addressing inverted indexes

In this section we compare our self-indexes with *II*s built over text compressed with either *SCDC* or *Huffword* and using the *rice-bitmap* approach to compress the posting lists.

We focus our comparison on three different setups of the indexes with different memory requirements, so that the indexes could occupy around 43%, 38%, and 35% of the size of the original text. We will refer to this scenarios as *large-index (l)*, *medium-index (m)*, and *small-index (s)* and to the corresponding indexes as $II^{l|m|s}$, $WCSA^{l|m|s}$, and $WSSA^{l|m|s}$. Such configurations were obtained by tuning the parameters of the indexes in the following way:

— *II*: As no sampling is done when *II* uses *rice-bitmap* to represent the posting lists, the only parameter for *II* is the block size. This parameter is set to 512 bytes in II^l, 32KiB in II^m, and 512KiB in II^s. Unless otherwise stated *II* will refer to the variant that compresses the text with *SCDC*.
— *WCSA*: We set the parameters $\langle t_A, t_{A^{-1}}, t_\Psi \rangle$ from the *WCSA* as follows: $\langle 32, 32, 16 \rangle$ for $WCSA^l$; $\langle 32, 64, 32 \rangle$ for $WCSA^m$; and $\langle 64, 64, 64 \rangle$ for $WCSA^s$.
— *WSSA*: We set the parameters $\langle t_{pos/suff}, t_{bit_1}, t_{bit_2} \rangle$ to $\langle 32, 16, 16 \rangle$, $\langle 64, 32, 32 \rangle$, and $\langle 128, 64, 64 \rangle$ to build $WSSA^l$, $WSSA^m$, and $WSSA^s$, respectively.

We compare the times for operations *count*, *locate*, and *extract*.

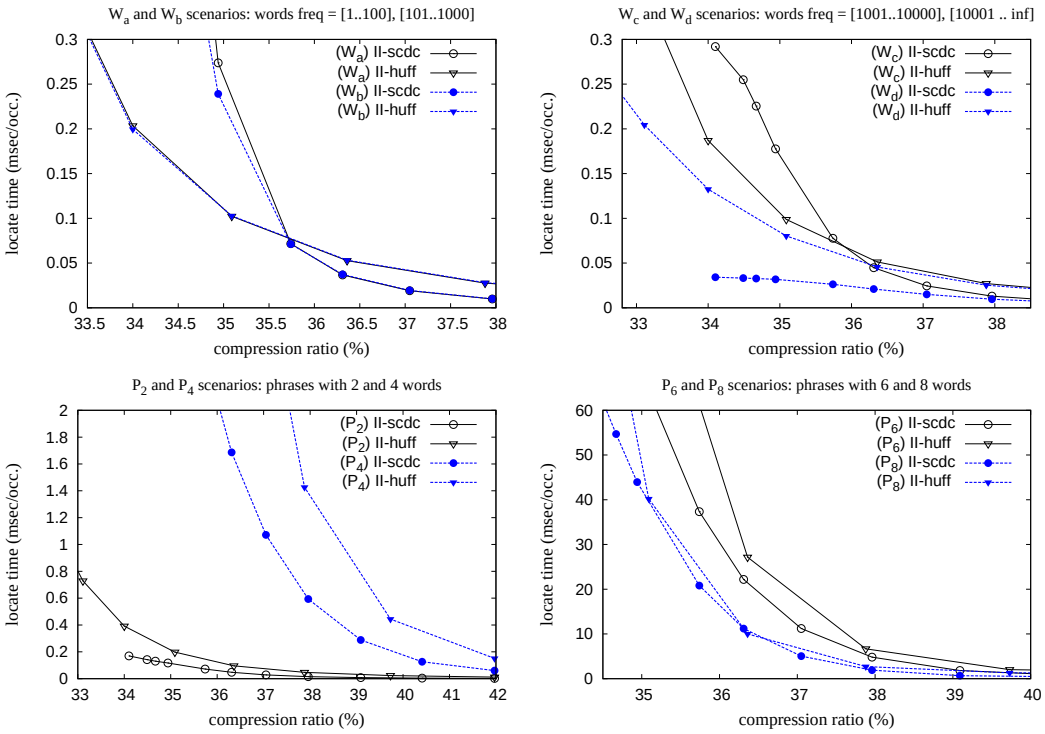

Fig. 10. Space/time tradeoff for *locate* obtained by the *II* using either SCDC or Huffword as text compressor.

Table IV. Counting time comparison of *II* with *WCSA* and *WSSA*.

		II			*WCSA*			*WSSA*		
		l	m	s	l	m	s	l	m	s
Ratio (%)		43.848	37.964	34.670	43.704	38.079	34.596	43.069	37.620	34.901
time (μsec/pat)	W_a	12.164	112.876	5955.999	1.720	1.700	1.720	1.112	1.184	1.336
	W_b	282.722	3071.000	121319.970	1.699	1.699	1.699	1.119	1.199	1.380
	W_c	2944.898	33849.553	582098.743	1.807	1.807	1.703	1.136	1.187	1.368
	W_d	25145.629	218489.597	743410.284	1.819	1.819	1.819	1.137	1.137	1.364
	P_2	3041.457	31101.224	263350.624	7.109	11.390	22.094	5.130	5.897	7.190
	P_4	1679.687	26205.939	185649.999	19.000	32.200	57.500	11.980	13.620	16.360
	P_6	1237.500	6945.000	119825.001	23.800	39.600	71.700	20.160	22.620	27.040
	P_8	1335.808	3230.001	92420.001	30.600	50.800	92.400	27.740	31.020	36.720

Count time. Table IV shows the results. One important advantage of our new self-indexes is that the time needed to count the occurrences of a pattern is $O(\log n)$, independent of the number of occurrences, whereas in the *II* the same operation as for locating the occurrences is required (i.e., an intersection of lists followed by the scanning of candidate blocks). In this operation the *II* cannot compete with our self-indexes. A possible way of partially avoiding this problem is to include the *number of occurrences* for each word along with the vocabulary (worsening compression ratio by around 0.75 percentage points). However, counting on phrases would not be improved. This feature makes our *WSI*s very interesting, for example, to estimate relevance of phrases at very low cost.

Among our self-indexes, the *WSSA* stands as the best choice for *count* in all scenarios. The *WCSA* is around 25-40% slower than *WSSA* on single-word patterns and 50-300% slower on phrase-patterns.

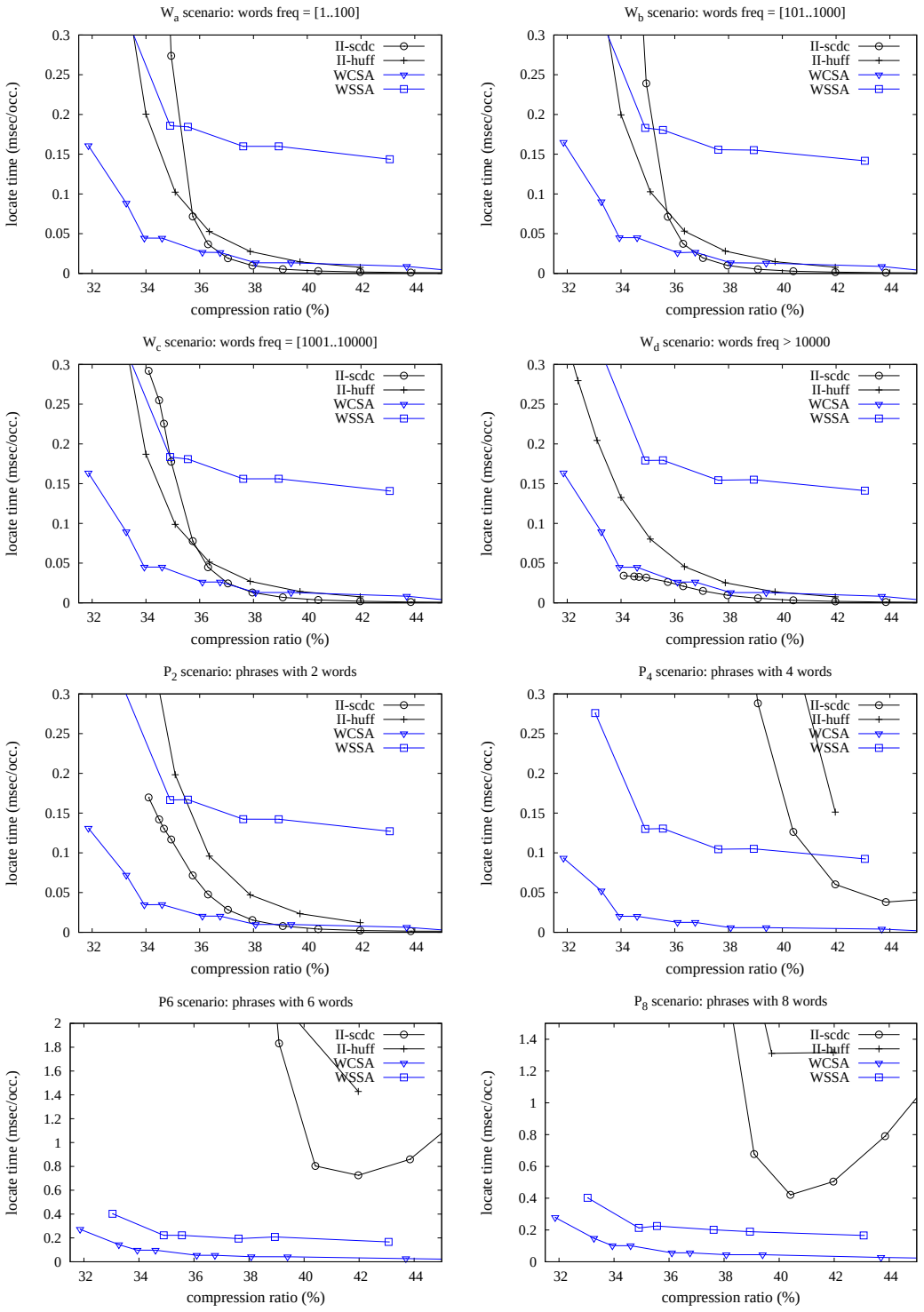

Fig. 11. Locate time/space tradeoff for *II*, *WCSA*, and *WSSA*.

Locate time. Figure 11 compares *locate* times. Again, we have included not only the 3 previous setups for our indexes, but also have added more dots in the plots by varying their parameters. In this operation the *WSSA* is clearly overcome by the *WCSA* in all cases. For example, with ratios around 43%, $WCSA^l$ is 5 to 20 times faster than $WSSA^l$. These gaps decrease slightly as we reduce the memory utilization of the self-indexes.

The *II* is clearly the best choice to search for single-word patterns when compression ratios are above 38%, or for very frequent patterns (where most of the blocks are probably being scanned and many occurrences are found in each block). In practice, II^l overcomes $WCSA^l$ by around 8-9 times, and II^m is still slightly faster than $WCSA^m$ in most cases. However, for ratios below 35% II^s can only overcome $WCSA^s$ for very frequent words (the set W_d). When searching for phrase-patterns, on the other hand, the *WCSA* is the clear winner, with the unique exception of short phrases (P_2) for compression ratios over 39%.

Finally, we can also see that our assumption that compressed text with *SCDC* was to be preferred over *Huffword* holds also for *locate*. Recall that the only exception is the search for single words when compression ratios are below 36%. Yet, we can see that in such scenario *II* s are clearly overcome by *WCSA* for operation *locate*.

Extract time. Recall that self-indexes contain an implicit representation of the indexed text. In this section we compare the *extractWSI*(s,e) time for *WCSA* and *WSSA*, that is, the time needed to recover the substring from the *s*th to the *e*th word in the source text. Note that *II* does not implement *extractWSI* as it indexes blocks rather than keeping complete positional information. Therefore, *II* cannot extract the same intervals as our self-indexes. However, as a rough approximation, we are also including *extract* time measures for our *II*s. In this case, we generated random byte-offsets within the *SCDC* compressed data and bit-offsets within the *Huffword* compressed data. For each of those offsets, *II*s pay for synchronization from the previous block beginning up to the given position, and finally extract 10, 100, or 1000 words from there on.

Figure 12 shows the results for different setups of *WSSA*, *WCSA*, and *II*s. On the one hand, by showing average extract time (in μsec) per character extracted, we observe that times decrease as the width of the interval increases. This happens because the extraction of the first word is very costly (recall it requires a synchronization step due to the sampling in the *ISI* layer, or from the previous block beginning), and from there on, extracting the following words proceeds sequentially. On the other hand, results show that *WCSA* is 3-8 times faster than *WSSA* for operation *extract* (Figure 12-bottom-right focuses on the comparison of *WCSA* with *WSSA*). Also we can observe that *II*s are typically faster than the self-indexes when using small block-sizes (hence synchronization is not a expensive operation) due to the fast decompression of *SCDC* and *Huffword*. For ratios $\leq$ 36%, the *SCDC*-based *II* is clearly overcome by the *WCSA*. The *Huffword*-based *II* behaves much better and overcomes *WCSA* in almost all cases. The only exception is when only 10-word snippets are extracted.

Although the whole collection will not probably be extracted in a real-case scenario, we have also measured the time needed by *WCSA* and *WSSA* to recover the whole original text. That is, we run *extractWSI*(0,228707233). It took 183, 243, and 376 seconds for $WCSA^l$, $WCSA^m$, and $WCSA^s$ respectively, whereas 1046, 1197, and 1366 seconds were needed for $WSSA^l$, $WSSA^m$, and $WSSA^s$ respectively. Of course, these times are much worse than those of a typical sequential text decompression such as the ones in our *II* structures. For example, *SCDC* decompresses our 1GiB text in just 10.6 seconds, and our *Huffword* implementation requires 20.5 seconds.

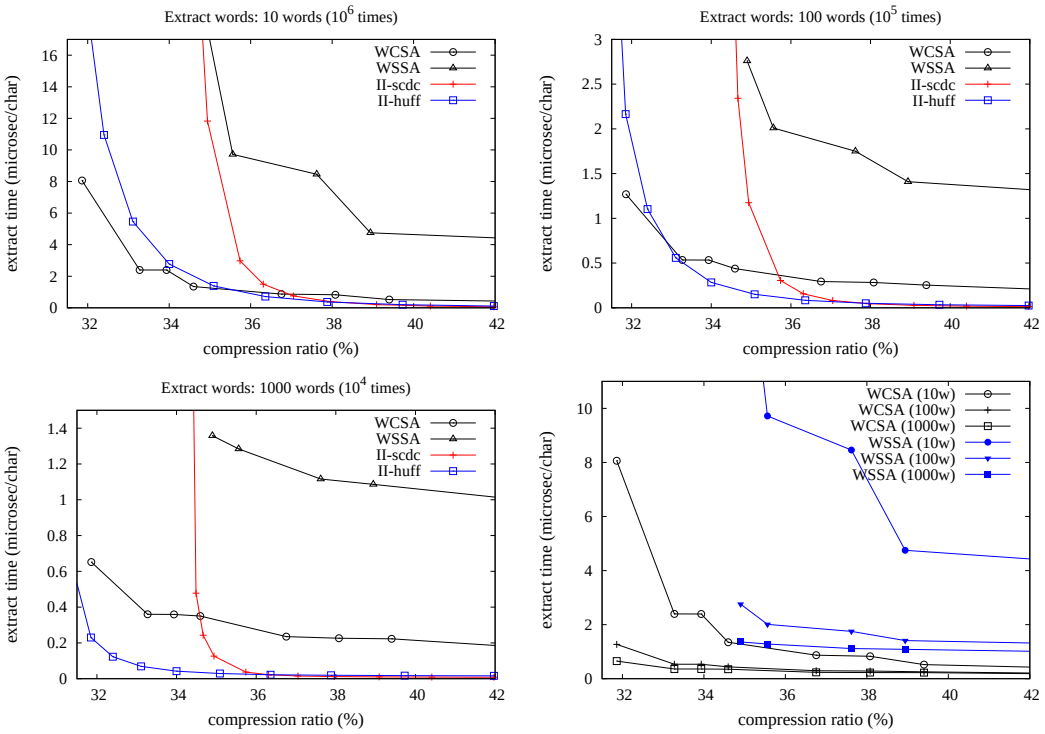

Fig. 12. Extract time/space tradeoff for *II*, *WCSA*, and *WSSA*.

5.4. Construction time and memory usage

We now present additional information regarding the construction cost of our indexes in our 1GiB text. We show both CPU user-time (in seconds), measured with command *time*, and peak memory usage (in MiB) obtained from file `/proc/<pid>/status`.

Table V. Construction time and peak memory usage for *II*, *WCSA*, and *WSSA*.

	Construction time (sec.)		Peak memory (MiB)		Comp. ratio (%)	
	large	small	large	small	large	small
$FWCSA^s$	189	191	1,807	1,806	40.326	34.191
WCSA	172	171	3,007	2,925	43.704	34.596
WSSA	792	951	5,023	4,928	43.069	34.901
II (*SCDC*)	66	57	1,476	592	43.848	34.670
II (*Huffword*)	110	102	1,408	559	39.433	31.035

We show two setups for each index, *large* and *small*, leading to different compression ratios. For *WCSA*, *WSSA*, and *II* (*SCDC*) we used exactly the same configurations as in Table IV. For *II* (*Huffword*) we set the block size to 512 bytes (large) and 256KiB (small), respectively, and compressed the posting lists with *rice-bitmap*. Finally, for $FWCSA^s$ we used the same setups as in Table III. Note that our programs for building the indexes are not heavily optimized to reduce memory utilization. We can see that the inverted indexes are the fastest and least memory demanding structures, requiring typically less memory than the size of the source file. The *WCSA* and *FWCSA* structures are built in around 3 minutes. In *WCSA* most of this time (around 130 sec.) is spent in the suffix sorting, whereas $FWCSA^s$ spends only around 70 sec. for suffix sorting and around 120 seconds for building

the presentation layer. Finally, *WSSA* is clearly the most time and memory demanding structure at construction.

5.5. Replicability and parameter tuning

Although we have considered one single collection for our experiments, the conclusions are most likely to apply in most other natural language scenarios. The collection is itself a standard one used in TREC benchmarks, and composed of various subcollections of different types.

Moreover, the performance of our indexes depend on rather stable characteristics of natural language collections. The space usage directly depends on the zero-order word-based entropy of the collection, which is always closely around 25% of the plain text size [Bell et al. 1990, Chapter 4], and on the vocabulary size, which is negligible on large collections [Heaps 1978]. Additional space depends deterministically on sampling parameters such as t_Ψ, etc. The time performance depends also deterministically on the pattern length and various sampling steps. That is, the effect of those parameters on the space/time performance is independent of the text data.

Finally, although there are various parameters in the indexes, these affect different aspects of the retrieval performance (such as text extraction speed, time to locate one occurrence, etc.), therefore they are easily tuned to fit different retrieval needs. We have explored various options for these, usually considering powers of 2, to cover a significant range of space/time tradeoffs.

As an external example of the robustness of our conclusions, recently Transier and Sanders [2010] compared our WCSA with inverted indexes in main memory. Using a denser sampling, they achieved compression ratios in the range 42%–53%, and confirmed that the WCSA is superior to inverted indexes for phrase searches.

6. CONCLUSIONS AND FUTURE WORK

We have shown how well-known self-indexes such as the *CSA* and the *SSA*, typically built over characters, can be adapted to deal with very large alphabets obtaining what we call *integer self-indexes* (*ISI*). We have also discussed how to apply them for self-indexing natural language text. The resulting word-based self-indexes (*WCSA* and *WSSA*) have shown to be very appealing alternatives to the traditional inverted indexes (*II*). Self-indexes compress the text *and* the index within a unique data structure. For compression ratios over 38-40% the *II* is still a relevant alternative, being actually the best choice for locating very frequent patterns. However, with compression ratios around 32-38%, the *WCSA* was shown to be the best choice to search for words and phrases.

The improvement in the case of phrases is particularly striking, because our self-indexes search for them in the same way as for words, whereas the *II* must carry out expensive intersection processes. This is probably the most important aspect where the new self-indexes deserve attention.

In addition, the new self-indexes overcome by several orders of magnitude the performance of the *II* at counting the number of occurrences of a given pattern, which can be relevant for example to estimate relevance of phrases at very low cost. Among them, the *WSSA* overcomes the *WCSA* at counting in all scenarios.

We have also presented in detail our *flexible word-based self-indexing* approach. We show that by plugging a *presentation layer* over an *ISI* we obtain new self-indexes that allow one to use typical operations of natural language retrieval, such as stemming, case-folding, skipping stopwords, etc. In addition, the resulting indexes are still very compact and fast, being directly comparable to their *non-flexible* counterparts.

As future work, following several recent research [Sadakane 2007; Hon et al. 2009; Gagie et al. 2010; Culpepper et al. 2010], we are targeting at enabling word-based self-indexes to

solve document-based queries instead of performing only text searches. Queries like listing the documents where a pattern appears (instead of all of its individual occurrences), determining the frequency of a pattern in a document, the number of distinct documents where it appears, or listing a few documents where the pattern is most frequent, can be used as a basis to carry out meaningful Information Retrieval operations on the text collection, for example implementing tf-idf based ranking. These kind of activities have been traditionally supported using a variant of inverted indexes [Baeza-Yates and Ribeiro-Neto 2011]. However, some recent results [Culpepper et al. 2010] suggest that the novel sequence-oriented indexes may be able to overcome inverted indexes on natural language collections. A recent example of adapting a word-based self-index to solve some document retrieval queries is given by Arroyuelo et al. [2010].

Albeit, as explained in the Introduction, indexes that run in main memory are of interest nowadays, it might still be interesting to try to compete with inverted indexes in secondary memory, which is the scenario in several applications. Our (F)WCSA is not a promising candidate as such. However, it is possible to combine recent advances in self-indexes for external memory [González and Navarro 2007; Chien et al. 2008; Sinha et al. 2008] with a word-based scheme.

Another point is the space needed to build the index in main memory. While the inverted index can be built in almost the same space of the final index, our (F)WCSA prototype needs to build the suffix array first. Recent advances in building self-indexes within compressed space [González and Navarro 2008; Okanohara and Sadakane 2009; Hon et al. 2009] should easily carry on to our scenario as well.

ACKNOWLEDGMENTS

Funded in part 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); by NSERC Canada and David R. Cheriton Scholarships Program (F.C.); and by Fondecyt grants 1-080019 and 1-110066, Chile (G.N).

REFERENCES

Apostolico, A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*. NATO ISI Series. Springer-Verlag, 85–96.

Arroyuelo, D., González, S., and Oyarzún, M. 2010. Compressed self-indices supporting conjunctive queries on document collections. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 6393. 43–54.

Baeza-Yates, R. 2004. A fast set intersection algorithm for sorted sequences. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 3109. 400–408.

Baeza-Yates, R. and Navarro, G. 2000. Block-addressing indices for approximate text retrieval. *Journal of the American Society for Information Science (JASIS) 51,* 1, 69–82.

Baeza-Yates, R. and Navarro, G. 2004. Modeling text databases. In *Recent Advances in Applied Probability*. Springer, 1–25.

Baeza-Yates, R. and Ribeiro-Neto, B. 2011. *Modern Information Retrieval* 2nd Ed. Addison-Wesley.

Baeza-Yates, R. and Salinger, A. 2005. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 3772. 13–24.

Barbay, J. and Kenyon, C. 2002. Adaptive intersection and t-threshold problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 390–399.

Barbay, J. and Kenyon, C. 2008. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms 4,* 1, article 4.

Barbay, J., López-Ortiz, A., and Lu, T. 2006. Faster adaptive set intersections for text searching. In *Proc. 5th International Workshop on Experimental Algorithms (WEA)*. 146–157.

Barbay, J., López-Ortiz, A., Lu, T., and Salinger, A. 2009. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics 14,* 3, article 7.

Bell, T., Cleary, J., and Witten, I. 1990. *Text compression*. Prentice Hall.

Brisaboa, N., Cánovas, R., Claude, F., Martínez-Prieto, M., and Navarro, G. 2011. Compressed string dictionaries. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*. LNCS 6630. 136–147.

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

Brisaboa, N., Fariña, A., Navarro, G., Places, A., and Rodríguez, E. 2008. Self-indexing natural language. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5280. 121–132.

Burrows, M. and Wheeler, D. J. 1994. A block-sorting lossless data compression algorithm. Tech. rep., Digital Equipment Corporation.

Chien, Y.-F., Hon, W.-K., Shah, R., and Vitter, J. 2008. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proc. Data Compression Conference (DCC)*. 252–261.

Claude, F. and Navarro, G. 2008. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5280. 176–187.

Culpepper, J. and Moffat, A. 2007. Compact set representation for information retrieval. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 4726. 137–148.

Culpepper, J. S. and Moffat, A. 2010. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS) 29,* 1, 1:1–1:25.

Culpepper, S. and Moffat, A. 2005. Enhanced byte codes with restricted prefix properties. In *Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 3772. 1–12.

Culpepper, S., Navarro, G., Puglisi, S., and Turpin, A. 2010. Top-k ranked document search in general text databases. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*. LNCS 6347. 194–205 (part II).

Demaine, E. and Munro, I. 2000. Adaptive set intersections, unions, and differences. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 743–752.

Ferragina, P., González, R., Navarro, G., and Venturini, R. 2009. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics 13*, article 12.

Ferragina, P. and Manzini, G. 2000. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*. 390–398.

Ferragina, P. and Manzini, G. 2005. Indexing compressed texts. *Journal of the ACM 52,* 4, 552–581.

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

Gagie, T., Navarro, G., and Puglisi, S. J. 2010. Colored range queries and document retrieval. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 6393. 67–81.

González, R., Grabowski, S., Mäkinen, V., and Navarro, G. 2005. Practical implementation of rank and select queries. In *Proc. Posters 4th Workshop on Efficient and Experimental Algorithms (WEA)*. 27–38.

González, R. and Navarro, G. 2007. A compressed text index on secondary memory. In *Proc. 18th International Workshop on Combinatorial Algorithms (IWOCA)*. College Publications, UK, 80–91.

González, R. and Navarro, G. 2008. Improved dynamic rank-select entropy-bound structures. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*. LNCS 4957. 374–386.

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

Grossi, R. and Vitter, J. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*. 397–406.

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

Hon, W. K., Sadakane, K., and Sung, W. K. 2009. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *SIAM Journal on Computing 38,* 6, 2162–2178.

Hon, W.-K., Shah, R., and Vitter, J. S. 2009. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th IEEE Symposium on Foundations of Computer Science (FOCS)*. 713–722.

Huffman, D. 1952. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E. 40,* 9, 1090–1101.

Jacobson, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*. 549–554.

Larsson, N. J. and Sadakane, K. 2007. Faster suffix sorting. *Theoretical Computer Science 387,* 3, 258–272.

Mäkinen, V. and Navarro, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12,* 1, 40–66.

Mäkinen, V. and Navarro, G. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG) 4,* 3, article 32.

Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22,* 5, 935–948.

Manzini, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

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

Moffat, A. and Culppeper, S. 2007. Hybrid bitvector index compression. In *Proc. 12th Australasian Document Computing Symposium (ADCS)*. 25–31.

Moffat, A. and Turpin, A. 1997. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications (TCOM) 45,* 10, 1200–1207.

Moura, E., Navarro, G., Ziviani, N., and 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 *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. LNCS 1180. 37–42.

Navarro, G. and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, article 2.

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

Okanohara, D. and Sadakane, K. 2009. A linear-time burrows-wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5721. 90–101.

Porter, M. F. 1980. An algorithm for suffix stripping. *Program 14,* 3, 130–137.

Puglisi, S. J., Smith, W. F., and Turpin, A. 2006. Inverted files versus suffix arrays for locating patterns in primary memory. In *Proc. 13th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 4209. 122–133.

Raman, R., Raman, V., and Rao, S. S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 233–242.

Sadakane, K. 2003. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms 48,* 2, 294–313.

Sadakane, K. 2007. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms (JDA) 5,* 1, 12–22.

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

Sinha, R., Puglisi, S., Moffat, A., and Turpin, A. 2008. Improving suffix array locality for fast pattern matching on disk. In *Proc. ACM SIGMOD Conference*. 661–672.

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

Transier, F. and Sanders, P. 2010. Engineering basic algorithms of an in-memory text search engine. *ACM Transactions on Information Systems (TOIS) 29*, article 2.

Weiner, P. 1973. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*. 1–11.

Williams, H. E. and Zobel, J. 1999. Compressing integers for fast file access. *The Computer Journal 42,* 3, 193–201.

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

Zobel, J. and Moffat, A. 2006. Inverted files for text search engines. *ACM Computing Surveys 38,* 2.