Dynamic Entropy-Compressed Sequences and Full-Text Indexes ⋆

Veli Mäkinen [1]

Department of Computer Science, University of Helsinki, Finland.
`vmakinen@cs.helsinki.fi`

Gonzalo Navarro [2]

Department of Computer Science, University of Chile.
`gnavarro@dcc.uchile.cl`

Abstract

We give new solutions to the Searchable Partial Sums with Indels problem. Given a sequence of n k-bit numbers, we present a structure taking $kn+o(kn)$ bits of space, able of performing operations *sum*, *search*, *insert*, and *delete*, all in $O(\log n)$ worst-case time, for any $k = O(\log n)$. This extends previous results by Hon et al. [ISAAC 2003] achieving the same space and $O(\log n/\log\log n)$ time complexities for the queries, yet complexities for *insert* and *delete* are amortized and worse than ours, and supported only for $k = O(1)$. Our result matches an existing lower bound for large values of k.

We also give new solutions to the Dynamic Sequence problem. Given a sequence of n symbols in the range $[1, \sigma]$ with binary zero-order entropy H_0, we present a dynamic data structure that requires $nH_0 + o(n \log \sigma)$ bits of space, which is able of performing *rank* and *select*, as well as inserting and deleting symbols at arbitrary positions, in $O(\log n \log \sigma)$ time. Our result is the *first* entropy-bound dynamic data structure for *rank* and *select* over general sequences.

In the case $\sigma = 2$, where both previous problems coincide, we improve the dynamic solution of Hon et al. in that we compress the sequence. The only previous result with entropy-bound space for dynamic binary sequences is by Blandford and Blelloch [SODA 2004], which has the same complexities as our structure, but does not achieve constant 1 multiplying the entropy term in the space complexity.

Finally, we present a new dynamic compressed full-text self-index, for a collection of texts over an alphabet of size σ, of overall length n and h-th order empirical entropy H_h. The index requires $nH_h + o(n \log \sigma)$ bits of space, for any $h \le \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. It can count the number of occurrences of a pattern of length m in time $O(m \log n \log \sigma)$. Reporting each such occurrence can be supported in $O(\log^2 n \log\log n)$ time, and displaying a context of length ℓ from a text takes time $O(\log n(\ell \log \sigma + \log n \log\log n))$. Insertion/deletion of a text to/from the collection takes $O(\log n \log \sigma)$ time per symbol. This largely improves the space of a previous result by Chan et al. [CPM 2004] in exchange for a slight complexity penalty. We achieve at the same time the *first* dynamic index requiring essentially nH_h bits of space, and the *first* construction of a compressed full-text self-index within that working space. Previous results achieve at best $O(nH_h)$ space with constants larger than 1 (Ferragina and Manzini [FOCS 2000], Arroyuelo and Navarro [ISAAC 2005]) and higher time complexities.

An important result we prove in this paper is that the wavelet tree of the Burrows-Wheeler transform of a text, if compressed with a technique achieving zero-order compression locally (e.g., Raman et al. [SODA 2002]), automatically achieves h-th order entropy space for any h. This unforeseen relation is essential for the results of the previous paragraph, but it also derives into significant simplifications on many existing static compressed full-text self-indexes that build on wavelet trees.

1 Introduction and Related Work

The study of compressed data structures aims to represent classical structures like trees, graphs, text indexes, etc., in the smallest possible space without challenging the functionality of the structure; the original operations should be supported efficiently without decompressing the whole structure.

The well-known Searchable Partial Sums problem consists of maintaining a sequence A of nonnegative integers $a_1 \ldots a_n$, each of k bits, supporting queries on the prefix sums and limited updates on the values. An extension [22] called Searchable Partial Sums with Indels allows insertions and deletions of values as well.

The restricted version where $k = 1$, and thus the numbers are actually bits, is called the Dynamic Bit Vector problem. In this case the update operation means flipping the bit. Also, queries on the prefix sums correspond to the well-known *rank* and *select* queries on bit sequences. The extension to the Dynamic Bit Vector with Indels problem is immediate.

Some recent articles deal with the problem of designing succinct dynamic data structures, requiring $kn + o(kn)$ bits of space, for these problems [31,22]. The best current result for $k = O(1)$ [22] requires $kn+o(kn)$ bits of space, $O(\log_b n)$

* A preliminary partial version of this paper appeared in *Proc. CPM 2006, LNCS 4009.*
^{1} Funded by the Academy of Finland under grant 108219.
^{2} Partially funded by Fondecyt Grant 1-050493, Chile.

time for *rank* and *select*, and $O(b)$ time for flips, for any $b \geq \log n / \log \log n$.[3] Insertions and deletions can be supported in $O(b)$ amortized time, for $b \geq (\log n / \log \log n)^2$. These results scale well to larger k, obtaining the same worst-case complexities for prefix sum queries as well as for updates, but insertions and deletions are not supported anymore. The best time complexities achieved for this problem are worst-case $O(\log n / \log(w/\delta))$, where updates that add/subtract a number of δ bits are permitted [30]. They show that this complexity is optimal, but insertions and deletions are not considered.

In this paper we extend this result by achieving $kn + o(kn)$ bits of space and $O(\log n)$ worst case time complexity for all the operations, for any $k = O(\log n)$. For larger $k = O(w)$, where w is the machine word size under the RAM model of computation, our time complexity raises to $O(w/\log^{1-\varepsilon} n + \log n)$, for any constant $\varepsilon > 0$. Moreover, we refine this result for the case of strictly positive numbers: the total space is not anymore kn bits, but the sum of the exact number of bits necessary to represent each number. For the case $k = \Theta(\log n) = \Theta(w)$ the obtained time complexity is optimal, given the lower bound $\Omega(\log n / \log(w/k)) = \Omega(\log n)$ for the more restricted problem with (arbitrary) updates [30][4].

Let us return to the Dynamic Bit Vector with Indels problem. We go further than representing them using $n+o(n)$ bits, and achieve a representation that uses $nH_0 + o(n)$ bits of space, where $0 < H_0 \leq 1$ is the empirical zero-order entropy of the sequence. This is an improvement over an $O(nH_0)$ result by Blandford and Blelloch [4] since, although their results are more general, they do not achieve constant 1 multiplying the entropy term in the space complexity. The comparison of time complexities against partial sums with $k = 1$ [22] stays as above, but now we compress the sequence. Our space complexity has been only achieved in the static scenario [31], where no updates are possible.

We further generalize this result to sequences of symbols over an alphabet $[1, \sigma]$, where operations *rank* and *select* generalize to $rank_a(A, i)$, counting the occurrences of symbol a in $A[1, i]$, and $select_a(A, j)$, giving the position of the j-th a in A. By using wavelet trees [17] we achieve $nH_0 + o(n \log \sigma)$ bits of space and $O(\log n \log \sigma)$ time complexities. This space has been previously achieved only for static data structures, with query time $O(\lceil \log \sigma / \log \log n \rceil)$ time for *rank* and *select* but no support for updates [32,15]. By using multiary wavelet trees, we can reduce the query times to $O(\frac{1}{\epsilon} \log n \lceil \log \sigma / \log \log n \rceil)$ in

[3] In this paper log stands for $\log_2$.

[4] Note that, even if we wish to restrict our updates to $\delta < k$ bits, insertions and deletions would permit simulating general updates, thus the stated lower bound must hold anyway if insertions and deletions are permitted, unless we restrict these in a rather unnatural way.

exchange for increasing the update times to $O(\frac{1}{\epsilon}\log^{1+\epsilon} n/\log\log n)$, for any $\epsilon > 0$.

Moreover, all our results work under weaker assumptions on the RAM model than many previous results on dynamic settings. Instead of assuming $\log n = \Theta(w)$ as it is frequent in the literature, we opt for the weaker condition $\log n = O(w)$. We show that the results stay the same in terms of time and space complexities, except for $O(w)$ extra bits of space that are required for a constant number of system pointers.

Let us now regard sequences of symbols with another semantics. The indexed string matching problem is that of, given a long text $T[1,n]$ over an alphabet Σ of size σ, building a data structure called *full-text index* on it, to solve two types of queries: (a) Given a short pattern $P[1,m]$ over Σ, *count* the occurrences of P in T; (b) *locate* those *occ* positions in T. There are several classical full-text indexes requiring $O(n\log n)$ bits of space which can answer counting queries in $O(m\log\sigma)$ time (like suffix trees [1]) or $O(m+\log n)$ time (like suffix arrays [26]). Both locate each occurrence in constant time once the counting is done. Similar complexities are obtained with modern compressed data structures [12,17,15], requiring space $nH_h+o(n\log\sigma)$ bits (for some small h), where $H_h \leq \log\sigma$ is the h-th order empirical entropy of T. These indexes are often called *compressed self-indexes* refering to their space requirement and to their ability to work without the text and even fully replace it, by delivering any text substring without accessing T.

There exist dynamic self-indexes by Chan, Hon, and Lam [7]. One requires $O(n\sigma)$ bits of space, and it can count the number of occurrences of a pattern of length m in time $O(m\log n)$. Insertions and deletions require $O(\sigma\log n)$ and $O((\sigma+\log n)\log n)$ time per character, respectively. The other requires $O(n\log\sigma)$ bits of space and counts in time $O(m\log^2 n)$. Insertions and deletions take $O(\log n)$ time per character. In either case, each occurrence position can be retrieved in $O(\log^2 n)$ time. The structures can be combined so as to get $O(\sigma n)$ bits of space, $O(m\log n)$ counting time, and $O(\sigma\log n)$ insertion and deletion time per character.

The main building block in compressed self-indexes is function $rank_a$. Actually, our dynamic compressed $rank_a$ structure can be used to implement a dynamic compressed self-index that takes $nH_h+o(n\log\sigma)$ bits of space, for any $h \leq \alpha\log_\sigma n$ and constant $0 < \alpha < 1$. This is obtained by plugging our structure for symbol sequences in the dynamic self-index of [7]. Our counting time is $O(m\log n\log\sigma)$. We can locate each occurrence in time $O(\log^2\log\log n)$, and display a text context of length ℓ in time $O(\log n(\ell\log\sigma+\log n\log\log n))$. Insertion and deletion of a text to the collection takes $O(\log n\log\sigma)$ time per symbol. Compared with the original indexes of Chan et al., we obtain a significant space saving and, depending on the case, better update or better counting times.

The fact that plugging our nH_0-bits sequence representation into the self index of [7] yields h-th order compression stems from the fact that the sequence we are representing is the Burrows-Wheeler transform of the text collection [5,27]. This is a striking result we prove in this paper. For several years, much effort has been spent in designing sophisticated (static) data structures on top of the plain wavelet tree so as to reduce its nH_0-bit size to nH_h bits [17,15,24]. In this paper we show that this is automatically achieved by the *original* wavelet tree without any further effort! Thus, as a byproduct, we obtain a significant simplification in the design of static data structures for this problem.

Ours is the *first* dynamic compressed self-index with space essentially equal to the h-th order empirical entropy of the text collection, which in addition can be built within this working space. We know only of two previous dynamic full-text self-indexes. The older [12] requires $O(nH_h)$ bits of space (with constant 5 at least), $O(m \log^3 n)$ counting time, $O(\log n)$ amortized insertion time per character, and $O(\log^2 n)$ amortized deletion time per character. A newer one [6] requires $O(n)$ bits of space, $O(m \log n)$ counting time, $O(\log^2 n)$ locating time per occurrence, and $O(\log n)$ insertion/deletion time per character.

As a plus, we obtain an $O(n \log n \log \sigma)$ time construction algorithm for static self-index requiring $nH_h + o(n \log \sigma)$ bits *working space* during construction (the same as the final structure). Previous construction algorithms within entropy space achieve $O(nH_0)$ bits of space and $O(n \log n)$ time [21], or $O(nH_h)$ bits of space (with constant larger than 4) and $O(\sigma n)$ time [2].

Several other compressed indexes can be obtained using our algorithm. Moreover, it is very easy to obtain the Burrows-Wheeler transform of T from the index we build, within the same $O(n \log n \log \sigma)$ time. A recent result [23] achieves $n \log \sigma + O(n)$ bits and $O(n \log^2 n)$ time.

2 Definitions

To simplify notation, we ignore roundings. When refering to number of bits, we use simply $\log n$ to refer to $\lfloor (\log n) + 1 \rfloor$. That is, $\log \log n$ bits means actually $\lfloor (\log \lfloor (\log n) + 1 \rfloor) + 1 \rfloor$ bits. Similarly $(\log n)/2$ is the integer nearest to $\lfloor (\log n) + 1 \rfloor / 2$, and so on.

We assume our sequence $A = a_1 \ldots a_n$ to be drawn from an alphabet $\{0, 1, \ldots \sigma - 1\}$. Let n_c denote the number of occurrences of symbol c in A, i.e., $n_c = |\{i \mid a_i = c\}|$. Then the zero-order *empirical entropy* is defined as $H_0(A) = \sum_{0 \le c < \sigma} \frac{n_c}{n} \log \frac{n}{n_c}$. This is the lower bound for the average code word length of any compressor that fixes the code words to the symbols independently of the context they appear in. A tighter lower bound for sequences is the *h-th order empirical entropy* $H_h(A)$, where the compressor can fix the code word based on the h-symbol context following the symbol to be coded.[5] Formally, it can be defined as $H_h(A) = \sum_{w \in \Sigma^h} \frac{n_w}{n} H_0(A|w)$, where n_w denotes the number of occurrences of substring w in A and $A|w$ denotes the concatenation of the symbols appearing immediately before those n_w occurrences [27]. Substring $w = A[i+1, i+h]$ is called a *h-context* of symbol a_i. We take A here as a *cyclic string*, such that a_n precedes a_1, and thus the amount of h-contexts is exactly n.

We assume a random access machine with word size w; typical arithmetic operations on w-bit integers are assumed to take constant time. We make the minimal assumption that $\log n = O(w)$, instead of the common stronger assumption $\log n = \Theta(w)$.

We study the following problems in this paper:

The Dynamic Sequence with Indels problem is to maintain a (virtual) sequence $A = a_1 \ldots a_n$, $a_i \in \{0, 1, \ldots, \sigma - 1\}$, supporting the operations:

- $rank_c(A, i)$ returns the number of occurrences of symbol c in $a_1 \cdots a_i$;
- $select_c(A, j)$ returns the index i containing j-th occurrence of c;
- $insert(A, c, i)$ inserts $c \in \{0, 1, \ldots \sigma - 1\}$ between a_{i-1} and a_i; and
- $delete(A, i)$ deletes a_i from the sequence.

The Dynamic Bit Vector with Indels problem is a restriction of the above to alphabet $\{0, 1\}$ (i.e., $\sigma = 2$). Then we use short-hand notation $rank(A, i) = rank_1(A, i)$ and $select(A, i) = select_1(A, i)$. Notice that $rank_0(A, i) = i - rank_1(A, i)$, but the same does not apply for $select_0(A, j)$; so both *select* queries must be handled.

The Searchable Partial Sums problem consists of maintaining a sequence A of nonnegative integers $a_1 \ldots a_n$, each of k bits, so that we can perform the following queries and operations on them:

- $sum(A, i)$ returns $\sum_{t=1}^{i} a_t$;
- $search(A, j)$ returns the smallest i such that $sum(A, i) \geq j$; and
- $update(A, i, \Delta)$ increases a_i by Δ, assuming $a_i + \Delta$ is within bounds and $\Delta = O(\text{polylog}(n))$.

[5] It is more logical (and hence customary) to define the context as the h symbols preceding a symbol, but we use the reverse definition for technical convenience. If this is an issue, the sequences can be handled in reverse order to obtain results on the more standard definition. It is anyway known that both definitions do not differ by much [14].

A more general problem called Searchable Partial Sums with Indels includes also the following operations:

- $insert(A, i, x)$ inserts x between a_{i-1} and a_i.
- $delete(A, i)$ deletes a_i from the sequence.

Notice that the Searchable Partial Sums with Indels problem with $k = 1$ is equivalent to the Dynamic Bit Vector with Indels problem (*sum* being *rank*, *search* being *select*).

A problem related to the Dynamic Sequence with Indels problem is the Dynamic Text Collection problem, defined as follows: Maintain a dynamic collection $\mathcal{C}$ of texts $\{T_1, T_2, \ldots, T_m\}$, where each $T_i \in \{1, 2, \ldots \sigma\}^*$, supporting the following operations:

- $count(\mathcal{C}, P)$ returns the number of times pattern P occurs as a substring in the collection;
- $locate(\mathcal{C}, P)$ returns the occurrence positions of P in the collection;
- $substring(\mathcal{C}, j, l, r)$ returns $T_j[l, r]$;
- $j = insert(\mathcal{C}, T)$ inserts text T into the collection, returning a handle j to it (that is, from now on $T = T_j$); and
- $delete(\mathcal{C}, j)$ deletes text T_j from the collection.

3 Previous Results

Our new solutions build on top of various previous results. We explain part of these previous results in detail, in order to present our contribution in self-contained manner.

3.1 Static Entropy-Bound Structures for Bit Vectors

Raman et al. [32] proposed a data structure to solve *rank* and *select* queries in constant time over a static bit vector $A = a_1 \ldots a_n$ with binary zero-order entropy H_0. The structure requires $nH_0 + o(n)$ bits.

The idea is to split A into *superblocks* $S_1 \ldots S_{n/s}$ of $s = \log^2 n$ bits. Each superblock S_i is in turn divided into $2 \log n$ blocks $B_i(j)$, of $t = (\log n)/2$ bits each, thus $1 \leq j \leq s/t$. Each such block B_i is said to belong to *class* c if it has exactly c bits set, for $0 \leq c \leq t$. For each class c, a universal table G_c of $\binom{t}{c}$ entries is precomputed. Each entry corresponds to a possible block belonging to class c, and it stores all the local *rank* answers for that block. Overall all the G_c tables add up $2^t = \sqrt{n}$ entries, and $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits.

Each block $B_i(j)$ of the sequence is represented by a pair $D_i(j) = (c, o)$, where c is its class and o is the index of its corresponding entry in table G_c. A block of class c thus requires $\log(c+1) + \log\binom{t}{c}$ bits. The first term is $O(\log\log n)$, whereas all the second terms add up $nH_0 + O(n/\log n)$ bits. To see this, note that $\log\binom{t}{c_1} + \log\binom{t}{c_2} \leq \log\binom{2t}{c_1+c_2}$, and that $nH_0 \geq \log\binom{t(n/t)}{c_1+\ldots+c_{n/t}}$. The pairs $D_i(j)$ are of variable length and are all concatenated into a single sequence.

Each superblock S_i stores a pointer P_i to its first block description in the sequence (that is, the first bit of $D_i(1)$) and the *rank* value at the beginning of the superblock, $R_i = rank(A, (i-1)s)$. P and R add up $O(n/\log n)$ bits. In addition, S_i contains s/t numbers $L_i(j)$, giving the initial position of each of its blocks in the sequence, relative to the beginning of the superblock. That is, $L_i(j)$ is the position of $D_i(j)$ minus P_i. Similarly, S_i stores s/t numbers $Q_i(j)$ giving the *rank* value at the beginning of each of its blocks, relative to the beginning of the superblock. That is, $Q_i(j) = rank(A, (i-1)s + (j-1)t) - R_i$. As those relative values are $O(\log n)$, sequences L and Q require $O(n \log\log n/\log n)$ bits.

To solve $rank(A, p)$, we compute the corresponding superblock $i = 1 + \lfloor p/s \rfloor$ and block $j = 1 + \lfloor (p-(i-1)s)/t \rfloor$. Then we add the *rank* value of the corresponding superblock, R_i, the relative *rank* value of the corresponding block, $Q_i(j)$, and complete the computation by fetching the description (c, o) of the block where p belongs (from bit position $P_i + L_i(j)$) and performing a (precomputed) local *rank* query in the universal table, $rank(G_c(o), p-(i-1)s-(j-1)t)$.

The overall space requirement is $nH_0 + O(n \log\log n/\log n)$ bits, and *rank* is solved in constant time. We do not cover *select* because it is not necessary to follow this paper.

The scheme extends to sequences over small alphabets as well [15]. Let $B = a_1 \ldots a_t$ be the symbols in a block, and call n_a the number of occurences of symbol $a \in [1, q]$ in B. We call $(n_1, \ldots, n_q)$ the *class* of B. Thus, in our (c, o) pairs, c will be a number identifying the class of B and o an index within the class. A simple upper bound to the number of classes is $(t+1)^q$ (as a class is a tuple of q numbers in $[0, t]$, although they have to add up t). Thus $O(q \log\log n)$ bits suffice for c (a second bound on the number of classes is q^t as there cannot be more classes than different sequences). Just as in the binary case, the sum of the sizes of all o fields adds up $nH_0(A) + O(n/\log_q n)$ [15].

We now extend the result of the previous section to larger alphabets. The idea is to build a wavelet tree [17] over sequences represented using *rank* and *select* structures for small alphabets.

A binary wavelet tree is a balanced binary tree whose leaves represent the symbols in the alphabet. The root is associated with the whole sequence $A = a_1 \cdots a_n$, its left child with the subsequence of A obtained by concatenating all positions i having $a_i < \sigma/2$, and its right child with the complementary subsequence (symbols $a_i \geq \sigma/2$). This subdivision is continued recursively, until each leaf contains a repeat of one symbol. The sequence at each node is represented by a bit vector that tells which positions (those marked with 0) go to the left child, and which (marked with 1) go to the right child. It is easy to see that the bit vectors alone are enough to determine the original sequence: To recover a_i, start at the root and go left or right depending on the bit vector value B_i at the root. When going to the left child, replace $i \leftarrow rank_0(B, i)$, and similarly $i \leftarrow rank_1(B, i)$ when going right. When arriving at the leaf of character c it must hold that the original a_i is c. This requires $O(\log \sigma)$ *rank* operations over bit vectors.

It also turns out that operations *rank* and *select* on the original sequence can be carried out via $O(\log \sigma)$ operations of the same type on the bit vectors of the wavelet tree [17]. For example, to solve $rank_c(A, i)$, start at the root and go to the left child if $c < \sigma/2$ and to the right child otherwise. When going down, update i as in the previous paragraph. When arriving at the leaf of c, the current i value is the answer. For $select_c(A, j)$, the algorithm starts at the leaf of c and goes upwards until the root, updating $j \leftarrow select_b(B, j)$ with $b = 0$ or 1 depending on whether we descend from the parent (owning vector B) from the left or right child.

A multiary wavelet tree, of arity q, is used in [15]. In this case the sequence of each wavelet tree node ranges over alphabet $[1, q]$, and symbol rank/select queries are needed over those sequences. One needs $\log_q \sigma$ operations on those sequences to perform the corresponding operation on the original sequence.

Either for binary or general wavelet trees, it can be shown that the H_0 entropies in the representations of the sequences at each level add up to $nH_0(A)$ bits [17,15]. However, as we have $O(\sigma)$ bit vectors, the sublinear terms sum up to $o(\sigma n)$. The space occupancy of the sublinear structures can be improved to $o(n \log \sigma)$ by concatenating all the bit vectors of the same level into a single sequence, and handling only $O(\log \sigma)$ such sequences[6]. It is straightforward to do *rank*, as well as obtaining symbol a_i, without any extra information [15].

[6] Note that $o(n \log \sigma)$ is sublinear in the size of A measured in bits.

For *select* one still needs pointers to the parent of each node and its starting position in the sequence of its level. This information adds up $O(\sigma \log n)$ bits of space.

If we now represent the concatenated bit vectors of the binary wavelet tree by the *rank* structures explained in the previous section, we obtain a structure requiring $nH_0(A) + O(n \log\log n / \log_\sigma n) = nH_0(A) + o(n \log \sigma)$ bits, solving *rank* in $O(\log \sigma)$ time. Within the same bounds one can solve *select* as well [32,17].

One can also use multiary wavelet trees and represent the sequences with alphabet size q using the techniques for small alphabets (see the end of previous section). With a suitable value for q, one obtains a structure requiring the same $nH_0(A) + o(n \log \sigma)$ bits of space, but answering *rank* and *select* in constant time when $\sigma = O(\text{polylog}(n))$, and $O(\lceil \log \sigma / \log\log n \rceil)$ time in general [15].

3.3 Dynamic Structures for Bit Vectors

Hon et al. [22] show how to handle a bit vector $A = a_1 \dots a_n$ in $n + o(n)$ bits of space, so that *rank* and *select* can be solved in $O(\log_b n)$ time, while insertions and deletions to the sequence can be handled in $O(b)$ time, for any parameter $b \geq \log n / \log\log n$. Hence, they provide a solution to the Dynamic Bit Vector with Indels problem. Their main structure is a weight-balanced B-tree (WBB) [9,31].

Our goal is to obtain $nH_0 + o(n)$ bits of space and $O(\log n)$ time for all the operations above. We build over a simplified version of their structure, which uses standard balanced trees and achieves $O(\log n)$ time and $O(n)$ bits of space [7]. This is described below.

Consider a balanced binary tree on A whose leftmost leaf contains bits $a_1 a_2 \cdots a_{\log n}$, second left-most leaf contains bits $a_{\log n+1} a_{\log n+2} \cdots a_{2\log n}$, and so on. Each node v contains counters $p(v)$ and $r(v)$ telling the number of positions stored and the number of bits set in the subtree rooted at v, respectively. Note that this tree, with all its $\log n$-size pointers and counters, requires $O(n)$ bits.

To perform $rank(A, i)$, we enter the tree to find the leaf containing position i. We start with $rank \leftarrow 0$. If $p(left(v)) \geq i$ we enter the left subtree, otherwise we enter the right subtree with $i \leftarrow i - p(left(v))$ and $rank \leftarrow rank + r(left(v))$. In $O(\log n)$ time we reach the desired leaf and complete the rank query in $O(\log n)$ time by scanning the bit sequence corresponding to that node. For *select* we proceed similarly, except that the roles of $p()$ and $r()$ are reversed. For $select_0$ the computation is analogous.

Insertions and deletions are handled by entering to the correct leaf as in *rank*, and replacing its bit sequence with the new content. Then the $p(v)$ and $r(v)$ counters in the path from the leaf to the root are changed accordingly. When a leaf is updated to contain $2 \log n$ bits, it is split into two leaves, each containing $\log n$ bits. When a leaf is updated to contain $(\log n)/2$ bits, it is merged with its sibling. If this merging produces a leaf with more than $2 \log n - 1$ bits, this leaf is again split into two equal-size halves. After splitting and merging, the tree needs to be rebalanced and the counters updated in the nodes on the way to the root.

3.4 Dynamic Entropy-Bound Structures for Bit Vectors

Blandford and Blelloch [4] design a general scheme to convert a space-demanding data structure into one that requires $O(nH_0)$ bits of space. The data structures considered can solve a subset of a wide range of problems related to ordered sets, and the main idea is to represent such sets using *gap encoding* (see next). In particular, if one applies the idea to the structure described above for solving bit vector *rank*, *select*, *insert*, and *delete* in $O(\log n)$ time, the only difference is that bit vectors are represented in compressed form in the leaves of the binary tree.

We note that gap encoding has also been used to achieve zero-order entropy in static schemes. In [19] they explore the idea of inserting some information into the encoding so as to permit solving *rank* and *select* queries in logarithmic time via binary searches. In [20] they improve this result and reach time $o((\log \log n)^2)$, close to the lower bound on the predecessor problem when the space depends on the number of bits set and only logarithmically on the total number of bits. In [25] they achieve constant time on gap encoding, yet they have a higher $o(n)$-type dependence on the total number of bits.

3.4.1 Gap Encoding

Let $A = 0^{g_0}10^{g_1}1 \ldots 0^{g_{\ell-1}}10^{g_\ell}$, where 0^{g_i} represents a sequence of g_i 0-bits (called a gap). Gap encoding represents A as $\delta(g_0)\delta(g_1) \ldots \delta(g_{\ell-1})\delta(g_\ell)$, where $\delta(x)$ is an encoding for the nonnegative integer x. This encoding must satisfy two properties: (*i*) $|\delta(x)| = \log x + o(\log x)$; (*ii*) we can univocally distinguish x and D from $\delta(x)D$, being D any bit sequence.

A well-known encoding satisfying the above properties is Elias' δ [10,3]. To represent x, let $l = \lceil \log(x+1) \rceil$ be the number of bits necessary to encode x, and let $ll = \lceil \log(l+1) \rceil$ be the number of bits necessary to code l. Then $\delta(x)$ is formed by three parts: (*a*) ll 0-bits followed by a 1-bit, (*b*) the $ll - 1$ least significant bits of the binary representation of l (this part is empty if $l < 2$), and (c) the $l-1$ least significant bits of the binary representation of x (this part is empty if $x<2$). For example, for $x=0$, we have $l=0$ and $ll=0$, thus $\delta(0)=1$; for $x=1$, $l=1$ and $ll=1$, thus $\delta(1)=01$; for $x=2$, $l=2$ and $ll=2$, thus $\delta(2)=001\ 0\ 0$; whereas $\delta(3)=001\ 0\ 1$; $\delta(4)=001\ 1\ 00$ since $l=3$ and $ll=2$; and so on.

It is clear that $|\delta(x)|=\log x+2\log\log x+O(1)=\log x+o(\log x)$. The total length of this representation of A is therefore

$$\begin{aligned}\sum_{i=0}^{\ell}\log g_i+o(\log g_i) &\leq \sum_{i=0}^{\ell}\log\frac{n-\ell}{\ell+1}+o(\log\frac{n-\ell}{\ell+1})\\ &\leq (\ell+1)\log\frac{n}{\ell}+(\ell+1)o(\log\frac{n}{\ell}),\end{aligned}$$

where we have used the fact that $\sum_{i=0}^{\ell}g_i=n-\ell$, and thus the summation of those convex functions achieves its maximum value when all $g_i=\frac{n-\ell}{\ell+1}$. As the binary entropy of A is $H_0=\frac{\ell}{n}\log\frac{n}{\ell}+\frac{n-\ell}{n}\log\frac{n}{n-\ell}$, the first term of the result is $\ell\log\frac{n}{\ell}+O(\log n)\leq nH_0+O(\log n)$. The second term is $O(\ell)$ if $\ell=\Theta(n)$, and $o(nH_0+\log n)$ otherwise. Therefore, the total size of the gap representation is $n'=nH_0(1+o(1))+O(\ell+\log n)$ bits. For Elias' representation, this is more precisely $n'=\ell\log\frac{n}{\ell}+O(\ell\log\log\frac{n}{\ell}+\log n)$.

3.4.2 A Dynamic Structure based on Gap Encoding

Consider the balanced binary search tree of Section 3.3 built on the gap encoded bit vector A: The encoded bit vector $\delta(g_0)\delta(g_1)\ldots\delta(g_{\ell-1})\delta(g_\ell)$ of length n' is partitioned into blocks of approximately $\log n$ bits, each block containing as many full $\delta(g_i)$ codes as can be accommodated into $\log n$ bits. These blocks form the leaves of the binary search tree. To answer *rank* and *select* queries one can proceed just like in the uncompressed case, except that the final scanning in the leaves requires decoding the gap encoding. The time complexity remains $O(\log n)$.

To support *insert* and *delete* on this gap-encoded sequence one can proceed as in the uncompressed case, splitting and merging leaves when necessary. To insert a bit a preceding position i inside a block, we sequentially look for the gap where a should be inserted. Say that a must be inserted at relative position i' within $0^{g_k}1$, $1\leq i'\leq g_k+1$. If $a=1$ we must replace $\delta(g_k)$ by $\delta(i'-1)\delta(g_k-i'+1)$. Otherwise, if $a=0$ we must replace $\delta(g_k)$ by $\delta(g_k+1)$. All the δ-codes that follow must be shifted to make room for the new code. The replacement and shifting can be easily done in $O(\log n)$ time if there is enough empty space left in the block. If not, the block needs to be split into two. Notice that on a single insert the space needed can at most double. Deletions are handled analogously.

Space is improved from $O(n)$ to $O(nH_0)$ bits. The gap encoding itself takes essentially nH_0 bits, the unused empty space in the leaves occupies in the worst case other nH_0 bits, as blocks can be half-full[7], and the tree pointers occupy $(nH_0/\log n) \times O(\log n) = O(nH_0)$ bits.

3.5 Static Full-Text Self-Indexes

Many static full-text self-indexes are based on representing the Burrows-Wheeler transform [5] of a text using wavelet trees to support efficient substring searches. We will later consider dynamic wavelet trees to solve the Dynamic Text Collection problem, hence we introduce the basic concepts here. We follow closely the description given in [24].

3.5.1 The Burrows-Wheeler Transform

The *Burrows-Wheeler transform (BWT)* [5] of a text T produces a permutation of T, denoted by T^{bwt}. We assume that T is terminated by an endmarker "\$" $\in \Sigma$, smaller than other symbols. String T^{bwt} is the result of the following transformation: (1) Form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string T, call F its first column and L its last column; (2) sort the rows of $\mathcal{M}$ in lexicographic order; (3) the transformed text is $T^{bwt} = L$.

The BWT is reversible, that is, given T^{bwt} we can obtain T. Note the following properties [5]:

a. Given the i-th row of $\mathcal{M}$, its last character $L[i]$ precedes its first character $F[i]$ in the original text T, that is, $T = \ldots L[i]F[i]\ldots$.
b. Let $L[i] = c$ and let r_i be the number of occurrences of c in $L[1,i]$. Let $\mathcal{M}[j]$ be the r_i-th row of $\mathcal{M}$ starting with c. Then the character corresponding to $L[i]$ in the first column F is located at $F[j]$ (this is called the *LF mapping*: $LF(i) = j$). This is because the occurrences of character c are sorted both in F and L using the same criterion: by the text following the occurrences.

The BWT can then be reversed as follows:

(1) Compute the array $C[1,\sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text T. Notice that $C[c]+1$ is the position of the first occurrence of c in F (if any).
(2) Define the *LF mapping* as follows: $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$.
(3) Reconstruct T backwards as follows: set $s = 1$ (since $\mathcal{M}[1] = \$t_1t_2\ldots t_{n-1}$) and, for each $i \in n-1, \ldots, 1$ do $T[i] \leftarrow L[s]$ and $s \leftarrow LF[s]$. Finally put

[7] As described blocks could be 25% full, but it is easy to force them to be half-full.

the endmarker $T[n] = \$$.

The BWT transform by itself does not compress T, it just permutes its characters. However, this permutation is more compressible than the original T. Actually, it is not hard to compress T^{bwt} to $O(nH_h + \sigma^h)$ bits, for any $h \geq 0$ [27]. The idea is as follows (we will reuse it in Section 7.3): Partition L into minimum number of pieces $L^1 L^2 \cdots L^\ell$ such that the symbols inside each piece $L^k = L[i_k, j_k]$ have the same *h-context*. Note that the h-context of a symbol $L[i]$ is $\mathcal{M}[i][1,h]$. By the definition of h-th order entropy, it follows that $|L^1|H_0(L^1) + |L^2|H_0(L^2) + \cdots + |L^\ell|H_0(L^\ell) = nH_h$. That is, if one is able to compress each piece upto its zero-order entropy, then the end result is h-th order entropy. Using, say, arithmetic coding on each piece, one achieves $nH_h + \sigma^{h+1}\log n$ bits encoding of T. The latter term comes from the encoding of the symbol frequences in each piece separately.

3.5.2 Suffix Arrays

The *suffix array* $\mathcal{A}[1,n]$ of text T is an array of pointers to all the suffixes of T in lexicographic order. Since T is terminated by the endmarker "\$", all lexicographic comparisons are well defined. The i-th entry of $\mathcal{A}$ points to text suffix $T[\mathcal{A}[i],n] = t_{\mathcal{A}[i]}t_{\mathcal{A}[i]+1}\ldots t_n$, and it holds $T[\mathcal{A}[i],n] < T[\mathcal{A}[i+1],n]$ in lexicographic order.

Given the suffix array, the occurrences of the pattern $P = p_1p_2\ldots p_m$ can be counted in $O(m\log n)$ time. The occurrences form an interval $\mathcal{A}[sp,ep]$ such that suffixes $t_{\mathcal{A}[i]}t_{\mathcal{A}[i]+1}\ldots t_n$, for all $sp \leq i \leq ep$, contain the pattern P as a prefix. This interval can be searched for using two binary searches in time $O(m\log n)$. Once the interval is obtained, a locating query is solved simply by listing all its pointers in constant time each.

We note that the suffix array $\mathcal{A}$ is essentially the matrix $\mathcal{M}$ of the BWT (Section 3.5.1), as sorting the cyclic shifts of T is the same as sorting its suffixes given the endmarker "\$": $\mathcal{A}[i] = j$ if and only if the i-th row of $\mathcal{M}$ contains the string $t_jt_{j+1}\ldots t_{n-1}\$t_1\ldots t_{j-1}$.

3.5.3 Backward Search

The FM-index [12] is a self-index based on the Burrows-Wheeler transform. It solves counting queries by finding the interval of $\mathcal{A}$ that contains the occurrences of pattern P. The FM-index uses the array C and function $rank_c(L,i)$ of the LF mapping to perform backward search on the pattern. Fig. 1 shows the counting algorithm. Using the properties of the BWT, it is easy to see that the algorithm maintains the following invariant [12]: At the i-th phase, variables sp and ep point, respectively, to the first and last row of $\mathcal{M}$ prefixed by $P[i, m]$. The correctness of the algorithm follows from this observation. Note that P is processed backwards, from p_m to p_1.

```
**Algorithm** FMCount($P[1,m]$,$L[1,n]$)
(1)  $i \leftarrow m$;
(2)  $sp \leftarrow 1$; $ep \leftarrow n$;
(3)  while ($sp \leq ep$) and ($i \geq 1$) do
(4)      $c \leftarrow P[i]$;
(5)      $sp \leftarrow C[c] + rank_c(L, sp-1)+1$;
(6)      $ep \leftarrow C[c] + rank_c(L, ep)$;
(7)      $i \leftarrow i - 1$;
(8)  if ($ep < sp$) then return 0 else return $ep - sp + 1$;
```

Fig. 1. FM-index algorithm for counting the number of occurrences of $P[1, m]$ in $T[1, n]$.

Note that array C can be explicitly stored in little space, and for $rank_c(L, i)$ we can directly use the wavelet tree as explained in Section 3.2. The space usage is $nH_0 + o(n \log \sigma)$ bits and the m steps of backward search take overall $O(m \log \sigma)$ time [24].

Let us now consider how to locate the positions in $\mathcal{A}[sp, ep]$. The idea is that T is sampled at regular intervals, so that we explicitly store the positions in $\mathcal{A}$ pointing to the sampled positions in T (note that the sampling is not regular in $\mathcal{A}$). Hence, using the LF mapping, we move backward in T until finding a position that is known in $\mathcal{A}$. Then it is easy to infer our original text position. Fig. 2 shows the pseudocode.

```
**Algorithm** FMlocate($i$,$L[1,n]$)
(1)  $i' \leftarrow i$, $t \leftarrow 0$;
(2)  while $\mathcal{A}[i']$ is not known do
(3)      $i' \leftarrow LF(i') = C[L[i']] + rank_{L[i']}(L, i')$;
(4)      $t \leftarrow t + 1$;
(5)  return "text position is $\mathcal{A}[i'] + t$".
```

Fig. 2. FM-index algorithm for locating the occurrence $\mathcal{A}[i]$ in T.

We note that, in addition to C and $rank$, we need access to characters $L[i']$ as well. These can be found using the same wavelet tree built for $rank$. Finally, if we sample one out of $\log^{1+\varepsilon} n$ positions in T, for any constant $\varepsilon > 0$, and use $\log n$ bits to represent each corresponding known $\mathcal{A}$ value, we require $O(n/\log^{\varepsilon} n) = o(n)$ additional bits of space and can locate each occurrence of P in $O(\log \sigma \ \log^{1+\varepsilon} n)$ time.

Finally, let us consider displaying text contexts. To retrieve $T[l, r]$, we start at the position in $\mathcal{A}$ that points to the lowest marked text position following

r. This position in $\mathcal{A}$ is known from the sampling. From there, we perform $O(\log^{1+\varepsilon} n)$ steps, using the LF mapping, until reaching r. Then we perform $\ell = r - l$ additional LF steps to collect the desired text characters. The resulting complexity is $O(\log \sigma \ (\ell + \log^{1+\varepsilon} n))$.

All the $O(\log \sigma)$ terms in the time complexities can be made $O(\lceil \log \sigma / \log \log n \rceil)$ (which is constant if $\sigma = O(\mathrm{polylog}(n))$), by using multiary wavelet trees.

3.6 Dynamic Full-Text Self-Indexes

Chan, Hon, and Lam [7] show how to use a solution to DYNAMIC SEQUENCE WITH INDELS problem to obtain a solution to the DYNAMIC TEXT COLLECTION problem. One of the ideas is to simulate the above backward search algorithm: They use $A = bwt(\mathcal{C})$ on a text collection $\mathcal{C}$ (seen as a concatenation of texts over $[0, \sigma]$, where alphabet symbol 0 is reserved for separating contiguous texts and somehow plays the role of "\$").

They show that one can dynamically maintain a collection of texts, by keeping a data structure supporting *rank*, *insert* and *delete* on A, in addition to a dynamic version of table C of Section 3.5. Adding new text T (preceded by 0) triggers $|T|$ insertions to A: The insertion points can be found in $O(|T| g(|\mathcal{C}|))$ time, where $g(n)$ is the time to access C and to answer *rank* on a collection of length n. The process consists of inserting the suffixes of $T = t_1 \ldots t_{|T|}$ one by one in backward fashion. The initial insertion point, for $t_{|T|}$, can be at $C[t_{|T|}]+1$ (note that, as there might be repeated suffixes, the position is not unique; this does not affect the correctness of the BWT-based scheme), thus we have to *insert* symbol $t_{|T|-1}$ (preceding $t_{|T|}$) at $a_{C[t_{|T|}]+1}$ (that is, position $C[t_{|T|}] + 1$ in A). In general, once we have inserted t_i at position j in A corresponding to $T_{i+1,|T|}$, we use the LF mapping to find the next insertion point (that is, the A position corresponding to $T_{i,|T|}$). This is done with an access to C and a *rank* operation. At the end, the symbol 0 preceding t_1 is inserted and we move to insert $t_{|T|}$ at the position in A corresponding to suffix $0 \cdot T$ (T is seen as a circular string).

Deleting a text T triggers $|T|$ deletions from A: The deletion points can be found in $O(|T| g(|\mathcal{C}|) \log |\mathcal{C}|)$ time. In principle one would mimic the very same insertion process, this time doing deletions. The problem is that the operation receives the text T to delete but cannot know which of the 0's of A correspond to it. Thus they search backwards for T in A, and assuming it is unique, they find the position in A corresponding to t_1. Then they have to find t_2 and so on using the inverse of LF, which can be computed in $O(\log |\mathcal{C}|)$ time.

If we call $n = |\mathcal{C}|$, their original structure (called `COUNT`) takes $O(n\sigma)$ bits of space, and supports *rank*, *insert*, and *delete* in $g(n) = O(\log n)$ time. They give another structure that uses $O(n \log \sigma)$ bits of space in exchange of an $O(\log n)$ penalty factor in all time complexities. Using both structures they still have $O(\sigma n)$ bits and can handle both insertion and deletion in $O(|T| \log n)$ time. Searches for P cost $O(|P| \log n)$ time.

Their index is extended to support locating of occurrences using a structure called `MARK`, which samples one out of $\log n$ collection positions and stores the position in A that corresponds to the sampled collection position. This requires $O(\log n)$ bits per sample, for a total of $O(n)$ further bits. To locate an occurrence at $A[i]$, they look for it in `MARK`. If it is not present, they use the LF mapping repeatedly (which traverses $\mathcal{C}$ backwards) until a marked position is found. Then the original value $A[i]$ is the sampled one plus the number of LF steps performed. This takes overall $O(\log^2 n)$ time.

Finally, to recover a substring of some text T in the collection, the user must somehow know the lexicographic position of T within the other texts (this is a strong assumption as it would solve the problem of the extra $O(\log n)$ factor for deletions!). The lexicographically j-th text has its character $t_{|T|}$ at a_j [8]. By locating $t_{|T|}$ in $\mathcal{C}$ they can translate a relative position within a text into an absolute position in $\mathcal{C}$. Then they take the next sampled position in $\mathcal{C}$ and use the LF mapping to discover the desired characters backward from there. This takes $O(\log n(\ell + \log n))$ time, where ℓ is the length of the text piece to display.

4 Dynamic Succinct Structures for Bit Vectors and Partial Sums

In this section we design a data structure to represent a bit sequence $A = a_1 \dots a_n$ using $n + o(n)$ bits of space and performing operations *rank*, *select*, *insert* and *delete* all in $O(\log n)$ time. This already improves previous results [22], and serves as a basis for the entropy-bound structures developed in the next sections.

4.1 High-Level Hierarchy

Section 3.3 shows how to obtain the desired time complexities using $O(n)$ bits of space. To achieve $n + o(n)$ bits, we use the same tree organization, except that it is built on $\omega(\log n)$-size leaves. Thus the tree has $o(n/ \log n)$ internal nodes which, with all their pointers, require only $o(n)$ bits of space. The two

[8] To see this, note that $t_{|T|}$ in $a_j = L[j]$ of the BWT corresponds to $F[j] = 0$. The 0's of all the texts are in the beginning of F, sorted lexicographically by the texts.

problems to solve are (i) we cannot process the leaves bitwise in $O(\log n)$ time; (ii) we cannot store $(1+\epsilon)s$ bits for leaves and use only s bits from those. This is essentially the technique used in Section 3.3 to ensure that bit insertions/deletions are handled with $O(1)$ tree node updates.

We divide A into blocks and superblocks, as in Section 3.1. Each superblock S will maintain $s = f(n)\log n$ bits (for some $f(n) = O(\log n)$ to be determined later), and will be stored in a tree leaf, without any extra bits of space. Each superblock will hold exactly $2f(n)$ whole blocks of $t = (\log n)/2$ bits each. All the s bits of the superblock will thus be stored contiguously in plain form, without any extra structure in principle (later we add a few pointers per leaf). From now on we will use the term “leaf” and “superblock” interchangeably.

4.2 Queries Inside a Superblock

A $rank(S, i)$ query inside a superblock is handled in $O(\log n)$ time by using a universal table R, which receives a t-bit sequence and gives the total number of 1-bits in it. Thus we traverse the superblock in a blockwise manner, adding 1-bits until we reach the block that contains position i. Within that block we count the 1-bits one by one until reaching the i-th position. The whole process takes $O(f(n) + t) = O(\log n)$ time. More formally, if $a_1 \dots a_s$ are the bits of the leaf, then $b = 1 + \lfloor i/t \rfloor$ is the block i belongs to, and we compute

$$\sum_{1\le q<b} R[a_{(q-1)t+1}\dots a_{qt}] + \sum_{(b-1)t+1\le q\le i} a_q.$$

A $select(S, j)$ query is solved similarly: We add up successive R values until we exceed j at some block b. Then we rescan block b bitwise until reading the j'-th 1-bit, where $j' = j - \sum_{1\le q<b} R[a_{(q-1)t+1}\dots a_{qt}]$. This also takes $O(f(n)+\log n)$ time. We remark that table R is universal and does not depend on the sequence A nor on the particular leaf; just on t. Moreover, table R is very small, requiring just $O(\sqrt{n}\log\log n)$ bits.

4.3 Updates Inside a Superblock

To insert a bit a at position i of S, we simply shift to the right the bit vector $a_i \dots a_s$, to make room for $a_i = a$. This shift can be done by chunks of $\Theta(\log n)$ bits under a RAM model that permits bit shifts, or multiplication and division (as, say, dividing by two is equivalent to shifting the bits to the right). Otherwise, it is easy to build a small universal table to perform the shifts by chunks of t bits. Thus the new bit is accomodated within the leaf in $O(f(n))$ time.

We note that, after the shift, former bit a_s overflows to the next leaf. We now insert a_s at the beginning of the next leaf, which causes a new overflow, and so on. This brings two problems: (*i*) doing the propagation efficiently within a leaf, and (*ii*) limiting the propagation to a reasonable number of leaves.

To achieve efficiency within each leaf, we redefine them as circular arrays of bits. A pointer telling the position of the first bit in the leaf is stored within the leaf, and it requires $O(\log\log n)$ bits. This amounts to $O(n \log\log n/ (f(n)\log n)) = o(n/f(n))$ wasted space. The advantage is that, if leaves are circular arrays, then inserting the overflown bit at their beginning and taking out their last bit that overflows to the next leaf is easily done in constant time.

To limit the propagation of overflows across leaves, every $f(n)$ leaves we permit the formation of a *partial* leaf, which reserves $f(n)\log n$ bits but might be partially full. Those partial leaves amount to $n/f(n)$ wasted bits overall. Partial leaves are not circular, so the pointer of the previous paragraph can be reused to store their current number of bits. An additional bit, stored for each leaf, tells whether it is full or partial.

Partial leaves ensure that we never traverse more than $f(n)$ leaves in the overflow propagation process. Thus the overall insertion time (considering just the work within leaves) is $O(f(n))$ in the leaf that receives the insertion, plus $O(1)$ per leaf to propagate the overflow across $O(f(n))$ full leaves, plus $O(f(n))$ to insert the overflown bit in the partial leaf (as it is not circular, it is necessary to shift its values). This adds up $O(f(n))$.

To ensure the desired density of partial leaves, we first check whether there is a partial leaf among the next $2f(n)$ leaves. If there is one, we carry out the propagation up to it. Otherwise, we propagate $f(n)$ leaves and create a new empty partial leaf. In both cases we work over $O(f(n))$ leaves, and guarantee that every partial leaf is $f(n)$ leaves away from any other. We note that partial leaves may end up overflowing, at which point they are not considered partial anymore.

For deletions we proceed similarly, bringing back the first bit of the next leaf and propagating the underflow. If there is a partial leaf within the next $2f(n)$ leaves, we propagate the underflow until there. Otherwise, we propagate the underflow for $f(n)$ leaves, and declare the $f(n)$-th leaf partial. A partial leaf that gets empty should be removed. Thus deletions are also handled in $O(f(n))$ time.

4.4 Operations in the Tree

Section 3.3 shows how to perform *rank* and *select* up to the leaves in $O(\log n)$ time. We have shown in Section 4.2 how to manage inside the leaves in time $O(f(n) + \log n)$ time. The wasted space is $O(n/f(n))$ within the leaves, and also $O(n/f(n))$ for the internal tree nodes. By choosing $f(n) = \log n$ we obtain $O(\log n)$ time for both query operations, and $O(n/\log n) = o(n)$ wasted bits of space. This completes the solution for those queries.

The update time within leaves is $O(\log n + f(n)) = O(\log n)$ according to Section 4.3. Let us now consider the tree adjustments required upon updates.

Inserting and deleting bits requires rewriting the $p()$ and $r()$ values from the affected superblock(s) through the root. Creation and deletion of leaves and internal tree nodes is easily handled together with the maintenance of $r()$ and $p()$. We note, however, that although each bit insertion/deletion can produce at most one tree leaf insertion/deletion, it can affect the bits of $O(f(n))$ leaves, as well as all their $r()$ and $p()$ values upwards the root. Yet, we note that those $O(f(n))$ leaves are contiguous in the tree and therefore their total number of ancestors do not exceed $O(f(n)) + O(\log n) = O(\log n)$. It is not hard to organize those updates so as to work $O(\log n)$ time overall. Thus we achieve $O(\log n)$ overall time complexity for insertions and deletions as well.

4.5 Changing $\log n$

Our result so far assumes that $\log n$ stays constant during the operations. This value fixes the superblock/block hierarchy and the global preprocessed tables. This assumption can be removed in two ways: (1) performing a global rebuild whenever $\log n$ changes; (2) maintaining partial structures ready for values $(\log n) - 1$, $\log n$, and $(\log n) + 1$ (which we call the *previous*, *current*, and *next*).

Approach (1) is easy to implement. We can rebuild all structures in $O(n)$ time when necessary to accommodate the new value of $\log n$. Amortized over all insertions and deletions, this costs only $O(1)$ time per operation.

Approach (2) is more complex but is inspired on a standard mechanism to convert amortized complexity into worst-case complexity. The idea is to split the current elements among the *previous*, *current*, and *next* structures, so that the first elements are in *previous*, the last are in *next*, and *current* holds the middle elements. It is trivial to run *rank* and *select* queries on this split structure. When n is zero or a power of 2, all the elements are in *current*, and the other two are empty. Upon an insertion, the size of *next* must grow by

2 unless it is already full, and *previous* must shrink by 1 unless it is already empty; a deletion must cause the opposite effect; and *current* acts as a variable-size buffer.

To achieve this, let us denote $x \rightarrow y$ or $x \leftarrow y$ the movement of one element among structures, for $x, y \in \{p, c, n\}$, e.g. $p \leftarrow c$ means moving the first element of *current* to *previous*. If the source structure is empty, the movement is just ignored. Then, we insert (delete) in the proper structure and then, depending on where the insertion (deletion) point lies, we move elements as follows:

- *previous*: $p \rightarrow c$, $p \rightarrow c$, $c \rightarrow n$, $c \rightarrow n$ ($c \leftarrow n$, $c \leftarrow n$, $p \leftarrow c$, $p \leftarrow c$).
- *current*: $p \rightarrow c$, $c \rightarrow n$, $c \rightarrow n$ ($c \leftarrow n$, $c \leftarrow n$, $p \leftarrow c$).
- *next*: $p \rightarrow c$, $c \rightarrow n$ ($c \leftarrow n$, $p \leftarrow c$).

It is easy to see that, after n net insertions, *next* will hold all the $2n$ elements, and that after $n/2$ net deletions, *previous* will hold all the $n/2$ remaining elements. This is true even if the insertions and deletions are intermixed. When *next* holds all the elements, it becomes *current* and the new *previous* and *next* structures are empty; similarly when *previous* holds all the elements. At those points, precisely, n is a new power of 2 and $\log n$ has changed its value. The space requirement is still $n + o(n)$, even when the tree pointers of *next* require $\log(2n)$ bits.

Note, however, that we do not have time to build the right R table corresponding to the new *next* or *previous* structures, as we need it immediately. The solution is to maintain tables R ready for 5 values of $\log n$, from $(\log n)-2$ to $(\log n)+2$. Thus, upon a change in $\log n$, we have the new R table we need immediately available. For example, if $\log n$ increases, we have ready the 3 tables for $\log n$, $(\log n)+1$ and $(\log n)+2$. Yet, to maintain the invariant, we should now build the R table for $(\log n)+3$, but we have plenty of time to build it in parallel with the new operations: After $O(\sqrt{n})$ operations we have managed to build the new R table (as there are $O(\sqrt{n})$ cells and each is easily built in $O(\log n)$ time). This is much less than the time necessary for $\log n$ to change again. If, on the other hand, $\log n$ shrinks back before we build the new R table, we can just discard the partial work done.

Finally, note that we have only two types of nodes, so memory can be easily managed in constant time per allocation or deallocation request. We obtain the following result, where we recall that w is the number of bits in the word of the RAM model.

Theorem 1 *The* Dynamic Bit Vector with Indels *problem under RAM model with constraint* $\log n = \Theta(w)$ *can be solved using* $n + O(n/\log n)$ *bits of space supporting the operations rank, select, insert, and delete, in* $O(\log n)$ *worst-case time.*

4.6 Handling the Case $\log n = o(w)$

In Section 4.5 we assumed that $\log n = \Theta(w)$. This permits us using the system memory, with w-bit pointers, and assume that each such pointer takes $O(\log n)$ bits. This is a common assumption in the literature, but we find it too restrictive in the dynamic setting. In this section we extend our result to the case where $\log n = o(w)$. We use different solutions for the internal tree nodes and for the leaves.

Let us start with the tree nodes. Assume our tree has u internal nodes and holds n bits. Each time tree *next* becomes *current* in Section 4.5, we allocate $u' = 2n/(f(2n)\log(2n))$ cells of space for the new *next* tree (those cells require $(\log u')$-bit pointers). This ensures that, when the number of bits we handle reaches $2n$, all the tree will be in *next*, *next* will become *current*, and thus the new *current* memory area will be completely full. Similarly, when *previous* becomes *current*, we allocate space for $u'' = (n/2)/(f(n/2)\log(n/2))$ cells in the new *previous* tree. Note also that, according to the rules to move elements in Section 4.5, *current* never increases its size after it is created. Thus, there is no flaw in creating it with exactly the number of cells to hold its current number of elements. Overall, we are using $O(n/f(n))$ bits of space (more precisely, about $3.5\,n/f(n)$ bits) for all the trees, plus $3w$ bits necessary for the pointers to the three memory areas.

Within each memory area, we must provide a mechanism to manage tree node insertions and deletions in constant time. This is very easy because all the nodes have the same size within each area. An implicit list of free cells, where the list pointers use the same free space they mark, is sufficient for memory management within each area. The above paragraph shows that overflows do not occur within these areas.

We must use a different mechanism for the tree leaves, as they add up n bits and thus we cannot afford allocating $3.5\,n$ bits of space. Let us focus in one individual structure (say, *current*). Each leaf takes $f(n)\log n$ bits of space. A separate memory area will store all those leaves in array form, so an $O(\log n)$-bits index into the array suffices as a pointer from an internal tree node to a leaf. This array must be kept in compact form upon insertions and deletions, which is easily achieved by storing the parent of each leaf in the array area (this permits moving the last leaf to the hole left by a deletion and updating the tree node that points to that leaf).

The problem is how to allocate memory when this array grows. We divide the whole memory for the leaves into $\sqrt{n/w}$ *chunks* of $\sqrt{nw}$ bits each[9]. All chunks will be full except for the last one, costing us $O(\sqrt{wn})$ extra bits. As

[9] This partitioning does not change along the lifetime of the structure, that is, we

they are all of the same size, it is easy to allocate and deallocate chunks from system memory in constant time. Any pointer to a leaf is composed of two parts: the first $\frac{1}{2}\log\frac{n}{w}$ bits indicate the chunk number, and the other $\frac{1}{2}\log(nw)$ bits give the offset within the chunk. To achieve constant-time access from a pointer, we need a global array of the chunk addresses in system memory, requiring $O(w\sqrt{n/w}) = O(\sqrt{nw})$ bits. The overall extra space for the chunk mechanism is $O(\sqrt{nw})$, which is $o(n)$ unless n is very small, $n = O(w)$. In this case $\sqrt{nw} = O(w)$, and as we are already paying $O(w)$ bits for a constant number of system memory pointers, we can spend the same space as if we had w bits in our structure (using a single chunk).

Thus we can reexpress the result of Theorem 1 under our weaker model of computation as follows:

Theorem 2 *The* Dynamic Bit Vector with Indels *problem under RAM model with constraint* $\log n = O(w)$ *can be solved using* $n + O(n/\log n + \sqrt{nw} + w) = n + o(n) + O(w)$ *bits of space, supporting the operations rank, select, insert, and delete, in* $O(\log n)$ *worst-case time.*

We note that the only price we are finally paying is $O(w)$ extra bits of space. This is asymptotically optimal if we assume that at least it is necessary to have a single system pointer to any dynamic structure.

4.7 Searchable Partial Sums with Indels

Assume now that our sequence $A = a_1 \ldots a_n$ is formed by k-bit numbers. Now the length of A in bits is kn. Over this sequence we wish to support the operations *sum*, *search*, *insert*, and *delete*.

We can apply essentially the same technique as for bits (case $k = 1$), by choosing blocks of $t = (\log n)/(2k)$ numbers, so that we can handle $(\log n)/2$ bits in constant time (let us for now assume $k \leq (\log n)/2$). We maintain $s = f(n)\log(n)/k$ numbers (that is, $f(n)\log(n)$ bits) in a superblock, so we can still handle it in $O(f(n))$ time. Now table R receives a sequence of t numbers and delivers their sum. Table R still requires $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits. The process to compute *sum* within a leaf is completely analogous to *rank* in Section 4.2: we add up R values until reaching the block, and then add up $O(\log(n)/k)$ k-bit numbers. Likewise, *search* is analogous to *select*. Both operations are carried out in $O(f(n))$ time within a block.

For $k > (\log n)/2$ (but still $k = \Theta(\log n)$), we do not use table R, but simply add up the numbers in the leaf one by one. This still solves *sum* and *search*

have to interpret n as $2^{\lfloor \log n \rfloor}$ here.

in $O(f(n))$ time.

The updates and the tree work exactly as for bits, using circular arrays. For the tree, the $r()$ values in the nodes are now the sums of the numbers in the node subtree, and all the management is exactly analogous. We thus achieve $O(\log n + f(n))$ time for the operations, and $O(nk/f(n))$ bits of extra space. We can choose $f(n) = \log n$ as before. (Note that the $r()$ values also fit in $O(\log n)$ bits, as we have n numbers of k bits, which add up at most $n2^k$, and $\log(n2^k) = k + \log n = O(\log n)$ bits.)

Changing $\log n$ can be handled smoothly, just as when dealing with bits. Note that, when $\log n$ changes, we may switch from using R to not using it, or vice versa. If we assume that $\log n = \Theta(w)$, we have the following result.

Theorem 3 *The* SEARCHABLE PARTIAL SUMS WITH INDELS *problem under RAM model with constraint* $\log n = \Theta(w)$ *with k-bit numbers,* $k = O(\log n)$*, can be solved using* $kn + o(kn)$ *bits of space, supporting the operations sum, search, insert, and delete, in* $O(\log n)$ *worst-case time.*

The best current result permitting indels [22] works only for $k = O(1)$, and also requires $kn + o(kn)$ bits of space. It needs $O(\log_b n)$ time for *sum* and *search*, and $O(b)$ *amortized* time for *insert* and *delete*, for $b = \Omega(\text{polylog}(n))$. For example they can achieve $O(\log n/ \log\log n)$ time for the queries, yet the updates seem to require at least amortized time $\Theta((\log n/ \log\log n)^2)$. Our result achieves slightly worse complexity for queries and better complexity for updates. In addition, all of our complexities are worst-case and we can handle any $k = O(\log n)$ value.

A final point is to consider the case $\log n = O(w)$. We use the same memory arrangement of Section 4.6 to achieve the same time complexities and only $O(w)$ extra space (the same analysis applies verbatim with nk instead of n bits). Yet, we could like to handle k values as large as $k = \Theta(w)$. In this case the tree requires $O(w \cdot kn/(f(n) \log n))$ bits of space for the $r()$ values (the r values are now sums of $O(n)$ w-bit numbers, and thus they require $O(w + \log n) = O(w)$ bits). We must choose $f(n) = \omega(w/ \log n)$ to achieve sublinear extra space. Let us choose $f(n) = w/ \log^{1-\varepsilon} n$, for any constant $\varepsilon > 0$. The time complexity raises to $O(w/ \log^{1-\varepsilon} n + \log n)$.

Theorem 4 *The* SEARCHABLE PARTIAL SUMS WITH INDELS *problem under RAM model with constraint* $\log n = O(w)$ *with k-bit numbers,* $k = O(w)$*, can be solved using* $kn + o(kn) + O(w)$ *bits of space, supporting the operations sum, search, insert, and delete, in* $O(w/ \log^{1-\varepsilon} n + \log n)$ *worst-case time, for any constant* $\varepsilon > 0$.

5 Dynamic Entropy-Bound Structures for Bit Vectors

We design two data structures to represent a bit sequence $A = a_1 \dots a_n$ of binary zero-order entropy H_0, using essentially nH_0 bits of space and performing operations *rank*, *select*, *insert* and *delete* in $O(\log n)$ time.

Our two solutions differ in the extra space they achieve on top of the nH_0 bits. Their common parts are as follows. Both use balanced trees with leaves reserving $s = f(n)\log n$ bits of space, where $O(\log n)$ bits can be wasted within each leaf. Both share the $O(\log n)$-time mechanism for the operations in the tree, differing only on how they manage within the leaves. Both use the mechanism of propagation to partial leaves to ensure that most leaves are almost full, and at most one out of $f(n)$ leaves is partial, so $O(f(n))$ leaves are affected by an insertion or a deletion. Both use precomputed tables to process, in constant time, $\Theta(\log n)$ bits within leaves.

We note that, since now the sequence is not directly available, we must provide a way to retrieve any bit a_i from A. In a binary sequence this is easy, as $a_i = rank(A,i) - rank(A,i-1)$, so we can do it also in $O(\log n)$ time. Actually, in our second solution, we can retrieve an $O(\log^2 n)$-bit chunk from A within the same $O(\log n)$ time.

5.1 Gap Encoding

The first mechanism is suitable for sequences where 1-bits are sparse (the complementary technique can be used when 0-bits are sparse). Let ℓ be the number of 1-bits in A, then the space we require with this method is essentially $\ell \log \frac{n}{\ell}(1 + o(1)) + O(\ell) \leq nH_0(1 + o(1)) + O(\ell)$ bits.

Recall from Section 3.4 the structure by Blandford and Blelloch [4]. We show how to improve its constant in the entropy term in space requirement to 1.

5.1.1 Operations Inside a Superblock

We maintain as many complete gaps (δ-codes) as possible within each leaf of s bits, and assume that the 1-bit after the last encoded gap belongs to the leaf (thus the last gap of A requires special treatment). This representation may leave up to $\log n + O(\log\log n)$ unused bits at the end of the leaf because the next δ-code does not fit in it, and thus it is written at the next leaf. We also need $O(\log\log n)$ bits to record the number of bits used in the leaf, as well as to mark the beginning of the circular array. We do not use the concept of block in this solution, just superblocks (that is, leaves) formed by gaps.

Note that a leaf of $s = f(n) \log n$ bits may represent as many as $O(s)$ gaps, and as few as $f(n)(1+o(1))$. In order to process a whole leaf in $O(f(n))$ time, we need universal tables that let us process it by chunks of $\Theta(\log n)$ bits. Let G be a table that receives $t = (\log n)/2$ bits as follows: $G(x) = (b, r, l)$ indicates that it is possible to decode the first l bits of x (that is, the final $t - l$ bits do not make up a complete δ-code), and that in those l bits there are r gaps that add up b. Note that it is possible that a δ-code is longer than t bits. If x is the prefix of such a δ-code, then $G(x) = (0, 0, 0)$ indicates that G is unable to decode it. Table G requires $O(\sqrt{n}\ \log n)$ bits of space.

To decode δ-codes longer than t bits we use a different table which decodes only parts (a) and (b) of the δ-code (see Section 3.4.1). $U(x) = (d, l)$ means that the first δ-code represented in x (or of which x is a prefix) represents a number of d bits, and that parts (a) and (b) of its representation require l bits. A further access for the next $d \leq \log n$ bits (once we skip the first l bits of x) completes the decoding of the long gap. Table U handles entries of $O(\log\log n)$ bits, and thus it needs $O(\text{polylog}(n))$ bits of space. Using G and U we can, in a constant number of accesses, decode at least one gap and at least $t = \Theta(\log n)$ bits from the leaf.

A $rank(S, i)$ query inside a leaf is handled in $O(f(n)+\log n)$ time by decoding successive gaps using G (and occasionally U) and adding the r values (that is, 1-bits), as long as the sum of the $b + 1$ values (that is, gap lengths plus their terminating 1-bit) does not exceed i. The l values delivered by G are used to advance in the δ-encoded sequence. Once the next G access exceeds i, we reread those bits code-wise, using U (even for short codes) and adding 1 per gap to the result, until we read the gap where position i is exceeded. Overall we spend $O(f(n))$ time with G and $O(\log n)$ time with U.

Similarly $select(S, j)$ is solved by adding values $b + 1$ until the sum of the r values exceeds j, and then rereading the last argument of G code-wise until j is reached. For $select_0(S, j)$ we must add values $b + 1$ until the sum of the b values exceeds j, and the rest is straightforward.

To insert a bit a preceding position i in S, we sequentially look for the gap where a should be inserted (using G and U as before). Say that a must be inserted at relative position i' within $0^{g_k}1$, $1 \leq i' \leq g_k + 1$. If $a = 1$ we must replace $\delta(g_k)$ by $\delta(i'-1)\delta(g_k-i'+1)$ (see Section 3.4.2). Otherwise, if $a = 0$ we must replace $\delta(g_k)$ by $\delta(g_k + 1)$. All the δ-codes that follow must be shifted to make room for the new code. The replacement can be easily done in $O(\log n)$ time and the shifting can be carried out in $O(f(n))$ time as in Section 4.2. Deletion of a bit is analogous.

We note that insertion of a new bit can expand the code sequence within the leaf by $O(\log n)$ bits, which may overflow and require that (other) $O(\log n)$ bits formed by whole overflowing codes be moved to the next leaf. This propagation is identical to that of Section 4. The fact that we move $O(\log n)$ bits instead of one bit changes nothing under the RAM model: To copy $O(\log n)$ bits from the previous leaf, one first makes room for them by taking out $O(\log n)$ bits from the end of the circular array; then the desired bits are copied just before the beginning of the circular array; and the bits that were moved out overflow to the next leaf. All this is handled in a constant amount of $\Theta(\log n)$-bit moves. Thus the insertion of a bit is handled in $O(\log n + f(n))$ time. Deletion is analogous.

5.1.2 *Changing* $\log n$ *and Handling the Case* $\log n = o(w)$

Case $\log n = o(w)$ can be treated analogously to Sections 4.5 and 4.6. We must have tables G and U ready for 5 values of $\log n$, from $(\log n) - 2$ to $(\log n) + 2$, and have plenty of time to build them for the next change of $\log n$. The way to handle the case $\log n = o(w)$ is analogous too.

We note that the extra space of the tree and partially full leaves adds up $n'/f(n)$, not $n/f(n)$. Also, the $\sqrt{nw}$ space complexity of Section 4.6 is actually $\sqrt{n'w}$. By choosing again $f(n) = \log n$ we achieve the following result.

Theorem 5 *The* DYNAMIC BIT VECTOR WITH INDELS *problem under RAM model with constraint* $\log n = O(w)$ *can be solved using* $nH_0(1 + o(1)) + O(\ell + \sqrt{n}\ \mathrm{polylog}(n) + w) = nH_0 + o(n) + O(\ell + w)$ *bits of space supporting the operations rank, select, insert, and delete, in* $O(\log n)$ *worst-case time. Here,* $H_0 \leq 1$ *is the empirical zero-order entropy of the sequence and* ℓ *the number of bits set.*

To compare the extra space against the (static) structure of [20], we rewrite $nH_0 + O(\ell)$ into the more precise form $\ell \log \frac{n}{\ell} + O(\ell \log\log \frac{n}{\ell})$. Our $1 + o(1)$ is actually $1 + O(1/\log n)$, so the product of both is still $\ell \log \frac{n}{\ell} + O(\ell \log\log \frac{n}{\ell})$, just as in [20]. In addition we have a dependence on the uncompressed stream size, yet this is mild, $O(\sqrt{n}\ \mathrm{polylog}(n))$. In Section 5.2 this dependence becomes stronger, but in exchange the extra space $O(\ell)$ is removed, which is relevant if $\ell = \Theta(n)$.

5.1.3 *Searchable Partial Sums Revisited*

Consider again the problem of managing a sequence A of k-bit *positive* numbers a_i, $1 \leq a_i \leq 2^k$. Assume we represent it as a binary sequence A' of $\sum_{i=1}^{n} a_i$ bits. In A' we set bits $sum(A, i)$ for all i. Then, it holds $sum(A, i) = select(A', i)$ and $search(A, j) = 1 + rank(A', j - 1)$. Inserting a number a into A is equivalent to inserting a whole gap $\delta(a - 1)$. This can be done in a form completely analogous as how we inserted individual bits (and even slightly simpler). The same holds for deletion.

Thus all the operations are supported in $O(\log n)$ time. As for the space, calling $n' = \sum_{i=1}^{n} \log a_i \leq kn$, the number of bits in A' is upper bounded by $n' + o(n') + O(n)$. The following result is immediate (for the details related to w recall Section 4.7).

Theorem 6 *The* Searchable Partial Sums with Indels *problem under RAM model with constraint* $\log n = O(w)$ *with* k*-bit positive numbers can be solved using* $n' + o(n') + O(n + w)$ *bits of space, where* $n' \leq kn$ *adds up the exact number of bits needed to represent each number in the sequence. This representation supports the operations sum, search, insert, and delete, in* $O(w/\log^{1-\varepsilon} n + \log n)$ *worst-case time for any constant* $\varepsilon > 0$*. In particular, this is* $O(\log n)$ *if* $\log n = \Theta(w)$*.*

Note that this result is similar to that of Theorem 4 if all the numbers are at least 2^{k-1}. Yet, when there are small and large numbers together, this theorem achieves a more compact representation.

5.2 Block Identifier Encoding

The second mechanism to compress bit sequences is slightly more complex, yet it removes the $O(\ell)$ term from the space complexity. This is important when the sequence is sufficiently dense of 1-bits.

The solution in this section uses a scheme close to the one described in Section 3.1, albeit simplified because we do not need to achieve constant time within a leaf. We divide A into blocks and superblocks, where superblocks (the tree leaves) reserve $s = f(n) \log n$ bits of space and maintain as many complete blocks as possible. Each block represents $t = (\log n)/2$ bits, but it is stored in fewer bits using its (c, o) identifier. We do not represent the L and Q sequences of Section 3.1, just the D sequence of block identifiers. Each leaf has at most $t + O(\log \log n)$ wasted bits, for the unused space at the end and to store the exact length of the D sequence within the block. This amounts to $O(n/f(n))$ wasted bits overall.

5.2.1 Queries Inside a Superblock

A table G, similar in spirit to that of Section 5.1, is used to decode $\Theta(\log n)$ bits from the leaf in constant time. $G(x) = (b, r, l)$ indicates that it could decode up to $l \leq t$ bits from x (since the rest did not encode a whole block), where it found b encoded blocks, adding up r 1-bits overall. When $G(x) = (0, 0, 0)$, we are in presence of a long code (of length $> t$), which is decoded in constant time as follows. We first read the $O(\log\log n)$ bits of c in constant time. Then, a small universal table $C(c) = \left\lceil \log \binom{t}{c} \right\rceil$ tells us the number of bits of the o entry. We read in constant time the next $C(c)$ bits, which gives us o. Finally, a table $U(c, o) = (x, r)$ gives us the explicit t-bit content x of the block encoded as (c, o), and its total number r of 1-bits. Thus, in constant time we decode $\Theta(\log n)$ bits from the leaf, and at least one entry. Tables G and U require $O(\sqrt{n}\ \text{polylog}(n))$ bits of space, whereas C requires $O(\text{polylog}(n))$ bits.

A $rank(S, i)$ query inside a leaf is handled in $O(f(n) + \log n)$ time by decoding successive blocks using G and adding up the r values (that is, 1-bits), as long as the sum of the $t \cdot b$ values (that is, processed block lengths) does not exceed i. The l values delivered by G are used to advance in the encoded sequence. Once the sum of tb values exceeds i after a G access, we reread those bits block-wise using C and U (even for short codes), and add up the r values given by U, until we read the block that contains position i. This last block is reprocessed bitwise using the x value given by U. Overall we spend $O(f(n) + \log n)$ time.

Similarly $select(S, j)$ is solved by adding values tb until the sum of the r values exceeds j, then rereading the last argument of G block-wise until r is exceeded again, and finally processing the last block bit-wise. For $select_0(S, j)$ we must add values tb until the sum of $tb - r$ values exceeds j.

5.2.2 Inserting and Deleting Bits

To insert a bit a preceding position i in S we sequentially find, using G, C, and U, the block b where the insertion is to take place, $b = 1 + \lfloor i/t \rfloor$. All the $D(1 \ldots b-1)$ entries are direcly copied into a new memory area where the updated representation of S is to be built. On a RAM machine this copying can be done in $O(f(n))$ time.

The block $D(b) = (c, o)$ to modify is obtained in constant time with tables C and U. Let $B = a_1 \ldots a_t$ be the bits of this block, and let $i' = i - (b - 1)t$ be the position to insert the bit a within B. Thus we compute $B' = a_1 \ldots a_{i'-1} a a_{i'} \ldots a_{t-1}$ and save a_t for later. Again, B' can be computed in constant time using bit shifts. To compress B' we use a universal table H, which given a t-bit block gives its (c, o) representation. H requires $O(\sqrt{n}\ \log n)$ bits, and gives $H(B') = (c', o')$ in constant time. This description $D(b)' = (c', o')$ is appended at the updated copy of S we are constructing.

We must now take care of the remaining blocks to the right. We have a bit a_t that fell off B. To perform all this propagation in $O(f(n))$ time, we use yet another universal table $J(a, x)$, where a is a bit to insert at the beginning of the next block and x is the sequence of the first (compressed) t bits of $D(b + 1 \ldots)$. $J(a, x) = (D', a')$ means that, if we decode from x as many integral blocks as we can, append bit a at the beginning, and reencode them, we obtain sequence D' and bit a' falls off at the end of D'. Another table $V(x) = r$ tells us how many bits we could use from x, so we can advance in the processing of sequence D by r bits after having copied D' to the new version of S we are constructing. If $V(x) = 0$, this means that x starts a long block (that is, whose compressed representation occupies more than t bits). In this case we treat the block individually: We decode it using C and U, insert bit a at its beginning, call a' the bit that overflows at its end, and recompress it using H. Therefore, in constant time we process $\Theta(\log n)$ bits from the leaf, and at least one entry. The process continues until we complete the leaf and then replace S by its updated version. Note we still have one overflown bit.

Tables J and V require $O(\sqrt{n} \text{ polylog}(n))$ bits. With the occasional help of C, U, and H, they process the leaf in $O(f(n))$ time, plus the time necessary to write the modified leaf by $\Theta(\log n)$-bit chunks.

Let us consider how much can the superblock grow by the insertion of a single bit. If a new block is started (which can occur only in a partial leaf), we need $O(\log \log n)$ more bits. In addition, the D entry of a block may grow because its (c, o) descriptor changes. The maximum value of $\lceil \log \binom{t}{c+1} \rceil - \lceil \log \binom{t}{c} \rceil$ is $\lceil \log t \rceil$, achieved when $c = 0$. Propagated over at most $O(f(n) \log n / \log \log n)$ blocks, the sequence of D values might be increased by $O(f(n) \log n)$ bits. This is as large as a whole superblock, and means that a single bit insertion might double the size of the superblock in some extreme cases. For example, if the sequence is $(0^t 1^t)^r$, all the c values will be 0 or t, and the o indexes will be empty, thus we will store $f(n) \log n / \log \log n$ blocks in the superblock. If we now insert a 1 at the beginning of the sequence, each o descriptor becomes $\log t = O(\log \log n)$ bits wide, which adds up $f(n) \log n$ extra bits. Still, the new superblock is also $O(f(n) \log n)$ size and can be output using J and V in $O(f(n))$ time.

5.2.3 Overflow to the Next Superblock

At the end of the operation, it might be that the new sequence does not fit within the s bits allocated to the leaf. If so, we take out as many blocks as necessary from the end of the leaf, so as to move them to the beginning of the next leaf. We have seen that we might have to move up to $O(f(n) \log n)$ bits. In addition we must insert the excess bit at the next leaf (after the blocks we are moving, if any).

The circular array mechanism is not useful this time. The process completely rewrites the next leaf S'. We move the overflowing D entries to the beginning of S'. Then we must insert the carry bit at the beginning of the original entries of S'. This can be carried out in $O(f(n))$ time using tables J and V. Yet, this bit insertion may produce another $O(f(n)\log n)$-bits overflow, in addition to the original $O(f(n)\log n)$ bits. We can create a new leaf as soon as we have enough overflown bits. This ensures that at most s bits are ever propagated to the next leaf. The propagation can thus be carried out in $O(f(n))$ time per leaf rewritten/created. Moreover, as a leaf of s bits can grow up to size $O(s)$, each leaf can trigger the creation of $O(1)$ further leaves. The mechanism of partial leaves (Section 4.3) limits the propagation among leaves: only $O(f(n))$ leaves are rewritten or created in the process.

For deletions we proceed similarly, using a table J' very similar to J: $J'(x,a)$ deletes the first bit of the blocks represented by x and adds bit a at their end. The bit a we give to J' is obtained in constant time using C and U, as the first bit of the block encoded in D at offset $V(x)$ from the current position. Also, we ensure that leaves are as full as possible. If some space is left at the end of the leaf, we check that the first blocks from the next leaf can be moved back, and propagate the underflow similarly as the overflows. Partial leaves are handled as before upon deletions. Note that, just as whole leaves can be created due to an insertion, up to $O(f(n))$ whole leaves can disappear due to a deletion (as their contents can shrink so as to be packed within fewer leaves).

Note that, because of the changes in $|o|$ widths, an insertion can actually produce an underflow and a deletion can produce an overflow. This is not problematic. Overall (still not considering how to manage leaves), we have $O(1/f(n))$ extra space per bit and $O(f(n)^2)$ insertion/deletion time.

5.2.4 Final Global Aspects

Creation and deletion of leaves and internal tree nodes is easily handled together with the maintenance of $r()$ and $p()$ in the tree, as in Section 4.4. We note, however, that we permit that a single update affects $O(f(n))$ leaves, and it creates/deletes $O(f(n))$ leaves. At this point, we opt for a red-black tree as our balanced tree structure. Once the leaf to be inserted or deleted is located, the red-black tree needs constant time to rebalance, so this adds up $O(f(n))$ time per insertion or deletion. As for propagating the red-black coloring upwards the root, the same reasoning used for blocked $r()$ and $p()$ updates (Section 4.4) applies. Thus the total work in the tree is $O(\log n)$.

Handling changes in $\log n$ is totally analogous to Section 5.1.2. We must have tables G, U, C, H, J, J' and V ready for 5 values of $\log n$, and we have plenty of time to build them for the next change of $\log n$. The way to handle the case $\log n = o(w)$ is analogous too.

In this case it is also true that the extra space of the tree and partially full leaves adds up $n'/f(n)$, not $n/f(n)$ (where n' is the compressed sequence length). Since now times are up to $O(f(n)^2)$, we have to choose $f(n) = \sqrt{\log n}$

to obtain $O(\log n)$ time and $O(n'/\sqrt{\log n})$ space.

Theorem 7 *The* DYNAMIC BIT VECTOR WITH INDELS *problem under RAM model with constraint* $\log n = O(w)$ *can be solved using* $nH_0(1 + o(1)) + O(n \log\log n/ \log n + w) = nH_0 + o(n) + O(w)$ *bits of space supporting the operations rank, select, insert, and delete, in* $O(\log n)$ *worst-case time. Here,* $H_0 \leq 1$ *is the empirical zero-order entropy of the sequence.*

6 Handling Sequences of Symbols

We show now how our last result on bit sequences (Section 5.2) can be extended to sequences of symbols over a general alphabet $[1, q]$. Note that this looks similar to the k-bit version of Section 4.7, but the operations to support here are quite different.

6.1 Queries Inside a Superblock

We use the general scheme of Section 5.2, adapting it to handle larger alphabets. We use superblocks of $s = f(n) \log n$ bits (or $f(n) \log_q n$ symbols). Blocks are of $t = (\log_q n)/2$ symbols, and thus span $(\log n)/2$ bits. We use the encoding of [15], where the (c, o) pairs of Section 5.2 are extended to handle non-binary sequences (recall end of Section 3.1).

A table G similar to that of Section 5.2 decodes $\Theta(\log n)$ bits from the leaf in constant time. $G(x) = (b, r_1, \ldots, r_q, l)$ indicates that it could decode up to $l \leq t \log q$ bits from x (since the rest did not encode a whole block), where it found b encoded blocks, adding up r_a occurrences of each symbol $a \in [1, q]$ overall. When $b = 0$, we are in presence of a long code (of more than $t \log q$ bits), which is decoded in constant time as follows. We first read the bits of c in constant time (those are at most $(\log n)/2$ bits according to the second bound at the end of Section 3.1). Then, the rest is handled with tables analogous to C and U of Section 5.2.1: now $C(c)$ tells the length of o entries of class c, and $U(c, o) = (x, r_1, \ldots, r_q)$ gives the explicit t-symbol content of class (c, o) and all the r_a values within the block. Thus, in constant time we decode $\Theta(\log n)$ bits from the leaf, and at least one entry. All these tables require $O(\sqrt{n}\ (\log n + q \log\log n))$ bits of space.

Queries $rank_a(S, i)$ and $select_a(S, j)$ inside a leaf are handled in $O(f(n) + \log n)$ time just as in Section 5.2, adding up r_a values. We can also retrieve a_i within the same complexity, by just locating the right block and using U to obtain the explicit symbols of it. Actually we can obtain, within the same time complexity, any chunk of $\log^2 n / \log q$ consecutive symbols.

6.2 Inserting and Deleting Symbols

The mechanism is totally analogous to Section 5.2. Table H that recompresses the new block B' in constant time still requires $O(\sqrt{n}\ \log n)$ bits. Propagation of symbols to next leaves is analogous as well. Tables J and V also require $O(\sqrt{n}\ \text{polylog}(n))$ bits.

Let us consider how much can the superblock grow by the insertion of a single bit. If a new block is started in a partial leaf, we need $O(\log n)$ bits for its c entry [10]. On the other hand, the growth of the D entries is still limited by $\log t = O(\log \log n)$ bits. The upper bound of $f(n) \log n / \log \log n$ blocks per superblock still holds, as the entries c require at least of $\Theta(\log \log n)$ bits as in the binary case. Thus, added over all possible blocks, we have that the block expansion is limited by $O(f(n) \log n)$, of the same order of the current block size. All the rest on handling overflows is as in Section 5.2. For deletions we use table J', of the same size of J.

Overall (not yet considering how to manage the tree) we have, on top of nH_0, $O((n \log q)/f(n) + (n \log q) q \log \log n / \log n)$ extra space. The first term is the sequence length divided by the overhead due to partial leaves and unused space at full leaves. The second is due to the c entries, $(n/t)q \log \log n$ (first bound at the end of Section 3.1). All the operations are handled within $O(\log n + f(n)^2)$ time. We can choose, as before, $f(n) = \sqrt{\log n}$ to obtain $O(\log n)$ time and $O(n \log q / \sqrt{\log n})$ extra space for the first term. The second term is $o(n \log q)$ for $q = o(\log n / \log \log n)$.

6.3 Managing the Tree

In each internal node of the tree we must now store the total occurrences of each symbol within the node subtree, $r_a()$. This requires $O(qnH_0/f(n))$ additional bits of space. This is $o(n \log q)$ as long as $q = o(\sqrt{\log n})$.

The search time within the tree is still $O(\log n)$, as in all cases only one $r_a()$ value is involved. Updates, however, are more complicated. A single symbol insertion/deletion may involve moving many symbols to the next leaf, and this

[10] This comes from the second bound at the end of Section 3.1. A natural question is which is the point of compressing $b = (\log n)/2$ bits into (c, o) if just c takes so much space. Yet, this is just a brutal bound that is sometimes convenient. The other bound we are using is $O(q \log \log n)$ bits.

in turn involves updating many $r_a()$ values upwards. Although those movements to the next leaf cancel out at their common ancestor, it is not hard to build examples where we need to update $\Theta(q \log n)$ values of $r_a()$ (imagine moving a block from the last leaf of the left root child to the first leaf of the right root child: since $q < t$ we can have q updates whose common ancestor is $\log n$ nodes away). Therefore, insertions and deletions cost $O(q \log n)$.

6.4 Changing $\log n$

This is analogous to Section 5.2.4. We must have tables G, U, C, H, J, J' and V ready for 5 values of $\log n$. As long as those tables take sublinear space, we have time to build them for the next change of $\log n$ (as this requires $\Theta(n)$ operations, among which we can deamortize the construction of the small tables). The way to handle the case $\log n = o(w)$ is analogous too.

In this case it holds again that the extra space of the tree and partially full leaves adds up $O(nH_0/f(n))$, not $O(n/f(n))$. By choosing again $f(n) = \sqrt{\log n}$ we achieve the following result.

Theorem 8 *The* Dynamic Sequence with Indels *problem under RAM model with constraint* $\log n = O(w)$ *and symbols in* $[1, q]$*, for* $q = o(\sqrt{\log n})$*, can be solved using* $nH_0 + o(n \log q) + O(w)$ *bits of space, supporting the operations rank and select in* $O(\log n)$ *worst-case time, and insert and delete in* $O(q \log n)$ *worst-case time. Here,* $H_0 \leq \log q$ *is the empirical zero-order entropy of the sequence.*

6.5 Handling Larger Alphabets

We now extend the result of the previous section to alphabets of size σ, larger than $q = o(\sqrt{\log n})$. The idea is to build a wavelet tree [17] (recall Section 3.2) over sequences represented using Theorem 8 [15].

Let us assume we represent the sequence for each wavelet tree level using the dynamic solution of Theorem 8. We have the restriction $q = o(\sqrt{\log n})$. The wavelet tree has $O(\log_q \sigma)$ levels. Time complexities for the operations is the number of levels times the cost per level. This is $O(\log n \log_q \sigma)$ for the query operations, and $O(q \log n \log_q \sigma)$ for the update operations.

Changes in $\log n$ occur simultaneously in all symbol sequences, and they are smoothly encapsulated within the tree of each level. Handling the case $\log n = o(w)$ is also analogous. We share a single memory area for all the $O(\log \sigma)$ sequences, so that we still need only $O(1)$ w-bit pointers.

Theorem 9 *The* DYNAMIC SEQUENCE WITH INDELS *problem under RAM model with constraint* $\log n = O(w)$ *and symbols in* $[1, \sigma]$ *can be solved using* $nH_0 + o(n \log \sigma) + O(w)$ *bits of space, supporting the operations rank and select in* $O(\log n \log_q \sigma)$ *worst-case time, and insert and delete in* $O(q \log n \log_q \sigma)$ *worst-case time, for any* $q = o(\sqrt{\log n})$*. Here,* $H_0 \leq \log \sigma$ *is the empirical zero-order entropy of the sequence, and we assume* $\sigma = o(n)$*.*

The case $q = 2$ corresponds to bit sequences, thus the wavelet tree is built directly over the representation of Section 5.2. In this case the wavelet tree has $O(\log \sigma)$ levels and all the operations cost $O(\log n \log \sigma)$. Another interesting choice is $q = \log^{\epsilon} n$, for $0 < \epsilon < \frac{1}{2}$. The height of the wavelet tree is $O(\frac{1}{\epsilon} \log \sigma / \log \log n)$. The time complexities are $O(\frac{1}{\epsilon} \log n \log \sigma / \log \log n)$ for the query operations, and $O(\frac{1}{\epsilon} \log^{1+\epsilon} n \log \sigma / \log \log n)$ for the update operations.

7 Dynamic Full-Text Indexes

In this section we extend the result of Chan, Hon, and Lam [7] for DYNAMIC TEXT COLLECTION problem (recall Section 3.6) in several aspects. The most important is a considerable reduction in space. We also change the model of operation and simplify some structures.

7.1 Reducing Space and Increasing Time

The most immediate improvement to [7] is to replace their `COUNT` structure by the one of Theorem 9 (with $q = 2$ in principle, although other tradeoffs could be interesting too). This converts the time for *rank*, *insert* and *delete* into $O(\log n \log \sigma)$, and requires only $nH_0 + o(n \log \sigma)$ bits of space. Note that H_0 refers to the zero-order entropy of A, but this coincides with the zero-order entropy of $\mathcal{C}$ as they are a permutation of each other. We immediately obtain an entropy-bound index for counting pattern occurrences on a dynamic collection of texts.

To locate occurrences and display text substrings, we can use their same `MARK` structure, yet sampling one out of $\log_{\sigma} n \log \log n$ text positions, so as to have $o(n \log \sigma)$ extra bits of space for it. With this sampling and our *rank* structure we can report each occurrence in time $O(\log^2 n \log \log n)$. Displaying a text substring of length ℓ can be carried out in time $O(\log n(\ell \log \sigma + \log n \log \log n))$.

We obtain, almost automatically, the following compressed version of their structure.

Theorem 10 *The* Dynamic Text Collection *problem can be solved with a data structure of size $nH_0(\mathcal{C}) + o(n \log \sigma) + O(w)$ bits supporting counting of occurrences of a pattern P in $O(|P| \log n \log \sigma)$ time, and inserting and deleting a text T in $O(|T| \log n \log \sigma)$ time. After counting, any occurrence can be located in time $O(\log^2 n \log \log n)$. Any substring of length ℓ from any T in the collection can be displayed in time $O(\log n(\ell \log \sigma + \log n \log \log n))$. Deletion and displaying times assume that we know the lexicographic position of T within the other texts in the collection. Here n is the length of the concatenation $\mathcal{C} = 0\ T_1 0\ T_2 \cdots\ 0\ T_m$, and we assume $\sigma = o(n)$.*

Note that we have already used the fact, pointed out in Section 3.6, that knowing the lexicographic position j of T within the others is sufficient to locate its last character in $A[j]$. We also point out that the restriction $\sigma = o(n)$ comes from structure C, which needs $O(\sigma \log n)$ bits. This is $o(n \log \sigma)$ as long as $\sigma = o(n)$.

It is good time to give simple descriptions for C and `MARK`. We note that $C[c]$ is the number of occurrences of characters smaller than c in T. Let us consider $K[c]$ as the number of occurrences of c in T, and build a Searchable Partial Sums structure for it. Now $C[c] = \sum_{c'<c} K[c]$, which is a *sum* query, and upon text insertions/deletions we must increase/decrease by 1 some entry in K. Using Theorem 3, we require $\sigma \log n(1 + o(1))$ bits for K and can answer $C[c]$ and perform the updates in $O(\log n)$ time. This does not affect the given time complexities.

For `MARK`, we will maintain a text sampling so that the distance between consecutive samples is between $(\log n)/2$ and $\log n$. There are two queries to handle. The first is, given a position in $\mathcal{A}$, know whether it is sampled or not, and if it is, know the corresponding value. We maintan an array S_A with the differences between consecutive sampled positions in $\mathcal{A}$ (starting with an artificial 1), and a Searchable Partial Sums with Indels structure over S_A. To know whether $\mathcal{A}[i]$ is sampled, we ask whether $sum(S_A, search(S_A, i)) = i$. If it is, then it is the $search(A, i)$-th sample in the set. The sampled $\mathcal{A}[i]$ values are stored in a balanced tree in order of increasing i so that they can easily be found by position.

The second query to handle is for displaying. Given a text position, we wish to know which is the nearest sampled position following it in $\mathcal{C}$. For this sake we maintain array S_C, storing differences between consecutive samples in $\mathcal{C}$, and also processed for Searchable Partial Sums with Indels. The text sample following j is thus $sum(S_C, search(S_C, j))$. We also use $search(S_C, j)$ to access a balanced tree storing the samples in text position order and storing the corresponding $\mathcal{A}$ position.

Upon a character insertion in $\mathcal{A}[i]$, corresponding to position j in $\mathcal{C}$, we must increase $S_A[search(S_A, i)]$ and $S_C[search(S_C, j)]$ by 1. If the latter exceeds $\log n$, we must insert a new sample within it. This corresponds to deleting the S_C entry and inserting 2 new entries replacing it. In this case, a corresponding entry must be inserted in S_A, thus replacing the mentioned S_A entry by two. The balanced trees that contain the samples are updated too. Similarly, when a character is removed from the collection, entries are decremented and might have to be merged if an S_C entry falls below $(\log n)/2$.

All these operations are carried out in $O(\log n)$ time and $o(n \log \sigma)$ bits of space, which does not affect time nor space complexities. This description for `MARK` is simpler than the one in [7].

7.2 An Improved Model for Handling the Collection

We find that asking the users of the data structure to know the lexicographic position of their texts within the collection is delegating a problem the same data structure should solve. In this section we give a more friendly model that maintains the same time complexities and requires $O(m \log n)$ additional bits of space. This extra space should be irrelevant unless the texts are very short [11] .

When the user inserts a new text T into $\mathcal{C}$, we return a *handle* for it. The handle is a $\log m$-bits number, where m is the current number of texts in the collection. To delete T later, we only require its original handle. Pattern occurrences are given in the form (i, j), where i is the handle of the text where the occurrence lies and j is the position within that text. Finally, to retrieve text substrings we only need the handle of the text to display and the positions within it.

To implement this we store a balanced tree `HANDLE`, where the handles are the keys and are stored at the leaves, and another balanced tree `LEX` where the leaves store the same handles in lexicographical order of their corresponding texts. Associated to each key in `HANDLE` we store a pointer to the leaf corresponding to it in `LEX`. Each internal node in `LEX` contains the size of its subtree, which together with parent pointers, easily permits discovering the lexicographic position of a leaf in `LEX` by an upwards traversal (adding up the size of left subtrees of parent nodes we arrive at from the right child).

[11] Removing this term was the explicit goal in [7], precisely for the case of many short texts, so we are addressing different situations. Yet, as mentioned in Section 3.6, they could have converted their $O(\log^2 n)$ deletion time into $O(\log n)$ within their model, and without resorting to Ψ.

Together, both trees permit determining the lexicographic position of a text given its handle in $O(\log m) = O(\log n)$ time, and require $O(m \log n)$ bits of space.

After a new text T is inserted in the collection, we must determine its lexicographical position among the other texts, so as to insert a new corresponding leaf at the correct position in `LEX`. This is easy, as the lexicographic position corresponds to the position where $t_{|T|}$ was inserted in A. Once we do this, we insert the new text handle in `HANDLE` and point to the newly created `LEX` leaf. All the operations in `LEX` take $O(\log m)$ time.

Let us now switch our attention to `HANDLE`. Upon a text insertion we find the smallest unused handle number (this guarantees that the handle will require $\log m$ bits as promised). This is easily achieved by storing in each internal node of `HANDLE` the subtree size and the maximum handle value stored within: When the numbers in the left subtree differ (as there are holes in there) we descend to the left, otherwise to the right. These data at internal nodes are easy to maintain upon tree updates, all in $O(\log m)$ time.

Still, note that there is a potential problem with the handle numbers we manage. It is possible to insert m collections and then delete all but the last one, so that we have only one collection but maintain a handle of $\log m$ bits. The only way to fix this is to permit the structure to modify handles upon deletions, even those for texts that do not participate in the operations. That is, upon a deletion, we should inform that the largest existing handle has been renamed to use the value of the deleted handle. This ensures that all handles are within $[1, m]$. Otherwise the space required by the structure is $O(m \log(n + M))$, where M is the largest number of texts in the collection we have ever had.

The only missing piece is how to report occurrence positions in the format *(handle, local position)* instead of absolute position in the collection. A new balanced tree `POS` stores the handles in text position order. Each handle stores the distance in $\mathcal{C}$ to the previous leaf, and internal nodes accumulate these distances. Then it is immediate that the absolute position obtained using `MARK` can be converted into its handle plus relative position in a root-to-leaf traversal on the tree. Similarly, a display request for $T_j[l, r]$ is converted using `POS` into a display request for $\mathcal{C}[l', r']$, by having a pointer from `HANDLE` leaves to their corresponding leaves in `POS`, and traversing `POS` from the leaf to the root. Tree `POS` can easily be maintained in $O(\log m)$ time for each text insertion and deletion.

Finally, we must consider the case of $\log n$ changing. In this case we can use a standard method: We maintain three copies of all the extra structures (including those for `MARK` and the dynamic C table of [7]), for $(\log n) - 1$, $\log n$, and $(\log n) + 1$. When the change occurs we switch to the new structures, and have sufficient time to build new structures for $(\log n)+1$ or $(\log n)-1$ before $\log n$ changes again. The case $\log n = o(w)$ is handled essentially as for the tree in Section 4.6, as we can afford a constant factor in the space overhead of these structures.

Theorem 11 *The* Dynamic Text Collection *problem can be solved with a data structure of size $nH_0(\mathcal{C}) + o(n \log \sigma) + O(m \log n + w)$ bits supporting counting of occurrences of a pattern P in $O(|P| \log n \log \sigma)$ time, and inserting and deleting a text T in $O(|T| \log n \log \sigma)$ time. After counting, any occurrence can be located in time $O(\log^2 n \log \log n)$. Any substring of length ℓ from any T in the collection can be displayed in time $O(\log n(\ell \log \sigma + \log n \log \log n))$. Here n is the length of the concatenation $\mathcal{C} = 0\ T_1 0\ T_2 \cdots\ 0\ T_m$, and we assume $\sigma = o(n)$.*

Note that we have assumed that the method of modifying the handles is acceptable, otherwise the $O(m \log n)$ extra space is $O(m \log(n + M))$ as explained. The time complexities do not change.

7.3 Space is Actually h-th Order Entropy

Recall the partitioning of L into ℓ pieces $L^1 L^2 \cdots L^\ell$ according to the h-contexts (end of Section 3.5.1): It is sufficient to achieve zero-order entropy within each partition to obtain h-th order entropy overall. Previous work [15] on a static setting made use of this property by building wavelet trees over the partitions, so as to obtain h-th order entropy from the sum of zero-order entropies of the wavelet trees. Trying to maintain such an optimal partitioning under a dynamic setting seems to be very difficult because the partitioning can change abruptly due to a single character insertion. It is still possible to maintain a dynamic partition for a given fixed context length h, by keeping a trie of the existing contexts with a local wavelet tree at each trie leaf. In this section, however, we prove a much more striking result: We show that the solution obtained in the previous section, with just a single wavelet tree for all the text, *is* indeed an nH_h-bits space solution.

For the proof, let us first state formally a couple of results reviewed earlier in the paper. The first is mentioned in Section 3.2, and the second in Section 3.5.1.

Lemma 1 ([17]) *Let L be a string and B_v the corresponding binary sequence for each node v of the wavelet tree of L. Then $\sum_v |B_v| H_0(B_v) = |L| H_0(L)$.*

Lemma 2 ([27]) *Let $L = L^1 L^2 \ldots L^\ell$ be a partition of L, the BWT of T, according to contexts of length h in $\mathcal{M}$. Then $\sum_{1 \le i \le \ell} |L^j| H_0(L^j) = nH_h(T)$.*

We are now ready to prove our main Lemma.

Lemma 3 *Let $L = L^1 L^2 \cdots L^\ell$ be any partition of L, the BWT of T. The number of bits used by a partition L^j in the wavelet tree of L is upper bounded by $|L^j| H_0(L^j) + O(|L^j| \log \sigma / \log n + \sigma \log n)$.*

Proof. The bits corresponding to L^j form a substring of the bit vectors at each node of the wavelet tree, as their positions are mapped to the left and right child using $rank_0$ or $rank_1$, thus order is preserved. Let us consider a particular node of the wavelet tree and call B its bit sequence. Let us also call B^j the substring of B corresponding to partition L^j, and assume B^j has l^j bits set. Consider the blocks of b bits that compose B, according to the partitioning of [32] (Section 3.1). Let $B^j_{blk} = B^j_1 B^j_2 \ldots B^j_t$ be the concatenation of those bit blocks that are *fully contained* in B^j, so that B^j_{blk} is a substring of B^j of length $b \cdot t$. Assume B^j_i has l^j_i bits set, so that B^j_{blk} has $l^j_1 + \ldots + l^j_t \leq l^j$ bits set. The space the o fields of the (c, o) representations of blocks B^i_j take in the compressed B^j_{blk} is

$$\sum_{i=1}^{t} \left\lceil \log \binom{b}{l^j_i} \right\rceil \leq \log \binom{b \cdot t}{l^j_1 + \ldots l^j_t} + t \leq \log \binom{|B^j|}{l^j} + t \leq |B^j| H_0(B^j) + t$$

where all the inequalities hold by simple combinatorial arguments [29] and have been reviewed in Section 3.1.

Note that those B^j bit vectors are precisely those that would result if we built the wavelet tree just for L^j. According to Lemma 1, adding up those $|B^j| H_0(B^j)$ over all the $O(\sigma)$ wavelet tree nodes gives $|L^j| H_0(L^j)$. To this we must add two space overheads. The first is the extra t bits above, which add up $O(|L^j| \log \sigma / \log n)$ over the whole wavelet tree because $b \cdot t \leq |B^j|$ and the $|B^j|$ lengths add up $|L^j|$ at each wavelet tree level. The second overhead is the space of the blocks that overlap with B^j and thus were not counted: As B^j is a substring of B, there can be at most 2 such blocks per wavelet tree node. At worst they can take $O(\log n)$ bits each, adding up $O(\sigma \log n)$ bits over the whole wavelet tree. $\square$

The above lemma lets us split the wavelet tree "horizontally" into pieces. Let us add up all the zero-order entropies for the pieces. If we partition L according to contexts of length h in $\mathcal{M}$, and add up all the space due to all partitions in the wavelet tree, we get $\sum_{1 \leq j \leq \ell} |L^j| H_0(L^j) = n H_h(T)$ (Lemma 2). To this we must add (*i*) $O(|L^j| \log \sigma / \log n)$, which sums up to $O(n \log \sigma / \log n) = o(n \log \sigma)$ bits over all the partitions; and (*ii*) $O(\sigma \log n)$ bits per partition, which gives $O(\ell \sigma \log n)$. In the partitioning we have chosen we have $\ell \leq \sigma^h$, thus the upper bound $n H_h + o(n \log \sigma) + O(\sigma^{h+1} \log n)$ holds for the total number of bits spent in the wavelet tree. The next theorem immediately follows.

Theorem 12 *The space required by the wavelet tree of L, the BWT of T, if the bitmaps are compressed using [32], is $n H_h(T) + o(n \log \sigma) + O(\sigma^{h+1} \log n)$*

bits for any $h \geq 0$. This is $nH_h(T) + o(n \log \sigma)$ bits for any $h \leq \alpha \log_\sigma n - 1$ and any constant $0 < \alpha < 1$. Here n is the length of T and σ its alphabet size.

Note that this holds *automatically and simultaneously for any h*, and we do not even have to care about h in the index. The next improvement over Theorem 11 is now immediate.

Theorem 13 *The* Dynamic Text Collection *problem can be solved with a data structure of size $nH_h(\mathcal{C}) + o(n \log \sigma) + O(\sigma^{h+1} \log n + m \log n + w)$ bits, simultaneously for all h. It supports all the operations of Theorem 11 with the same time complexities. For $h \leq \alpha \log_\sigma n - 1$, for any constant $0 < \alpha < 1$, the space complexity simplifies to $nH_h(\mathcal{C}) + o(n \log \sigma) + O(m \log n + w)$ bits.*

Something striking about the above result is that it holds for the static full-text self-indexes in the literature that build on the wavelet tree of the BWT of the text [24,15], but this has gone unnoticed and more complicated arrangements have been made to reach h-th order entropy. In [24], they first run-length compress the BWT in order to reduce its length to $O(nH_h)$ and then apply the BWT. In [15] they explicitly cut the BWT into pieces L^j so that the sum of nH_0 sizes of the pieces adds up nH_h. In both papers, the simpler version they build on (just the wavelet tree of the BWT) would have been sufficient. Thus, we have achieved a significant simplification in the design of static full-text indexes as well. (There are other results in those papers, some of which we have used here.)

In [13] they propose an algorithm to cut A optimally, so as to minimize the sum of local zero-order entropies plus the overheads of maintaining the separate structures. The optimum partitioning might not correspond to any fixed h value, but rather use longer contexts in some parts of A and shorter in others. What we have shown is that the space produced by *any* splitting of A into pieces is achieved in the simple arrangement having just one wavelet tree, without the need of finding such an optimal partitioning. Their technique, on the other hand, is more general as it works for any zero-order compressor.

Also the paper where the wavelet tree is originally proposed [17] as an internal tool to design one of the most space-efficient compressed full-text indexes, would benefit from our simplification. They cut A into a table of *lists* (columns) and *contexts* (rows). All the entries across a row correspond to a contiguous piece of A, that is, some context L^j. A wavelet tree is built over each table row so as to ensure, again, that the sum of zero-order entropies over the rows adds up to global h-th order entropy. Our finding implies that all rows could have been concatenated into a single wavelet tree and the same space would have been achieved. This would greatly simplify the original arrangement and possibly expose the deep relationship with the BWT-based approaches [28].

Interestingly, in [18] they find out that, if they use gap encoding over the successive values along a *column*, and they then concatenate all the columns, the total space is $O(nH_h)$ without any table partitioning as well. Both findings share the same source: the sum of zero-order entropies of the table cells, no matter the order, adds up to nH_h.

Finally, it is interesting to point out that, in a recent paper [11], the possibility of achieving h-th order compression when applying wavelet trees over the BWT is explored (among many other results), yet they resort to run-length compression to achieve this. Once more, our finding is that this is not really necessary to achieve h-th order compression if the levels of the wavelet tree are represented using the technique of block identifier encoding [31].

Another consequence of our result is that we obtain an $O(n \log n \log \sigma)$ time construction algorithm for a compressed self-index requiring $nH_h + o(n \log \sigma)$ bits *working space* during construction: This is obtained by just inserting text T into an empty collection. This index can be easily converted into a more efficient static self-index, where a static wavelet tree requires the same space and reduces the $O(\log n \log \sigma)$ time complexities to $O(\lceil \log \sigma / \log \log n \rceil)$ [15].

Therefore, we have obtained the *first* compressed self-index with space essentially equal to the h-th order empirical entropy of the text collection, which in addition can be built within this working space. Alternative dynamic indexes or constructions of self-indexes [12,22,2,6] achieve at best $O(nH_h)$ bits of space (with constants larger than 4), and in many cases worse time complexities, as explained in the Introduction.

Note also that, from the dynamic index just built, it is very easy to obtain the BWT of T. It is a matter of finding the characters of A one by one. This takes $O(n \log n \log \sigma)$ time, just as the construction, and gives an algorithm to build the BWT of a text within entropy bounds. The best result we know of, in terms of space complexity [23], achieves $O(n \log^2 n)$ time ($O(n \log n)$ on average) using $O(n)$ bits in addition to the $n \log \sigma$ bits of the text.

8 Final Remarks

We have introduced a technique to maintain a dynamic bit sequence of length n using $nH_0 + o(n)$ bits of space, where $0 \leq H_0 \leq 1$ is the zero-order entropy of the sequence. The structure answers *rank* and *select* queries and permits insertions and deletions of bits, in worst-case logarithmic time. This is the first dynamic data structure achieving this space. Closely related lower bounds [30] suggest that the time complexities are optimal, yet a proof is missing for this particular set of operations.

From this central result we have uncovered many connections with other problems and derived a surprising number of results in their dynamic setups, using less space and/or time compared to the best existing solutions. We have obtained improved update times and slightly better space for searchable partial sums with indels; the first results on dynamic sequences over alphabets of size σ (achieving zero-order entropy space with times of the form $O(\log n \log \sigma)$ per operation); compressed dynamic full-text self-indexes and compressed construction of full-text self-indexes (achieving high-order entropy space and $O(\log n \log \sigma)$ worst-case time per character in all the operations).

All our results are worst-case and support varying $\log n$, being valid even for the case $\log n = o(w)$, where w is the size of the machine word. The traditional techniques to support varying $\log n$ cannot be directly adapted because we cannot afford the extra space to maintain several copies of the data structure.

Some of the results we have achieved match existing lower bounds. Yet, others seem to be improvable. In particular, it should be possible to improve the $O(\log n \log \sigma)$ time complexity for accessing and updating dynamic wavelet trees, perhaps with a fractional cascading mechanism. This would immediately affect several other results we achieved.

Finally, we have shown that wavelet trees, when built over the BWT of a text, automatically achieve high-order entropy. This translates into a significant simplification to many existing self-indexes that achieve high-order entropy (e.g., [24,15]), by showing that the base technique they build on naturally achieves the result without need of any further engineering. Our finding also impact several other works that use this technique in one form or ahother [17,13,11].

Still, the results in [24,15] have practical value. In their actual implementation (`http://pizzachili.dcc.uchile.cl` or `http://pizzachili.di.unipi.it`), zero-order entropy is achieved by using *uncompressed* bit streams over a *Huffman-shaped* wavelet tree, as this requires less space overhead and implementation effort than using the technique of [32] over balanced wavelet trees. In this case the locality property does not hold, and h-th order entropy would not be achieved if just the simple wavelet tree of the BWT was used. Now, our findings suggest that implementing the technique of [32] over a balanced wavelet tree is indeed promising, as in exchange for its (sublinear) space overhead and implementation effort, it would need no extra data structure to achieve higher order compression. Thus we expect it to constitute a simple and competitive alternative in practice.

This, in particular, would immediately derive into a simple and powerful practical algorithm to build, within entropy bounds, different compressed self-indexes, the BWT of a text, and so on. Moreover, this can have a noticeable practical impact on difficult real-life problems such as building indexes for texts that do not fit in main memory, even compressed. In practice, one of the best algorithms for this problem [8] is still a multi-pass technique with I/O complexity $O(n^2/M)$ [16], where M is the maximum text size that can be indexed in main memory. The use of our compressed construction technique on main memory translates into much larger values of M, and thus fewer passes over the disk, in exchange for (much less important) higher CPU times within each pass.

This is connected with possibly the most important current challenge for compressed data structures, and for compressed full-text self-indexes in particular. Compressed data structures mainly aim at avoiding the use of disk whenever possible, usually in exchange for slower operation in main memory. This pays off by far because main memory is much faster than secondary memory (and this happens at any level of the memory hierarchy, e.g., one can achieve better cache usage just because more compressed data fit in the cache, even if the access patterns are not particularly cache-friendly). Yet, when the data does not fit in main memory anyway, one wishes to have a compressed data structure with good locality of reference, so as to minimize the I/O complexity. This is challenging because better space usage does not automatically translate into fewer block accesses. Indeed, many of the existing solutions suffer from poor locality of reference.

References

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

[2] D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proc. ISAAC'05*, LNCS 3827, pages 1143–1152, 2005.

[3] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[4] D. Blandford and G. Blelloch. Compact representations of ordered sets. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–19, 2004.

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

[6] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 2006. To appear.

[7] W.-L. Chan, W.-K. Hon, and T.-W. Lam. Compressed index for a dynamic collection of texts. In *Proc. 15th Annual Symposium on Combinatorial Patter Matching (CPM)*, LNCS 3109, pages 445–456, 2004.

[8] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[9] P. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. WADS'89*, pages 39–46, 1989.

[10] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–20, 1975.

[11] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. In *Proc. 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, 2006. To appear.

[12] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pages 390–398, 2000.

[13] P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 655–663, 2004.

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

[15] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, 2006. To appear. Preliminary versions in *Proc. SPIRE 2004* and Tech. Rep. TR/DCC-2004-5, Dept. of Computer Science Univ. of Chile, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequences.ps.gz`.

[16] G. Gonnet, R. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 3: New indices for text: Pat trees and Pat arrays, pages 66–82. Prentice-Hall, 1992.

[17] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850, 2003.

[18] R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–645, 2004.

[19] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[20] A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. Compressed data structures: dictionaries and data-aware measures. In *Proc. 5th International Workshop on Experimental Algorithms (WEA)*, pages 158–169, 2006.

[21] W.-K. Hon, T.-W. Lam, K. Sadakane, and W.-K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 240–249, 2003.

[22] W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums. In *Proc. ISAAC'03*, LNCS 2906, pages 505–516, 2003.

[23] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. In *Proc. DIMACS Working Group on the Burrows-Wheeler Transform: Ten Years Later*, 2004.

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

[25] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 2006. To appear.

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

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

[28] G. Navarro and V. Mäkinen. Compressed full-text indexes. Technical Report TR/DCC-2006-6, Dept. of Computer Science, University of Chile, April 2006. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survcompr2.ps.gz`. Submitted to a journal.

[29] R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. 26th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 595–604, 1999.

[30] M. Patrascu and E. Demaine. Tight bounds for the partial-sums problem. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 20–29, 2004.

[31] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proc. WADS'01*, pages 426–437, 2001.

[32] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'02*, pages 233–242, 2002.