A simple storage scheme for strings achieving entropy bounds[*]

Paolo Ferragina[†] Rossano Venturini[‡]

Abstract

We propose a storage scheme for a string $S[1,n]$, drawn from an alphabet Σ, that requires space close to the k-th order empirical entropy of S, and allows to retrieve any ℓ-long substring of S in optimal $O(1+\frac{\ell}{\log_{|\Sigma|} n})$ time. This matches the best known bounds [14, 7], via the use of binary encodings and tables only. We also apply this storage scheme to prove new time *vs* space trade-offs for compressed self-indexes [5, 12] and the Burrows-Wheeler Transform [2].

1 Introduction

Starting from [5], the design of compressed (self)indexes for strings became an active field of research (see [12] and references therein). The key problem addressed in these papers consists of representing a string $S[1,n]$ drawn from an alphabet Σ within compressed space, and still be able to answer various types of queries (e.g. substring, approximate, ...) in efficient time, without incurring in the whole decompression of the compressed data. In these results, compressed space usually means space close to the k-th order empirical entropy of S,[1] and efficient time means something depending on the length of the searched string and as much independent as possible of S's length. Various trade-offs are known, and for them we refer the reader to [12].

Recently, Sadakane and Grossi [14] addressed the foundational problem of designing a compressed storage scheme for a string S which is *provably better* than storing S as a plain array of symbols. Here, the query operation to be supported is the retrieval of any ℓ-long substring of S in optimal $O(1+\frac{\ell}{\log_{|\Sigma|} n})$ time. Previous solutions [12], based on compressed indexes, incurred in an additional sub-logarithmic time overhead. Conversely, the Sadakane-Grossi's storage scheme is able to achieve the optimal time bound and occupy a number of bits upper bounded by the following function:[2]

$$(1.1)\quad nH_k(S)+O(\frac{n}{\log_{|\Sigma|} n}\,(k\log|\Sigma|+\log\log n)\,)$$

This storage scheme is based on a sophisticated combination of various techniques: Ziv-Lempel's string encoding [16], succinct dictionaries [13], and some novel succinct data structures for supporting navigation and path-decoding in Lz-tries. Since storing S by means of a *plain* array of symbols takes $\Theta(n\log|\Sigma|)$ bits, the scheme in [14] is effective when $k=o(\log_{|\Sigma|} n)$.

More recently, González and Navarro [7] proposed a simpler storage scheme achieving the same time and space bounds above, but exploiting a statistical encoder (namely, Arithmetic) on some of S's substrings. Unlike [14], this storage scheme requires to fix the order k of the entropy bound in advance.

In what follows we propose a very simple storage scheme that: (1) drops the use of any compressor (either statistical or Lz-like), and deploys only binary encodings and tables; (2) matches the space bound of Eqn. (1.1) simultaneously over all $k=o(\log_{|\Sigma|} n)$. As a corollary of this main result, we will prove new space/time trade-offs on the design of compressed indexes [5, 12] and the Burrows-Wheeler transform [2].

2 Some background

Let $S[1,n]$ be a string drawn from the alphabet $\Sigma=\{a_1,\ldots,a_h\}$. For each $a_i\in\Sigma$, we let n_i be the number of occurrences of a_i in S. Let $\{P_i=n_i/n\}_{i=1}^{h}$ be the empirical probability distribution for the string S. The *zero-th order empirical entropy* of the string S is defined as[3]

$$(2.2)\qquad H_0(S)=-\sum_{i=1}^{h}P_i\,\log P_i$$

[*] Author emails: ferragina@di.unipi.it and venturin@cli.di.unipi.it. Partially supported by Italian MIUR grants Italy-Israel FIRB "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics", PRIN "Algorithms for the Next Generation Internet and Web" (ALGO-NEXT), and by Yahoo! Research grant on "Data compression and indexing in hierarchical memories".

[†] Dipartimento di Informatica, University of Pisa, Italy

[‡] Dipartimento di Informatica, University of Pisa, Italy

[1] This is a lower bound to the space achieved by any k-th order compressor.

[2] As stated in [7], the term $k\log|\Sigma|$ appears erroneously as k in [14]. We therefore use the correct bound in this note.

[3] Throughout this paper we assume that all logarithms are taken to the base 2, whenever not explicitly indicated, and we assume $0\log 0=0$.

For any length-k string w, we denote by w_S the string of single symbols following the occurrences of w in S, taken from left to right. For example, if S = mississippi and w = si, we have w_S = sp since the two occurrences of si in S are followed by the symbols s and p, respectively.
The k-th order empirical entropy of S is defined as:

$$H_k(S) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_S| \, H_0(w_S). \tag{2.3}$$

Not surprisingly, for any $k \geq 0$ we have $H_k(S) \geq H_{k+1}(S)$. The value $|S| H_k(S)$ is a lower bound to the output size of any compressor that encodes each symbol with a code that only depends on the symbol itself and on the k immediately preceding symbols [10].

For simplicity of exposition, we will use $\mathcal{B} = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ to denote the infinite set of binary strings *canonically* ordered, where ϵ is the empty string. In other words, $\mathcal{B}$'s strings are ordered first by length, and then lexicographically by their content.

2.1 The Burrows-Wheeler transform. In [2] Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation now called the *Burrows-Wheeler Transform* (BWT from now on). The BWT transforms the input string S into a new string that is usually easier to compress. The BWT consists of three basic steps (see Figure 1):

1. append at the end of S a special symbol # smaller than any other symbol of Σ;
2. form a *conceptual* matrix $\mathcal{M}_S$ whose rows are the cyclic shifts of string $S\#$ in lexicographic order;
3. construct the transformed string L by taking the last column of the sorted matrix $\mathcal{M}_S$.

Notice that every column of $\mathcal{M}_S$, hence also the transformed string L, is a permutation of $S\#$. In particular the first column of $\mathcal{M}_S$, call it F, is obtained by lexicographically sorting the symbols of $S\#$ (or, equally, the symbols of L). Note also that when we sort the rows of $\mathcal{M}_S$ we are essentially sorting the suffixes of S because of the presence of the special symbol #. This shows that: (1) there is a strong relation between $\mathcal{M}_S$ and the *suffix array* data structure built on S; (2) symbols following the same substring (*context*) in S are grouped together in L, thus giving raise to clusters of nearly identical symbols. Property 1 is crucial for designing compressed indexes (see e.g. [12]), whereas property 2 is the key for the design of modern data compressors (see e.g. [10, 4]).

```
                              F             L
 mississippi#                 # mississipp i
 ississippi#m                 i #mississip p
 ssissippi#mi                 i ppi#missis s
 sissippi#mis                 i ssippi#mis s
 issippi#miss                 i ssissippi# m
 ssippi#missi       ⟹         m ississippi #
 sippi#missis                 p i#mississi p
 ippi#mississ                 p pi#mississ i
 ppi#mississi                 s ippi#missi s
 pi#mississip                 s issippi#mi s
 i#mississipp                 s sippi#miss i
 #mississippi                 s sissippi#m i
```

Figure 1: Example of Burrows-Wheeler transform for the string S = mississippi. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the column L; in this example the string ipssm#pissii.

The matrix $\mathcal{M}_S$ has also other remarkable properties. To illustrate them we introduce the following notation:

- $C[\cdot]$ denotes an array of length $|\Sigma|$ such that $C[c]$ stores the total number of S's symbols which are alphabetically smaller than c.
- $rank_c(L, i)$ is a function that returns the number of occurrences of symbol c in the prefix $L[1, i]$.
- $select_c(L, i)$ is a function that returns the position of the ith occurrence of symbol c in L.

As an example, in Figure 1 we have $C[\mathtt{s}] = 8$ because S contains 8 symbols smaller than s, $rank_s(L, 10) = 4$ because $L[1, 10]$ contains 4 occurrences of the symbol s, and $select_s(L, 3) = 9$ because $L[9] = \mathtt{s}$ is the third occurrence of s in L. The following properties of $\mathcal{M}_S$ have been proven in [2]:

a. Given the i-th row of $\mathcal{M}_S$, its last symbol $L[i]$ immediately precedes its first symbol $F[i]$ in the original string S.

b. Let $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$. $LF(\cdot)$ stands for *Last-to-First column mapping* since the symbol $L[i]$ is located in F at position $LF(i)$. For example in Figure 1 we have $LF(10) = C[\mathtt{s}] + rank_s(L, 10) = 12$ and both $L[10]$ and $F[LF(10)] = F[12]$ correspond to the first s in the string mississippi.

c. If $S[k]$ is the i-th symbol of L then $S[k-1] = L[LF(i)]$. For example in Figure 1 we have that

$S[3] = \mathtt{s}$ is the 10th symbol of L and we correctly have $S[2] = L[LF(10)] = L[12] = \mathtt{i}$.

Property **c.** makes it possible to retrieve S from L as follows. Initially set $k = n$, $i = 1$ and $S[n] = L[1]$ (since the first row of $\mathcal{M}_S$ is $\#S$). Then, for $k = n-1, \ldots, 1$ set $i = LF(i)$ and $S[k] = L[i]$. For example, to reconstruct the string $S = \mathtt{mississippi}$ of Figure 1 we start by setting $S[11] = L[1] = \mathtt{i}$. At the first iteration (i.e. $k = 10$ and $i = LF(1) = 2$), we get $S[10] = L[2] = \mathtt{p}$. At the second iteration (i.e. $k = 9$ and $i = LF(2) = 7$), we get $S[9] = L[7] = \mathtt{p}$, and so on.

The Burrows-Wheeler Transform has changed the way in which fundamental tasks for string processing and data retrieval, such as compression and indexing, are designed and engineered (see e.g. [4, 5, 8, 10, 12]). We are therefore interested in the succinct/compressed storage of the BWT because of its impact in all modern indexing and compression tools. In Sections 4 and 5 we will address this issue by using our storage scheme, detailed in the following section.

3 Our storage scheme for strings

Let $S[1,n]$ be a string drawn from an alphabet Σ, and assume that n is a multiple of $b = \lfloor \frac{1}{2} \log_{|\Sigma|} n \rfloor$. If this is not the case, we append to S the missing symbols taking them as the special *null symbol*.[4] We partition S into blocks S_i of size b each. Let $\mathcal{S}$ be the set of distinct blocks of S. The number of all blocks is $\frac{n}{b}$; the number of distinct blocks is $|\mathcal{S}| = O(|\Sigma|^b) = O(n^{1/2})$.

The encoding scheme. We sort the elements of $\mathcal{S}$ per decreasing frequency of occurrence in S's partition. Let $r(S_i)$ be the rank of the block S_i in this ordering, and let $r^{-1}(j)$ be its inverse function (namely, the one that returns the block having the given rank j). The storage scheme for S consists of the following information.

- Each block S_i is assigned a codeword $\mathtt{enc}(i)$ consisting of the binary string that has rank $r(S_i)$ in $\mathcal{B}$. Of course, $\mathtt{enc}(i)$ is not a *uniquely decodable code*, but the additional tables built below will allow us to decode it in constant time and within a space bounded by Eqn. (1.1).
- We build a bit sequence V obtained by juxtaposing the binary encodings of all S's blocks in the order of their appearance in S. Namely $V = \mathtt{enc}(1) \cdots \mathtt{enc}(\frac{n}{b})$.
- We store r^{-1} as a table of $O(|\Sigma|^b)$ entries, taking $O(|\Sigma|^b \log n) = o(n)$ bits.
- To guarantee constant-time access to the encodings of S's blocks and to ensure their decodings, we use a *two-level* storage scheme for the starting positions of $\mathtt{encs}$ (see [11]). Specifically, we *logically* group every $c = \Theta(\log n)$ contiguous blocks into one *superblock*, having thus size $bc \log |\Sigma| = \Theta(\log^2 n)$ bits. Table $T_{Sblk}[1, \frac{n}{bc}]$ stores the starting position of the encoding of every super-block in V, and table $T_{blk}[1, \frac{n}{b}]$ stores the starting position in V of the encoding of every block *relative to* the beginning of its enclosing super-block. Note that the starting position of each super-block is no more than $|V| = O(\frac{n}{b} \log n) = O(n \log |\Sigma|)$, whereas the relative position of each block is $O(\log^2 n)$. Consequently, tables T_{Sblk} and T_{blk} occupy $O(\frac{n}{bc} \log |V| + \frac{n}{b} \log \log n) = O(\frac{n \log \log n}{\log_{|\Sigma|} n})$ bits overall, and guarantee a constant-time access to every codeword $\mathtt{enc}(i)$ and its length.[5]

THEOREM 3.1. *Our storage scheme encodes $S[1,n]$ in $|V| + O(\frac{n \log \log n}{\log_{|\Sigma|} n})$ bits, which is upper bounded by Eqn. (1.1), simultaneously over all $k = o(\log_{|\Sigma|} n)$.*

Proof. We show that V is shorter than the compressed string produced by the encoding of [7], which was shown to achieve the bound stated in Eqn. (1.1). For completeness, that proof is reported in Appendix A.

The storage scheme in [7] encodes each block S_i by writing its first k symbols explicitly, i.e. $k \log |\Sigma|$ bits, and by then encoding the other $b - k$ symbols via a kth order statistical compressor E (which is then ensured to have k symbols to encode the next one). In the case that this encoding is longer than $\lfloor (1/2) \log n \rfloor$ bits, block S_i is written explicitly without any compression. To distinguish between these two cases, [7] keeps some extra tables and data structures.

Note that the codewords assigned by E are a subset of $\mathcal{B}$, whereas the codewords assigned by $\mathtt{enc}$ are the first $|\mathcal{S}|$ binary strings of $\mathcal{B}$. Given that $\mathcal{B}$ is the set of shortest codewords assignable to $\mathcal{S}$'s strings, our encoding $\mathtt{enc}$ is better than E because it follows the *golden rule* of data compression: it assigns shorter codewords to more frequent symbols. ■

We now show how to decode in constant time a generic block S_k. This will be enough to prove the result for any l-long substring of S.

We first derive the starting position $p(k)$ of the string $\mathtt{enc}(k)$ that encodes S_k in V. Namely, we compute the super-block number $h = \lceil k/c \rceil$ containing

[4] This will add to the entropy estimation a negligible additive term equal to $O(\log |\Sigma| \ \log_{|\Sigma|} n) = O(\log n)$ bits.

[5] It suffices to compute the starting position of $\mathtt{enc}(i)$ and $\mathtt{enc}(i+1)$, if any.

$\texttt{enc}(k)$, and its starting bit-position $y = T_{Sblk}[h]$ within V. Then, we compute $x = T_{blk}[k]$ as the relative bit-position of $\texttt{enc}(k)$ within its enclosing super-block. Thus $p(k) = x + y$.

Similarly, we derive the starting position $p(k+1)$ of $\texttt{enc}(k+1)$ in V (if any, otherwise we set $p(k+1) = |V|+1$). We can thus fetch $\texttt{enc}(k) = V[p(k), p(k+1)-1]$ in constant time since $|\texttt{enc}(k)| = p(k+1) - p(k) = O(\log n)$.

We finally decode $\texttt{enc}(k)$ as follows. Let v be the integer value represented by the binary string $\texttt{enc}(k)$, where $v = 0$ if $\texttt{enc}(k) = \epsilon$. Because of the canonical ordering of $\mathcal{S}$, S_k is computed as the block having rank $z = 2^{|\texttt{enc}(k)|} + v$. That is, $S_k = r^{-1}(z)$.

Theorem 3.2. *Our storage scheme retrieves any ℓ-long substring of S in optimal $O(1 + \frac{\ell}{\log_{|\Sigma|} n})$ time.*

Proof. The algorithm described above allows to retrieve any block S_k in constant time. The theorem follows by observing that any l-long substring $S[j, j+l-1]$ spans $O(1 + \frac{l}{\log_{|\Sigma|} n})$ blocks of S. ∎

4 BWT compression and access

In Section 2 we introduced the Burrows-Wheeler Transform of a string $S[1,n]$, denoted by $\texttt{bwt}(S)$, and highlighted its interesting properties and applications to data compression and compressed data indexing. In this section we show that our storage scheme can be used upon the string $\texttt{bwt}(S)$ in order to achieve an interesting *compressed-space bound* which depends on both the kth order entropy of the string S and the kth order entropy of its BW-transformed string $\texttt{bwt}(S)$. The relation between these two entropies will be commented below.

Theorem 4.1. *Our storage scheme applied on $L = \texttt{bwt}(S)$ takes no more than $n \min\{H_k(L), H_k(S)\} + o(n \log |\Sigma|)$ bits, simultaneously over all $k = o(\log_{|\Sigma|} n)$. Any ℓ-long substring of L can be retrieved in optimal $O(1 + \frac{\ell}{\log_{|\Sigma|} n})$ time.*

Proof. Let E_k^R be a (semi-static) kth order statistical compressor for the reversed string S^R; and let $C_k(j)$ be the k-long prefix of the jth row of $\mathcal{M}_S$. By the properties of $\texttt{bwt}(S)$ (see Section 2), $C_k(j)$ follows $L[j]$ and thus $C_k(j)$ can be denoted as the *following* k-long context of $L[j]$. Of course, if we consider the reversed string S^R, the string $[C_k(j)]^R$ is the *preceding* k-long context of $L[j]$.

We partition L into b-long substrings $L_1 L_2 \ldots L_{n/b}$ (called hereafter *blocks*), where $b = \lfloor \frac{1}{2} \log_{|\Sigma|} n \rfloor$. Note that each block L_i corresponds to the range $[(i-1)b + 1, ib]$ of b rows in the BWT-matrix $\mathcal{M}_S$. We say that the block L_i is *k-prefix-equal* iff the k-long prefixes of all rows in $\mathcal{M}_S[(i-1)b+1, ib]$ are equal. Otherwise, L_i is said to be *k-prefix-different*. Notice that, the total number of k-prefix-different blocks is $O(|\Sigma|^k)$ because that is the number of distinct k-long strings drawn from alphabet Σ. From the BWT-properties recalled above, we know that the symbols of a given k-prefix-equal block L_i have the same *preceding* k-long context in S^R, i.e. $[C_k(bi)]^R$.

To prove the theorem we consider a preliminary encoding scheme for L's blocks. Each k-prefix-equal block L_i is encoded as follows: first, write explicitly the string $C_k(bi)$, using $k \log |\Sigma|$ bits; and then, encode each individual symbol of L_i by using E_k^R and the knowledge of its k-long context $[C_k(ib)]^R$ (which is equal for all symbols in L_i). Conversely, the k-prefix-different blocks are written without any compression using $O((b+k)\log|\Sigma|) = O(\log n + k \log |\Sigma|)$ bits: k occurrences of a special *null* symbol plus the b-long block itself.

Let us now evaluate the space occupancy of this encoding scheme. Since there are $O(|\Sigma|^k)$ k-prefix-different blocks, their (plain) encoding takes $O(|\Sigma|^k(\log n + k \log |\Sigma|)) = O(|\Sigma|^k \log n) = o(n)$ bits, since we assumed $k = o(\log_{|\Sigma|} n)$. As far as the evaluation of the space required by the encoding of the k-prefix-equal blocks is concerned, we proceed as follows. Let ρ be a k-long string, and let us denote by $L\langle\rho\rangle$ the sequence of k-prefix-equal blocks in L having context ρ. Clearly, the length of $L\langle\rho\rangle$ is a multiple of b. Additionally, since the symbols in $L\langle\rho\rangle$ follow the string ρ^R in S^R, we can conclude that $L\langle\rho\rangle$ is a subsequence of the string $(\rho^R)_{S^R}$ (see definition of w_S in Section 2). In particular, $|L\langle\rho\rangle| \leq |(\rho^R)_{S^R}|$.

If we concatenate the strings $L\langle\rho\rangle$, as ρ varies over all k-long strings in Σ^k (in lexicographic order), we obtain a string which is equal to L except that some of its blocks are missing, namely the k-prefix-different blocks. Now, recall that we have used E_k^R to encode each individual symbol of $L\langle\rho\rangle$, by using the knowledge of the context ρ^R. For the symbols of $L\langle\rho\rangle$, the compressor E_k^R has totally emitted no more than $|L\langle\rho\rangle|\ H_0((\rho^R)_{S^R}) + O(kl_\rho \log |\Sigma|)$ bits, where l_ρ is the number of blocks forming $L\langle\rho\rangle$. Indeed the term $H_0((\rho^R)_{S^R})$ derives from the fact that E_k^R is a semi-static k-order statistical compressor which therefore builds, for each k-long context $\rho \in \Sigma^k$, a 0th order model over all symbols following ρ^R in S^R (which form exactly the string $(\rho^R)_{S^R}$). We sum this bound over all ρs, by recalling three things: (1) the definition of k-th order entropy in Eqn. 2.3, (2) $|L\langle\rho\rangle| \leq |(\rho^R)_{S^R}|$, (3) $\sum_{\rho \in \Sigma^k} l_\rho \leq n/b$. Thus we obtain $(\sum_{\rho \in \Sigma^k} |\rho_{S^R}|\ \ H_0(\rho_{S^R})) + O((n/b) k \log |\Sigma|) =$

$nH_k(S^R) + O((n/b)k \log|\Sigma|)$ bits. Finally, by noting that the difference between $H_k(S)$ and $H_k(S^R)$ is negligible [5], we get the bound in terms of $H_k(S)$, namely $nH_k(S) + O((n/b)k\log|\Sigma|)$ bits.

Now we turn this encoding scheme into another one which will be easily related to our encoder $\mathtt{enc}$. Consider the set $\mathcal{S} = \{\mathcal{L}_1, \ldots, \mathcal{L}_r\}$ of distinct blocks of L, with $r = O(n^{\frac{1}{2}})$. A block $\mathcal{L}_i$ may occur many times in L and all the corresponding rows in $\mathcal{M}_S$ may have different k-long prefixes. Therefore, the block $\mathcal{L}_i$ can have up to $O(|\Sigma|^k)$ different encodings because it may have at most $|\Sigma|^k$ distinct k-contexts (as a k-prefix-equal block) and at most $|\Sigma|^k$ plain encodings (as a k-prefix-different block). We reassign to $\mathcal{L}_i$ the shortest of these codewords. As a result, each distinct block $\mathcal{L}_i$ will have one unique encoding, independently of its type (k-prefix-equal or k-prefix-different), and still the previous space bound will hold.

Let us now take our storage scheme $\mathtt{enc}$ of Section 3, apply it onto string L, and compare the length of the resulting compressed string against the previous encoding. By Theorem 3.1, we know that $\mathtt{enc}$ encodes L within $nH_k(L) + O((n/b)k\log|\Sigma|)$ bits. As far as the bound with $H_k(S)$ is concerned, we observe that $\mathtt{enc}$ will perform better than the previous encoding, because the codewords assigned to each block by that encoder are a subset of $\mathcal{B}$. Therefore, by an argument similar to the one used in the proof of Theorem 3.1, we can conclude that $\mathtt{enc}$ will take less than $nH_k(S) + O((n/b)k\log|\Sigma|)$ bits. Combining these two space bounds, we derive the statement of the theorem. ∎

The relation between $H_k(S)$ and $H_k(L)$ is still unknown. However, it is possible to provide examples of strings for which one entropy is smaller than the other. For example, let us consider the string $S = (bba)^d$ and set $k = 1$. By Eqn. 2.3 we have

$$nH_1(S) = \quad (d-1)H_0(b^{d-1}) + 2dH_0((ba)^d) = 2d = \tfrac{2}{3}n$$

On the other hand, since $L = \mathtt{bwt}(S) = b^{2d}a^d$, we have

$$\begin{aligned} nH_1(L) &= (d-1)H_0(a^{d-1}) + 2dH_0(b^{2d-1}a) \\ &= -(2d-1)\log\tfrac{2d-1}{2d} - \log\tfrac{1}{2d} \\ &= 2d\log 2d - (2d-1)\log(2d-1) \\ &= O(\log n) \end{aligned}$$

which is exponentially smaller than $nH_1(S)$, for any $d > 1$. Now, we show an example in which $nH_1(L) > nH_1(S)$. Let $S = (a_1a_2\ldots a_d)^d$ and $k = 1$. We have

$$nH_1(S) = 0$$

Since $L = \mathtt{bwt}(S) = a_d^d a_1^d \ldots a_{d-1}^d$, we have

$$\begin{aligned} nH_1(L) &= -(d-1)((d-1)\log\tfrac{d-1}{d} + \log\tfrac{1}{d}) \\ &= \Theta(\sqrt{n}\ \log\sqrt{n}) \end{aligned}$$

5 Compressed indexing

[5] showed that the compressed indexing problem over a text S can be *reduced* into the, possibly easier, problem of supporting fast *rank/select* operations over the string $L = \mathtt{bwt}(S)$. The literature is now full of results on compressed data structures for fast rank/select primitives [12]. In this volume, Barbay *et al.* [3] improved the known results for strings drawn from arbitrary alphabets. They proposed an algorithmic scheme for supporting *rank/select* that may take advantage from any effective compressed storage scheme for strings (like ours!).

Theorem 5.1. *Let $S[1,n]$ be a string over an alphabet Σ and assume that we have a storage scheme for S supporting access to any of its symbols in $f(n,|\Sigma|)$ time. There exists a succinct index requiring additional $n \cdot o(\log|\Sigma|)$ bits that supports $rank_c$ in $O(\log\log|\Sigma|\log\log\log|\Sigma|(f(n,|\Sigma|) + \log\log|\Sigma|))$ time, and $select_c$ in $O(\log\log\log|\Sigma|(f(n,|\Sigma|) + \log\log|\Sigma|))$ time.*

We can then combine this result with our storage scheme for $\mathtt{bwt}(S)$ (Theorem 4.1), and thus obtain:

Lemma 5.1. *Let $S[1,n]$ be a string over an alphabet Σ, and let $L = \mathtt{bwt}(S)$ be its BW-transformed string. There exists a compressed index of size $n \cdot (\min\{H_k(L), H_k(S)\} + o(\log|\Sigma|))$ bits, that supports rank/select operations over the string L in $o((\log\log|\Sigma|)^3)$ time. The space bound holds simultaneously over all $k = o(\log_{|\Sigma|} n)$.*

To be precise, the time bound of $rank_c$ is $O((\log\log|\Sigma|)^2\log\log\log|\Sigma|)$, and the time bound of $select_c$ is $O((\log\log|\Sigma|)\ \log\log\log|\Sigma|)$.

The FM-index [5] is a class of compressed self-indexes. The main search functionalities supported by FM-indexes are count and locate. The count query returns the number *occ* of occurrences of a pattern $P[1,m]$ in the indexed string S; the locate query lists the *occ* positions of P's occurrences in S. [5] has shown that the implementation of these query operations essentially boils down to the space-time efficient design of data structures for computing $C[c]$, $L[i]$ and $rank_c(L,i)$ (see Section 2). Consequently, by using the data structure of Lemma 5.1, we can design an FM-index that supports count in $O(m(\log\log|\Sigma|)^2\log\log\log|\Sigma|)$ time, and locate in $O(occ \cdot \log^{1+\epsilon} n \cdot (\log\log|\Sigma|)^2 \log\log\log|\Sigma|)$ time. As in [5], supporting locate queries needs additional $O(\frac{n}{\log^\epsilon n}) = o(n)$ bits, where $0 < \epsilon < 1$.

Theorem 5.2. *Let $S[1,n]$ be a string over alphabet Σ, there exists an implementation of the FM-index taking $n \cdot (\min\{H_k(L), H_k(S)\} + o(\log|\Sigma|))$ bits of storage, for* *any $k = o(\log_{|\Sigma|} n)$, and supporting the* count *query in $O(m(\log\log|\Sigma|)^2 \log\log\log|\Sigma|)$ time, and the* locate *query in $O(occ\ (\log^{1+\epsilon} n)\ (\log\log|\Sigma|)^2 \log\log\log|\Sigma|)$ time.*

6 Conclusions

The simplification we have proposed in this paper to the results of [14, 7] drives us to two possible considerations. One is that we now have a *class-note* solution for the string storage problem that, as deeply illustrated in [14], may find successful applications into many other interesting contexts: e.g. it may turn succinct or 0-th order entropy data structures into k-th order entropy data structures (see [3, 9, 14] and references therein). The second consideration refers to future research. Namely, all known solutions are far from being usable in practice because of the additive term which usually *dominates* the kth order entropy term. More research is therefore needed to either achieve a space bound close to the one attainable with the k-th order compressors of the Bzip-family [10, 8, 4], for which the additive term is $O(|\Sigma|^k \log n)$ bits, or to show a lower bound related to kth order entropy, in the vein of [1, 6]. Since our storage scheme, unlike [14, 7], does not use any sophisticated data compression machinery, we are led to think that there is room for improvement!

References

[1] P. Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Procs ACM-SIAM SODA*, 11-12, 2005.

[2] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[3] J. Barbay, J. I. Munro, M. He and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Procs ACM-SIAM SODA*, 2007 (this volume).

[4] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

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

[6] A. Golynski. Optimal lower bounds for rank and select indexes. In *Procs ICALP*, LNCS 4051, 370-381, 2006.

[7] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Procs CPM*, LNCS 4009, 295–306, 2006.

[8] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Procs ACM-SIAM SODA*, 841–850, 2003.

[9] J. Jansson, K. Sadakane, and K.K. Sung. Ultra-succinct representation of ordered trees. In *Procs ACM-SIAM SODA*, 2007 (this volume).

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

[11] I. Munro. Tables. In *Procs FST-TCS*, LNCS 1180, 37–42, 1996.

[12] G. Navarro and V. Mäkinen. Compressed full-text indexes. Technical Report TR/DCC-2006-6, Dept. of Computer Science, University of Chile, April 2006.

[13] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Procs ACM-SIAM SODA*, 233–242, 2002.

[14] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Procs ACM-SIAM SODA*, 1230–1239, 2006.

[15] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[16] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transaction on Information Theory*, 24:530–536, 1978.

A Bounding the space of [7]'s scheme

For completeness of exposition we report here the proof of [7] showing that their storage scheme actually achieves the space bound stated in Eqn. (1.1).

Let us denote by f_i the number of occurrence of each symbol $S[i]$ given its preceding kth order context $S[i-k, i-1]$ divided by n. Note that $n \times f_i$ is the number of times symbol $S[i]$ occurs after $S[i-k, i-1]$. Therefore, according to the notation in Section 2, $n \times f_i$ is the number of times symbol $S[i]$ occurs within w_S, where $w = S[i-k, i-1]$. It is easy to see that a (semi-static) k-order modeler can compute all the frequencies f_i via two passes over S, hence in $O(n)$ time.

Arithmetic encoding is one of the most effective statistical encoders [15]. Given the f_is, it represents the string S with a range of size $F = f_1 \times f_2 \times \cdots \times f_n$. It is well known [15] that $2 + \log(1/F) = 2 + \sum_{i=1}^{n} \log(1/f_i)$ bits are enough to distinguish a number within that range. The binary representation of this number is the Arithmetic compression of S. If we compute $\sum_{i=k+1}^{n} \log(1/f_i)$, and then we group all the terms with the same kth order context, we obtain a summation upper bounded by $nH_k(S)$. Additionally, since $f_i \geq 1/n$, we have that $\sum_{i=1}^{k} \log(1/f_i) = O(k \log n)$. As a result, a (semi-static) kth order Arithmetic encoder compresses the whole S within $nH_k(S) + 2 + O(k \log n)$ bits.

The storage scheme of [7] compresses the blocks S_i of S individually: the first k symbols of S_i are represented explicitly, the remaining $b - k$ symbols of S_i are compressed via the k-order Arithmetic encoder (hence using their kth order frequencies fs). This blocking approach increases the Arithmetic compression cost of the whole S, shown above, by $O((n/b)k \log |\Sigma|)$ bits, which accounts for the cost of explicitly storing $S_i[1, k]$.

To decode in constant time every block S_i, [7] uses a table indexed by the pair $\mathcal{P}[i] = \langle\ S_i[1, k]$, Arithmetic encoding of $S_i[k + 1, b]\ \rangle$. It is easy to observe that $\mathcal{P}[i]$ uniquely identifies S_i. A small technical point is that, if the encoding of S_i is longer than $\lfloor (1/2) \log n \rfloor$ bits, then block S_i is written explicitly without any compression. This table uses $O(2^{k \log |\Sigma| + (1/2) \log n} \log n) = O(|\Sigma|^k\ n^{1/2} \log n)$ bits.

Summing up the cost of the block's encodings and the space occupancy of the decoding table, we get the space bound of Eqn. (1.1), whenever $k = o(\log_{|\Sigma|} n)$.