Dynamic Compressed Strings with Random Access

Roberto Grossi[1], Rajeev Raman[2], S. Srinivasa Rao[3], and Rossano Venturini[1]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Department of Computer Science, University of Leicester, UK
[3] School of Computer Science and Engineering, Seoul National University, Korea

Abstract We consider the problem of storing a string S in dynamic compressed form, while permitting operations directly on the compressed representation of S: access a substring of S; replace, insert or delete a symbol in S; count how many occurrences of a given symbol appear in any given prefix of S (called rank operation) and locate the position of the ith occurrence of a symbol inside S (called select operation). We discuss the time complexity of several combinations of these operations along with the entropy space bounds of the corresponding compressed indexes. In this way, we extend or improve the bounds of previous work by Ferragina and Venturini [TCS, 2007], Jansson et al. [ICALP, 2012], Nekrich and Navarro [SODA, 2013].

1 Introduction

The volume of unstructured, semi-structured, and replicated data, such as textual data, text with markup, and data backup and log analysis, has been growing much faster than structured (relational) data in recent years [IDC's "Digital Universe Survey", 2011]. Such non-relational data is commonly viewed as a (often highly compressible) string, and the processing of this data is not always amenable to external-memory solutions. Considerations like these, and the common practice of storing important data in the main memory of a computing cluster, have motivated the development of *compressed string storage schemes* [1,2,3], which store a string $S[1,n]$ from the alphabet $[\sigma] = \{1,\ldots,\sigma\}$ in compressed form, while allowing random access to the string via the operation:

$\mathsf{Access}(i,m)$**:** return the substring $S[i, i+m-1]$ for $m \geq 1$ and $1 \leq i \leq n-m+1$.

These schemes have been developed on the standard *RAM* model with a word size of $\Theta(\lg n)$ bits. They support Access optimally (as a substring of length $\ell = \Theta(\lg_\sigma n)$ fits in $O(1)$ words, $\mathsf{Access}(i,\ell)$ executes in $O(1)$ time) and aim to minimize the *space* in bits, expressed as the sum of two components that, when S is compressible, is smaller than the $n \lg \sigma$ bits used by S in 'raw' form:

– $nH_k(S)$ where $H_k(S)$ is the *k-th order empirical entropy* of S (a measure of compressibility, see the end of this section), and $k \geq 0$ is an integer, and
– the *redundancy*, or any additional space required to support random access.

Table 1: Summary of discussed results. Here $\ell = \Theta(\lg_\sigma n)$, $\rho = (k \lg \sigma + \lg \lg n)/\lg_\sigma n$, $\lambda = \lg n/\lg \lg n$, $\delta = \min\{\lg_\sigma n, (k+1)\lambda\}$, $\dagger$ = time to Access *one* symbol.

Access(i, ℓ)	Replace	Insert/Delete	Rank/Select	space (bits)	ref.
$O(1)$	—	—	—	$n(H_k + O(\rho))$	[1,2,3]
$O(1)$	$O(\delta)$	—	—	$n(H_k + O(\rho))$	[5]
$O(1)$	$O(1/\epsilon)$	—	—	$n(H_k + O(\epsilon(k+1) \lg \sigma + \rho))$	[5]
$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	—	—	$\boldsymbol{n(H_k + O(\rho))}$	**Thm.1**
$O(\lambda)$	$O(\lambda)$	$O(\lambda)$	—	$n(H_0 + O(\lg \lg n/\lg_\sigma n))$	[5]
$O(\lambda)$	$O(\lambda)$	$O(\lambda)$	—	$n(H_k + O(\rho + k \lg \lg n/\lg n))$	[5]
$\boldsymbol{O(\lambda)}$	$\boldsymbol{O(\lambda)}$	$\boldsymbol{O(\lambda)}$	—	$\boldsymbol{n(H_k + O(\lg \lg n/\lg_\sigma n))}$	**Thm.2**
$O(\lambda)^\dagger$	$O(\lambda)$	$O(\lambda)$	$O(\lambda)$	$n(H_0 + O(1)) + O(\sigma(\lg \sigma + (\lg n)^{1+\epsilon}))$	[6]
$\boldsymbol{O(1)}$	$\boldsymbol{O(\lambda)}$	—	$\boldsymbol{O(\lambda)}$	$\boldsymbol{n(H_k + O(\lg \lg \sigma + \rho \lg \lg n))}$	**Cor.1**

The redundancy is a quantity of significant fundamental interest, particularly for lower bounds (see [4] and references therein), and is critical in practice. The best space upper bound is currently $n(H_k(S) + O(\rho(k, \sigma, n)))$ bits, which holds *simultaneously* for all $0 \le k \le \lg_\sigma n$, where $\rho(k, \sigma, n) = \frac{k \lg \sigma + \lg \lg n}{\lg_\sigma n}$. As k increases, the H_k term decreases, but ρ increases. However, so long as $k = o(\lg_\sigma n)$, $\rho = o(n \lg \sigma)$ is asymptotically smaller than S in 'raw' form. The data structure of [1] attains the above space bound and supports Access in $O(1)$ time. We now describe our contributions in the context of related work.

Dynamic Access*-only Sequences.* The storage schemes of [1,2,3] are all *static*, i.e., do not permit changes to S (see Table 1, first row). Although there has been work on storing dynamic sequences in a compressed format, the gap (in compression performance and Access time) with the static storage schemes remained large, until a recent breakthrough result by Jansson et al. [5]. They considered expanding the repertoire to include, in addition to Access:

Replace(i, c)**:** replace $S[i]$ by a symbol $c \in [\sigma]$.
Insert(i, c)**:** insert the symbol c into S between positions $i-1$ (if it exists) and i, and make S one symbol longer.
Delete(i)**:** delete $S[i]$ and make S one symbol shorter.

Jansson et al. gave two schemes (Table 1, rows 2 and 3) that achieve "high-order" compression and $O(1)$ time for Access(i, ℓ), but effectively do not support $O(1)$-time Replace. To achieve a redundancy of $O(\rho(k, \sigma, n))$, Replace takes $O(\min\{\lg_\sigma n, (k+1) \lg n/\lg \lg n\})$ time (Table 1, row 2). Unfortunately, the redundancy needed to get $O(1)$-time Replace (take $1/\epsilon = O(1)$ in row 3) is rather large: the first term is asymptotically larger even than S in 'raw' form, unless k is constant. Jansson et al. state that removing the first term while obtaining $O(1)$-time Access(i, ℓ) and Replace is an open question, which we resolve here.

– We give a representation whose space usage is $n(H_k(S) + O(\rho(k, \sigma, n)))$, which matches the redundancy of the best known static schemes [1,2,3], and supports Access(i, ℓ) and Replace in $O(1)$ time (Table 1, row 4). The data structure is simple and has potential to be practical.

Jansson et al. also gave a storage scheme (Table 1, rows 5 and 6) supporting all four operations in $O(\lg n/\lg \lg n)$ time, which is optimal [7].

– We give a scheme with space usage $n(H_k(S) + O(\lg \lg n / \lg_\sigma n))$ bits, for all $k \leq \frac{1}{8} \lg_\sigma n$, that supports all four operations in $O(\lg n / \lg \lg n)$ time (Table 1, row 7). While the redundancy of Jansson et al. is greater than that of the static schemes of [1,2,3], ours is *less*. Our redundancy has (surprisingly) no significant dependency on k: increasing k does not affect space usage.

Dynamic Rank/Select *Sequences.* In other recent work, Navarro and Nekrich [6] considered dynamic sequences that support further operations on S:

$\mathsf{Rank}(c, i)$: return $|\{j \leq i | S[j] = c\}|$, for any $c \in [\sigma]$ and $1 \leq i \leq n$.
$\mathsf{Select}(c, i)$: return the position of the i-th occurrence of c in S (-1 if there are fewer than i occurrences of c in S), for any $c \in [\sigma]$ and $i \geq 1$.

Rank and Select operations are fundamental components of a number of space-efficient data structures used in a variety of applications, such as indexing compressed XML documents [8] or compressed text [9,10]. Improving on a number of earlier papers, Navarro and Nekrich finally achieved an optimal [7] (amortized) time of $O(\lg n / \lg \lg n)$ (Table 1, second-to-last row). A key shortcoming is that they are unable to achieve "higher-order" entropy compression. Also, their Access operation costs $O(\lg n / \lg \lg n)$ time *per* symbol retrieved.

– We give a scheme that occupies $nH_k(S) + O(n(\lg \lg \sigma + (k \lg \sigma \lg \lg n) / \lg_\sigma n)$ bits and supports $\mathsf{Access}(i, \ell)$ in $O(1)$ time, Rank and Select in $O(\lg n / \lg \lg n)$ time, and Replace in $O(\lg n / \lg \lg n)$ amortized time (Table 1, last row).

Compared to [6], we achieve higher-order entropy compression, which had not been achieved before for dynamic strings supporting Rank/Select and we can retrieve substrings of length ℓ quickly. However, our redundancy is worse and the Replace operation is weaker than a general Insert/Delete. We obtain this result by a technique that could be used in other problems. Compressed string storage schemes are the basis of an effective way of designing space-efficient data structures [11]: de-couple the storage of the data (S) from the *succinct indices* used to support operations. Our technique is to 'encapsulate' a *static* Rank/Select succinct index [12] within a 'dynamization' index, while updating S using the dynamic storage scheme of Theorem 1.

Empirical Entropy. Here we recall this notion. For each $c \in [\sigma]$, let $p_c = n_c/n$ be its empirical probability of occurring in S, where n_c is its number of occurrences. The *zero-th order empirical entropy* of S is defined as $H_0(S) = -\sum_{c=1}^{\sigma} p_c \lg p_c$. For any string w of length k, let w_S be the string of single symbols following the occurrences of w in S, taken from left to right. The kth order empirical entropy of S is defined as $H_k(S) = \frac{1}{n} \sum_{w \in [\sigma]^k} |w_S| \, H_0(w_S)$. Not surprisingly, for any $k \geq 0$ we have $H_k(S) \geq H_{k+1}(S)$. The value $nH_k(S)$ is a lower bound to the output size of any compressor that encodes each symbol of S only considering the symbol itself and the k immediately preceding symbols [13].

2 Entropy bounds for dynamic storage of strings

This section presents two main results on storing a dynamic compressed string.

Theorem 1. *Given a string $S[1,n]$ over the alphabet $\Sigma = [\sigma] = \{1,\ldots,\sigma\}$, there exists an index storing S that supports* $\mathsf{Access}(i,m)$ *in $O(1+\frac{m}{\lg_\sigma n})$ time and* $\mathsf{Replace}$ *in $O(1)$ time (bounds are worst-case and optimal). The overall space occupancy is $nH_k(S) + O(n\frac{k\lg\sigma+\lg\lg n}{\lg_\sigma n})$ bits for all $k = o(\lg_\sigma n)$.*

Theorem 2. *Given a string $S[1,n]$ over the alphabet $\Sigma = [\sigma]$, there exists an index storing S that supports* $\mathsf{Access}(i,m)$ *in $O(\frac{\lg n}{\lg\lg n} + \frac{m}{\lg_\sigma n})$ time, and* $\mathsf{Replace}$, Insert, Delete *in $O(\frac{\lg n}{\lg\lg n})$ time (bounds are worst-case and optimal). The overall space occupancy is $nH_k(S) + O(n\frac{\lg\lg n}{\lg_\sigma n})$ bits for all $k \le \frac{1}{8}\lg_\sigma n$.*

2.1 Supporting Access and Replace (Theorem 1)

Squeezing the current string S into S_ℓ. At any time we conceptually represent S as a sequence S_ℓ of $n' = \lceil n/\ell \rceil$ macro-symbols over the macro-alphabet Σ^ℓ, where $\ell = \lceil \frac{1}{2}\lg_\sigma n \rceil$. Each macro-symbol is made up of ℓ consecutive symbols in S (namely, $S_\ell[i] = S[(i-1)\cdot\ell+1,\ldots,i\cdot\ell]$, for any $1 \le i \le n'$) and thus the macro-alphabet has size $\sigma^\ell = \Theta(\sqrt{n})$. In this way, the 0-th order entropy-encoding of S_ℓ gives the k-th order entropy-encoding of S as stated below.

Lemma 1 ([1]). *For any ℓ, with $1 \le \ell \le n$, it holds $n'H_0(S_\ell) \le nH_k(S) + O(n'k\lg\sigma)$, simultaneously over all $k \le \ell$.*

Lemma 1 implies that if we can maintain a dynamic compressed representation of S_ℓ in $n'H_0(S_\ell) + O(n'\lg\lg n)$ bits, we obtain a dynamic compressed representation of S in $nH_k(S) + O(n\frac{k\lg\sigma+\lg\lg n}{\lg_\sigma n})$ bits.

Codewords for the macro-alphabet Σ^ℓ of S_ℓ. We now focus on a dynamic encoding of the macro-symbols in Σ^ℓ to obtain a 0-th order entropy-encoding of S_ℓ. We divide the whole set of assigned codewords into classes C_j, where $0 \le j \le \frac{1}{2}\lg n'$, and each of the codewords in C_j is of fixed length $j+3$ bits. We also assign a nonempty *set* Γ_c of codewords to each macro-symbol $c \in \Sigma^\ell$ (and remark that c can be encoded by any codeword in Γ_c). We want to preserve the following invariants (on the number of codewords):

1. $|C_j| < 2^{j+3}$, for any $0 \le j \le \frac{1}{2}\lg n'$.
2. $|C_j \cap \Gamma_c| \le 1$, for any class C_j and any macro-symbol $c \in \Sigma^\ell$.
3. $|\Gamma_c \cap \Gamma_{c'}| = 0$, for any two distinct macro-symbols $c, c' \in \Sigma^\ell$.

We will discuss the rationale of the invariants 1–3 and how to maintain them in the next paragraphs. From a data structure point of view, for each class C_j we keep (i) a decoding table (an array of 2^{j+3} entries) D_j that maps each codeword to its assigned macro-symbol and (ii) a free-list F_j of available codewords that can be still assigned to that class if required (where the next available codeword is the head of F_j). Moreover, for each macro-symbol c we keep *(iii)* an array W_c of $\frac{1}{2}\lg n'$ entries that represent Γ_c: entry $W_c[j]$ contains the codeword of C_j that has been assigned to c, or $\perp$ when there is no such codeword; and *(iv)* a counter f_c that stores the number of occurrences of c within the current S_ℓ. The space required for storing *(i)*–*(iv)* is $O(\sum_{j=0}^{\frac{1}{2}\lg n'} 2^{j+3}(\lg \sigma^\ell + \lg n) + \sigma^\ell \lg^2 n' + \sigma^\ell \lg n') = O(\sqrt{n}\lg n \lg n') = o(n')$ bits.

Dynamic 0*-th order entropy-encoding of* S_ℓ*.* The encoding of S_ℓ is obtained by concatenating the codewords of its n' macro-symbols: given any occurrence of macro-symbol c, this occurrence is encoded by any chosen codeword from Γ_c. However the codewords may change during the lifetime of S_ℓ, and thus we cannot simply store the resulting encoding of S_ℓ as a binary string: we need to access and modify codewords, and to increase or reduce the space reserved to them. Hence, we use the data structure in [5, Th.6] that stores a set of n' binary strings of up to $\lg n'$ bits each, and supports the following two operations: *(1)* Address(i) returns a memory pointer to the ith binary string $(1 \leq i \leq n')$, and *(2)* Realloc(i, b) changes the length of the ith binary string to b bits $(b \leq \lg n')$, as restated next using our parameters.

Lemma 2 ([5]). *Let B be a set of n' binary strings, each of length at most $\lg n'$, and let s be the total number of bits for all the strings in B. We can store B in $s + O(n' \lg\lg n + \lg^4 n)$ bits while supporting* address *and* realloc *in $O(1)$ time.*

Our plan is to use Lemma 2 with $s = n'H_0(S_\ell) + O(n')$: by the discussion in the previous paragraphs, the total space occupancy of our encoding scheme is $n'H_0(S_\ell) + O(n' \lg\lg n)$ bits as claimed.

Initialization. The macro-symbols in S_ℓ are grouped into classes having approximately the same number of occurrences. Specifically, we scan S_ℓ and compute the number of occurrences f_c of each macro-symbol c that appear in it. Then we assign one class C_j to each each c using f_c, so that $\frac{n'}{2^j} < f_c \leq \frac{n'}{2^{j+1}}$.[4] We initialize each class C_j to be empty and set its free-list F_j to contain all the 2^{j+3} binary codewords of fixed length $j + 3$ bits. We also initialize each array W_c to contain all $\perp$s (see points *(i)*–*(iv)* in a previous paragraph). Then, for each macro-symbol c with $f_c > 0$, say of class C_j, we extract a codeword w from F_j and set $D_j[w] = c$ and $W_c[j] = w$. Note that after the initialization, we have a single codeword shared by all the occurrences of the same macro-symbol but, as we will see shortly, this is not mandatory during the rest of the lifetime of S_ℓ.

Operation Access(i, m)*.* Let $p = \lceil i/\ell \rceil$. We retrieve the pth macro-symbol and its next $O(m/\ell)$ macro-symbols (if needed), taking $O(1)$ time per macro-symbol (i.e. ℓ symbols in S) as follows. We first retrieve the pth codeword, say w, and its

[4] An exception is the last class C_j for $j = \frac{1}{2}\lg n'$, containing the macro-symbols with less than $\frac{n'}{2^{j+1}} = O(\sqrt{n'})$ occurrences.

length, say j', using address as in Lemma 2. In this way we infer that $w \in C_{j'-3}$ and return the ℓ symbols of Σ that are stored in table entry $D_{j'-3}[w]$ in $O(1)$ time (since they take a total of $O(\lg n)$ bits). We repeat this task $O(1 + m/\ell)$ times for $p, p+1, \ldots,$ thus attaining a cost of $O(1 + \frac{m}{\lg_\sigma n})$ time.

Operation Replace(i, c)*.* We first describe an amortized implementation, which will then be deamortized. During an execution of Replace, the number of occurrences of a macro-symbol can change. Thus, macro-symbols may move to different classes in the lifetime of the data structure. Once a macro-symbol enters a class for the first time, it is assigned an available codeword of that class. Since the number of available codewords in any class is limited, it may happen that the last available codeword is consumed in this way. For the moment, we rebuild the whole data structure from scratch in that case. We have thus two conflicting goals. On one hand, the codewords of a class should be as large as possible to postpone the rebuilding. On the other hand, these codewords should be as small as possible to limit the loss with respect to entropy. We will show that rebuilding is needed only after $\Omega(n')$ Replace operations and, simultaneously, that the loss in the entropy is just $O(1)$ bits per macro-symbol of S_ℓ.

Operation Replace(i, c) must change the pth macro-symbol in S_ℓ, where $p = \lceil i/\ell \rceil$. Wlog assume that $S_\ell[p] = x$ has to be replaced by macro-symbol y (i.e., y is obtained from x by substituting x's symbol in position $i - (p-1)\ell$ with c). We perform the following steps and maintain Invariants 1–3 mentioned earlier:

1. Set $f_x = f_x - 1$ and $f_y = f_y + 1$ (data structures (iv)).
2. Let C_j be the current class of macro-symbol y (for the updated f_y):
   (a) If there is a codeword in C_j assigned to y, let e be such a codeword.
   (b) If such a codeword does not exist, extract e from F_j and assign it to y.
3. Encode $S_\ell[p]$ with e (see Lemma 2) and update data structures (i)–(iii).
4. If $|C_j| = 2^{j+3}$ (i.e., F_j is empty), rebuild all the data structures from scratch.

Lemma 3. *$\Omega(n')$* Replace *operations are required before $|C_j| = 2^{j+3}$, for any j.*

Proof. Initially, at most 2^j codewords of class C_j are used. Indeed, a macro-symbol c is in class C_j iff its number of occurrences f_c is in $(\frac{n'}{2^j}, \frac{n'}{2^{j+1}}]$. Pessimistically, assume that all the macro-symbols in the classes C_{j-1} and C_{j+1} move to class C_j. We have at most 2^{j+2} codewords assigned to macro-symbols in classes C_{j-1}, C_j and C_{j+1}. Any other macro-symbol c has a number of occurrences $\Theta(\frac{n'}{2^j})$ away from the interval $(\frac{n'}{2^j}, \frac{n'}{2^{j+1}}]$. Thus, to use the remaining (at least) $2^{j+3} - 2^{j+2} = 2^{j+2}$ codewords, we need at least $2^{j+2} \times \Theta(\frac{n'}{2^j}) = \Omega(n')$ Replace operations. □

The time complexity of Replace is clearly dominated by Step 4. Indeed, Steps 1–3 take $O(1)$ time while Step 4 requires $O(n')$ time. Lemma 3 implies that we can amortize this cost to $O(1)$ time. To deamortize the cost, we employ an incremental rebuilding scheme to obtain our claimed bounds.

We scan the macro-symbols in S_ℓ and update their codewords and data structures, namely, the codeword of the macro-symbol at hand is replaced with the codeword of its current class. We can release and reassign any codeword that is no longer in use in the encoding of S_ℓ using the free-lists. More precisely, we fix a constant d and, at the rth Replace, we update the codewords of the d macro-symbols of S_ℓ in positions $1+(d\cdot r) \mod n', \ldots, 1+(d\cdot r+d-1) \mod n'$. If a codeword is replaced, we check if it is no longer in use and, if so, we release it to F_j, where its length is $j+3$. By Lemma 3, it follows that this mechanism guarantees that no class can use all its codewords because the reassignment is faster by a constant factor and terminates before any condition $|C_j| = 2^{j+3}$ may happen (i.e., the condition in Step 4 does not hold anymore at this point).

Bounding the space occupancy. It remains to prove that in Lemma 2, our s is at most $\sum_{c\in\Sigma^\ell}(f_c \lg \frac{n'}{f_c} + O(f_c)) = n'H_0(S_\ell) + O(n')$ bits. (Recall that this gives the bound of Theorem 1 by Lemma 1.) It suffices to prove that the overall length of the codewords in Γ_c representing the f_c occurrences of any macro-symbol c in S_ℓ can be bounded by $f_c \lg \frac{n'}{f_c} + O(f_c)$ bits.

Lemma 4. *For any macro-symbol $c \in \Sigma^\ell$, the overall space required by the f_c codewords of c in the encoding of S_ℓ is $f_c \lg \frac{n'}{f_c} + O(f_c)$ bits.*

Proof. Assume that C_j is the current class of macro-symbol c (i.e., $\frac{n'}{2^j} < f_c \leq \frac{n'}{2^{j+1}}$). In the ideal scenario, all the f_c occurrences are encoded with the $j+3$-length codeword of class C_j. In this case, each of them would require $j+3 = \lg \frac{n'}{f_c} + O(1)$ bits and the thesis would follow. However, there are occurrences of c encoded with codewords from other classes. For each class C_i, let $f_{c,i}$ be the number of occurrences of c encoded with a codeword of class C_i. The amount of additional space over the ideal scenario above can be bounded by $\sum_{i=0}^{\frac{1}{2}\lg n'} f_{c,i} \cdot (i-j) \leq \sum_{i>j}^{\frac{1}{2}\lg n'} f_{c,i} \cdot (i-j) = O(f_c)$, where the latter equality follows by observing that $f_{c,i} \leq \frac{f_c}{2^{i-j-1}}$, for any $i > j$. □

2.2 Supporting Access, Replace, Insert and Delete (Theorem 2)

Different rules of the game. As previously mentioned, the lower bound of $\Omega(\frac{\lg n}{\lg\lg n})$ time in [7] applies to the operations in this setting. Keeping this in mind, we can orchestrate a different data layout than the one described in Section 2.1 as we have $O(\frac{\lg n}{\lg\lg n})$ time per operation. This, in turn, allows us to reduce the redundancy from $O(n\frac{k\lg\sigma+\lg\lg n}{\lg_\sigma n})$ to $O(n\frac{\lg\lg n}{\lg_\sigma n})$ bits and, moreover, the resulting redundancy is now *independent* of k.

We need an alternative to Lemma 1 that uses the *first* order entropy H_1 and *variable-length* blocks. Consider *any* partition of S as a sequence $S_{m,M}$ of n' blocks, where $0 \leq m \leq M \leq n$. Each block is a substring of S of length ranging from m to M and can be seen as belonging to the macro-alphabet $\Lambda = \cup_{i=m}^{M}\Sigma^i$. Note that we use the term 'block' rather than 'macro-symbol' (as in Section 2.1) because blocks are now of variable length and will be split and merged when necessary; thus, we will refer to $S_{m,M}$ as a sequence of blocks. We now relate $H_1(S_{m,M})$ to $H_k(S)$ as follows.

Lemma 5. *For any m, M, with $0 \leq m \leq M \leq n$, it holds $n' H_1(S_{m,M}) \leq nH_k(S) + O\left(n'(1 + \lg(M - m + 1)) + k \lg \sigma\right)$, simultaneously over all $k \leq m$.*

Proof. To obtain this inequality we define a k-order encoder E giving a compression size for S that is lower bounded by $n' H_1(S_{m,M})$ bits and upper bounded by $nH_k(S) + O(n'(1 + \lg(M - m + 1)) + k \lg \sigma)$ bits (so the claim will follow).

We first discuss the upper bound. For every position i ($k \leq i \leq n$), let p_i denote the empirical conditional probability of seeing symbol $S[i]$ after the k-order context $S[i - k, i - 1]$. Given these p_i's, the k-order arithmetic encoder represents S within $\sum_{i=k}^{n} \lg p_i + 2 + k \lg \sigma \leq nH_k(S) + 2 + k \lg \sigma$ bits (see e.g., [1,3]).[5] Using this fact, our encoder E encodes the blocks of S individually: *(1)* it writes the length of the block by using $O(1 + \lg(M - m + 1))$ bits; *(2)* it encodes the symbols in the block with the aforementioned k-order arithmetic encoder. This approach increases the above encoding by $O(n' + n' \lg(M - m + 1))$ bits. The size is at most $nH_k(S) + O(n' + n' \lg(M - m + 1)) + O(k \lg \sigma)$ bits, as claimed.

We now discuss the lower bound. Note that the information content of $S_{m,M}$ and S is the same as they represent the same string, and E assigns to each block of $S_{m,M}$ its own binary codeword. These codewords do not uniquely identify a block (i.e., there may exist two different blocks that have been assigned the same codeword). However, since the k-order context of any symbol is within its own block or the preceding block, for any block B, these codewords uniquely identify all the blocks that follows B in $S_{m,M}$. Thus, the compression size of E has to be at least $n' H_1(S_{m,M})$ bits, where blocks of $S_{m,M}$ are seen as macro-symbols from the alphabet Λ: the first-order entropy is a lower bound for any compressor which encodes each macro-symbol with a codeword that only depends on the macro-symbol itself and the immediately preceding one [13]. $\square$

Toy case. Armed with Lemma 5, let us first study the static case to achieve the first order entropy for $S_{\ell,\ell}$, with $\ell = m = M = \frac{1}{8} \lg_\sigma n$, and $O(\lg n / \lg \lg n)$ time for Access. Lemma 5 implies that we obtain a compressed representation of S in $nH_k(S) + O(n \frac{\lg \lg n}{\lg_\sigma n})$ bits.

We divide $S_{\ell,\ell}$ into super-blocks of $b = \lg n / \lg \lg n$ blocks each. (a) For each block $c \in S_{\ell,\ell}$, we construct a table T^1_c that stores the blocks following c in $S_{\ell,\ell}$, sorted by their *conditional* frequencies. Each of these blocks has been assigned a codeword from the set $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$: the larger is the conditional frequency of a block, the shorter is its codeword.

In order to achieve $n' H_1(S_{\ell,\ell})$ plus lower order terms, each super-block is encoded as follows: the first block of the super-block is written uncompressed; each remaining block c' is encoded by its codeword $T^1_c[c']$, where c is the block preceding c' in the super-block.

Apart from the tables in (a), the encoding of $S_{\ell,\ell}$ comprises (b) the concatenation of the aforementioned encodings of its super-blocks, and (c) a suitable encoding of the starting positions in (b) of the individual encodings of the super-blocks and their blocks (this is necessary as the codewords are not prefix-free).

[5] The term $k \lg \sigma$ accounts for the cost of writing explicitly the first k symbols of S, which do not have any k-order context.

As for the space occupancy, we have that the tables in (a) require $O(\sigma^{2\ell} \lg n) = o(n^{1/2})$ bits; the encoding in (b) requires at most $H_1(S_{\ell,\ell})$ bits plus $O(\lg \sigma^\ell) = O(\lg n)$ bits per super-block and $O(1)$ bits per block; the encoding in (c) requires $O(n \frac{\lg \lg n}{\lg_\sigma n})$ bits using a standard two-level solution with succinct dictionaries [14]. Summing up, the space for this static scheme is $n' H_1(S_{\ell,\ell}) + O(n \frac{\lg \lg n}{\lg_\sigma n}) \leq nH_k(S) + O(n \frac{\lg \lg n}{\lg_\sigma n})$ bits by Lemma 5 (setting $\ell = m = M = \frac{1}{8} \lg_\sigma n$).

Operation $\mathsf{Access}(i, m)$ identifies the pth block of $S_{\ell,\ell}$ as in Section 2.1 but it now decompresses the whole super-block containing that block. This requires $O(\lg n / \lg \lg n)$ time. After that, the decoding of the next $O(m/\ell)$ blocks adds $O(m/\ell)$ time to the latter cost.

Fully dynamic representation. We maintain a *flexible* partition $S_{\ell,2\ell}$ of S, where super-blocks contains between b and $2b$ blocks of $S_{\ell,2\ell}$ (i.e., between $b\ell$ and $4b\ell$ symbols of S). We do not give here the implementation details that are of common use in dynamic data structures. For example, once a block (or a super-block) becomes too large or too small, it is split or merged with the preceding block (or super-block). Instead, we discuss how to maintain the compressed representation for $S_{\ell,2\ell}$ in terms of the data structures (a)–(c) previously discussed.

As for (c), we adopt the dynamic binary vector in [15] using $O(n \lg \lg n / \lg_\sigma n)$ bits, and supporting Rank, Select, Insert, and Delete in $O(\lg n / \lg \lg n)$ time.

As for (a) and (b), observe that when any $\mathsf{Replace}$, Insert or Delete of a symbol is performed in S, we have to change at most 2 blocks of $S_{\ell,2\ell}$. Consequently, some conditional frequencies should be changed in the tables T^1_c in (a) but we cannot afford soon the time cost of this change. Here is our solution in short.

Be lazy. Let us consider the snapshot of the tables T^1_c in (a) for a certain instance of $S_{\ell,2\ell}$ during its lifetime. We use them *anyway* to encode the blocks in (b) of the current instance, called $S'_{\ell,2\ell}$, obtained after say r updates of S. This potentially introduces a loss in space as we are using the outdated statistics of $S_{\ell,2\ell}$ to encode also the blocks of $S'_{\ell,2\ell}$. The result in [5, Th.4] shows that this loss can be bounded by $O(r \lg n)$ bits.[6] By choosing $r = \Theta(n \lg \sigma \lg \lg n / \lg^2 n)$, this loss remains dominated by the redundancy of Theorem 2.

Use amortization and deamortize. By our choice of r, we have $\Theta(r \frac{\lg n}{\lg \lg n}) = \Theta(n/\ell)$ credits after r update operations: they are enough to cover the cost of updating the tables in (a) and re-encoding all the $\Theta(n/\ell)$ blocks (and their super-blocks) in (b) and (c). We thus have a scheme supporting all the operations in $O(\frac{\lg n}{\lg \lg n})$ amortized time. From this point, the deamortization scheme follows the same idea in [5], which consists in dividing updates into phases and spreading the cost of the re-encoding within each phase. We refer to [5] for the details.

3 Dynamizing static Rank/Select succinct indexes

This section presents a dynamization result and an example of its application. A Rank/Select succinct index for a static string is a data structure that supports

[6] We measure the loss with respect to $|S'_{\ell,2\ell}| H_1(S'_{\ell,2\ell})$ of the current partition.

Rank and Select on a string S but reads symbols from S *solely* through (constant-time) *probe* operations. A Rank/Select succinct index for a dynamic string S also supports Rank and Select on S, also only probes S, but is notified when S is modified by a Replace operation. We show:

Theorem 3. *Let $\mathcal{I}$ be a succinct index for a static string $S[1..n]$ over the alphabet $[\sigma]$ that occupies at most $f(n, \sigma)$ bits of space, where f is a convex function of n. Further, suppose that $\mathcal{I}$ can be constructed in linear time using $O(f(n, \sigma))$ bits of space. Then, $\mathcal{I}$ can be used to build a succinct index for a dynamic string S, supporting* Rank *and* Select *with an additive overhead of $O(\lg n / \lg\lg n)$ time per operation and* Replace *on S in $O(\lg n / \lg\lg n)$ amortized time. The resulting index requires $O(n \frac{(\lg \sigma + \lg\lg n) \lg\lg n}{\lg n})$ bits of additional space, plus $O(f(n, \sigma))$ bits of temporary working space.*

Choosing the Rank/Select index from [12, Theorem 4(a)] as $\mathcal{I}$ and Theorem 1 to store S, we obtain the following result via Theorem 3:

Corollary 1. *A string $S[1, n]$ over the alphabet $\Sigma = [\sigma]$ can be stored in $nH_k(S) + O(n(\lg\lg \sigma + (k \lg \sigma \lg\lg n) / \lg_\sigma n)$ bits for all $k = o(\lg_\sigma n)$, so as to support* Rank *and* Select *in $O(\frac{\lg n}{\lg\lg n})$ time,* Access(i, m) *in $O(1 + \frac{m}{\lg_\sigma n})$ time, and* Replace *in $O(\frac{\lg n}{\lg\lg n})$ amortized time.*

3.1 Proof of Theorem 3

Two-level solution. We partition the string S into chunks of $m = \sigma \lg^2 n$ symbols each, and we use $M = n/m$ to denote the number of these chunks ($M = 1$ if $\sigma = \Omega(n/\lg^2 n)$). The first level maintains dynamically the cumulative number of occurrences of each symbol in each chunk while the second level builds a static instance of $\mathcal{I}$ on each chunk together with additional data structures to correct its potentially wrong answers. The first level routes a Rank or Select query to the appropriate chunk; the second level handles the query for that chunk.

First level. For each symbol c, we maintain an array $A_c[1, M]$, where $A_c[i]$ stores the number of occurrences of symbol c in the ith chunk, for $1 \leq i \leq M$. Every time we replace a symbol c in the ith chunk with a new symbol c', we increase $A_{c'}[i]$ and decrease $A_c[i]$, both by 1. We then construct an instance of the dynamic partial-sums data structure in [16] on each of these arrays. This data structure answers prefix sum queries, predecessor queries on the prefix sums and ± 1 updates on the arrays in $O(\lg n / \lg\lg n)$ worst-case time. For each symbol c, it is easy to see that these queries suffice to reduce Rank and Select on S to those on a chunk. As the data structures use a linear number of words of memory, the overall space usage for all A_c's is $O(M\sigma \lg n) = O(n/\lg n)$ bits overall.

Second level. The second level supports Rank, Select and Replace on each chunk C (of size m). The key idea in our solution is to build a static index $\mathcal{I}_C$ on C, whose local operations are denoted by Static_Rank and Static_Select to distinguish them from Rank and Select that we want to support dynamically. The index $\mathcal{I}_C$ is rebuilt from scratch as soon as $\Theta(m \lg\lg n/\lg n)$ updates occur in C: since rebuilding takes $O(m)$ time, the rebuilding phase costs $O(\lg n/\lg\lg n)$ amortized time per update. Note that while $\mathsf{Static\_Rank}$ and $\mathsf{Static\_Select}$ need to probe the original content of C, $\mathsf{Replace}$ operations could have changed C meanwhile. We show how to solve this problem in Appendix A.

Now, consider Rank and Select queries on the current content of C. They are solved by first querying the static index $\mathcal{I}_C$ with $\mathsf{Static\_Rank}$ and $\mathsf{Static\_Select}$. However, they may report incorrect answers since $\mathcal{I}_C$ is built on the original content of C. We give two solutions to correct these potentially incorrect answers, depending on the alphabet size σ. The solution for small alphabets, namely, when $\lg\sigma = O(\lg n/\lg\lg n)$, is presented in Appendix B. Below we discuss the solution for large alphabets, namely, when $\lg\sigma = \omega(\lg n/\lg\lg n)$ and so we can use $O(n)$ additional bits without increasing the space complexity in Theorem 3.

Operations for large alphabets. Here we implement the operations as follows.

Rank is supported by indexing two dynamic sequences $A[1,m]$ and $D[1,m]$ that keep track of the modifications occurred in the chunk C. Initially, when there are no updates, $A = \perp^m$ and $D = \perp^m$ for a special symbol $\perp$ that denotes no symbol change. An update that modifies the i-th position in C with symbol c is recorded by setting $A[i] = c$ and $D[i] = C[i]$: subsequent modifications of the i-th position change only $A[i] = c$. Supporting Rank on A and D suffices for correcting the (potentially) wrong results of $\mathsf{Static\_Rank}$. Indeed, we have $\mathsf{Rank}(c,i) = \mathsf{Static\_Rank}(c,i) + \mathsf{Rank}_A(c,i) - \mathsf{Rank}_D(c,i)$. We observe that A and D contain only $O(m\lg\lg n/\lg n) = O(\sigma \lg n \lg\lg n)$ occurrences of symbols different from $\perp$, before the rebuilding. This sparseness allows us to remain within the space bound of Theorem 3 by using the solution of Navarro and Nekrich [6] on A and D. Indeed, it requires $mH_0(A) + O(m + \sigma \lg^{1+\epsilon} m) = O(m)$ bits for A, as $m = \sigma \lg^2 n$, and an analogous argument holds for D.

Select is supported by maintaining a dynamic ternary vector T_c for each symbol c over the alphabet $\{\perp, +, -\}$ as follows. Initially, we start with the sequence $T_c = \perp^{n_c}$, where n_c is the number of occurrences of c in C. When the j-th occurrence of c in the sequence is replaced by some other symbol, we change the j-th occurrence of a symbol from $\{\perp, +\}$ in T_c to a $-$. When a symbol other than c at position i is replaced by a c, then first we count the number of occurrences j of c before the position j, using a Rank operation, and then insert a $+$ after the j-th occurrence of a symbol from $\{\perp, +\}$ in T_c. We use the index in [15] for supporting $\mathsf{Select}_{T_c}(\perp/+, i)$ and $\mathsf{Rank}_{T_c}(\perp/+, j)$ in $O(\lg n/\lg\lg n)$ time. In this case the space usage is: $\sum_{c\in[\sigma]} O(n_c \lg 3) = O(m)$ bits. Operation $\mathsf{Select}(c,i)$ is then the following one:

1. Set $j = \mathsf{Select}_{T_c}(\perp/+, i)$ and $k = \mathsf{Rank}_{T_c}(+, j)$.
2. If $T_c[j] = \perp$, return $\mathsf{Static\_Select}(c, j-k)$; else, return $\mathsf{Select}_A(c,k)$.

Finally, $\mathsf{Replace}$ is supported by changing the content of A, D and T_c's, as discussed above.

References

1. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theor. Comput. Sci. **372** (2007) 115–121
2. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. SODA, ACM Press (2006) 1230–1239
3. González, R., Navarro, G.: Compressed text indexes with fast locate. CPM. Volume 4580 of Lecture Notes in Computer Science., Springer (2007) 216–227
4. Patrascu, M., Viola, E.: Cell-probe lower bounds for succinct partial sums. In Charikar, M., ed.: SODA, SIAM (2010) 117–122
5. Jansson, J., Sadakane, K., Sung, W.K.: CRAM: Compressed random access memory. ICALP (1). Volume 7391 of LNCS, Springer (2012) 510–521
6. Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations. Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). (2013)
7. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. STOC. (1989) 345–354
8. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM **57** (2009)
9. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52** (2005) 552–581
10. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Computing **35** (2005) 378–407
11. Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms **7** (2011) 52
12. Grossi, R., Orlandi, A., Raman, R.: Optimal trade-offs for succinct string indexes. ICALP (1). Volume 6198 of LNCS, Springer (2010) 678–689
13. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48** (2001) 407–430
14. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM TALG **3** (2007)
15. Sadakane, K., Navarro, G.: Fully-functional succinct trees. SODA. (2010) 134–149
16. Pǎtraşcu, M., Demaine, E.D.: Logarithmic lower bounds in the cell-probe model. SIAM J. Comput. **35** (2006) 932–963
17. Dietzfelbinger, M., Karlin, A.R., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., Tarjan, R.E.: Dynamic perfect hashing: Upper and lower bounds. SIAM J. Comput. **23** (1994) 738–761
18. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51** (2004) 122–144
19. Pagh, R.: A new trade-off for deterministic dictionaries. **1851** (2000) 22–31
20. Brodnik, A., Carlsson, S., Demaine, E.D., Munro, J.I., Sedgewick, R.: Resizable arrays in optimal time and space. WADS. LNCS 1663, Springer (1999) 37–48

APPENDIX

A Probing original symbols in $O(1)$ time

The static index $\mathcal{I}_C$ solves its queries by probing symbols of the original chunk C, which may change due to updates. When $\mathsf{Static\_Rank}$ and $\mathsf{Static\_Select}$ probe a non-constant number of symbols of C to answer a query, we want to avoid to increase the Rank and Select cost stated in Theorem 3. This forces us to be conservative and simulate any probe in constant time by maintaining a mapping from the modified positions in C to their original symbols. A probe is performed by returning the original symbol whenever the probed position has been modified or by accessing the current version of C otherwise. Let U be the set of modified positions in the chunk (i.e., U is a subset of $[m]$), where each position in U is associated with its original symbol c. We want to solve the dictionary problem on the set U, namely, we want to support the following two operations:

- $\mathsf{Lookup}(i)$ which returns the symbol c associated with position i whenever $i \in U$, $\perp$ otherwise;
- $\mathsf{Add}(i, c)$ which adds the position i to U associated with symbol c.

Unfortunately, known data structures solving this problem are unsatisfactory in our setting: dynamic perfect hashing schemes (e.g., Dynamic FKS [17] or Cuckoo Hashing [18]) exhibit constant time lookup but require randomization for updates, while deterministic dynamic dictionaries (see e.g., [19]) achieve only $\omega(1)$ time for lookups and polylogarithmic time for updates. Hence we resort to a two-level index that supports Lookup in $O(1)$ time, and Add in $O(\lg n/\lg\lg n)$ worst-case time by requiring $O(|U|(\lg\sigma + \lg\lg n) + m\lg m \cdot \lg\lg n/\lg n)$ bits of space. Thus, in our scenario the overall space occupancy is $O(m\frac{(\lg\sigma+\lg\lg n)\lg\lg n}{\lg n})$ bits, since $|U| = O(m\lg\lg n/\lg n)$ and $\lg m = O(\lg\sigma + \lg\lg n)$. The details are as follows.

Consider the characteristic vector $V[1, m]$ of U, namely, $V[i]$ is equal to 1 if and only if the position i belongs to U. We split the vector V into $\Theta(m\lg\lg n/\lg n)$ blocks of size $\frac{\lg n}{2\lg\lg n}$ each. The description of each block is divided in two parts: the *header*, which stores the offset of each 1 within the block, and the *content*, which stores the original symbol associated with each 1 in the block. Both offsets and symbols of a block are stored by keeping their relative order. Headers and contents are stored in two collections of resizable arrays [20]: $H_1, H_2, \ldots, H_{\frac{\lg n}{2\lg\lg n}}$ and $C_1, C_2, \ldots, C_{\frac{\lg n}{2\lg\lg n}}$. The two arrays H_h and C_h store, respectively, headers and contents of blocks having exactly h modified positions and, thus, they have entries of size $O(h\lg\lg n)$ bits and $O(h\lg\sigma)$ bits respectively. For each block, we also store two $O(\lg m)$-bit pointers that point to its header and content in their corresponding arrays.

The space occupancy of the scheme is $O(|U|(\lg\sigma + \lg\lg n))$ bits for storing headers and contents, plus $O(\lg m)$ bits for each of the $O(\frac{m\lg\lg n}{\lg n})$ pointers.

$\mathsf{Lookup}(i)$. Given the header of i's block, we use a lookup on a precomputed table which returns the rank r of i within its block if $i \in U$, or $\perp$ otherwise. In the former case, we access the rth symbol from the position pointed by i's block, whereas in the latter case, we simply probe the corresponding symbol from the original chunk as the position i is unmodified. Observe that the required precomputed table has sublinear size because we have only $2^{1/2 \lg n} = O(\sqrt{n})$ possible headers.

$\mathsf{Add}(i, c)$. We access the header and the content of i's block which are modified by inserting the offset of i in the header, and symbol c in the content. If the block has now h modified positions, then the header and content of the block are moved from arrays H_{h-1} and C_{h-1} to two newly allocated entries in the arrays H_h and C_h. This operation leaves 2 unused entries in H_{h-1} and C_{h-1}. We move into these positions the last entries of corresponding arrays (in order to avoid fragmentation), and shrink the arrays H_{h-1} and C_{h-1}, if necessary. Thus, Add operation moves at most $2h = O(\frac{\lg n}{\lg \lg n})$ header and content values, and modifies the pointers of at most two blocks. Resizable arrays [20] guarantee that the allocation and deallocation of h entries costs $O(h)$ amortized time.

B Rank and Select operations for $\lg \sigma = O(\lg n / \lg \lg n)$ in Theorem 3

For each symbol c, we define the sets ADD_c and DEL_c that keep track of the positions in C which are, respectively, currently equal to the symbol c but originally set to $c' \neq c$, and currently equal to the symbol $c' \neq c$ but originally set to c. Formally,

$$\mathsf{ADD}_c = \{i \mid i \in [m] : C[i] = c \text{ and } C[i] \text{ was originally set to } c' \neq c\}, \text{and}$$

$$\mathsf{DEL}_c = \{i \mid i \in [m] : C[i] = c' \neq c \text{ and } C[i] \text{ was originally set to } c\}.$$

For each symbol c, we maintain an augmented balanced binary tree T_c whose leaves corresponds to the positions in $\mathsf{ADD}_c \cup \mathsf{DEL}_c$. Node u of T_c stores the following values:

1. the value $S(u) = \mathsf{Static\_Rank}(c, \ell)$ of the largest position ℓ in u's subtree;
2. the number $A(u)$ of positions in u's subtree that belong to ADD_c;
3. the number $D(u)$ of positions in u's subtree that belong to DEL_c.

$\mathsf{Rank}(c, i)$ is solved as follows. If $i \in \mathsf{ADD}_c \cup \mathsf{DEL}_c$, then $\mathsf{Rank}(c, i)$ is computed by traversing the path from the root to the leaf corresponding to i. Indeed, we keep two counters A_i and D_i. Every time the root-to-leaf path moves to the right child of a node u, we add $A(v)$ to A and $D(v)$ to D, where v is the left child of u. It follows that $\mathsf{Rank}(c, i) = \mathsf{Static\_Rank}(c, i) - A_i - D_i$. Instead, if $i \notin \mathsf{ADD}_c \cup \mathsf{DEL}_c$, we have $\mathsf{Rank}(c, i) = \mathsf{Static\_Rank}(c, i) - A_j - D_j$ where j is the predecessor of i in $\mathsf{ADD}_c \cup \mathsf{DEL}_c$.

$\mathsf{Select}(c,i)$ is solved by identifying the largest position j in $\mathsf{ADD}_c \cup \mathsf{DEL}_c$ such that $\mathsf{Rank}(c,j) \leq i$. There are two possible cases. If $\mathsf{Rank}(c,j)$ is exactly i and $j \in \mathsf{ADD}_c$, we report j as the result of the query. Otherwise, we report $\mathsf{Static\_Select}(c, i-k+k')$, where k and k' are the number of positions at most j, respectively, in ADD_c and in DEL_c.

Observe that all the operations requires $O(\lg n/\lg\lg n)$ time. Indeed, they are solved by traversing a root-to-a-leaf path in a tree whose height is $\lg m = O(\lg n/\lg\lg n)$ by our hypothesis on the alphabet size. The space required by these binary trees is $O(\lg m)$ bits for each modified positions. Since their number is $O(m \lg\lg n/\lg n)$, the space occupancy is within the one of Theorem 3.