A Compressed Text Index on Secondary Memory *

Rodrigo González † Gonzalo Navarro ‡

Deptartment of Computer Science, University of Chile.
Av. Blanco Encalada 2120, 3^{rd} floor, Santiago, Chile.
{rgonzale,gnavarro}@dcc.uchile.cl

Abstract

We introduce a practical disk-based compressed text index that, when the text is compressible, takes much less space than the suffix array. It provides good I/O times for searching, which in particular improve when the text is compressible. In this aspect our index is unique, as most compressed indexes are slower than their classical counterparts on secondary memory. We analyze our index and show experimentally that it is extremely competitive on compressible texts. As side contributions, we introduce a compressed rank dictionary for secondary memory operating in one I/O access, as well as a simple encoding of sequences that achieves high-order compression and provides constant-time random access, both in main and secondary memory.

1 Introduction and Related Work

Compressed full-text self-indexing [31] is a recent trend that builds on the discovery that traditional text indexes like suffix trees and suffix arrays 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

*Earlier partial versions of this paper appeared in [15] and [14].

†Funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

‡Partially funded by Fondecyt Grant 1-080019, Chile.

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 (orders of magnitude slower) secondary memory. In those situations a compressed index is extremely attractive.

There are, however, cases where even the compressed index 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 can be significantly faster than searching a plain text, because fewer disk blocks must be scanned [36]. 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, due to their poor locality of access.

A self-index built on a text $T_{1,n} = t_1 t_2 \ldots t_n$ over an alphabet A of size σ will support at least the following queries, where $P = P_{1,m}$ is a pattern over A:

- $count(T, P)$: counts the number of occurrences of pattern P in T.
- $locate(T, P)$: locates the positions of all those $occ = count(T, P)$ occurrences of P.
- $extract(T, l, r)$: extracts the substring $T_{l,r}$ of T, with $1 \leq l \leq r \leq n$.

The most relevant text indexes for secondary memory follow:

- The String B-tree [9] is based on a combination between B-trees and Patricia tries. In this index *locate* 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. 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) [6] 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.
- The disk-based Suffix Array [2] is a suffix array on disk plus some memory-resident structures that improve the cost of the search. The suffix array is divided into blocks of h elements, and for each block the first m symbols of its first suffix are stored. At best[1], it takes

[1] Here we are assuming m is known at indexing time, which is rather optimistic. Times would be worse otherwise, and this is the meaning of "at best". Still, we refer to worst cases.

$4+m/h$ times the text size, plus text, and needs $2(1+\log h)$ I/Os for counting and $1 + \lceil (occ-1)/\tilde{b} \rceil$ extra I/Os for locating[2]. This is not yet a compressed index.

- The disk-based Compressed Suffix Array (CSA) [26] adapts a main memory compressed self-index [33] to secondary memory. It requires $n(H_0 + O(\log\log\sigma))$ bits of space (H_k is the k-th order empirical entropy of T [28]). It takes $O(m \log_{\tilde{b}} n)$ I/O time for *count*. Locating requires $O(\log n)$ accesses per occurrence, which is too expensive.

- The disk-based LZ-Index [1] adapts another main-memory self-index [30]. It uses up to $8nH_k(T) + o(n\log\sigma)$ bits, for any $k = o(\log_\sigma n)$. It does not provide theoretical bounds on time complexity, but it is very competitive in practice.

- The disk-based Geometric Burrows-Wheeler Transform (GBWT) [5] uses $O(n\log\sigma)$ bits of space, with constant > 2. Locating takes $O(m/\tilde{b} + \log_\sigma n \log_{\tilde{b}} n + occ \log_{\tilde{b}} n)$ I/Os. There is no practical implementation as far as we know.

- The LOF-SA index [35] is a suffix array enriched with longest common prefix (LCP) information, the characters that distinguish each pair of consecutive suffixes, and a truncated suffix tree in RAM that distinguishes suffixes in consecutive disk blocks. The whole structure requires more than $13n$ bytes, text included. It permits counting with at most $2 + \lceil (m-1)/\tilde{b} \rceil$ accesses to disk. Locating takes $O(occ/b)$ I/Os, with constant at least 2.

In this paper we present a practical self-index for secondary memory, which is built from three components: for *count*, we develop a novel secondary-memory version of backward searching; for *locate* we adapt a recent technique to locally compress suffix arrays [16]; and for *extract* we present a technique to compress sequences to k-th order entropy while retaining random access. Depending on the available main memory, our data structure requires $2(m-1)$ to $4(m-1)$ accesses to disk for *count* in the worst case. It locates the occurrences in $1 + \lceil (occ-1)/\tilde{b} \rceil$ I/Os in the worst case, and on average in $1 + (cr \cdot occ - 1)/\tilde{b}$ I/Os, $0 < cr \leq 1$ being the *suffix array compression ratio* achieved: the compressed size divided by the original suffix array size. Similarly, the time to extract $T_{l,r}$ is at most $1 + \lceil (r-l)/b \rceil$ I/Os in the worst case (where b is the number of text symbols on a disk block). On average, this is $1 + (cs \cdot (r-l+1) - 1)/b$, $0 < cs \leq 1$ being the *text compression ratio* achieved: the compressed size divided by the original text size. With sufficient main memory our index

[2]In this paper $\log x$ stands for $\lceil \max(0, \log_2 x) \rceil$, and thus $0 \log 0 = 0$.

takes $O(H_k \log(1/H_k) n \log n)$ bits of space (see restrictions in the next section), which in practice can be up to 4 times smaller than classical 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* times are better 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.

Our technique to solve *extract* is of independent interest. We start with a data structure that offers constant-time random access to a text that is compressed up to its k-th order entropy, and then adapt it for secondary storage. Such a main-memory data structure already existed [34], however, it is based on Ziv-Lempel encoding and it is not obvious how to adapt it to secondary memory. We introduce an alternative data structure which achieves the same space and time bounds, is much simpler, and is easy to adapt to secondary memory. We build on semi-static k-th order modeling plus statistical encoding, just as a normal semi-static statistical compressor would process S. This technique is also used within the structure that solves *count*.

We also introduce a secondary-memory data structure for compressed bitmaps supporting *rank*. It takes basically the same space and CPU time of an existing data structure for main memory [20], yet just one I/O access.

2 Background and Notation

We assume that the symbols of our strings (or sequences) are drawn from an alphabet A of size σ. Although the distinction is somewhat arbitrary, we will in general write strings in the form $S = S_{1,n} = s_1 s_2 \ldots s_n$, substrings $S_{i,j} = s_i s_{i+1} \ldots s_j$, and symbols $S_i = s_i$, whereas for arrays of other types we will use the form $A = A[1, n]$, subarrays $A[i, j]$, and elements $A[i]$. In both cases we use $|S| = |A| = n$.

We will have different ways to express the size of a disk block: $\bar{b}$ will be the number of bits, $b = \bar{b}/\log \sigma$ the number of symbols, and $\tilde{b} = \bar{b}/\log n$ the number of integers in a block.

The k-th order empirical entropy [28] is defined using that of zero-order. Let $S_{1,n}$ be a string over alphabet A, then

$$H_0(S) = -\sum_{a \in A} \frac{n_S^a}{n} \log_2(\frac{n_S^a}{n}) \qquad (1)$$

with n_S^a the number of occurrences of symbol a in sequence S. This definition extends to $k > 0$ as follows. Let A^k be the set of all sequences of length k over A. For any string $w \in A^k$, called a context of size k, let w_S be the string consisting of the concatenation of characters following w in S. Then, the k-th order empirical entropy of S is

$$H_k(S) = \frac{1}{n} \sum_{w \in A^k} |w_S| H_0 \left(w_S \right). \qquad (2)$$

The k-th order empirical entropy captures the dependence of symbols upon their context. For $k \geq 0$, $nH_k(S)$ provides a lower bound to the output of any compressor that considers a context of size k to encode every symbol of S. Note that the uncompressed representation of S takes $n \log \sigma$ bits, and that $0 \leq H_k(S) \leq H_{k-1}(S) \leq \ldots \leq H_1(S) \leq H_0(S) \leq \log \sigma$.

Note that a semi-static k-th order *modeler* that yields the probabilities $p_1, p_2, \ldots, p_n$ for the symbols $s_1, \ldots, s_n$, will actually determine $p_i \approx P(s_i | s_{i-k} \ldots s_{i-1})$ using the formula $p_i = \frac{n_{w_S}^{s_i}}{|w_S|}$, where $w = s_{i-k} \ldots s_{i-1}$. It is not hard to see, by grouping all the terms with the same w in the summation [28, 18], that

$$-\sum_{i=k+1}^{n} \log p_i = nH_k(S). \qquad (3)$$

Operations rank and select We make heavy use of operations $rank_c(S, i)$ and $select_c(S, i)$ on sequences, where $rank_c(S, i)$ returns the number of times c appears in prefix $S_{1,i}$ and $select_c(S, i)$ returns the position of the i-th appearance of c within S. A particularly interesting case arises when S is a bitmap (i.e., a sequence over alphabet $\{0, 1\}$).

The suffix array $SA[1, n]$ of a text T [27] 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.

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

Figure 1: Backward search algorithm to find and count the suffixes in SA prefixed by P (or the occurrences of P in T).

The Burrows-Wheeler transform (BWT) is a reversible permutation T^{bwt} of T [4] 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: $T_i^{bwt} = T_{SA[i]-1}$.[3] This is the key reason why one can search using T^{bwt} instead of SA.

The inverse transformation is carried out via the so-called "LF mapping", defined as follows:

- 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 $c \in A$, $Occ(c,q) = rank_c(T^{bwt}, q)$ is the number of occurrences of character c in the prefix $T_{1,q}^{bwt}$.
- $LF(i) = C[T_i^{bwt}] + Occ(T_i^{bwt}, i)$, the "$LF$ mapping".

Backward searching is a technique to find the area of SA containing the occurrences of a pattern $P_{1,m}$ by traversing P backwards and making use of the BWT. It was first proposed for the FM-index [10], a self-index composed of a compressed representation of T^{bwt} and auxiliary structures to compute $Occ(c,q)$. Fig. 1 gives the pseudocode to get the area $SA[\text{First}, \text{Last}]$ with the occurrences of P. It requires at most $2(m-1)$ calls to Occ. Depending on the variant, each call to Occ can take constant time for small alphabets [10] or $O(\log \sigma)$ time in general [11], using wavelet trees (see below).

Bitmaps with rank/select. Both *rank* and *select* on a bitmap $B_{1,n}$ can be computed in constant time using $o(n)$ bits of space in addition to B [29, 13], or $nH_0(B) + o(n)$ bits overall using a numbering scheme for bit blocks [32]. In both cases the $o(n)$ term is $\Theta(n \log\log n / \log n)$.

[3] We write T_i^{bwt} for $(T^{bwt})_i$ and $T_{i,j}^{bwt}$ for $(T^{bwt})_{i,j}$.

Let s be the number of one-bits in B. Then $nH_0(B) = s \log \frac{n}{s} + O(s)$, and thus the $o(n)$ terms above are too large if s is much smaller than $n/2$. 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 (which will not be an issue for us). One such rank dictionary [20] 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 $s \log \frac{n}{s} + O(s \log \log \frac{n}{s})$ bits of space and supports *rank* in $O(\log s)$ time. This structure is called *BSGAP* (binary searchable gap encoding) [19, Section 4.3].

The wavelet tree [18] $wt(S)$ over a sequence $S_{1,n}$ is a perfect binary tree of height $\log \sigma$, 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.

The wavelet tree permits us to calculate $rank_c(S, i)$ 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 $i \leftarrow rank_1(B^v, i)$ and move to the right child of v. Similarly, if c belongs to the left child we update $i \leftarrow rank_0(B^v, i)$ 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 $rank_c(T^{bwt}, i)$. We can obtain S_i from the wavelet tree with a very similar process, except that we go down left or right depending on whether $B^v[i] = 0$ or 1, until we reach the leaf corresponding to symbol $c = S_i$. By traversing the tree upwards we can also solve $select_c(S, i)$. All the operations take $O(\log \sigma)$ time.

We will build wavelet trees over sequences $S = T^{bwt}$. A plain wavelet tree of S requires $n \log \sigma$ bits of space. If we compress the wavelet tree using the numbering scheme [32] we obtain $nH_k(T) + o(n \log \sigma)$ bits of space for any $k \le \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$ [25].

The locally compressed suffix array (LCSA) [16] is built on well-known regularity properties that show up in suffix arrays when the text they index is compressible [31]. The LCSA uses differential encoding on SA, which converts those regularities into true repetitions. Those repetitions are then factored out using Re-Pair [23], 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 [16] that the size of the LCSA is $O(H_k \log(1/H_k) n \log n)$ bits for any $k \leq \alpha \log_\sigma n$, any constant $0 < \alpha < 1$, and $H_k = o(1)$.

The LCSA consists of three substructures: the sequence of phrases SP, the compressed dictionary CD needed to decompress 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, but in the extended paper [17] they show how to build it on disk.

Statistical encoding. We are interested in the use of semi-static statistical encoders [3] to represent the text on disk. Thus, we are given a k-th order modeler as described earlier, which will yield the probabilities $p_1, p_2, \ldots, p_n$ for each symbol in S, and we will encode the successive symbols of S trying to use $-\log p_i$ bits for s_i. If we reach exactly $-\log p_i$ bits, the overall number of bits produced will be $nH_k(S) + O(k \log n)$, according to Eq. (3).

Different encoders give different approximations to the ideal $-\log p_i$ bits. The simplest encoder is probably Huffman's [21], while the one generating the least number of bits is Arithmetic coding [3].

Given a statistical encoder E and a semi-static modeler over sequence $S_{1,n}$ yielding probabilities $p_1, p_2, \ldots, p_n$, we call $E(S)$ the bitwise output of E for those probabilities, and $|E(S)|$ its bit length. We call $f(E, S) = |E(S)| - (-\sum_{1 \leq i \leq n} \log p_i)$ the extra space in bits needed to encode S using E, on top of the entropy of the model. We also define $f(E, n) = \max_{|S|=n} f(E, S)$. For example, the wasted space of Huffman encoding is bounded by 1 bit per symbol, and thus $f(\text{Huffman}, n) < n$ (tighter bounds exist [3] but are not useful for this paper). On the other hand, Arithmetic encoding approaches $-\log p_i$ as closely as desired, requiring only at most two extra bits to terminate the whole sequence (see next). Thus $f(\text{Arithmetic}, n) \leq 2$.

Arithmetic coding essentially expresses S using a number in $[0, 1)$ which lies within a range of size $P = p_1 \times p_2 \times \cdots \times p_n$. We need $-\log P = -\sum \log p_i$ bits to distinguish a number within that range (plus two extra bits for technical reasons [3, Sections 5.2.6 and 5.4.1]). Thus each symbol s_i, which appears within its context np_i times, requires $-\log p_i$ bits to be encoded. This totalizes $-\sum \log p_i + 2$ bits. Again, we can relate the model entropy of $p_1, p_2, \ldots, p_n$ with the empirical entropy of S using Eq. (3). This way, Arithmetic coding encodes S using at most $nH_k(S) + O(k \log n) + 2$ bits for any k.

There are usually some limitations to the near-optimality achieved by Arithmetic coding in practice [3]. One is that many bits are required to manipulate P, which can be cumbersome. This is mainly alleviated by emitting the most significant bits of the final number as soon as they are known, and thus scaling the remainder of the number again to the range $[0, 1)$ (that is, dropping the emitted bits from our number). Still, some symbols with very low probability may require many bits. To simplify matters, fixed precision arithmetic is used to approximate the real values, and this introduces a very small (yet linear) inefficiency in the coding. In this paper, we never run into this problem because, as seen later, we do not encode any sequence that requires more than $\frac{\log n}{2}$ bits. As soon as those bits are not precise enough to represent the encoding, we switch to plain symbol-wise encoding.

Another limitation applies to adaptive encoding, where some kind of aging technique is used to let the model forget symbols that have appeared many positions away in the sequence. In our case this does not apply, as we use semi-static encoding. Finally, we notice that we run into no efficiency problems at all at decoding time, as we will use the $\frac{\log n}{2}$-bit compressed stream as an index to a precomputed table that will directly yield the uncompressed symbols.

3 An 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 $BSGAP$ (Section 2). Otherwise we will store in secondary memory GAP, the δ-encoded form of B: We encode the gaps between consecutive 1's in B as variable-length integers, so that $0^{x-1}1$ is represented as the number x using $\log x + 2 \log\log x$ bits [3]. Let s be the number of one-bits in B. Then GAP uses at most $s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)$ bits of space. We split GAP 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 j, we also store the number of 1's accumulated up to block $j-1$; we call this value OB_j. To access GAP, we use in main memory an array B^a, where $B^a[j]$ is the number of bits of B represented in blocks 1 to $j-1$. B^a uses $(s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n))\frac{\log n}{\bar{b}}$ bits of

Structure	Space (bits)	CPU time for *rank*
BSGAP	$s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)$	$O(\log s)$
GAP+ B^a	$s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)$ + $(s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)) \frac{\log n}{b}$	$O(\log s + b$ $+ \log\log \frac{n}{s})$

Structure	Real space if $s = n/b$			
	$n =$ 1 Tb $b = 32$ KB	1 Gb 8 KB	1 Gb 4 KB	1 Mb 4 KB
BSGAP	100 MB	354 KB	667 KB	< 1KB
GAP+	93 MB	326 KB	613 KB	< 1KB
B^a	14 KB	< 1KB	< 1KB	< 1KB

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

space. We call $DGAP = B^a + GAP$ the whole structure.

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) Our initial position is $p \leftarrow B^a[j]$ and our initial rank is $r \leftarrow OB_j$. (3) We read block j from disk. (4) We decompress the δ-encodings x in block j, adding x to p, and adding 1 to r if $p \leq i$. (5) We stop when $p \geq i$; $rank_1(B, i)$ will be r.

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 3 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 *DGAP*, and for moderate-size bitmaps even the *BSGAP* option is good.

Theorem 1. *A bit sequence of length n with s bits set can be stored using $c = s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)$ bits in secondary memory, plus $c \cdot \frac{\log n}{b}$ bits in main memory (being $\bar{b}$ the disk block size in bits), so that $rank_1$ can be solved using one access to disk plus $O(\log s + \log\log n + \bar{b})$ CPU time.*

4 A Simple Entropy-Bounded Sequence Representation

Given a sequence $S_{1,n}$ over an alphabet A of size σ, we encode S into a compressed data structure S' within entropy bounds. To perform all the original operations over S under the RAM model, it is enough to allow extracting any aligned block of $\beta = \lfloor \frac{1}{2} \log_\sigma n \rfloor$ consecutive symbols of S, using S', in constant time. We then show how to adapt the structure to secondary memory.

4.1 Data structures for substring decoding

We describe our data structure to represent S in essentially $nH_k(S)$ bits, and to permit the access of any aligned substring of size $\beta = \lfloor \frac{1}{2} \log_\sigma n \rfloor$ in constant time. We assume $k < \beta$. This structure is built using any statistical encoder E as described in Section 2.

Structure. We divide S into blocks of length $\beta = \lfloor \frac{1}{2} \log_\sigma n \rfloor$ symbols. Each block will be represented using at most $\beta' = \lfloor \frac{1}{2} \log n \rfloor$ bits (and hopefully less). We define the following sequences indexed by block number $i = 1, \ldots, \lceil n/\beta \rceil$:

- $S^i = S_{\beta(i-1)+1,\beta i}$ is the sequence of symbols forming the i-th block of S.
- $C^i = S_{\beta(i-1)-k+1,\beta(i-1)}$ is the sequence of symbols forming the k-th order context of the i-th block (a dummy value is used for C^1).
- $E^i = E(S^i)$ is the encoded sequence for the i-th block of S, initializing the k-th order modeler with context C^i.
- $\ell_i = |E^i|$ is the size in bits of E^i.
- $\tilde{E}^i = \begin{cases} S^i & \text{if } \ell_i > \beta' \\ E^i & \text{otherwise} \end{cases}$, is the shortest sequence among E^i and S^i.
- $\tilde{\ell}_i = |\tilde{E}^i| = \min(\beta', \ell_i)$ is the size in bits of $\tilde{E}^i$.

The idea behind $\tilde{E}^i$ is to ensure that no encoded block is longer than β' bits (which could happen if a block contains many infrequent symbols). These special blocks are encoded explicitly.

Our compressed representation of S stores the following information:

- $W[1, \lceil n/\beta \rceil]$: A bit array such that
$$W[i] = \begin{cases} 0 \text{ if } \ell_i > \beta' \\ 1 \text{ otherwise} \end{cases},$$
with the additional $o(n/\beta)$ bits to answer rank queries over W in constant time [29].

- $C[1, rank_1(W, \lceil n/\beta \rceil)]$: $C[rank_1(W, i)] = C^i$, that is, the k-th order context for the i-th block of S iff $\ell_i \leq \beta'$, with $1 \leq i \leq \lceil n/\beta \rceil$.

- $U = \tilde{E}^1 \tilde{E}^2 \ldots \tilde{E}^{\lceil n/\beta \rceil}$: A bit sequence obtained by concatenating all the variable-length $\tilde{E}^i$ s.

- $DM : A^k \times 2^{\beta'} \longrightarrow 2^{\beta}$: A table defined as $DM[\alpha, \beta] = \gamma$, where α is any context of size k, β represents any encoded block of at most β' bits, and γ represents the decoded form of β, truncated to the first β symbols (as less than the β' bits will be usually necessary to obtain the β symbols of the block).

- Information to answer where each $\tilde{E}^i$ starts within U. We group together every $c = \log n$ consecutive blocks to form superblocks of size $\Theta(\log^2 n)$ and store two tables:
  - $R_g[1, \lceil n/(\beta c) \rceil]$ contains the absolute position of each superblock.
  - $R_l[1, \lceil n/\beta \rceil]$ contains the relative position of each block with respect to the beginning of its superblock.

4.2 Substring decoding algorithm

We want to retrieve $S^j = S_{(j-1)b+1, jb}$ in constant time. To achieve this, we take the following steps:

1. We calculate $h = \lceil j/c \rceil$, $h' = \lceil (j+1)/c \rceil$ and $u = U[R_g[h] + R_l[j], R_g[h'] + R_l[j+1] - 1]$, then
   - if $W[j] = 0$ then we have $S^j = u$.
   - if $W[j] = 1$ then we have $S^j = DM[C[rank_1(W, j)], u']$, where u' is u padded with $\beta' - |u|$ dummy bits.

   We note that $|u| \leq \beta'$ and thus it can be manipulated in constant time.

Lemma 2. *Our data structure can extract any aligned substring of b symbols from sequence S in $O(1)$ time.*

4.3 Space requirements

Let us now consider the storage size of our structures.

- We use the constant-time solution to answer rank queries [29] over W, totalizing $\frac{2n}{\log_\sigma n}(1+o(1))$ bits.
- Table C requires at most $\frac{2n}{\log_\sigma n} k \log\sigma$ bits.
- Sequence U takes $|U| = \sum_{i=1}^{\lceil n/\beta \rceil} |\tilde{E}^i| \le \sum_{i=1}^{\lceil n/\beta \rceil} |E^i| = nH_k(S) + O(k \log n) + \lceil n/\beta \rceil f(E,\beta)$ bits, which depends on the statistical encoder E used. For example, in the case of Huffman coding, we have $f(\text{Huffman},\beta) < \beta$, and thus we achieve $nH_k(S)+O(k\log n)+n$ bits. For the case of Arithmetic coding, we have $f(\text{Arithmetic},\beta) \le 2$, and thus we have $nH_k(S) + O(k\log n) + \frac{4n}{\log_\sigma n}$ bits, as described in Section 2.
- The size of DM is $\sigma^k 2^{\beta'} \beta \log\sigma = \sigma^k\ n^{1/2}\ \frac{\log n}{2}$ bits.
- Finally, let us consider tables R_g and R_l. Table R_g has $\lceil n/(\beta c) \rceil$ entries of size $\log n$, totalizing $\frac{2n}{\log_\sigma n}$ bits. Table R_l has $\lceil n/\beta \rceil$ entries of size $\log(\beta' c)$, totalizing $\frac{4n \log\log n}{\log_\sigma n}$ bits.

By considering that any substring of $\Theta(\log_\sigma n)$ symbols can be extracted in constant time by applying $O(1)$ times the procedure of Section 4.2, we have the final theorem.

Theorem 3. *Let $S_{1,n}$ be a sequence over an alphabet of size σ. Our data structure uses $nH_k(S) + O(\frac{n}{\log_\sigma n}(k\log\sigma + \log\log n))$ bits of space for any $k \le (1-\epsilon)\log_\sigma n$ and any constant $0 < \epsilon < 1$, and it supports access to any substring of S of size $\Theta(\log_\sigma n)$ symbols in $O(1)$ time.*

Note that, in our scheme, the size of DM can be neglected only if $k \le (\frac{1}{2} - \epsilon)\log_\sigma n$, but this can be pushed to $(1-\epsilon)\log_\sigma n$, for any constant $0 < \epsilon < 1$, by choosing $\beta = \frac{\epsilon}{2}\log_\sigma n$. Thus the size of DM will be $\sigma^k n^{\frac{\epsilon}{2}} \frac{\log n}{s}$, which is negligible for, say, $k \le (1 - \frac{\epsilon}{2} - \frac{\epsilon}{2})\log_\sigma n$. The price is that now we must decode $O(1/\epsilon)$ blocks to extract $O(\log_\sigma n)$ bits from S, but this is still constant.

Corollary 3.1. *The previous structure takes space $nH_k(S) + o(n\log\sigma)$ if $k = o(\log_\sigma n)$.*

These results match exactly those of [34][4]. Our method is simpler, but their result holds simultaneously for all k, while in our structure k must be chosen beforehand.

Note that we are storing some redundant information that can be eliminated. The last characters of block S^i are stored both within $\tilde{E}^i$ and as C^{i+1}. Instead, we can choose to explicitly store the first k characters of *all* blocks S^i, and encode only the remaining $\beta - k$ symbols, $S^i[k+1, \beta]$, either in explicit or compressed form. This improves the space in practice, but in theory it cannot be proved to be better than the scheme we have given.

Some extensions of this result to handle dynamism, and to apply it to encode wavelet trees, are studied in [14].

4.4 A secondary memory version

We now modify our data structure to operate on secondary memory.

Structures maintained in main memory. We store in main memory the data generated by the modeler, that is, table DM, which requires $\sigma^k n^{1/2} \frac{\log n}{2}$ bits. This restricts the maximum possible k to be used.

Structures in secondary memory. To store the structure in secondary memory we split the sequence $U = \tilde{E}^1 \tilde{E}^2 \ldots \tilde{E}^{\lceil n/\beta \rceil}$ and W into disk blocks of $\bar{b}$ bits (thus we lose at most $\frac{n}{b}\beta = O(\frac{n \log n}{b})$ bits due to alignments). Also each block will contain the context C^j (for some j) of order k of the first entry of U, $\tilde{E}^j$, stored in the disk block ($k \log \sigma$ bits).

To know where a symbol of S is stored we need a compressed rank dictionary ER (Section 3), in which we mark the beginning of each disk block. This replaces tables R_g and R_l. ER has $\frac{n}{b}$ bits set out of $\lceil n/\beta \rceil$, and it can be chosen to reside in main or in secondary memory, the latter choice requiring one more I/O access.

The algorithm to extract $S_{l,r}$ is: (1) Find the block $j = rank_1(ER, \lceil l/\beta \rceil)$ where S_l is stored. (2) Read block j and decompress it using DM and the context of the first entry. (3) Continue reading and decompressing them until reaching S_r.

[4]The term $k \log \sigma$ appears as k in [34], but this is a mistake (K. Sadakane and R. Grossi, personal communication). The reason is that they take from [22] an extra space of the form $\Theta(kt + t)$ as stated in Lemma 2.3, whereas the proof in Theorem A.4 gives a term of the form $kt \log \sigma + \Theta(t)$.

Using this scheme we have at most $1 + \lceil (r-l)/b \rceil$ I/O operations, which on average is $1 + ((r-l+1)H_k(S) - 1)/\bar{b}$. We add one I/O operation if we use the secondary memory version of the rank dictionary. The total CPU time is $O(\frac{r-l}{\log_\sigma n} + \bar{b} + \log n)$. Term $\bar{b}$ can be removed by directly accessing inside the block. This requires maintaining in each disk block the R_l of each $\tilde{E}^i$ stored inside the block, which adds other $o(n \log \sigma)$ bits of space. The next theorem considers only the basic variant, others are easy to derive.

Theorem 4. *Let $S_{1,n}$ be a sequence over an alphabet of size σ. Our secondary-memory data structure uses $nH_k(S) + O(\frac{n}{\log_\sigma n} + \frac{n}{b}(k \log \sigma + \log n))$ bits of space for any $k \leq (1-\epsilon)\log_\sigma n$ and any constant $0 < \epsilon < 1$, and $O(n^{1-\epsilon} \log n + \frac{n}{b} \log \frac{b}{\beta})$ bits in main memory. It supports access to any substring $S_{l,r}$ in $1 + \lceil (r-l)/b \rceil$ I/O accesses, which on average is $1 + ((r-l+1)H_k(S) - 1)/\bar{b}$. The total CPU time is $O(\frac{r-l}{\log_\sigma n} + \bar{b} + \log n)$.*

5 A Compressed Secondary Memory Structure

We introduce a 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 allows diverse trade-offs depending on how much main memory space they occupy.

5.1 Counting

We run the algorithm of Fig. 1 to answer a counting query. Table C uses $\sigma \log n$ bits and easily fits in main memory, thus the problem is how to calculate Occ over T^{bwt}.

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)[5] Let K be the total number of blocks. We define:

[5]Thus, in what follows, b will be the remaining space in the disk block. This is asymptotically irrelevant as long as $b \geq c \cdot \sigma \log(tb)$ for some $c > 1$. Otherwise the accesses to disk must be doubled.

- $E_c(j)$, for $1 \le j \le \lceil K/t \rceil$, is the number of occurrences of symbol c in blocks 1 to $(j-1)\cdot t$, with $E_c(1) = 0$.
- $E'_c(j)$, for $1 \le j \le K$, is the number of occurrences of symbol c in blocks from $\lceil j/t \rceil \cdot t - t + 1$ to j.

Now we can compute $Occ(c,i) = rank_c(T^{bwt}, i) = E_c(\lceil j/t \rceil) + E'_c(j) + rank_c(B_j, \mathit{offset})$, where j is the block where i belongs and *offset* is the position of i within block j. Now we explain four ways to represent T^{bwt}, each with its pros and cons. This will give us four different ways to calculate j, *offset*, and $rank_c(B_j, \mathit{offset})$.

Version 1. The simplest choice is to store T^{bwt} directly without any compression. As a disk block can store b symbols, we will have $K = \lceil n/b \rceil$ blocks. $rank_c(B_j, \mathit{offset})$ is calculated by traversing the block and counting the occurrences of c up to *offset*. As the layout of blocks is regular, we know that that T_i^{bwt} belongs to block $j = \lceil i/b \rceil$, and $\mathit{offset} = i - (j-1) \cdot b$.

Version 2. We represent the T^{bwt} chunks with a wavelet tree (Section 2) to speed up the scanning of the 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 (just like the plain storage of the chunk of T^{bwt}). In this case the division of T^{bwt} is uniform and uncompressed, thus we can still easily determine j and *offset*. Fig. 2 illustrates. Note that this propagation generates $2^{\ell-1}$ intervals at level ℓ of WT. Some definitions follow:

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

Some relationships hold: (1) $L_i^\ell = O_i^\ell + Z_i^\ell$. (2) $Z_i^\ell = rank_0(B_i^\ell, L_i^\ell)$. (3) $L_i^\ell = Z_{(i+1)/2}^{\ell-1}$ if i is odd (B_i^ℓ is a left child); $L_i^\ell = O_{i/2}^{\ell-1}$ otherwise. (4) $|D_\ell| = L_1^1 = b$ for $\ell < \lfloor \log \sigma \rfloor$, the last level can be different if σ is not a power of 2. With those properties, L_i^ℓ, O_i^ℓ and Z_i^ℓ are determined

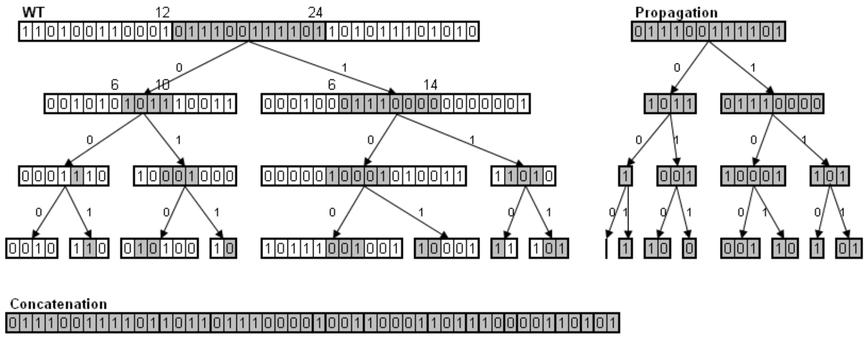

Figure 2: Block propagation over the wavelet tree. Making ranks over the first level B of WT ($rank_0(B,12)=6$, $rank_0(B,24)=10$ and $rank_1(\cdot,i)=i-rank_0(\cdot,i)$), we determine the propagation over the second level of WT, and so on.

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

Figure 3: Algorithm to obtain $rank_c(B,j)$, for version 2.

recursively from B and b. We only store B plus the structures to answer $rank$ on it in constant time. Note that any $rank$ over B_i^{ℓ} is answered via two $ranks$ over B.

Fig. 3 shows how we calculate $rank$ in $O(\log\sigma)$ constant-time steps. Some precisions are in order

1. Block D_ℓ begins at bit $(\ell-1)\cdot b+1$ of B, and $|B| = b\log\sigma$.
2. To know where B_i^{ℓ} begins, we only need to add to the beginning of D_ℓ the length of $B_1^{\ell},\ldots,B_{i-1}^{\ell}$. Each B_k^{ℓ}, with $1 \le k \le i-1$, belongs to a left branch that we do not follow to reach B_i^{ℓ} from the root. So, when we descend through the wavelet tree to B_i^{ℓ}, every time we take

a right branch we accumulate the number of bits of the left branch (zeros of the parent).

3. *node* is the number of the current interval at the current ℓ.
4. We do not calculate B^{ℓ}_{node}, we just maintain its position within B.

The extra space on top of the $n \log \sigma$ bits is still $O(\frac{n \log \sigma \log \log n}{\log n}) = o(n \log \sigma)$, even if encoding is local to the block. This is achieved by maintaining the block sizes of $\frac{1}{2} \log n$ bits in the *rank* structures (Section 2). The consequence is a small table of $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits in main memory.

Version 3. We aim at compressing T^{bwt} so as to achieve k-th order compression of T. We compress the blocks B from version 2 using the numbering scheme [32], yet without any structure for *rank*. In this case the division of T^{bwt} is not uniform; rather 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 at most $nH_k + \sigma^{k+1} \log n + o(n \log \sigma)$ bits for any k [25]. To calculate $rank_c(B, \mathit{offset})$, we apply the same algorithm of Version 2, but now the bitmap is not stored explicity. Constant time *ranks* on the bitmaps are supported by the compressed representation [32].

As the block size is variable, determining j is not as simple as before. Compression ensures that there are at most $(n + o(n))/b$ blocks. We use a binary sequence $EB_{1,n}$ to mark where each block starts. Thus the block of T^{bwt}_i is $j = rank_1(EB, i)$. We use an entropy-compressed rank dictionary (Section 2) for EB. If we need to use the $DGAP$ variant, we add up one more I/O per access to T^{bwt} (Section 3) .

Version 4. We aim at compressing T^{bwt} directly without wavelet trees. We represent T^{bwt} with our entropy-bounded data structure on secondary memory (Section 4.4). Again, the division of T^{bwt} is not uniform, rather we add symbols from T^{bwt} to the disk block as long as its compressed T^{bwt} fits in the block. By doing this, we compress T^{bwt} to $nH_k(T^{bwt}) + o(n \log \sigma)$ bits for $k = o(\log_\sigma n)$. To calculate $rank_c(B, \mathit{offset})$, we decompress block B by applying the decoding algorithm presented in Section 4.4.

Space usage of E and E'. In versions 1 and 2, if we sum up all the entries, E uses $\lceil K/t \rceil \cdot \sigma \log n$ bits and E' uses $K\sigma \log \frac{t \cdot n}{K}$ bits. In version 3, the numbering scheme [32] 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. In version

Version	Main Memory	Secondary Memory
1	$\frac{n}{bt}\cdot \sigma \log n$	$n \log \sigma + \frac{n}{b}\cdot \sigma \log(t\cdot b)$
2	$\frac{n}{bt}\cdot \sigma \log n + o(\sqrt{n} \log^2 n)$	$n \log \sigma(1 + o(1)) + \frac{n}{b}\cdot \sigma \log(t\cdot b)$
3a	$\frac{n}{bt}\cdot \sigma \log n$ $+o(\sqrt{n} \log^2 n) + bsgap$	$nH_k(T) + o(n \log \sigma) + \sigma^{k+1} \log n$ $+\frac{n}{b}\cdot \sigma \log(t \cdot b \log n)$
3b	$\frac{n}{bt}\cdot \sigma \log n$ $+o(\sqrt{n} \log^2 n) + gap\frac{\log n}{b}$	$nH_k(T) + o(n \log \sigma) + \sigma^{k+1} \log n$ $+gap + \frac{n}{b}\cdot \sigma \log(t \cdot b \log n)$
4a	$\frac{n}{bt}\cdot \sigma \log n$ $\sigma^k \sqrt{n}\frac{\log n}{2} + bsgap$	$nH_k(T^{bwt}) + o(n \log \sigma) + \sigma^{k+1} \log n$ $+\frac{n}{b}\cdot \sigma \log(t \cdot b \log n)$
4b	$\frac{n}{bt}\cdot \sigma \log n$ $\sigma^k \sqrt{n}\frac{\log n}{2} + gap\frac{\log n}{b}$	$nH_k(T^{bwt}) + o(n \log \sigma) + \sigma^{k+1} \log n$ $+gap + \frac{n}{b}\cdot \sigma \log(t \cdot b \log n)$

$gap = \frac{n}{b}(\log b + 2 \log \log b) + O(\log n) = O(\frac{n}{b} \log n)$
$bsgap = \frac{n}{b} \log b + O(\frac{n}{b} \log \log b) = O(\frac{n}{b} \log n)$.

Table 2: Different sizes (in bits) obtained to answer *count*.

4, there is no upper bound to how many original symbols can fit in a compressed block. To avoid an excessively large E', we can impose an artificial limit: if more than $b \log n$ symbols are compressed into a single disk block, we stop adding symbols there. This guarantees that $\log(t \cdot b \log n)$ bits are sufficient for each entry of E'. The growth in the compressed file we cause cannot be more than $\bar{b}$ bits per $b \log n$ symbols, that is, $O(\frac{n\bar{b}}{b \log n}) = o(n \log \sigma)$ bits overall.

Costs per call to $rank_c$**.** In Versions 1 and 2, we pay one I/O per call to $rank_c$. In Versions 3 and 4, we pay one or two I/Os per call to $rank_c$. In Versions 1 and 4, we spend $O(b)$ CPU operations per call to $rank_c$. In Versions 2 and 3, this is reduced to $O(\log \sigma)$ per call to $rank_c$.

Tables 2 and 3 show the different sizes and times, respectively, needed for our four versions. We added the times to do *rank* on the entropy-compressed bit arrays. Versions 3a and 4a use an in-memory rank dictionary *BSGAP*, while 3b and 4b use the *DGAP* variant (Section 3). The space complexity of version 3 depends on $H_k(T)$ but version 4 depends on $H_k(T^{bwt})$. There is no obvious connection between $H_k(T)$ and $H_k(T^{bwt})$. In Appendix A we prove that $H_1(T^{bwt}) \leq 1 + H_k(T) \log \sigma + o(1)$ for any $k \leq (1 - \epsilon) \log_\sigma n$ and any constant $0 < \epsilon < 1$.

Version	I/O	CPU
1	$2(m-1)$	$O(m \cdot b)$
2	$2(m-1)$	$O(m \log \sigma)$
3a	$2(m-1)$	$O(m(\log \sigma + \log \frac{n}{b}))$
3b	$4(m-1)$	$O(m(b + \log \frac{n}{b}))$
4a	$2(m-1)$	$O(m(b + \log \frac{n}{b}))$
4b	$4(m-1)$	$O(m(b + \log \frac{n}{b}))$

Table 3: Different times obtained to answer *count*.

5.2 Locating

Our locating structure will be a variant of the LCSA [16], 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. To minimize the I/Os, the dictionary will be maintained in main memory (in theory, it is sufficient to maintain $O(\log^2 n)$ bits in main memory for the dictionary in order to achieve the promised space bounds for the LCSA [16]; in the next section we see how this translates to practice). So we compress the differential suffix array until we reach the desired dictionary size. Finally, we need a compressed bitmap LB (Section 3) 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 $P_{1,m}$, we first use our counting substructure to obtain the interval $[\text{First}, \text{Last}]$ of the suffix array of T (see Section 2). Then we find the block index First belongs to, $j = rank_1(LB, \text{First})$. Finally, we read the necessary blocks until we reach Last, decompressing 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, $1 + \lceil (occ - 1)/\tilde{b} \rceil$ I/O accesses, plus one if we store LB in secondary memory. This I/O cost is optimal and on average improves, thanks to compression, to $1 + (occ' - 1)/\tilde{b}$. We perform $O(occ + \tilde{b})$ CPU operations to decompress the interval of SP.

5.3 Extracting

To extract arbitrary portions of the text we store T in compressed form using the variant of our entropy-bounded succinct data structure for secondary memory, see Section 4.4.

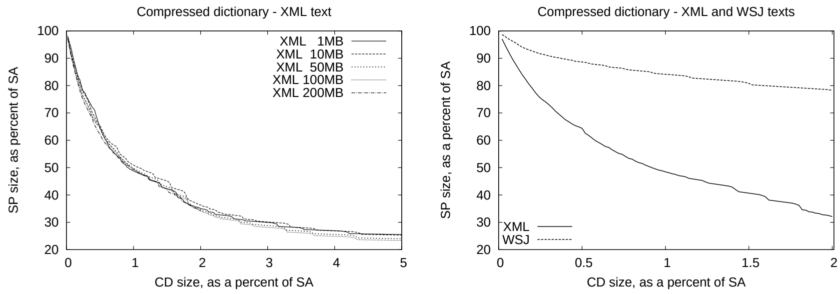

Figure 4: On the left, compression ratio achieved on XML (for different lengths) as a function of the percentage allowed to the dictionary (CD). On the right, on different texts. Both are percentages over the size of SA.

6 Experiments

We consider two text files for the experiments: the text WSJ (Wall Street Journal) from the TREC collection from year 1987, of size 126 MB, and the 200 MB XML file provided in the *Pizza&Chili Corpus*[6]. We searched for 5,000 random patterns, of length from 5 to 50, generated from these files. As in previous work [7, 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 (left) shows that the compressibility depends on the percentage $|CD|/|SA|$ and not on the absolute size $|CD|$. Fig. 4 (right) shows the relation between $|CD|/|SA|$ versus $|SP|/|SA|$ for the texts used in the next experiments. In the following, we let our CD use 2% of the suffix array size. For counting we use version 1 (Section 5.1) with $t = \log n$, and $BSGAP$ for the LB locating structure (Section 5.2). With this setting our index uses 16.13 MB of RAM for XML ($\sigma = 97$), and 10.58 MB for WSJ ($\sigma = 91$), for LB, CD, and E_c. 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 [9], Compact Pat Tree (CPT) [6], disk-based Suffix Array (SA) [2] and disk-based LZ-Index [1]. We omit the disk-based CSA [26] and the disk-based GBWT [5] as they are not implemented (even for simulations), but also because they can be predicted to be strictly worse than ours in these experiments. We also omit the LOF-SA index [35] because it largely exceeds our range of space consumption of interest, and it would not be competitive for locating for the same reason (fewer entries fit in a disk block). For counting it would need usually less than 4 accesses to disk.

[6] `http://pizzachili.dcc.uchile.cl`

We add our results to those of [1, Section 4]. Albeit our RAM usage is moderate, other data structures [9, 6, 1] can operate with a (small) constant number of disk blocks in main memory. We now consider the impact of giving these other structures the same amount of RAM we use, using it in the most reasonable (obvious) way we can devise. For the String B-tree, it was shown [8] that the arity of the tree is $b/12.25$ for the static version. For our page size $b = 8192$ integers, one would need less than 21 MB of RAM to hold the first tree level, thus we will subtract one disk access from the results given in [1]. For the CPT, we have that the pages are formed mostly by pointers to children and are filled to about 50% [6]. Thus one would need near 100 MB to fit the first level in RAM. The effect of fitting as much as possible in, say, 20 MB of RAM, is negligible and thus we have not changed the results used in [1]. For the disk-based LZ-index, in both texts one could store one level of $LZTrie$ and $RevTrie$ in about 12 MB of RAM. Yet, this time the top-down tree traversal is just a part of the total number of accesses, as there are also many direct accesses to the tries. As the potential benefit is hard to predict and implementations taking advantage of main memory do not exist yet[7], we do not change the results of [1]. Finally, the disk-based SA needs to hold the extra nm/h bytes in RAM, and thus we have extended the range studied in [1] to include the point where nm/h is as small as our RAM usage.

Fig. 5 (left) shows counting experiments (GN-index being ours). Our structure needs at most $2(m-1)$ disk accesses, but usually less as both ends of the suffix array interval tend to fall within the same disk block as the counting progresses. We show our index with and without the substructures for locating. It can be seen that our structure is extremely competitive for counting, being much smaller and/or faster than all the alternatives.

Fig. 5 (right) shows locating experiments. This time our structure grows due to the inclusion of the LCSA. Note that, for $m = 5$, we are able to report *more* occurrences than those the block could store in raw format. This time the competitiveness of our structure 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.

We have used texts up to 200 MB, but our results show that the compression ratio stays similar if we maintain a fixed percentage for the dictionary size (Fig. 4 (left)), that the counting cost is at most $2(m-1)$, and that the locating cost depends on the number of occurrences of P and on the compression ratio. Thus it is very easy to predict other scenarios.

[7] D. Arroyuelo. Personal communication.

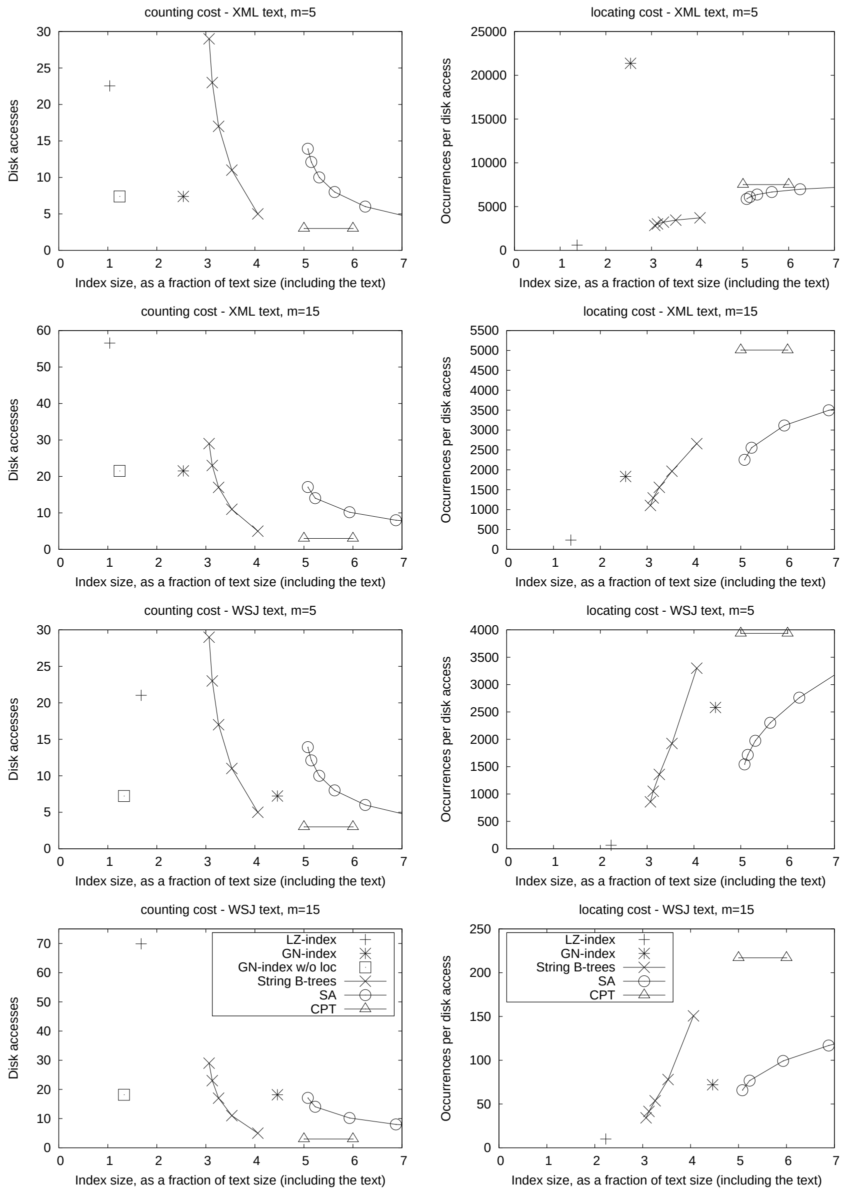

Figure 5: Search cost vs. space requirement for the different texts and indexes we tested. Counting on the left (lower is faster) and locating on the right (higher is faster). Recall that m is the pattern length.

7 Conclusions and Future Work

We have presented a practical self-index for secondary memory that, when the text is compressible, takes much less space than the suffix array. It also provides good I/O times for locating, which in particular improve when the text is compressible. In this aspect our index is unique, as most compressed indexes are slower than their classical counterparts on secondary memory. We show experimentally that our index is very competitive against the alternatives, offering very relevant space/time tradeoffs.

We have also presented a simple scheme based on k-th order modeling plus statistical encoding to convert the sequence S into a compressed data structure. This structure permits retrieving any string of S of $\Theta(\log_\sigma n)$ symbols in constant time. This is an alternative to the first work achieving the same result [34], which is based on Ziv-Lempel compression and more complex (yet, their result holds simultaneously for all $k = o(\log_\sigma n)$, whereas ours requires to fix k at compression time). We also show how to adapt our structure for secondary memory, and apply it to compress T^{bwt} and the text itself. We show a relationship between the entropies of $H_1(T^{bwt})$ and $H_k(T)$. Other relationships are studied in [14], together with some mechanisms to add text to the compressed sequence. Later work [12] builds on our result and simplifies it.

We have also presented a compressed rank dictionary for secondary memory, which takes basically the same space of the BSGAP structure [19, Section 4.3] and performs just one access to disk. It needs a negligible amount of RAM space.

As future work we plan to improve the counting time of our secondary memory index. In this line, we are working on merging the CPT structure [6] with our index. The former contains a small disk-based tree structure plus a suffix array. By replacing that suffix array with our index, we will achieve a significant space reduction over the CPT (actually the space will be only slightly more than that of our current index). The counting times will be as good as for the CPT, and the locating times as good as for our index.

Acknowledgement. We thank Diego Arroyuelo for his help on the experimental part.

References

[1] D. Arroyuelo and G. Navarro. A Lempel-Ziv text index on secondary storage. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 83–94, 2007.

[2] R. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

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

[4] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Tech.Rep. 124, DEC, 1994.

[5] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Geometric burrows-wheeler transform: Linking range searching and text indexing. In *Proc. Data Compression Conference (DCC)*, pages 252–261, 2008.

[6] D. Clark and I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[7] P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 373–382, 1996.

[8] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 373–382, 1996.

[9] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[10] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

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

[12] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 372(1):115–121, 2007.

[13] A. Golynski. Optimal lower bounds for rank and select indexes. In *Proc. 33th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4051, pages 370–381, 2006.

[14] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 295–306, 2006.

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

[16] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[17] R. González and G. Navarro. Locally compressed suffix arrays. Technical Report TR/DCC-2008-13, Department of Computer Science, University of Chile, September 2008. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/lcsa.ps.gz`.

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

[19] A. Gupta. *Succinct Data Structures.* PhD thesis, Duke University, USA, 2007.

[20] A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theoretical Computer Science (TCS)*, 387(3):313–331, 2007.

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

[22] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[23] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. of the IEEE*, 88(11):1722–1732, 2000.

[24] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal on Computing*, 12(1):40–66, 2005.

[25] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th String Processing and Information Retrieval (SPIRE)*, pages 229–241, 2007.

[26] V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proceedings 15th Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 681–692, 2004.

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

[28] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

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

[30] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.

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

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

[33] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[34] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.

[35] R. Sinha, S. Puglisi, A. Moffat, and A. Turpin. Improving suffix array locality for fast pattern matching on disk. In *Proc. 28th ACM International Conference on Management of Data (SIGMOD)*, pages 661–672, 2008.

[36] 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.

A Appendix

We show that there is a relationship between the k-th order entropy of a text T and the first-order entropy of T^{bwt}. For this sake, we will compress T^{bwt} with a first-order compressor, whose output size is an upper bound to $nH_1(T^{bwt})$.

A *run* in T^{bwt} is a maximal substring formed by a single letter. Let $rl(T^{bwt})$ be the number of runs in T^{bwt}. It was proved [24] that $rl(T^{bwt}) \leq nH_k(T) + \sigma^k$ for any k. Our first-order encoder exploits this property, as follows:

- If $i > 1$ and $s_i = s_{i-1}$ then we output bit 0.
- Otherwise we output bit 1 followed by s_i in plain form ($\log \sigma$ bits).

Thus we encode each symbol of T^{bwt} by considering only its preceding symbol. The total number is $n + rl(T^{bwt}) \log \sigma \leq n(1 + H_k(T^{bwt}) \log \sigma + \frac{\sigma^k \log \sigma}{n})$ bits. The latter term is negligible for $k \leq (1-\epsilon)\log_\sigma n$, for any $0 < \epsilon < 1$. On the other hand, the total space obtained by our first-order encoder cannot be less than $nH_1(T^{bwt})$. Thus we get our result:

Lemma 5. *Let $T_{1,n}$ be a text over an alphabet of size σ. Then $H_1(T^{bwt}) \leq 1 + H_k(T) \log \sigma + o(1)$ for any $k \leq (1-\epsilon)\log_\sigma n$ and any constant $0 < \epsilon < 1$.*

We can improve this upper bound if we use Arithmetic encoding to encode the 0 and 1 bits that distinguish run heads. Their zero-order probability is at most $p = H_k(T) + \frac{\sigma^k}{n}$, thus the 1 becomes $-p \log p - (1-p)\log(1-p) \leq 1$. Likewise, we can encode the run heads s_i up to their zero-order entropy. These improvements, however, do not translate into clean formulas.

This shows, for example, that we can get (at least) about the same results of the Run-Length FM-Index [24] by compressing T^{bwt} using a entropy-bounded succinct data structure.