A Compressed Text Index on Secondary Memory

Rodrigo González * and Gonzalo Navarro **

Department of Computer Science, University of Chile.
{rgonzale,gnavarro}@dcc.uchile.cl

Abstract. We introduce a practical disk-based compressed text index that, when the text is compressible, takes little more than the plain text size (and replaces it). It provides very good I/O times for searching, which in particular improve when the text is compressible. In this aspect our index is unique, as compressed indexes have been slower than their classical counterparts on secondary memory. We analyze our index and show experimentally that it is extremely competitive on compressible texts.

1 Introduction and Related Work

Compressed full-text self-indexing [22] is a recent trend that builds on the discovery that traditional text indexes like suffix trees [25] and suffix arrays [17] can be compacted to take space proportional to the compressed text size, and moreover be able to reproduce any text context. Therefore self-indexes replace the text, take space close to that of the compressed text, and in addition provide indexed search into it. Although a compressed index is slower than its uncompressed version, it can run in main memory in cases where a traditional index would have to resort to the (one million times slower) secondary memory. In those situations a compressed index is extremely attractive.

There are, however, cases where even the compressed text is too large to fit in main memory. One would still expect some benefit from compression in this case (apart from the obvious space savings). For example, sequentially searching a compressed text is much faster than a plain text, because much fewer disk blocks must be scanned [26]. However, this has not been usually the case on indexed searching. The existing compressed text indexes for secondary memory are usually slower than their uncompressed counterparts.

A self-index built on a text $T_{1,n} = t_1 t_2 \dots t_n$ supports at least the following queries:

- $count(P_{1,m})$: counts the number of occurrences of pattern P in T.
- $locate(P_{1,m})$: locates the positions of all the occ occurrences found by $count(P_{1,m})$.
- $extract(l, r)$: extracts the subsequence $T_{l,r}$ of T.

The most relevant text indexes for secondary memory follow:

- The String B-tree [7] is based on a combination between B-trees and Patricia tries. $locate(P_{1,m})$ takes $O(\frac{m+occ}{\tilde{b}} + \log_{\tilde{b}} n)$ worst-case I/O operations, where $\tilde{b}$ is the disk block size measured in integers. This time complexity is optimal, yet the string B-tree is not a compressed index. Its static version takes about 5–6 times the text size plus text.
- The Compact Pat Tree (CPT) [4] represents a suffix tree in secondary memory in compact form. It does not provide theoretical space or time guarantees, but the index works well in practice, requiring 2–3 I/Os per query. Still, its size is 4–5 times the text size, plus text.

* Work supported by Mecesup Grant UCH 0109, Chile.
** Work supported by Fondecyt Grant 1-050493, Chile.

- The disk-based Suffix Array [2] is a suffix array on disk plus some memory-resident structures that improve the cost of the search. It takes at best $4 + m/h$ times the text size, plus text, and needs $2(1 + \log h)$ I/Os for counting and $\lceil occ/\tilde{b} \rceil$ I/Os for locating (in this paper $\log x$ stands for $\lceil \log_2(x + 1) \rceil$). This is not yet a compressed index.
- The disk-based Compressed Suffix Array (CSA) [15] adapts the main-memory compressed self-index [24] to secondary memory. It requires $n(H_0 + O(\log \log \sigma))$ bits of space (H_k is the kth order empirical entropy of T [18]). It takes $O(m \log_{\tilde{b}} n)$ I/O time for $count(P_{1,m})$. Locating requires $O(\log n)$ access per occurrence, which is too expensive.
- The disk-based LZ-Index [1] adapts the main-memory self-index [21]. It uses $8nH_k(T)+o(n \log \sigma)$ bits, where σ is the alphabet size. It does not provide theoretical bounds on time complexity, but it is very competitive in practice.

In this paper we present a practical self-index for secondary memory, which is based on the FM-index [8]. Depending on the available main memory, our data structure requires $2m$ to $4m$ accesses to disk for $count(P_{1,m})$ in the worst case. It locates the occurrences in $\lceil occ/\tilde{b} \rceil$ I/Os in the worst case, and on average in $cr \cdot occ/\tilde{b}$ I/Os, $0 < cr \leq 1$ is the *compression ratio* achieved: the compressed divided by original text size. Similarly, the time to extract $P_{l,r}$ is $\lceil (r-l+1)/b \rceil$ I/Os in the worst case (where b is the number of symbols on a disk block), multiplying that time by cr on average. With sufficient main memory our index takes $O(H_k \log(1/H_k) n \log n)$ bits of space, which in practice can be up to 4 times smaller than suffix arrays. Thus, our index is the first in being compressed and at the same time taking advantage of compression in secondary memory, as its locate and extract time are faster when the text is compressible. Counting time does not improve with compression but it is usually better than, for example, disk-based suffix arrays and CSAs. We show experimentally that our index is very competitive against the alternatives, offering a relevant space/time tradeoff when the text is compressible.

2 Background and Notation

We assume that the symbols of T are drawn from an alphabet $A = \{a_1, \ldots, a_\sigma\}$ of size σ. We will have different ways to express the size of a disk block: b will be the number of symbols, $\bar{b}$ the number of bits, and $\tilde{b}$ the number of integers in a block.

The suffix array $SA[1, n]$ of a text T contains all the starting positions of the suffixes of T, such that $T_{SA[1]...n} < T_{SA[2]...n} < \ldots < T_{SA[n]...n}$, that is, SA gives the lexicographic order of all suffixes of T. All the occurrences of a pattern P in T are pointed from an interval of SA.

The Burrows-Wheeler transform (BWT) is a reversible permutation T^{bwt} of T [3] which puts together characters sharing a similar context, so that k-th order compression can be easily achieved. There is a close relation between T^{bwt} and SA: $T_i^{bwt} = T_{SA[i]-1}$. This is the key reason why one can search using T^{bwt} instead of SA.

To calculate the inverse transformation, we introduce the following notation:

- For $c \in A$, $C[c]$ is the total number of occurrences of symbols in T (or T^{bwt}) which are alphabetically smaller than c.
- For $c \in A$, $Occ(c, q)$ is the number of occurrences of character c in the prefix $T^{bwt}[1, q]$.
- $LF(i) = C[T^{bwt}[i]] + Occ(T^{bwt}[i], i)$, the "$LF$ mapping".

LF lets us recover T, but more importantly, it lets us search for P in T, as seen next.

Algorithm *count*($P[1,m]$)
$i \leftarrow m, c \leftarrow P[m], \text{First} \leftarrow C[c]+1, \text{Last} \leftarrow C[c+1]$;
while (First $\leq$ Last) **and** ($i \geq 2$) **do**
    $c \leftarrow P[i-1]$;
    $\text{First} \leftarrow C[c] + Occ(c, \text{First}-1) + 1$;
    $\text{Last} \leftarrow C[c] + Occ(c, \text{Last})$;
    $i \leftarrow i-1$;
if (Last < First) **then return** 0 **else return** Last − First + 1;

Fig. 1. Algorithm to find and count the suffixes in SA prefixed by P (or the occurrences of P in T).

The FM-index [8, 9] is a self-index composed of a compressed representation of T^{bwt} and auxiliary structures to compute $Occ(c, q)$ for any $c \in A$ and $q \in [1 \ldots n]$.

To solve $count(P)$, the FM-index finds the first and last row of M (or suffix array position) prefixed by P. Fig. 1 gives the pseudocode to get the rows of M prefixed by P. Its time cost is at most $2(m-1)$ calls to Occ. Depending on the variant, each call to Occ can take constant time for small alphabets [8] or $O(\log \sigma)$ time in general [9], using wavelet trees (see next).

A rank/select dictionary over a binary sequence $B_{1,n}$ is a data structure that supports functions $rank_c(B, i)$ and $select_c(B, i)$, where $rank_c(B, i)$ returns the number of times c appears in prefix $B_{1,i}$ and $select_c(B, i)$ returns the position of the i-th appearance of c within sequence B.

Both *rank* and *select* can be computed in constant time using $o(n)$ bits of space in addition to B [20, 10], or $nH_0(B) + o(n)$ bits [23]. In both cases the $o(n)$ term is $\Theta(n \log\log n/\log n)$.

Let s be the number of one bits in $B_{1,n}$. Then $nH_0(B) \approx s \log \frac{n}{s}$, and thus the $o(n)$ terms above are too large if s is not close to n. Existing lower bounds [19] show that constant-time *rank* can only be achieved with $\Omega(n \log\log n/\log n)$ extra bits. As in this paper we will have $s << n$, we are interested in techniques with less overhead over the entropy, even if not of constant-time (this will not be an issue for us). One such rank dictionary [13] encodes the gaps between successive 1's in B using δ-encoding and adds some data to support a binary-search-based *rank*. It requires $s(\log \frac{n}{s} + \frac{\log n}{\log s} + 2 \log\log \frac{n}{s}) + O(\log n)$ bits of space and supports *rank* in $O(\log s)$ time. We call this structure GR.

The wavelet tree [12] $wt(S)$ over a sequence $S[1, n]$ is a perfect binary tree of height $\lceil \log \sigma \rceil$, built on the alphabet symbols, such that the root represents the whole alphabet and each leaf represents a distinct alphabet symbol. If a node v represents alphabet symbols in the range $A^v = [i, j]$, then its left child v_l represents $A^{v_l} = [i, \frac{i+j}{2}]$ and its right child v_r represents $A^{v_r} = [\frac{i+j}{2} + 1, j]$. We associate to each node v the subsequence S^v of S formed by the characters in A^v. However, sequence S^v is not really stored at the node. Instead, we store a bit sequence B^v telling whether characters in S^v go left or right, that is, $B^v_i = 1$ if $S^v_i \in A^{v_r}$. The wavelet tree has all its levels full, except for the last one that is filled left to right.

In this paper S will be T^{bwt}. A plain wavelet tree of S requires $n \log \sigma$ bits of space. If we compress the wavelet tree using a numbering scheme [23] we obtain $nH_k(T) + o(n \log \sigma)$ bits of space for any $k \leq \alpha \log_\sigma n$ and any $0 < \alpha < 1$ [16].

The wavelet tree permits us to calculate $Occ(c, i)$ using binary ranks over the bit sequences B^v. Starting from the root v of the wavelet tree, if c belongs to the right side, we set $i \leftarrow rank_1(i)$ over vector B^v and move to the right child of v. Similarly, if c belongs to the left child we update $i \leftarrow rank_0(i)$ and go to the left child. We repeat this until reaching the leaf that represents c, where the current i value is the answer to $Occ(c, i)$.

The locally compressed suffix array (LCSA) [11], is built on well-known regularity properties that show up in suffix arrays when the text they index is compressible [22]. The LCSA uses differential encoding on SA, which converts those regularities into true repetitions. Those repetitions are then factored out using Re-Pair [14], a compression technique that builds a dictionary of phrases and permits fast local decompression using only the dictionary (whose size one can control at will, at the expense of losing some compression). Also, the Re-Pair dictionary is further compressed with a novel technique. The LCSA can extract any portion of the suffix array very fast by adding a small set of sampled absolute values. It is proved in [11] that the size of the LCSA is $O(H_k \log(1/H_k) n \log n)$ bits for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$.

The LCSA consists in three substructures: the sequence of phrases SP, the compressed dictionary CD needed to uncompress the phrases and the absolute sample values to restore the suffix array values. One disadvantage of the original structure is the space and time needed to construct it. In [5] they present a heuristic to overcome this, as it can run with limited main memory and performs sequential passes on disk. It might not choose the pairs to replace as well as the original algorithm, but it can trade construction time for precision.

3 A Compressed Secondary Memory Structure

We describe an FM-Index-like structure on secondary memory, which is able to answer count, locate and extract queries. It is composed of three substructures, each one responsible for one type of query, and allowing diverse trade-offs depending of how much main memory space they occupy.

3.1 Entropy-compressed rank dictionary on secondary memory

As we will require several bitmaps in our structure with few bits set, we describe an entropy-compressed rank dictionary, suitable for secondary memory, to represent a binary sequence $B_{1,n}$. In case it fits in main memory, we use GR (Section 2). Otherwise we will use DEB, the δ-encoded form of B: we encode the gaps between consecutive 1's in B as variable-length integers, so that x is represented using $\log x + 2 \log\log x$ bits. DEB uses at most $s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)$ bits of space [16, Sec. 3.4.1]. Let $\bar{b}$ be the number of bits contained in a disk block. We split DEB into blocks of at most $\bar{b}$ bits: if a δ-encoding spans two blocks we move it to the next block. Each block is stored in secondary memory and, at the beginning of block i, we also store the number of 1's accumulated up to block $i-1$; we call this value OB_i. To access DEB, we use in main memory an array B^a, where $B^a[i]$ is the number of bits of B represented in blocks 1 to $i-1$. B^a uses $(s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)) \frac{\log n}{b}$ bits of space.

To answer $rank_1(B, i)$ with this structure, we carry out the following steps: (1) We binary search B^a to find j such that $B^a[j] \leq i < B^a[j+1]$. (2) We read block j from disk. (3) We decompress the δ-encodings in block j until reaching position i, summing up the bits set. (4) $rank_1(B, i)$ will be the previous sum plus OB_i, the accumulator of 1's stored in the block.

Overall this costs $O(\log \frac{s}{b} + \log\log \frac{n}{s} + \bar{b}) = O(\log s + \log\log n + \bar{b})$ CPU time and just one disk access. When we use these structures in the paper, s will be $\Theta(n/b)$. Table 1 shows some real sizes and times obtained for the structures, when $s = n/b$. As it can be seen, we require very little main memory for the second scheme, and for moderate-size bitmaps even the GR option is good.

Table 1. Different sizes and times obtained to answer *rank*, for some relevant choices of n and b. DEB is stored in secondary memory and is accessed using B^a. B^a and GR reside in main memory. Tb, Gb, etc. mean terabits, gigabits, etc. TB, GB, etc. mean terabytes, gigabytes, etc.

Structure	Space (bits)	CPU time for $rank$	Real space if $s = n/b$			
			$n = 1$ Tb $b = 32$ KB	$n = 1$ Gb $b = 8$ KB	$n = 1$ Gb $b = 4$ KB	$n = 1$ Mb $b = 4$ KB
GR	$s \log \frac{n}{s} + s \frac{\log n}{\log s} + 2s \log\log \frac{n}{s} + O(\log n)$	$O(\log s)$	100 MB	354 KB	667 KB	677 B
$DEB+$	$s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)\ +$		93 MB	326 KB	613 KB	600 B
B^a	$(s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)) \frac{\log n}{b}$	$O(\log s + \log\log n + \bar{b})$	14 KB	153 B	575 B	1 B

3.2 Counting

To answer a counting query we need to run the algorithm of Fig. 1. Table C uses $\sigma \log n$ bits and easily fits in main memory, thus the problem is how to calculate *Occ*.

We describe four different structures to count depending on how we represent T^{bwt}. We enumerate the versions from 1 to 4. In versions 1 and 2, we use an uncompressed form of T^{bwt} and pay one I/O per call to *Occ*. In versions 3 and 4, we use a compressed form of T^{bwt} and pay one or two I/Os per call to *Occ*. In versions 1 and 3, we spend $O(b)$ CPU operations per call to *Occ*. In versions 2 and 4, this is reduced to $O(\log \sigma)$. Version 4 is omitted from now on as it is not competitive.

To calculate $Occ(c, i)$, we need to know the number of occurrences of symbol c before each block on disk. To do so, we store a two-level structure: the first level stores for every t-th block the number of occurrences of every c from the beginning, and the second level stores the number of occurrences of every c from the last t-th block. The first level is maintained in main memory and the second level on disk, together with the representation of T^{bwt} (i.e., the entry of each block is stored within the block). Let K be the total number of blocks. We define:

- $E_c(j)$: number of occurrences of symbol c in blocks 0 to $(j-1) \cdot t$, with $E_c(0) = 0$, $1 \leq j < \lfloor K/t \rfloor$.
- $E'_c(j)$: j goes from 0 to $K - 1$. For $j \bmod t = 0$ we have $E'_c(j) = 0$, and for the rest we have that $E'_c(j)$ is the number of occurrences of symbol c in blocks from $\lfloor j/t \rfloor \cdot t$ to $j - 1$.

Summing up all the entries, E uses $\lceil K/t \rceil \cdot \sigma \log n$ bits and E' uses $K \cdot \sigma \log(t \cdot n/K)$ bits of space in versions 1 and 2. In version 3, the numbering scheme [23] has a compression limit $n/K \leq b \cdot \log n/(2 \log\log n)$. Thus, for version 3, E' uses at most $K \cdot \sigma \log(t \cdot \frac{b \log n}{2 \log\log n})$ bits.

To use these structures, we first need to know in which block lies $T^{bwt}[i]$:

- In versions 1 and 2, where the block size is constant, we know directly that $T^{bwt}[i]$ belongs to block $\lfloor i/b \rfloor$, where b is the number of symbols that fit in a disk block.
- In version 3, the block size is variable. Compression ensures that there are at most n/b blocks. We use a binary sequence $EB_{1,n}$ to mark where each block starts. Thus the block of $T^{bwt}[i]$ is $rank_1(EB, i)$. We use an entropy-compressed rank dictionary (Section 2) for EB. If we need to use the DEB variant, we add up one more I/O per access to T^{bwt} (Section 3.1) .

With this sorted out, we can compute $Occ(c, i) = E_c(j \text{ div } t) + E'_c(j) + Occ'(B_j, c, \mathit{offset})$, where j is the block where i belongs, *offset* is the position of i within block j, and $Occ'(B_j, c, \mathit{offset})$ is the number of occurrences of symbol c within block B_j up to *offset*. Now we explain the three ways to represent T^{bwt}, and this will give us three different ways to calculate $Occ'(B_j, c, \mathit{offset})$.

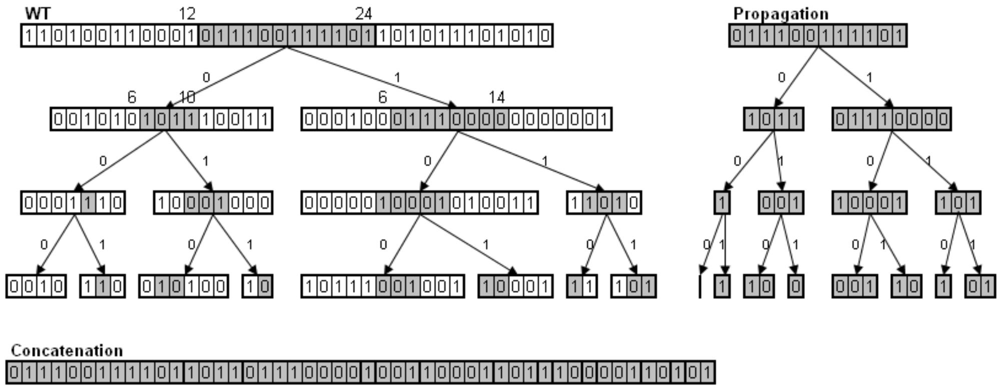


Fig. 2. Block propagation over the wavelet tree. Making ranks over the first level of WT ($rank_0(12) = 6$, $rank_0(24) = 10$ and $rank_1(i) = i - rank_0(i)$), we determine propagation over the second level of WT, and so on.

Version 1. We use directly T^{bwt} without any compression. If a disk block can store b symbols (ie, $b \log \sigma$ bits), we will have $K = \lceil n/b \rceil$ blocks. $Occ'(B_j, c, \mathit{offset})$ is calculated by traversing the block and counting the occurrences of c up to *offset*.

Version 2. Let b be the number of symbols within a disk block. We divide the first level of $WT = wt(T^{bwt})$ into blocks of b bits. Then, for each block, we gather its propagation over WT by concatenating the subsequences in breadth-first order, thus forming a sequence of $b \log \sigma$ bits. See Fig. 2. Note that this propagation generates 2^{j-1} intervals at level j of WT. Some definitions:

- B_i^j: the i-th interval of level j, with $1 \leq j \leq \lceil \log \sigma \rceil$ and $1 \leq i \leq 2^{j-1}$.
- L_i^j: the length of interval B_i^j.
- O_i^j/Z_i^j: the number of 1's/0's in interval B_i^j.
- $D_j = B_1^j \ldots B_{2^{j-1}}^j$ with $1 \leq j \leq \lceil \log \sigma \rceil$: all concatenated intervals from level j.
- $B = D_1 D_2 \ldots D_{\lceil \log \sigma \rceil}$: concatenation of all the D_j, with $1 \leq j \leq \lceil \log \sigma \rceil$.

Some relationships hold: (1) $L_i^j = O_i^j + Z_i^j$. (2) $Z_i^j = rank_0(B_i^j, L_i^j)$. (3) $L_i^j = Z_{(i+1)/2}^{j-1}$ if i is odd (B_i^j is a left child); $L_i^j = O_{i/2}^{j-1}$ otherwise. (4) $|D_j| = L_1^1 = b$ for $j < \lfloor \log \sigma \rfloor$, the last level can be different if σ is not a power of 2. With those properties, L_i^j, O_i^j and Z_i^j are determined recursively from B and b. We only store B plus the structures to answer $rank_1$ on it in constant time [10]. Note that any $rank_1(B_i^j)$ is answered via two ranks over B.

Fig. 3 shows how we calculate Occ' in $O(\log \sigma)$ constant-time steps. To navigate the wavelet tree, we use some properties:

1. Block D_j begins at bit $(j-1) \cdot b + 1$ of B, and $|B| = b \log \sigma$.
2. To know where B_i^j begins, we only need to add to the beginning of D_j the length of $B_1^j, \ldots, B_{i-1}^j$. Each B_k^j, with $1 \leq k \leq i-1$, belongs to a left branch that we do not follow to reach B_i^j from the root. So, when we descend through the wavelet tree to B_i^j, every time we take a right branch we accumulate the number of bits of the left branch (zeroes of the parent).
3. *node* is the number of the current interval at the current *level*.
4. We do not calculate B_{node}^{level}, we just maintain its position within B.

Algorithm $Occ'(B, c, j)$
$node \leftarrow 1; ans \leftarrow j; des \leftarrow 0; B_1^1 = B[1, b];$
for $level \leftarrow 1$ **to** $\lceil \log \sigma \rceil$ **do**
  if c belongs to the left subtree of $node$ **then**
    $ans \leftarrow rank_0(B_{node}^{level}, ans);$
    $len \leftarrow Z_{node}^{level};$
    $node \leftarrow 2 \cdot node - 1;$
  else $ans \leftarrow rank_1(B_{node}^{level}, ans);$
    $len \leftarrow O_{node}^{level}; \quad des \leftarrow Z_{node}^{level};$
    $node \leftarrow 2 \cdot node;$
  $B_{node}^{level} = B[level \cdot b + des + 1, level \cdot b + des + len];$
return ans;

Fig. 3. Algorithm to obtain the number of occurrences of c inside a disk block, for version 2.

Table 2. Different sizes and times obtained to answer $count(P_{1,m})$.

Version	Main Memory	Secondary Memory	I/O	CPU
1	$O(\frac{n}{bt} \cdot \sigma \log n)$	$n \log \sigma + O(\frac{n}{b} \cdot \sigma \log(t \cdot b))$	$2 \cdot m$	$O(m \cdot b)$
2	$O(\frac{n}{bt} \cdot \sigma \log n)$	$n \log \sigma + O(\frac{n}{b} \cdot \sigma \log(t \cdot b))$	$2 \cdot m$	$O(m \log \sigma)$
3a	$O(\frac{n}{bt} \cdot \sigma \log n + \frac{n}{b} \log n)$	$nH_k(T) + O(\sigma^{k+1} \log n) + O(\frac{n}{b} \cdot \sigma \log(t \cdot b \log n))$	$2 \cdot m$	$O(m(b + \log n))$
3b	$O(\frac{n}{bt} \cdot \sigma \log n + \frac{n}{b^2} \log n \log b)$	$nH_k(T) + O(\sigma^{k+1} \log n) + O(\frac{n}{b} \cdot \sigma \log(t \cdot b \log n))$	$4 \cdot m$	$O(m(b + \log n))$

Version 3. We compress block B from version 2 using a numbering scheme [23], yet without any structure for $rank$. In this case the division of T^{bwt} is not uniform, but we add symbols from T^{bwt} to the disk block as long as its compressed WT fits in the block. By doing this, we compress T^{bwt} to $nH_k + O(\sigma^{k+1} \log n + n \log \log n / \log n)$ bits for any k [16]. To calculate $Occ'(B, c, offset)$, we decompress block B and apply the same algorithm of version 2, in $O(b)$ time.

In Table 2 we can see the different sizes and times needed for our three versions. We added the times to do $rank$ on the entropy-compressed bit arrays.

3.3 Locating

Our locating structure will be the LCSA, see Section 2. The array SP from LCSA will be split into disk blocks of $\tilde{b}$ integers. Also, we will store in each block the absolute value of the suffix array at the beginning of the block. For optimization of I/O, the dictionary will be maintained in main memory. So we compress the differential suffix array until we reach the desired dictionary size. Finally, we need a compressed bitmap LB to mark the beginning of each disk block. LB is entropy-compressed and can reside in main or secondary memory.

For locating every match of a pattern $P_{1,m}$, first we use our counting substructure to obtain the interval [First, Last] of the suffix array of T (see Section 2). Then we find the block First belongs to, $j = rank_1(LB, \text{First})$. Finally, we read the necessary blocks until we reach Last, uncompressing them using the dictionary of the LCSA.

We define $occ = \text{Last} - \text{First} + 1$ and $occ' = cr \cdot occ$, where $0 < cr \leq 1$ is the compression ratio of SP. This process takes, without counting, $\lceil occ'/\tilde{b} \rceil$ I/O access, plus one if we store LB in secondary memory. This I/O cost is optimal and improves thanks to compression. We perform $O(occ + \tilde{b})$ CPU operations to uncompress the interval of SP.

3.4 Extracting

To extract arbitrary portions of the text we compress T using a semistatic statistical modeler of order k plus an encoder EN. This compresses the text to $nH_k(T) + f_{EN}(n)$, where $f_{EN}(n)$ is the redundancy of the encoder. For example, a Huffman coder has redundancy n, whereas an arithmetic encoder has redundancy 2. The data generated by the modeler, DM, is maintained in main memory, which requires $\sigma^{k+1} \log n$ bits. This restricts the maximum possible k to be used: If we have ME bits to store the data generated by the modeler then $k \leq \log_\sigma(ME/\log n) - 1$. To store the structure in secondary memory we split the compressed text into disk blocks of size $\bar{b}$ bits (thus the overhead over the entropy is $\frac{n}{b} f_{EN}(\bar{b})$). If we store less than $b = \bar{b}/\log \sigma$ symbols in a particular disk block, we rather store it uncompressed. An extra bit per block indicates whether this was the case. Also each block will contain the context of order k of the first symbol stored in the block ($k \log \sigma$ bits). To know where a symbol of T is stored we need a compressed rank dictionary ER, in which we mark the beginning of each block. This can be chosen to be in main memory or in secondary memory, the latter requiring one more I/O access.

The algorithm to extract $T_{l,r}$ is: (1) Find the block $j = rank_1(ER, l)$ where T_l is stored. (2) Read block j and decompress it using DM and the context of the k first symbols. (3) Continue reading blocks and decompressing them until reaching T_r.

Using this scheme we have at most $\lceil (j - i + 1)/b \rceil$ I/O operations, which on average is $\lceil (j - i + 1)H_k(T)/\bar{b} \rceil$. We add one I/O operation if we use the secondary memory version of the rank dictionary. The total CPU time is $O(j - i + b + \log n)$.

4 Experiments

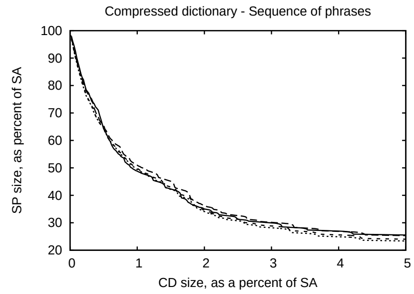
We consider two text files for the experiments: the text WSJ (Wall Street Journal) from the TREC collection from year 1987, of 126 MB, and the 200 MB XML file provided in the *Pizza&Chili Corpus* (`http://pizzachili.dcc.uchile.cl`). We searched for 5,000 random patterns, of length from 5 to 50, generated from these files. As in [6] and [1], we assume a disk page size of 32 KB.

We first study the compressibility we achieve as a function of the dictionary size, $|CD|$ (as CD must reside in RAM). Fig. 4 shows that the compressibility depends on the percentage $|CD|/|SA|$ and not on the absolute size $|CD|$. In the following, we let our CD use 2% of the suffix array size. For counting we use version 1 (GR, Section 3.2) with $t = \log n$. With this setting our index uses 19.15 MB of RAM for XML, and 12.54 MB for WSJ (for GR, CD, and DM). It compresses the SA of XML to 34.30% and that of WSJ to 80.28% of its original size.

We compared our results against String B-tree [7], Compact Pat Tree (CPT) [4], disk-based Suffix Array (SA) [2] and disk-based LZ-Index [1]. We add our results to those of [1, Sec. 4]. We omit the disk-based CSA [15] as it is not implemented, but that one is strictly worse than ours.

Fig. 5 (left) shows counting experiments. Our structure needs at most $2(m - 1)$ disk accesses. We show our index with and without the substructures for locating. Fig. 5 (right) shows locating experiments. For $m = 5$, we report *more* occurrences than those the block could store in raw format.

We can see that the result depends a lot on the compressibility of the text. In the highly-compressible XML our index occupies a very relevant niche in the tradeoff curves, whereas in WSJ it is subsumed by String B-trees. Thus, our index is very competitive on compressible texts. We have used texts up to 200 MB, but our results show that the outcome scales up linearly for the RAM needed, while the counting cost is at most $2(m - 1)$ and the locating cost depends on the number of occurrences of P. Thus it is very easy to predict other scenarios.



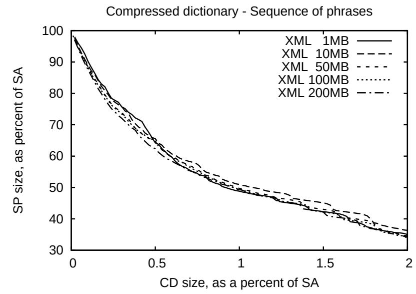


Fig. 4. Compression ratio achieved on XML asa function of the percentage allowed to the dictionary (CD). Both are percentage over the size of SA, the right plot shows a detail.

References

1. D. Arroyuelo and G. Navarro. A Lempel-Ziv text index on secondary storage. In *Proc. CPM*, 2007. To appear.
2. R. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Inf. Systems*, 21(6):497–514, 1996.
3. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Tech.Rep. 124, DEC, 1994.
4. D. Clark and I. Munro. Efficient suffix trees on secondary storage. In *Proc. SODA*, pages 383–391, 1996.
5. F. Claude and G. Navarro. A fast and compact web graph representation. Tech.Rep. TR/DCC-2007-4, CS Dept., Univ. Chile, Chile, 2007. ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/claudegraphs.ps.gz.
6. P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In *Proc. SODA*, pages 373–382, 1996.
7. P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.
8. P. Ferragina and G. Manzini. Indexing compressed texts. *J. ACM*, 52(4):552–581, 2005.
9. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. SPIRE*, LNCS 3246, pages 150–160, 2004. Extended version to appear in *ACM TALG*.
10. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. Posters WEA*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.
11. R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. CPM*, 2007. To appear.
12. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.
13. A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. Compressed data structures: dictionaries and data-aware measures. In *Proc WEA*, pages 158–169, 2006.
14. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.
15. V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. ISAAC*, LNCS 3341, pages 681–692, 2004.
16. V. Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. Tech.Rep. TR/DCC-2006-10, Dept. of CS, Univ. Chile, 2006. ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/dynamic.ps.gz.
17. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.
18. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
19. P. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. SODA*, pages 11–12, 2005.
20. I. Munro. Tables. In *Proc. FSTTCS*, LNCS 1180, pages 37–42, 1996.
21. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms*, 2(1):87–114, 2004.
22. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 2007. To appear. Meanwhile download from http://www.dcc.uchile.cl/gnavarro/ps/acmcs06.ps.gz.
23. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.
24. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algor.*, 48(2):294–313, 2003.
25. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.
26. N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.

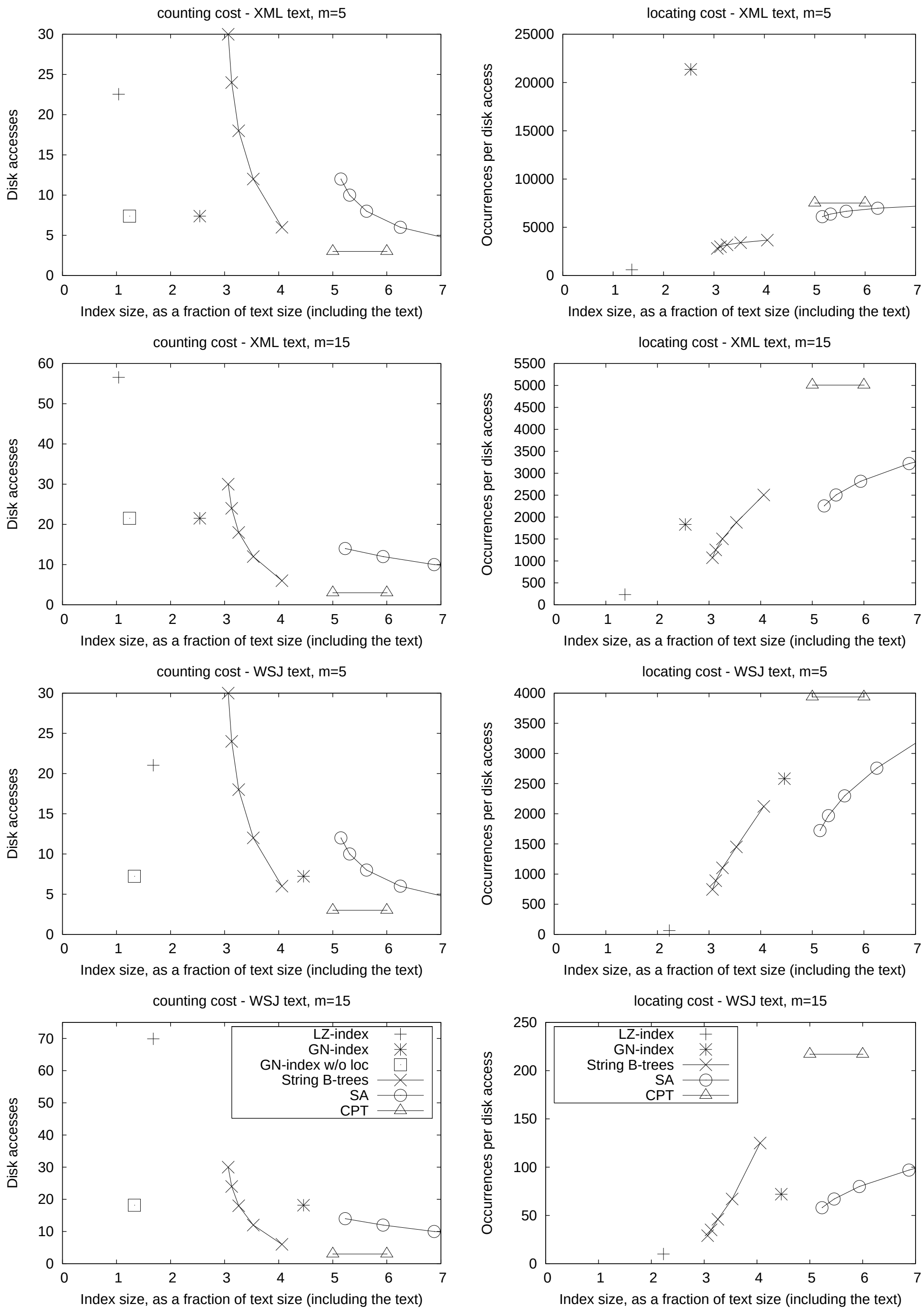


Fig. 5. Search cost vs. space requirement for the different indexes we tested. Counting on the left and locating on the right.