Optimal Dynamic Sequence Representations *

Gonzalo Navarro† Yakov Nekrich‡

Abstract

We describe a data structure that supports access, rank and select queries, as well as symbol insertions and deletions, on a string $S[1,n]$ over alphabet $[1..\sigma]$ in time $O(\lg n/\lg\lg n)$, which is optimal. The time is worst-case for the queries and amortized for the updates. This complexity is better than the best previous ones by a $\Theta(1+\lg\sigma/\lg\lg n)$ factor. Our structure uses $nH_0(S)+O(n+\sigma(\lg\sigma+\lg^{1+\varepsilon} n))$ bits, where $H_0(S)$ is the zero-order entropy of S and $0<\varepsilon<1$ is any constant. This space redundancy over $nH_0(S)$ is also better, almost always, than that of the best previous dynamic structures, $o(n\lg\sigma)+O(\sigma(\lg\sigma+\lg n))$. We can also handle general alphabets in optimal time, which has been an open problem in dynamic sequence representations.

1 Introduction

String representations supporting rank and select queries are fundamental in many data structures, including full-text indexes [20, 15, 18], permutations [18, 3], inverted indexes [10, 3], graphs [13], document retrieval indexes [35], labeled trees [18, 5], XML indexes [21, 14], binary relations [5], and many more. The problem is to encode a string $S[1,n]$ over alphabet $\Sigma=[1..\sigma]$ so as to support the following queries:

$$\begin{aligned}
\mathrm{rank}_a(S,i) &= \text{number of occurrences of } a\in\Sigma \\
&\quad \text{in } S[1,i], \text{ for } 1\le i\le n.\\
\mathrm{select}_a(S,i) &= \text{position in } S \text{ of the } i\text{-th occurrence}\\
&\quad \text{of } a\in\Sigma, \text{ for } 1\le i\le \mathrm{rank}_a(S,n).\\
\mathrm{access}(S,i) &= S[i].
\end{aligned}$$

There exist various representations of S that support these operations [20, 18, 15, 3, 7]. However, these representations are static, that is, S cannot change. In various applications one needs dynamism, that is, to insert and delete symbols in S. There are not many dynamic solutions, however. All are based on the *wavelet tree* representation [20]. The wavelet tree decomposes S hierarchically. In a first level, it separates larger from smaller symbols, by marking in a bitmap which symbols of S were larger and which were smaller. The two subsequences of S are recursively separated. The $\lg\sigma$ levels of bitmaps describe S, and access, rank and select operations on S are carried out via $\lg\sigma$ rank and select operations on the bitmaps. Insertions and deletions in S can also be carried out by inserting and deleting bits from $\lg\sigma$ bitmaps (see Section 2 for more details).

In the static case, rank and select operations on bitmaps take constant time, and therefore access, rank and select on S takes $O(\lg\sigma)$ time [20]. This can be reduced to $O(1+\lg\sigma/\lg\lg n)$ by using multiary wavelet trees [15]. These separate the symbols into $\rho=o(\lg n)$ ranges, and instead of a bitmap store a sequence over an alphabet of size ρ. In the dynamic case, however, the operations on those bitmaps or sequences are slowed down. Mäkinen and Navarro [27] obtained $O(\lg\sigma\lg n)$ time for all the operations, including updates, by using dynamic bitmaps that handled all the operations in time $O(\lg n)$. They simultaneously compress the sequence to $nH_0(S)+o(n\lg\sigma)$ bits. Here $H_0(S)=\sum_{a\in[1..\sigma]}(n_a/n)\lg(n/n_a)\le\lg\sigma$ is the zero-order entropy of S, where n_a is the number of occurrences of a in S. González and Navarro [19] improved the times to $O((1+\lg\sigma/\lg\lg n)\lg n)$ by extending the results to multiary wavelet trees. In this case, instead of dynamic bitmaps, they handled dynamic sequences over a small alphabet (of size ρ). Finally, He and Munro [22] and Navarro and Sadakane [30] obtained the currently best result, $O((1+\lg\sigma/\lg\lg n)\lg n/\lg\lg n)$ time, still within the same space. They did so by improving the times of the dynamic sequences on small alphabets to $O(\lg n/\lg\lg n)$, which is optimal even on bitmaps [17]. The $\Omega((\lg n/\lg\lg n)^2)$ lower bound for dynamic range counting in two dimensions [33], and the $O(\lg n/\lg\lg n)$ static upper bound using wavelet trees [9], suggest that no more improvements are possible in this line.

In this paper we show that this dead-end can be broken by abandoning the implicit assumption that, to provide access, rank and select on S, we *must* provide rank and select on the bitmaps (or sequences over $[1..\rho]$). We show that all what is needed is to *track* positions of S downwards and upwards along the wavelet tree. It turns out that this tracking can be done in *constant*

*Partially funded by Fondecyt grant 1-110066, Chile.

†Department of Computer Science, University of Chile. Email: gnavarro@dcc.uchile.cl.

‡Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS. This work was done while this author was at the University of Chile. Email: yakov.nekrich@googlemail.com.

time per level, breaking the $\Theta(\lg n/\lg\lg n)$ per-level barrier. A second tool to achieve full independence of σ, and to compress the redundancy space, is the alphabet partitioning technique [3], which we exploit in a novel way under a dynamic scenario.

As a result, we obtain the *optimal* time complexity $O(\lg n/\lg\lg n)$ for all the queries (worst-case) and update operations (amortized). This is $\Theta(1+\lg\sigma/\lg\lg n)$ times faster than what was believed to be the "ultimate" solution. We also improve upon the space by compressing the redundancy of $o(n\lg\sigma)+O(\sigma(\lg\sigma+\lg n))$ of previous dynamic structures. Our space is $nH_0(S)+O(n+\sigma(\lg\sigma+\lg^{1+\varepsilon} n))$ bits, for any constant $0<\varepsilon<1$.

Finally, we also handle general alphabets, such as $\Sigma=\mathbb{R}$, or $\Sigma=\Gamma^*$ for a symbol alphabet Γ, in optimal time. For example, in the comparison model for $\Sigma=\mathbb{R}$, the time is $O(\lg\sigma+\lg n/\lg\lg n)$, where σ is the number of distinct symbols that appear in S; in the case $\Sigma=\Gamma^*$ for general Γ, the time is $O(|p|+\lg\gamma+\lg n/\lg\lg n)$, where $|p|$ is the query length and γ the number of distinct symbols of Γ that appear in the elements of S. Handling varying alphabets has been a long-standing problem on dynamic sequences, since wavelet trees do not deal well with them. We work around this problem by means of our dynamic alphabet partitioning scheme.

At the end we describe several applications where our result offers improved time/space tradeoffs. These include compressed indexes for dynamic text collections, construction of the Burrows-Wheeler transform [11] and static compressed text indexes within compressed space, and compressed representations of dynamic binary relations, directed graphs, and inverted indexes.

2 The Wavelet Tree

Let S be a string over alphabet $\Sigma=[1..\sigma]$. We associate each $a\in\Sigma$ to a leaf v_a of a full balanced binary tree T. The essential idea of the wavelet tree structure is the representation of elements from a string S by bit sequences stored in the nodes of tree T. We associate a subsequence $S(v)$ of S with every node v of T. For the root v_r, $S(v_r)=S$. In general, $S(v)$ consists of all occurrences of symbols $a\in\Sigma_v$ in S, where Σ_v is the set of symbols assigned to leaf descendants of v. The wavelet tree does not store $S(v)$ explicitly, but just a bit vector $B(v)$. We set $B(v)[i]=t$ if the i-th element of $S(v)$ also belongs to $S(v_t)$, where v_t is the t-th child of v (the left child corresponds to $t=0$ and the right to $t=1$). This data structure (i.e., T and bit vectors $B(v)$) is called a *wavelet tree*.

For any symbol $S[i]=a$ and every node v such that $a\in\Sigma_v$, there is exactly one bit b_v in $B(v)$ that indicates in which child of v the leaf $v_{S[i]}$ is stored. We will say that such b_v *encodes* $S[i]$ in $B(v)$; we will also say that bit b_v from $B(v)$ *corresponds* to a bit b_u from $B(u)$ if both b_v and b_u encode the same symbol $S[i]$ in two nodes v and u. Identifying the positions of bits that encode the same symbol plays a crucial role in wavelet trees. Other, more complex, operations rely on our ability to navigate in the tree and keep track of bits that encode the same symbol.

To implement access(S,i) we traverse a path from the root to the leaf $v_{S[i]}$. In each visited node we read the bit b_v that encodes $S[i]$ and proceed to the b_v-th child of v. To compute $\mathrm{rank}_a(S,i)$, we identify the last bit b' that precedes $B(v)[i]$ and corresponds to some symbol in $S(v_a)$. To answer $\mathrm{select}_a(S,i)$, we identify the index of the bit b_v in $B(v)$ that corresponds to $S(v_a)[i]$.

The standard method used in wavelet trees for identifying the corresponding bits is to maintain rank/select data structures on the bit vectors $B(v)$. Let $B(v)[e]=t$; we can find the offset of the corresponding bit in the child of v by answering a query $\mathrm{rank}_t(B(v),e)$. If v is the r-th child of a node u, we can find the offset of the corresponding bit in u by answering a query $\mathrm{select}_r(B(u),e)$. This approach leads to $O(\lg\sigma)$ query times in the static case because rank/select queries on a bit vector can be answered in constant time. However, we need $\Omega(\lg n/\lg\lg n)$ time to support rank/select and updates on a bit vector [17], which multiplies the operation times. A slight improvement can be achieved by increasing the fan-out of the wavelet tree to $\Theta(\lg^{\varepsilon} n)$: as before, $B(v)[e]=t$ if the e-th element of $S(v)$ also belongs to $S(v_t)$ for the t-th child v_t of v. This enables us to reduce the height of the wavelet trees and the query time by a $\Theta(\lg\lg n)$ factor. However, it seems that further improvements that are based on dynamic rank/select queries in every node are not possible.

In this paper we use a different approach to identifying the corresponding elements. We partition sequences $B(v)$ into blocks, which are stored in compact list structures $L(v)$. Pointers from selected positions in $L(v)$ to the structure $L(u)$ in a parent node u (and vice versa) enable us to navigate between nodes of the wavelet tree in constant time. We extend the idea to multiary wavelet trees. While similar techniques have been used in some geometric data structures [31, 8], applying them on compressed data structures where the bit budget is severely limited is much more challenging.

3 Basic Structure

We start by describing the main components of our modified wavelet tree. Then, we show how our structure supports access(S,i) and $\mathrm{select}_a(S,i)$. In the third part of this section we describe additional structures that enable us to answer $\mathrm{rank}_a(S,i)$. Finally, we show how to support updates.

3.1 Structure We assume that the wavelet tree $\mathcal{T}$ has node degree $\rho = \Theta(\lg^{\varepsilon} n)$. We divide sets $B(v)$ into *blocks* and store those blocks in a doubly-linked list $L(v)$. Each block $G_j(v)$, except the last one, contains $\Theta(\lg^3 n)$ consecutive elements from $B(v)$; the last block contains $O(\lg^3 n)$ consecutive elements. For each $G_j(v)$ we maintain a data structure $R_j(v)$ that supports rank and select queries on elements of $G_j(v)$. Since a block contains a poly-logarithmic number of elements over an alphabet of size ρ, we can answer rank and select queries in $O(1)$ time using $O(|G_j(v)|/\lg^{1-\varepsilon} n)$ additional bits, for any constant $0 < \varepsilon < 1$ (see Appendix A for details).

A *pointer* to an element $B(v)[e]$ consists of two parts: a unique id of the block $G_j(v)$ that contains offset e and the index of e in $G_j(v)$. Such a pair (block id, local index) will be called the *position* of e in v.

We maintain pointers between selected corresponding elements in $L(v)$ and its children. If an element $B(v)[e] = t$ is stored in a block $G_j(v)$ and $B(v)[e'] \neq t$ for all $e' < e$ in $G_j(v)$, then we store a pointer from e to the offset e_t of the corresponding element $B(v_t)[e_t]$ in $L(v_t)$, where v_t is the t-th child of v. If $B(v)[e] = t$ and the corresponding e_t in $L(v_t)$ is the first offset in its block, then we also store a pointer from e to e_t. If there is a pointer from e in $L(v)$ to e_t in $L(v_t)$, then we also store a pointer from e_t to e. All these pointers will be called *inter-node pointers*. We describe how inter-node pointers are implemented later in this section.

It is easy to see that the number of inter-node pointers from e in $L(v)$ to e_t in $L(v_t)$, for any fixed t, is $\Theta(g(v))$, where $g(v)$ is the number of blocks in $L(v)$. Hence, the total number of pointers that point down from a node v is bounded by $O(g(v)\rho)$. Since this also equals the number of pointers that point up to v, the total number of pointers in the wavelet tree equals $O(\sum_{v\in\mathcal{T}} g(v)\rho) = O(n \lg \sigma / \lg^{3-\varepsilon} n + \sigma \lg^{\varepsilon} n)$.

The pointers from a block $G_j(v)$ are stored in a data structure $F_j(v)$. Using $F_j(v)$, we can find, for any offset e in $G_j(v)$ and any $1 \leq t \leq \rho$, the last $e' \leq e$ in $G_j(v)$ such that there is a pointer from e' to an offset e'_t in $L(v_t)$. We describe in Appendix A how $F_j(v)$ implements the queries in constant time.

For the root node v_r, we store a dynamic partial-sum data structure $K(v_r)$ that contains the number of positions in each block of $L(v_r)$. Using $K(v_r)$, we can find the block $G_j(v_r)$ that contains the i-th element of $S(v_r) = S$, as well as the number of elements in all the blocks that precede a given block $G_j(v_r)$. Both operations can be supported in $O(\lg n/\lg\lg n)$ time [23, 30]. The same data structures $K(v_a)$ are also stored in the leaves v_a of $\mathcal{T}$. We observe that we do not store a sequence $B(v_a)$ in a leaf node v_a. Nevertheless, we divide the (implicit) sequence $B(v_a)$ into blocks and store the number of positions in each block in $K(v_a)$; we maintain $K(v_a)$ only if $L(v_a)$ consists of more than one block. Moreover we store inter-node pointers from the parent of v_a to v_a and vice versa. Pointers in a leaf are maintained using the same rules of any other node.

$L(v)$	List of blocks storing $B(v)$
$G_j(v)$	j-th block of list $L(v)$
$R_j(v)$	Supports rank/select/access inside $G_j(v)$
$F_j(v)$	Pointers leaving from $G_j(v)$
$H_j(v)$	Pointers arriving at $G_j(v)$
$P_t(v)$	Predecessor in $L(v)$ containing symbol t
$K(v)$	Partial sums on block lengths for v_r and v_a
$D_j(v)$	Deleted elements in $G_j(v)$, for v_r and v_a
DEL	Global list of deleted elements in $\overline{S}$.

Table 1: Structures inside any node v of the wavelet tree $\mathcal{T}$, or only in the root node v_r and the leaves v_a.

For future reference, we provide the list of secondary data structures in Table 1.

3.2 Access and Select Queries Assume the position of an element $B(v)[e] = t$ in $L(v)$ is known, and let i_v be the index of offset e in its block $G_j(v)$. Then the position of the corresponding offset e_t in $L(v_t)$ is computed as follows. Using $F_j(v)$, we find the index i' of the largest $e' \leq e$ in $G_j(v)$ such that there is a pointer from e' to some e'_t in $L(v_t)$. Due to our construction, such e' must exist. Let i'_v and i'_t denote the indexes of e' and e'_t respectively, and let $G_\ell(v_t)$ denote the block that contains e'_t. Let $r_v = \mathrm{rank}_t(G_j(v), i_v)$ and $r'_v = \mathrm{rank}_t(G_j(v), i'_v)$. Due to our rules to define pointers, e_t also belongs to $G_\ell(v_t)$ and its index is $i'_t + (r_v - r'_v)$. Thus we can find the position of e_t in $O(1)$ time if the position of $B(v)[e] = t$ is known.

Analogously, assume we know a position $B(v_t)[e_t]$ at $G_j(v_t)$ and want to find the position of the corresponding offset e in its parent node v. Using $F_j(v_t)$ we find the last $e'_t \leq e_t$ in $G_j(v_t)$ that has a pointer to its parent, which exists by construction. Let e'_t point to e', with index i' in a block $G_\ell(v)$. Let i'_t and i_t be the indexes of e'_t and e_t in $G_j(v_t)$, respectively. Then, by our construction, e is also in $G_\ell(v)$ and its index is $\mathrm{select}_t(G_\ell(v), \mathrm{rank}_t(G_\ell(v), i') + (i_t - i'_t))$.

To solve access(S, i), we visit the nodes $v_0 = v_r, v_1 \ldots v_h = v_a$, where $h = \lg_\rho \sigma$ is the height of $\mathcal{T}$, v_k is the t_k-th child of v_{k-1} and $B(v_{k-1})[e_{k-1}] = t_k$ encodes $S[i]$. We do not find out the offsets $e_1, \ldots, e_h$, but just their positions. The position of $e_0 = i$ is found in $O(\lg n/\lg\lg n)$ time using the partial-sums structure $K(v_r)$. If the position of e_{k-1} is known, we can find that of e_k in $O(1)$ time, as explained above. When a leaf node $v_h = v_a$ is reached, we know that $S[i] = a$.

To solve $\text{select}_a(S, i)$, we set $e_h = i$ and identify its position in the list $L(v_a)$ of the leaf v_a, using structure $K(v_a)$. Then we traverse the path $v_h, v_{h-1}, \ldots, v_0 = v_r$ where v_{k-1} is the parent of v_k, until the root node is reached. In every node v_k, we find the position of e_{k-1} in $L(v_{k-1})$ that corresponds to e_k as explained above. Finally, we compute the number of elements that precede e_0 in $L(v_r)$ using structure $K(v_r)$.

Clearly, access and select require $O(\lg_\rho \sigma + \lg n/\lg\lg n) = O((\lg\sigma + \lg n)/\lg\lg n)$ worst-case time.

3.3 Rank Queries We need some additional data structures for the efficient support of rank queries. In every node v such that $L(v)$ consists of more than one block, we store a data structure $P(v)$. Using $P(v)$ we can find, for any $1 \le t \le \rho$ and for any block $G_j(v)$, the last block $G_\ell(v)$ that precedes $G_j(v)$ and contains an element $B(v)[e] = t$. $P(v)$ consists of ρ predecessor data structures $P_t(v)$ for $1 \le t \le \rho$. We describe in Section 4 a way to support these predecessor queries in constant time in our scenario.

Let the position of offset e be the i-th element in a block $G_j(v)$. $P(v)$ enables us to find the position of the last $e' \le e$ such that $B(v)[e'] = t$. First, we use $R_j(v)$ to compute $r = \text{rank}_t(G_j(v), i)$. If $r > 0$, then e' belongs to the same block as e and its index in the block $G_j(v)$ is $\text{select}_t(G_j(v), r)$. Otherwise, we use $P_t(v)$ to find the last block $G_\ell(v)$ that precedes $G_j(v)$ and contains an element $B(v)[e'] = t$. We then find the last such element in $G_\ell(v)$ using $R_\ell(v)$.

Now we are ready to describe the procedure to answer $\text{rank}_a(S, i)$. The symbol a is represented as a concatenation of symbols $t_0 \circ t_1 \circ \ldots \circ t_h$, where each t_k is between 1 and ρ. We traverse the path from the root $v_r = v_0$ to the leaf $v_a = v_h$. We find the position of $e_0 = i$ in v_r using the data structure $K(v_r)$. In each node v_k, $0 \le k < h$, we identify the position of the last element $B(v_k)[e'_k] = t_k$ that precedes e_k, using $P_{t_k}(v_k)$. Then we find the offset e_{k+1} in the list $L(v_{k+1})$ that corresponds to e'_k.

When our procedure reaches the leaf node v_h, the element $B(v_h)[e_h]$ encodes the last symbol a that precedes $S[i]$. We know the position of offset e_h, say index i_h in its block $G_\ell(v_h)$. Then we find the number r of elements in all the blocks that precede $G_\ell(v_h)$ using $K(v_h)$. Finally, $\text{rank}_a(S, i) = r + i_h$.

Since structures P_t answer queries in constant time, the overall time for rank is $O(\lg_\rho \sigma + \lg n/\lg\lg n) = O((\lg\sigma + \lg n)/\lg\lg n)$.

3.4 Updates Now we describe how inter-node pointers are implemented. We say that an element of $L(u)$ is *pointed* if there is a pointer to its offset. Unfortunately, we cannot store the local index of a pointed element in the pointer: when a new element is inserted into a block, the indexes of all the elements that follow it are incremented by 1. Since a block can contain $\Theta(\lg^3 n)$ pointed elements, we would have to update up to $\Theta(\lg^3 n)$ pointers after each insertion and deletion.

Therefore we resort to the following two-level scheme. Each pointed element in a block is assigned a unique id. When a new element is inserted, we assign it the id $max\_id + 1$, where max_id is the maximum id value used so far. We also maintain a data structure $H_j(v)$ for each block $G_j(v)$ that enables us to find the position of a pointed element if its id in $G_j(v)$ is known. Implementation of $H_j(v)$ is based on standard word RAM techniques and a table that contains ids of the pointed elements; details are given in Appendix A.

We describe now how to insert a new symbol a into S at position i. Let $e_0, e_1, \ldots, e_h$ be the offsets of the elements that will encode $a = t_0 \circ \ldots \circ t_h$ in $v_r = v_0, v_1, \ldots, v_h = v_a$. We can find the position of $e_0 = i$ in $L(v_r)$ in $O(\lg n/\lg\lg n)$ time using $K(v_r)$, and insert t_0 at that position, $B(v_r)[e_0] = t_0$. Now, given the position of e_k, in $L(v_k)$, where $B(v_k)[e_k] = t_k$, we find the position of the last $e'_k < e_k$ such that $B(v_k)[e'_k] = t_k$, in the same way as for rank queries. Once we know the position of e'_k in $L(v_k)$, we find the position of e''_{k+1} in $L(v_{k+1})$ that corresponds to e'_k. The element t_{k+1} must be inserted into $L(v_{k+1})$ immediately after e''_{k+1}, at the position of $e''_{k+1} + 1 = e_{k+1}$.

The insertion of a new element $B(v_k)[e_k] = t$ into a block $G_j(v_k)$ is supported by structure $R_j(v_k)$. We must also update structures $F_j(v_k)$, $H_j(v_k)$ and $P_t(v_k)$. These updates take $O(1)$ time, see Section 4 for structure $P_t(v_k)$ and Appendix A for the others. Since pointers are bidirectional, changes to $F_j(v_k)$ trigger changes in the F and H structures of v_{k-1} and v_{k+1}. If v_k is the root node or a leaf, we also update $K(v_k)$.

If the number of elements in $G_j(v_k)$ exceeds $2\lg^3 n$, we split $G_j(v_k)$ evenly into two blocks, $G_{j_1}(v_k)$ and $G_{j_2}(v_k)$. Then, we rebuild the data structures R, F and H for the two new blocks. Note that there are inter-node pointers to $G_j(v_k)$ that now could become dangling pointers, but all those can be known from $F_j(v_k)$, since pointers are bidirectional, and updated to point to the right places in $G_{j_1}(v_k)$ or $G_{j_2}(v_k)$. Finally, if v_k is the root or a leaf, then $K(v_k)$ is updated.

The total cost of splitting a block is dominated by that of building the new data structures R, F and H. These are easily built in $O(\lg^3 n)$ time. Since we split a block $G_j(v)$ at most once per sequence of $\Theta(\lg^3 n)$ insertions in $G_j(v)$, the amortized cost incurred by splitting a block is $O(1)$. Therefore the total cost of an insertion in $L(v)$ is $O(1)$. The insertion of a new symbol

leads to $O(\lg_\rho \sigma)$ insertions into lists $L(v)$. Updates of data structures $K(v_r)$ and $K(v_a)$ take $O(\lg n/\lg\lg n)$ time. Hence, the total cost of an insertion is $O(\lg_\rho \sigma + \lg n/\lg\lg n) = O((\lg\sigma + \lg n)/\lg\lg n)$.

We describe how deletions are handled in Section 4, where we also describe the data structure $P(v)$.

3.5 Space We show in Appendix A how to manage the data in blocks $G_j(v)$ so that all the elements stored in lists $L(v)$ use $n\lg\sigma$ bits. Since there are $O(n\lg\sigma/\lg^3 n + \sigma)$ blocks overall, all the pointers between blocks of the same lists add up to $O(n\lg\sigma/\lg^2 n + \sigma\lg n)$ bits. All the data structures $K(v)$ add up to $O(n/\lg^2 n)$ bits. We showed before that the number of inter-node pointers is $O(n\lg\sigma/\lg^{3-\varepsilon} n + \sigma\lg^\varepsilon n)$, hence all inter-node pointers (i.e., F_j and H_j structures) use $O(n\lg\sigma/\lg^{2-\varepsilon} n + \sigma\lg^{1+\varepsilon} n)$ bits. Structures $P_t(v)$ (Section 4) use $O(n\lg\sigma/\lg^{2-\varepsilon} n)$ bits as they have ρ integers per block. Finally, in Appendix A we show that each structure $R_j(v)$ uses $O(|G_j(v)|/\lg^{1-\varepsilon} n)$ extra bits. Hence, all $R_j(v)$s for all blocks and nodes use $O(n\lg\sigma/\lg^{1-\varepsilon} n)$ bits. Thus the overall space is $n\lg\sigma + O(n\lg\sigma/\lg^{1-\varepsilon} n + \sigma\lg^{1+\varepsilon} n)$ bits.

Finally, note that our structures depend on the value of $\lg n$, so they should be rebuilt when $\lceil\lg n\rceil$ changes. Mäkinen and Navarro [27] describe a way to handle this problem without affecting the space nor the time complexities, even in the worst-case scenario. The result is completed in the next section, where we describe the changes needed to implement the predecessor structures P_t.

4 Lazy Deletions and Data Structure $P(u)$

The main idea of our solution is based on lazy deletions: we do not maintain exactly S but a supersequence $\overline{S}$ of it. When a symbol $S[i] = a$ is deleted from S, we retain it in $\overline{S}$ but take a notice that $S[i] = a$ is deleted. When the number of deleted symbols exceeds a certain threshold, we expunge from the data structure all the elements marked as deleted. We define $\overline{B}(v)$ and the list $\overline{L}(v)$ for the sequence $\overline{S}$ in the same way as $B(v)$ and $L(v)$ are defined for S.

Since elements of $\overline{L}(v)$ are never removed, we can implement $P(v)$ as an insertion-only data structure. For any t, $1 \le t \le \rho$, we store information about all the blocks of a node v in a data structure $P_t(v)$. $P_t(v)$ contains one element for each block $G_j(v)$ and is implemented as an incremental split-find data structure that supports insertions and splitting in $O(1)$ amortized time and queries in $O(1)$ worst-case time [25]. The splitting positions in $P_t(v)$ are the blocks $G_j(v)$ that contain an occurrence of t, so the operation "find" in $P_t(v)$ allows us to locate, for any $G_j(v)$, the last block preceding $G_j(v)$ that contains an occurrence of t.

The insertion of a symbol t in $\overline{L}(v)$ may induce a new split in $P_t(v)$. Furthermore, overflows in a block $G_j(v)$, which convert it into two blocks $G_{j_1}(v)$ and $G_{j_2}(v)$, induce insertions in $P_t(v)$. Note that an overflow in $G_j(v)$ triggers ρ insertions in the $P_t(v)$ structures, but this $O(\rho)$ time amortizes to $o(1)$ because insertions occur every $\Theta(\lg^3 n)$ operations.

Structures $P_t(v)$ do not support "unsplitting" nor removals. The replacement of $G_j(v)$ by $G_{j_1}(v)$ and $G_{j_2}(v)$ is implemented as leaving in $P_t(v)$ the element corresponding to $G_j(v)$ and inserting one corresponding to either $G_{j_1}(v)$ or $G_{j_2}(v)$. If $G_j(v)$ contained t, then at least one of $G_{j_1}(v)$ and $G_{j_2}(v)$ contain t, and the other can be inserted as a new element (plus possibly a split, if it also contains t).

We need some additional data structures to support lazy deletions. A data structure $\overline{K}(v)$ stores the number of non-deleted elements in each block of $\overline{L}(v)$ and supports partial-sum queries. We will maintain $\overline{K}(v)$ in the root of the wavelet tree and in all leaf nodes. Moreover, we maintain a data structure $D_j(v)$ for every block $G_j(v)$, where v is either the root or a leaf node. $D_j(v)$ can be used to count the number of deleted and non-deleted elements before the i-th element in a block $G_j(v)$ for any query index i, as well as to find the index in $G_j(v)$ of the i-th non-deleted element. The implementation of $D_j(v)$ is described in Appendix A. We can use $\overline{K}(v)$ and $D_j(v)$ to find the index $\bar{i}$ in $\overline{L}(v)$ where the i-th non-deleted element occurs, and to count the number of non-deleted elements that occur before the index $\bar{i}$ in $\overline{L}(v)$.

We also store a global list *DEL* that contains, in any order, all the deleted symbols that have not yet been expunged from the wavelet tree. For any symbol $\overline{S}[i]$ in the list *DEL* we store a pointer to the offset e in $\overline{L}(v_r)$ that encodes $\overline{S}[i]$. Pointers in list *DEL* are implemented in the same way as inter-node pointers.

4.1 Queries Queries are answered very similarly to Section 3. The main idea is that we can essentially ignore deleted elements except at the root and at the leaves.

access(S, i)**:** Exactly as in Section 3, except that e_0 encodes the i-th non-deleted element in $\overline{L}(v_r)$, and is found using $\overline{K}(v_r)$ and $D_j(v_r)$.

select$_a(S, i)$**:** We find the position of the offset e_h of the i-th non-deleted element in $\overline{L}(v_h)$, where $v_h = v_a$, using $\overline{K}(v_a)$. Then we move up in the tree exactly as in Section 3. When the root node $v_0 = v_r$ is reached, we count the number of non-deleted elements that precede offset e_0 using $\overline{K}(v_r)$.

$\text{rank}_a(S,i)$**:** We find the position of the offset e_0 of the i-th non-deleted element in $\overline{L}(v_r)$. Let v_k, t_k be defined as in Section 3. In every node v_k, we find the last offset $e'_k \le e_k$ such that $\overline{B}(v_k)[e'_k] = t_k$. Note that this element may be a deleted one, but it still drives us to the correct position in $\overline{L}(v_{k+1})$. We proceed exactly as in Section 3 until we arrive at a leaf $v_h = v_a$. At this point, we count the number of non-deleted elements that precede offset e_h using $\overline{K}(v_a)$ and $D_j(v_a)$.

4.2 Updates Insertions are carried out just as in Section 3. The only difference is that we also update the data structure $D_j(v_k)$ when an element $B(v_k)[e_k]$ that encodes the inserted symbol a is added to a block $G_j(v_k)$. When a symbol $S[i] = a$ is deleted, we append it to the list *DEL* of deleted symbols. Then we visit each block $G_j(v_k)$ containing the element $B(v_k)[e_k]$ that encodes $S[i]$ and update the data structures $D_j(v_k)$. Finally, $\overline{K}(v_r)$ and $\overline{K}(v_a)$ are also updated.

When the number of symbols in the list *DEL* reaches $n/\lg^2 n$, we perform a *cleaning* procedure and get rid of all the deleted elements. Therefore *DEL* never requires more than $O(n/\lg n)$ bits.

Let $B(v_k)[e_k]$, $0 \le k \le h$, be the sequence of elements that encode a symbol $\overline{S}[i] \in DEL$. The method for tracking the elements $B(v_k)[e_k]$, removing them from their blocks $G_j(v_k)$, and updating the block structures is symmetric to the insertion procedure described in Section 3. In this case we do not need the predecessor queries to track the symbol to delete, as the procedure is similar to that for accessing $S[i]$. When the size of a block $G_j(v_k)$ falls below $(\lg^3 n)/2$ and it is not the last block of $L(v_k)$, we merge it with $G_{j+1}(v_k)$, and then split the result if its size exceeds $2\lg^3 n$. This retains $O(1)$ amortized time per deletion in any node v_k, and $O((\lg\sigma+\lg n)/\lg\lg n)$ amortized time to delete any $S[i]$.

Once all the pointers in *DEL* are processed, we rebuild from scratch the structures $P(v)$ for all nodes v. The total size of all the $P(v)$ structures is $O(\rho n \lg\sigma/\lg^3 n)$ elements. Since a data structure for incremental split-find is constructed in linear time, all the $P(v)$s are rebuilt in $O(n\lg\sigma/\lg^{3-\varepsilon} n)$ time. Hence the amortized time to rebuild the $P(v)$s is $O(\lg\sigma/\lg^{1-\varepsilon} n)$, which does not affect the amortized time $O((\lg\sigma + \lg n)/\lg\lg n)$ to carry out the effective deletions.

We are ready to state a first version of our result, not yet compressing and with times depending on σ. In Appendix A it is seen that the time for the operations is the constant $O(1/\varepsilon)$. Since the height of the wavelet tree is $\lg_\rho \sigma = O((1/\varepsilon)\lg\sigma/\lg\lg n)$, the time for all the operations on the string S is precisely $O(((1/\varepsilon^2)\lg\sigma + \lg n)/\lg\lg n)$. On the other hand, we have used blocks of size $\Theta(\lg^3 n)$ as this is the minimum that guarantees sublinear redundancy, but any larger exponent works as well. With size $\Theta(\lg^{c+3} n)$ we get the following result.

THEOREM 4.1. *A dynamic string $S[1,n]$ over alphabet $[1..\sigma]$ can be stored in a structure using $n\lg\sigma + O(n\lg\sigma/\lg^c n + \sigma\lg^{1+\varepsilon} n)$ bits, for any constants $c > 0$ and $0 < \varepsilon < 1$, and supporting queries* access, rank *and* select *in time $O(((c/\varepsilon^2)\lg\sigma + \lg n)/\lg\lg n)$. Insertions and deletions of symbols are supported in $O(((c/\varepsilon^2)\lg\sigma + \lg n)/\lg\lg n)$ amortized time.*

5 Compressed Space and Optimal Time

We now compress the space of the data structure to zero-order entropy ($nH_0(S)$ plus redundancy), while improving the time performance to the optimal $O(\lg n/\lg\lg n)$. We use Theorem 4.1 in combination with alphabet partitioning [3] to obtain the result. We then consider general alphabets, which is possible thanks to the fact that alphabet partitioning frees us from alphabet dependencies via a simple mapping.

5.1 Alphabet Partitioning We use a technique inspired by an alphabet partitioning idea [3]. To each symbol a we will assign a *level* $\ell = \lceil \lg(n/n_a) \rceil$, where a occurs n_a times in S, so that there are at most $\lg n$ levels. Additionally, we assign level $\lceil \lg n \rceil + 1$ to the symbols of Σ not present in S. For each level ℓ we will create a sequence $S^\ell[1, n_\ell]$ containing the subsequence of S formed by the symbols of level ℓ, with their alphabet remapped to $[1..\sigma_\ell]$, where σ_ℓ is the number of distinct symbols of level ℓ. We will also maintain a sequence of levels S^{lev}, so that $S^{lev}[i]$ is the level of $S[i]$. We represent S^{lev} and the S^ℓ strings using Theorem 4.1. A few arrays handle the mapping between global symbols of Σ and local symbols in strings S^ℓ: $M[1,\sigma]$ gives the level of each symbol, $N[1,\sigma]$ gives the position of that symbol inside the local alphabet of its level, and local arrays $M^\ell[1,\sigma_\ell]$ map local to global symbols. All these are represented as plain arrays. Thus a symbol $a \in \Sigma$ is represented in string S^ℓ, at level $\ell = M[a]$, where it is written as symbol $a' = N[a]$. Conversely, a symbol a' in S^ℓ corresponds to symbol $a = M^\ell[a'] \in \Sigma$.

Barbay et al. [3] show how operations access, rank, and select on S are carried out via a constant number of operations in S^{lev} and in some S^ℓ. We now extend them to insertions and deletions. To insert symbol a at position i in S, we find its level $\ell = M[a]$ and its translation $a' = N[a]$ inside S^ℓ. Now we insert ℓ at position i in S^{lev}, and a' at position $\text{rank}_\ell(S^{lev}, i)$ in S^ℓ. Deletion is similar: after mapping, we delete the position $S^\ell[\text{rank}_\ell(S^{lev}, i)]$ and then the position $S^{lev}[i]$.

If the symbol a we are inserting did not exist in S, it will be assigned the last level $\ell = \lceil \lg n \rceil + 1$ and will not appear in M^ℓ. In this case we add a at the end of M^ℓ, $M^\ell[\sigma_\ell + 1] = a$, increase σ_ℓ, set $N[a] = \sigma_\ell$ and $M[a] = \ell$. Then we proceed as in a normal insertion. Instead, if a deletion removes the last occurrence of a, we use a more global update mechanism we explain next.

Actually, we maintain levels $\ell = \lceil \lg(n/n_a) \rceil$ only approximately. First, since $\lceil \lg n \rceil$ is fixed in our data structure (see the end of Section 3.5), if we call $n' = 2^{\lceil \lg n \rceil}$, it holds $\lfloor n'/2 \rfloor < n < n'$, and use level $\ell = \lceil \lg(n'/n_a) \rceil$ for a. We also keep track of the current frequency in S of each symbol $a \in \Sigma$, n_a, and the frequency a had when it was assigned its current level, n'_a. We retain the level ℓ assigned to a as long as $n'_a/2 < n_a < 2n'_a$. When $n_a = 2n'_a$ or $n_a = \lfloor n'_a/2 \rfloor$, we move a to a new level $\ell' = \lceil \lg(n'/n_a) \rceil = \ell \pm 1$, as follows. We compute the mapping $a' = N[a]$ of a in S^ℓ, change $M[a]$ to ℓ', and compute the new mapping $a'' = \sigma_{\ell'} + 1$ of a in $S^{\ell'}$. Now, for each of the n_a occurrences of a' in S^ℓ, say $S^\ell[i] = a'$ (found using $i = \mathrm{select}_{a'}(S^\ell, 1)$), we compute its position $j = \mathrm{select}_\ell(S^{lev}, i)$ in S^{lev}, change $S^{lev}[j]$ to ℓ', remove symbol $S^\ell[i]$, and insert symbol a'' in $S^{\ell'}$ at position $\mathrm{rank}_{\ell'}(S^{lev}, j)$. We also update the mappings: we set $M^{\ell'}[a''] = a$ and $N[a] = a''$, and move the last element of M^ℓ to occupy the empty slot left by a: $M^\ell[a'] = M^\ell[\sigma_\ell]$ and $N[b] = a'$, where $b = M^\ell[\sigma_\ell]$. We find all the occurrences of σ_ℓ in S^ℓ and replace them by a'. Finally, we increase $\sigma_{\ell'}$ and decrease σ_ℓ. When $n_a = 0$, we delete it from S^ℓ instead of moving it. Finally, we also rebuild each sequence S^ℓ periodically: we remember the number of symbols n'_ℓ in S^ℓ at the last time we built it, and rebuild S^ℓ when $n_\ell = \lfloor n'_\ell/2 \rfloor$ or $n_\ell = 2n'_\ell$.

The number of insertions or deletions that must occur until we change the level of a is $n'_a/2 = \Theta(n_a)$. Therefore, the process of changing a symbol from one level to another, which costs $O(n_a)$ update operations on S^{lev}, S^ℓ, M^ℓ, M and N, is amortized over $\Theta(n_a)$ updates. The same occurs with the symbol b mapped to σ_ℓ in S^ℓ, whose occurrences have to be re-encoded as a': Since $\lceil \lg(n'/n'_b) \rceil = \lceil \lg(n'/n'_a) \rceil$, it holds $n_b = \Theta(n_a)$. The rebuilds of S^ℓ and S amortize in the same way.

Note that we are letting the alphabet of the sequences S^ℓ grow and shrink, which our wavelet trees do not support. Rather, we create them with the maximum possible alphabet size $\overline{\sigma}_\ell \geq \sigma_\ell$. Since $\ell = \lceil \lg(n'/n'_a) \rceil = \lceil \lg(n'/n'_b) \rceil$ for any pair of symbols a, b mapped to S^ℓ, it follows that $n'_b > n'_a/2$. Since we retain that level ℓ for them as long as $n'_b/2 < n_b < 2n'_b$, it follows that $n_b > n'_a/4$, and thus there cannot be more than $4n_\ell/n'_a$ distinct symbols in S^ℓ. Since, on the other hand, $n_\ell < 2n'_\ell$, we can safely set the maximum alphabet size for S^ℓ to $\overline{\sigma}_\ell = 8n'_\ell/n'_a$ for any a. A bound in terms of ℓ is $n'_a \geq n'/2^\ell$, thus we set $\overline{\sigma}_\ell = 2^{\ell+3} n'_\ell/n'$. Note it holds $\overline{\sigma}_\ell = O(n_\ell/n_a)$ for any a mapped to S^ℓ. Note also that the *effective* alphabet (i.e., symbols actually occurring) of S^ℓ is of size $\sigma_\ell \geq n_\ell/(4n'_a) \geq n'_\ell/(8n'_a) = \overline{\sigma}_\ell/64$.

5.2 Time and Space The queries on S^{lev} take $O(\lg n/\lg\lg n)$ time, because its alphabet is of size $O(\lg n)$. Queries on S^ℓ take $O((\lg \overline{\sigma}_\ell + \lg n)/\lg\lg n) = O(\lg n/\lg\lg n)$ time, since $\overline{\sigma}_\ell = O(n)$. The accesses to M, N, and M^ℓ are constant-time. Therefore, we reach the optimal worst-case time $O(\lg n/\lg\lg n)$ for the three queries. Likewise, updates cost $O(\lg n/\lg\lg n)$ amortized time.

Let us now consider the space. Each symbol a with frequency n_a will be stored at a level $\ell = \lceil \lg(n/n_a) \rceil \pm 2$, in a sequence over an alphabet of size $\overline{\sigma}_\ell$. Therefore, we will spend $n_a \lg \overline{\sigma}_\ell + O(n_a \lg \overline{\sigma}_\ell / \lg^2 n)$ bits for it, according to Theorem 4.1 (we use $\lg n$ instead of $\lg n_a$ to define superblock sizes; we consider soon the rest of the space overhead). This is $n_a \lg(n_\ell/n_a) + O(n_a)$ bits, which added over the whole S^ℓ yields $\sum n_a \lg(n_\ell/n_a) + O(n_a)$ bits. Now consider the occurrences of symbol ℓ in S^{lev}, which we will also charge to S^ℓ. These cost $n_\ell \lg(n/n_\ell) + O(n_\ell \lg\lg n/\lg^2 n) = n_\ell \lg(n/n_\ell) + o(n_\ell)$. Added to the space spent at S^ℓ itself, and since the sum of the n_a's is n_ℓ, we obtain $\sum n_a \lg(n/n_a) + O(n_\ell)$ bits. Now, adding over the symbols a of all the levels, we obtain the total space $nH_0(S) + O(n)$.

Theorem 4.1 also involves a cost of $O(\overline{\sigma}_\ell \lg^{1+\varepsilon} n)$ bits per level ℓ, which add up to $O(\sigma \lg^{1+\varepsilon} n)$ since $\overline{\sigma}_\ell = \Theta(\sigma_\ell)$, and $\sum_\ell \sigma_\ell = \sigma$.

In addition we spend $O(\sigma(\lg\lg n + \lg\sigma))$ bits for the arrays M, N and M^ℓ. Finally, recall that we also spend space in storing deleted symbols, but these are at most $O(n/\lg^2 n)$, and thus they cannot increase the entropy by more than $O(n/\lg n)$. This gives the final result.

THEOREM 5.1. *A dynamic string $S[1,n]$ over alphabet $[1..\sigma]$ can be stored in a structure using $nH_0(S) + O(n + \sigma(\lg\sigma + \lg^{1+\varepsilon} n))$ bits, for any constant $0 < \varepsilon < 1$, and supporting queries* access, rank *and* select *in optimal time $O((1/\varepsilon^2)\lg n/\lg\lg n)$. Insertions and deletions of symbols are supported in $O((1/\varepsilon^2)\lg n/\lg\lg n)$ amortized time.*

5.3 Handling General Alphabets Our time results do not depend on the alphabet size σ, yet our space does, in a way that ensures that σ gives no problems as long as $\sigma = O(n/\lg^{1+\varepsilon} n)$ for some constant $\varepsilon > 0$.

Let us now consider the case where the alphabet Σ is much larger than the *effective* alphabet of the string, that is, the set of symbols that actually appear in S at a given point in time. Let us now use $\sigma \leq n$ to denote

the effective alphabet size. Our aim is to maintain the space within $nH_0(S) + O(n + \sigma \lg^{1+\varepsilon} n)$ bits, even when the symbols come from a large universe $\Sigma = [1..|\Sigma|]$, or even from a general ordered universe such as $\Sigma = \mathbb{R}$ or $\Sigma = \Gamma^*$ (i.e., Σ are words over another alphabet Γ).

Our arrangement into strings S^ℓ gives a simple way to handle a sequence over an unbounded ordered alphabet. By changing tables M and N to custom structures to search Σ, and storing elements of Σ in arrays M^ℓ, we obtain the following result.

THEOREM 5.2. *A dynamic string $S[1, n]$ over a general alphabet Σ can be stored in a structure using $nH_0(S) + O(n + \mathcal{S}(\sigma) + \sigma \lg^{1+\varepsilon} n)$ bits, for any constant $0 < \varepsilon < 1$, and supporting queries* access, rank *and* select *in time $O(\mathcal{T}(\sigma) + (1/\varepsilon^2) \lg n / \lg \lg n)$. Insertions and deletions of symbols are supported in $O(\mathcal{U}(\sigma) + (1/\varepsilon^2) \lg n / \lg \lg n)$ amortized time. Here σ is the number of distinct symbols of Σ occurring in S, $\mathcal{S}(\sigma)$ is the number of bits used by a dynamic data structure to search over σ elements in Σ plus to refer to σ elements in Σ, $\mathcal{T}(\sigma)$ is the worst-case time to search for an element among σ of them in Σ, and $\mathcal{U}(\sigma)$ is the amortized time to insert/delete symbols of Σ in the structure.*

For example, if $\Sigma = \mathbb{R}$ we have $O(\lg \sigma + \lg n / \lg \lg n)$ times, which is optimal in the comparison model.

An interesting particular case is $\Sigma = \Gamma^*$ on a general alphabet Γ, where we can store the effective set of strings in a data structure by Franceschini and Grossi [16], so that operations involving a string p take $O(|p| + \lg \gamma + \lg n / \lg \lg n)$, where γ is the number of symbols of Γ actually in use.

Another particular case is that Σ is an integer range $[1..|\Sigma|]$, then time can be reduced to $O(\lg \lg |\Sigma| + \lg n / \lg \lg n)$ and the space increases by $O(\sigma \lg |\Sigma|)$ bits, by using y-fast tries [36].

Yet another important particular case is when we maintain a contiguous effective alphabet $[1..\sigma]$, and only insert new symbols $\sigma+1$. In this case there is no penalty for letting the alphabet grow dynamically.

6 Applications

Our new results impact in a number of applications that build on dynamic sequences. We describe several here.

6.1 Dynamic Sequence Collections The standard application of dynamic sequences, stressed out in several previous papers [12, 27, 19, 30], is to maintain a collection $\mathcal{C}$ of texts, where one can carry out indexed pattern matching, as well as inserting and deleting texts from the collection. Plugging in our new representation we can improve the time and space of previous work (yet our update time is amortized).

THEOREM 6.1. *There exists a data structure for handling a collection $\mathcal{C}$ of texts over an alphabet $[1, \sigma]$ within size $nH_h(\mathcal{C}) + O(n + \sigma^{h+1} \lg n + m \lg n)$ bits, simultaneously for all h. Here n is the length of the concatenation of m texts, $\mathcal{C} = T_1 \circ T_2 \cdots \circ T_m$, and we assume that the alphabet size is $\sigma = o(n)$. The structure supports counting of the occurrences of a pattern P in $O(|P| \lg n / \lg \lg n)$ time. After counting, any occurrence can be located in time $O(\lg^2 n / \lg \lg n)$. Any substring of length ℓ from any T in the collection can be displayed in time $O((\ell + \lg n) \lg n / \lg \lg n))$. Inserting or deleting a text T takes $O(\lg n + |T| \lg n / \lg \lg n)$ amortized time. For $0 \le h \le (\alpha \lg_\sigma n) - 1$, for any constant $0 < \alpha < 1$, the space simplifies to $nH_h(\mathcal{C}) + O(n + m \lg n)$ bits.*

The theorem refers to $H_h(\mathcal{C})$, the h-th order empirical entropy of sequence $\mathcal{C}$ [28]. This is a lower bound to any semistatic statistical compressor that encodes each symbol as a function of the h preceding symbols in the sequence, and it holds $H_h(\mathcal{C}) \le H_{h-1}(\mathcal{C}) \le H_0(\mathcal{C}) \le \lg \sigma$ for any $h > 0$. To offer search capabilities, the Burrows-Wheeler Transform (BWT) [11] of $\mathcal{C}$, $\mathcal{C}^{bwt}$, is represented, not $\mathcal{C}$. Kärkkäinen and Puglisi [26] showed that, if $\mathcal{C}^{bwt}$ is split into superblocks of size $\Theta(\sigma \lg^2 n)$, and a zero-order compressed representation is used for each superblock, the total bits are $nH_h(\mathcal{C}) + o(n)$.

We use their partitioning, and Theorem 5.1 to represent each superblock. The superblock sizes are easily maintained upon insertions and deletions of symbols, by splitting and merging superblocks and rebuilding the structures involved, without affecting the amortized time per operation. They also need to manage a table storing the rank of each symbol up to the beginning of each superblock. This is arranged, in the dynamic scenario, with σ partial sum data structures containing $O(n/(\sigma \lg^2 n))$ elements each, plus another one storing the superblock lengths. This adds $O(n / \lg n)$ bits and $O(\lg n / \lg \lg n)$ time per operation.

Finally, the locating and displaying overheads are obtained by marking one element out of $\lg n$, so that the space overhead of $O(n)$ is maintained.

6.2 Burrows-Wheeler Transform Another application of dynamic sequences is to build the BWT of a text T, T^{bwt}, within compressed space, by starting from an empty sequence and inserting each new character, $T[n]$, $T[n-1]$, ..., $T[1]$, at the proper positions. The result is stated as the compressed construction of a static FM-index [15], a compressed index that consists essentially of a (static) wavelet tree of T^{bwt}. Our new representation improves upon the best previous result on compressed space [30].

THEOREM 6.2. *The Alphabet-Friendly FM-index [15],*

as well as the BWT [11], of a text $T[1,n]$ over an alphabet of size σ, can be built using $nH_h(T) + O(n)$ bits, simultaneously for all $1 \leq h \leq (\alpha \lg_\sigma n) - 1$ and any constant $0 < \alpha < 1$, in time $O(n \lg n / \lg\lg n)$. It can also be built within the same time and $nH_0(T) + O(n + \sigma(\lg\sigma + \lg^{1+\varepsilon} n))$ bits, for any constant $\varepsilon > 0$ and any alphabet size σ.

We are using Theorem 6.1 for the case $h > 0$, and Theorem 5.1 to obtain a less alphabet-restrictive result for $h = 0$. This is the first time $o(n \lg n)$ time is obtained within compressed space. Other space-conscious results that achieve better time complexity (but more space) are Okanohara and Sadakane [32], who achieved optimal $O(n)$ time within $O(n \lg\sigma \lg\lg_\sigma n)$ bits, and Hon et al. [24], who achieved $O(n \lg\lg\sigma)$ time and $O(n \lg\sigma)$ bits.

6.3 Binary Relations Barbay et al. [4] show how to represent a binary relation of t pairs relating n "objects" with σ "labels" by means of a string of t symbols over alphabet $[1..\sigma]$ plus a bitmap of length $t+n$. The idea is to traverse the matrix, say, object-wise, and write down in a string the labels of the pairs found. Meanwhile we append a 1 to the bitmap each time we find a pair and a 0 each time we move to the next object. Then queries like: find the objects related to a label, find the labels related to an object, and tell whether an object and a label are related, are answered via access, rank and select operations on the string and the bitmap.

A limitation in the past to make this representation dynamic was that creating or removing labels implied changing the alphabet of the string. Now we can use Theorem 5.1 and the results of Section 5.3 to obtain a fully dynamic representation. We illustrate the case where labels and objects come from finite universes.

THEOREM 6.3. *A dynamic binary relation consisting of t pairs relating n objects from $[1..N]$ with σ labels from $[1..L]$ can support the operations of counting and listing the objects related to a given label, counting and listing the labels related to a given object, and telling whether an object and a label are related, all in time $O(\lg\lg(NL) + \lg(n+t)/\lg\lg(n+t))$ per delivered datum. Pairs, objects and labels can also be added and deleted in amortized time $O(\lg\lg(NL) + \lg(n+t)/\lg\lg(n+t))$. The space required is $tH + n\lg N + \sigma\lg L + O(t + n + \sigma(\lg\sigma + \lg^{1+\varepsilon} t))$ bits, where $\varepsilon > 0$ is any constant and $H = \sum_{1\leq i\leq\sigma}(t_i/t)\lg(t/t_i) \leq \lg\sigma$, where t_i is the number of objects related to label i. Only labels and objects with no related pairs can be deleted.*

6.4 Directed Graphs A particularly interesting and general binary relation is a directed graph with n nodes and e edges. Our binary relation representation allows one to navigate it in forward and backward direction, and modify it, within little space.

THEOREM 6.4. *A dynamic directed graph consisting of n nodes in $[1..N]$ and e edges can support the operations of counting and listing the neighbors pointed from a node, counting and listing the reverse neighbors pointing to a node, and telling whether there is a link from one node to another, all in time $O(\lg\lg N + \lg(n + e)/\lg\lg(n + e))$ per delivered datum. Nodes and edges can be added and deleted in amortized time $O(\lg\lg N + \lg(n + e)/\lg\lg(n + e))$. The space required is $eH + n\lg N + O(e + n(\lg n + \lg^{1+\varepsilon} e))$ bits, where ε is any constant and $H = \sum_{1\leq i\leq n}(e_i/e)\lg(e/e_i) \leq \lg n$, where e_i is the outdegree of node i.*

If we only modify edges and the nodes are fixed, the overheades related to N disappear. Note also that we can change "outdegree" by "indegree" in the theorem by representing the transposed graph, as operations are symmetric. We can similarly transpose general binary relations.

6.5 Inverted Indexes Finally, we consider an application where the symbols are words. Take a text T as a sequence of n *words*, which are strings over a set of letters Γ. The alphabet of T is $\Sigma = \Gamma^*$, and its effective alphabet is called the *vocabulary* V of T, of size $|V| = \sigma$. A *positional inverted index* is a data structure that, given a word $w \in V$, tells the positions in T where w appears [1].

A well known way to simulate a positional inverted index within no extra space on top of the compressed text is to use a compressed sequence representation for T (over alphabet Σ), so that operation $\text{select}_w(T, i)$ simulates access to the ith position of the list of word w, whereas access to the original T is provided via $\text{access}(T, i)$. Operation rank can be used to emulate various inverted index algorithms, particularly for intersections [6]. The space is the zero-order entropy of the text seen as a sequence of words, which is very competitive in practice. Our new technique permits modifying the underlying text, that is, it simulates a dynamic inverted index. For this sake we use the technique of Section 5.3 and tries to handle a vocabulary over a fixed alphabet.

THEOREM 6.5. *A text of n words with a vocabulary of σ words and total length ν over a fixed alphabet can be represented within $nH_0(T)+O(n+\nu\lg n+\sigma\lg^{1+\varepsilon} n)$ bits of space, where $\varepsilon > 0$ is an arbitrary constant and $H_0(T)$ is the word-wise entropy of T. The representation outputs any word $T[i] = w$ given i, finds the position of the ith occurrence of any word w, and tells the number of occurrences of any word w up to position*

i, all in time $O(|w| + \lg n/\lg\lg n)$. *A word* w *can be inserted or deleted at any position in* T *in amortized time* $O(|w| + \lg n/\lg\lg n)$.

Another kind of inverted index, a *non-positional* one, relates each word with the documents where it appears (not to the exact positions). This can be seen as a direct application of our binary relation representation [2], and our dynamization theorems apply to it as well.

7 Conclusions and Further Challenges

We have obtained $O(\lg n/\lg\lg n)$ time for all the operations that handle a dynamic sequence on an arbitrary (known) alphabet $[1..\sigma]$, matching lower bounds that apply to binary alphabets [17]. Our structure is faster than previous work [22, 30] by a factor of $\Theta(1 + \lg\sigma/\lg\lg n)$. It also reduces the redundancy space, using $nH_0(S) + O(n + \sigma(\lg\sigma + \lg^{1+\varepsilon} n))$ bits, instead of $nH_0(S)+o(n\lg\sigma)+O(\sigma(\lg\sigma+\lg n))$ of previous work. We also show how to handle general alphabets. Our result can be applied to a number of problems; we have described several ones.

The only remaining advantage of previous work [22, 30] is that their update times are worst-case, whereas in our structure they are amortized. Obtaining optimal worst-case time complexity for updates is an interesting future challenge.

Another challenge is to simulate other operations than access, rank and select. Obtaining the full functionality of wavelet trees with better time than the current dynamic ones [22, 30] is unlikely, as discussed in the Introduction. Yet, there may be some intermediate functionality of interest.

References

[1] R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011.

[2] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. 9th LATIN*, LNCS 6034, pages 170–183, 2010.

[3] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st ISAAC*, pages 315–326 (part II), 2010.

[4] J. Barbay, A. Golynski, I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theor. Comp. Sci.*, 387(3):284–297, 2007.

[5] J. Barbay, M. He, I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. *ACM Trans. Alg.*, 7(4):article 52, 2011.

[6] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. 26th STACS*, pages 111–122, 2009.

[7] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. 20th ESA*, LNCS 7501, pages 181–192, 2012.

[8] G. E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proc. 19th SODA*, pages 894–903, 2008.

[9] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proc. 11th WADS*, pages 98–109, 2009.

[10] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st SIGIR*, pages 139–146, 2008.

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

[12] H. Chan, W. Hon, T. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Trans. Alg.*, 3(2):21, 2007.

[13] F. Claude and G. Navarro. Extended compact Web graph representations. In *Algorithms and Applications (Ukkonen Festschrift)*, pages 77–91. Springer, 2010.

[14] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[15] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.

[16] G. Franceschini and R. Grossi. A general technique for managing strings in comparison-driven data structures. In *Proc. 31st ICALP*, LNCS 3142, pages 606–617, 2004.

[17] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.

[18] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.

[19] R. González and G. Navarro. Rank/select on dynamic compressed sequences and applications. *Theor. Comp. Sci.*, 410:4414–4422, 2009.

[20] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

[21] A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. A framework for dynamizing succinct data structures. In *Proc. 34th ICALP*, pages 521–532, 2007.

[22] M. He and I. Munro. Succinct representations of dynamic strings. In *Proc. 17th SPIRE*, pages 334–346, 2010.

[23] W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums. In *Proc. 14th ISAAC*, pages 505–516, 2003.

[24] W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *SIAM J. Comp.*, 38(6):2162–2178, 2009.

[25] H. Imai and T. Asano. Dynamic segment intersection search with applications. In *Proc. 25th FOCS*, pages 393–402, 1984.

[26] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. 18th SPIRE*, LNCS 7024, pages 174–184, 2011.
[27] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Tran. Alg.*, 4(3):article 32, 2008.
[28] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
[29] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log n)$ time. *J. Comp. Sys. Sci.*, 33(1):66–74, 1986.
[30] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *CoRR*, abs/0905.0768v5, 2010. To appear in *ACM Trans. Alg.*
[31] Y. Nekrich. A dynamic stabbing-max data structure with sub-logarithmic query time. In *Proc. 22nd ISAAC*, pages 170–179, 2011.
[32] D. Okanohara and K. Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *Proc. 16th SPIRE*, LNCS 5721, pages 90–101, 2009.
[33] M. Patrascu. Lower bounds for 2-dimensional range counting. In *Proc. 39th STOC*, pages 40–46, 2007.
[34] R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th ICALP*, pages 357–368, 2003.
[35] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th CPM*, pages 205–215, 2007.
[36] D. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Proc. Lett.*, 17(2):81–84, 1983.

A Data Structures for Handling Blocks

We describe the way the data is stored in blocks $G_j(v)$, as well as the way the various data structures inside blocks operate. All the data structures are based on the same idea: We maintain a tree with node degree $\lg^\delta n$ and leaves that contain $o(\lg n)$ elements. Since elements within a block can be addressed with $O(\lg\lg n)$ bits, each internal node and each leaf fits into one machine word. Moreover, we can support searching and basic operations in each node in constant time.

A.1 Data Organization The block data is physically stored as a sequence of *miniblocks* of $\Theta(\lg_\rho n)$ symbols. Thus there are $O(\lg^2 n \lg\rho) = O(\lg^2 n \lg\lg n)$ miniblocks in a block. These miniblocks will be the leaves of a τ-ary tree T, for $\tau = \Theta(\lg^\delta n)$ and some constant $0 < \delta < 1$. The height of this tree is constant, $O(1/\delta)$. Each node of T stores τ counters telling the number of symbols stored at the leaves that descend from each child. This requires just $O(\tau \lg\lg n) = o(\lg n)$ bits. To access any position of $G_j(v)$, we descend in T, using the counters to determine the correct child. When we arrive at a leaf, we know the local offset of the desired symbol within the leaf, and can access it directly. Since the counters fit in less than a machine word, a small universal table gives the correct child in constant time, therefore we have $O(1)$ time access to any symbol (actually to any $\Theta(\lg_\rho n)$ consecutive symbols).

Upon insertions or deletions, we arrive at the correct leaf, insert or delete the symbol (in constant time because the leaf contains $\Theta(\lg n)$ bits overall), and update the counters in the path from the root (in constant time as they have $o(\lg n)$ bits). The leaves may have $\lg n$ to $2\lg n$ bits. Splits/merges upon overflows/underflows are handled as usual, and can be solved in a constant number of $O(1)$-time operations (T operates as a B-tree; internal nodes may have τ to 2τ children).

The space overhead due to the nodes of T is $O(|G_j(v)|\lg^\delta n \lg\lg n/\lg n)$ bits, where we also measure $|G_j(v)|$ in bits, not symbols. We consider now the space used by the data itself.

In order not to waste space, the miniblock leaves are stored using a memory management technique by Munro [29]. For our case, it allows us to allocate, free, and access miniblocks of length $\lg n$ to $2\lg n$ in constant time. Its space waste, given that our pointers are of $O(\lg\lg n)$ bits, is $O(\lg\lg n)$ per allocated miniblock, which adds up to $O(|G_j(v)|\lg\lg n/\lg n)$, plus a global redundancy of $O(\lg^2 n)$ bits. We use one structure per block, handling its miniblocks, so the global redundancy adds just $O(n\lg_\rho\sigma/\lg n)$ bits overall.

Each structure uses a memory area of fixed-size cells (inside which the variable-length miniblocks are stored) that grows or shrinks at the end as miniblocks are created or destroyed. A structure giving that functionality is called an *extendible array (EA)* [34]. We need to handle a set of $O(n\lg_\rho\sigma/\lg^3 n)$ EAs, what is called a *collection of extendible arrays*. Its functionality includes accessing any cell of any EA, letting it grow or shrink by one cell, and create and destroy EAs. The following lemma, simplified from the original [34, Lemma 1], and using words of $\lg n$ bits, is useful.

LEMMA A.1. *A collection of a EAs of total size s bits can be represented using $s + O(a\lg n + \sqrt{sa\lg n})$ bits of space, so that the operations of creation of an empty EA and access take constant worst-case time, whereas grow/shrink take constant amortized time. An EA of s' bits can be destroyed in time $O(s'/\lg n)$.*

In our case $a = O(n\lg_\rho\sigma/\lg^3 n)$ and $s = O(n\lg\sigma)$, so the space overhead posed by the EAs is $O(n\lg_\rho\sigma/\lg^2 n + n\lg\sigma/(\lg n\sqrt{\lg\lg n})) = o(n\lg\sigma/\lg n)$.

A.2 Structure $R_j(v)$ To support rank and select we enrich T with further information per node. We store ρ counters with the number of occurrences of each symbol

in the subtree of each child. The node size becomes $O(\tau\rho \lg\lg n) = O(\lg^{\varepsilon+\delta} n \lg\lg n) = o(\lg n)$ as long as $\varepsilon + \delta < 1$. This dominates the total space overhead, which becomes $O(|G_j(v)| \lg^{\varepsilon+\delta} n \lg\lg n / \lg n)$.

With this information on the nodes we can easily solve rank and select in constant time, by descending on T and determining the correct child (and accumulating data on the leftward children) in $O(1)$ time using universal tables. Nodes can also be updated in constant time even upon splits and merges, since all the counters can be recomputed in $O(1)$ time.

A.3 Structure $F_j(v)$ This structure stores all the inter-node pointers leaving from block $G_j(v)$, to its parent and to any of the ρ children of node v.

The structure is a tree T_f very similar in spirit to T. The pointers stored are inter-node, and thus require $\Theta(\lg n)$ bits. Thus we store a constant number of pointers per leaf. For each pointer we store the position in $G_j(v)$ holding the pointer (relative to the starting position of the leaf node inside $G_j(v)$) and the target position. The internal nodes, of arity τ, maintain information on the number of positions of $G_j(v)$ covered by each child, and the number of pointers of each kind ($1 + \rho$ counters) stored in the subtree of each child. This requires $O(\tau\rho \lg\lg n) = o(\lg n)$ bits, as before. To find the last position before i holding a pointer of a certain kind (parent or t-th wavelet tree child, for any $1 \le t \le \rho$), we traverse T_f from the root looking for position i. At each node u, it might be that the child u' where we have to enter holds pointers of that kind, or not. If it does, then we first enter into child u'. If we return with an answer, we recursively return it. If we return with no answer, or there are no pointers of the desired kind below u', we enter into the last sibling to the left of u' that holds a pointer of the desired kind, and switch to a different mode where we simply go down the tree looking for the rightmost child with a pointer of the desired kind. It is not hard to see that this procedure visits $O(1/\delta)$ nodes, and thus it is constant-time because all the computations inside nodes can be done in $O(1)$ time with universal tables. When we arrive at the leaf, there may be at most two pointers associated to the desired position (one to the parent and another to a wavelet tree child), so we can scan for the desired pointer in constant time.

The tree T_f must be updated when a symbol t is inserted before any other occurrence of t in $G_j(v)$, when a symbol is inserted at the first position of $G_j(v)$ and, due to the bidirectionality, when pointers to $G_j(v)$ are created from the parent or a child of v. It must be updated analogously when deletion of pointers occur. Those updates work just like on the tree T. T_f is also updated upon insertions and deletions of symbols, even if they do not have pointers, to maintain the positions up to date. In this case we traverse T_f looking for the position of the update, change the offsets stored at the leaf, and update the subtree sizes stored at the nodes.

A.4 Structure $H_j(v)$ This structure manages the inter-node pointers that point inside $G_j(v)$. As explained in Section 3.4, we give a handle to the outside nodes, that does not change over time, and $H_j(v)$ translates handles to positions in $G_j(v)$.

We store a tree T_h that is just like T_f, where the incoming pointers are stored. T_h is simpler, however, because at each node we only need to store the number of positions covered by the subtree of each child. Also, it is possible to traverse T_h from a leaf to the root. We also manage a table Tbl so that $Tbl[h]$ points to the leaf where the pointer corresponding to handle h is stored. At the leaves we store, for each pointer, a backpointer to Tbl and the position in $G_j(v)$ (in relative form). Given a handle h, we go to the leaf, find in constant time the one pointing back to h, and move upwards up to the root, adding to the position the number of positions covered by leftward children of each node. At the end we have obtained the position in constant time.

When pointers to $G_j(v)$ are created or destroyed, we insert or remove pointers in T_h. We maintain a list of empty cells in Tbl for future handles. We must also update T_h upon symbol insertions and deletions in $G_j(v)$, to maintain the positions up to date. When a leaf splits or merges, we update the pointers from a constant number of positions in Tbl, found with the backpointers.

Tbl may contain up to $\Theta(\lg^3 n)$ pointers of $O(\lg\lg n)$ bits, but there can be only $O(n \lg\sigma / \lg^3 n)$ pointers in the structure, adding up to $s = O(n \lg\sigma \lg\lg n / \lg^3 n)$ bits, spread across $a = O(n \lg_\rho \sigma / \lg^3 n)$ tables Tbl. Using again Lemma A.1, a collection of EAs poses an overhead of $o(n \lg\sigma / \lg^2 n)$.

A.5 Structure $D_j(v)$ This is a simple tree T_d similar to T, storing at each node the number of positions and the number of non-deleted positions below each child. It should be obvious how it operates.

A.6 The Final Result While the raw data adds up to $n \lg\sigma$ bits, the space overhead adds up to $O(n \lg\sigma \lg^{\varepsilon+\delta} n \lg\lg n / \lg n)$. By rewriting $\delta = \varepsilon$ as the original value of $\varepsilon/2$ and adjusting it infinitesimally, we have that the overhead is $O(n \lg\sigma / \lg^{1-\varepsilon} n)$ bits, for any $0 < \varepsilon < 1$. The time for the operations is, in all cases, $O(1/\delta) = O(1/\epsilon)$.