Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation

Philip Bille[1], Patrick Hagge Cording[2], Inge Li Gørtz[3], Frederik Rye Skjoldjensen[4], Hjalte Wedel Vildhøj[5], and Søren Vind[6]

1 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
2 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
3 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
4 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
5 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
6 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**

Abstract

Given a static reference string R and a source string S, a relative compression of S with respect to R is an encoding of S as a sequence of references to substrings of R. Relative compression schemes are a classic model of compression and have recently proved very successful for compressing highly-repetitive massive data sets such as genomes and web-data. We initiate the study of relative compression in a dynamic setting where the compressed source string S is subject to edit operations. The goal is to maintain the compressed representation compactly, while supporting edits and allowing efficient random access to the (uncompressed) source string. We present new data structures that achieve optimal time for updates and queries while using space linear in the size of the optimal relative compression, for nearly all combinations of parameters. We also present solutions for restricted and extended sets of updates. To achieve these results, we revisit the dynamic partial sums problem and the substring concatenation problem. We present new optimal or near optimal bounds for these problems. Plugging in our new results we also immediately obtain new bounds for the string indexing for patterns with wildcards problem and the dynamic text and static pattern matching problem.

1998 ACM Subject Classification E.4 Coding and Information Theory, E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Relative compression, dynamic compression, dynamic partial sum, substring concatenation, external macro compression

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2016.18

1 Introduction

Given a static reference string R and a source string S, a *relative compression of S with respect to R* is an encoding of S as a sequence of references to substrings of R. Relative compression (or *external macro compression*) is a classic model of compression defined by Storer and Szymanski [34, 35] in 1978 and has since been used in a wide range of compression scenarios [26, 27, 23, 24, 6, 9, 19]. To compress massive highly-repetitive data sets, such as biological sequences and web collections, relative compression has been shown to be very practical [23, 24, 19].

Relative compression is often applied to compress multiple similar source strings. In such settings relative compression is superior to compressing the source strings individually. For

instance, human genomes are 99% similar and hence relative compression might be used to compress a large collection of sequenced genomes using, e.g., the human reference genome as the static reference string. We focus on the case of compressing a single source string, but our results trivially generalize to compressing multiple source strings.

In this paper we initiate the study of relative compression in a *dynamic setting*, where the compressed source string S is subject to edit operations (insertions, deletions, and replacements of single characters). The goal is to maintain the compressed representation compactly, while supporting edits and allowing efficient random access to the (uncompressed) source string. Efficient data structures supporting these operations allow us to avoid costly recompression of massive data sets after updates.

We provide the first non-trivial bounds for this problem. We present new data structures that achieve *optimal* time for updates and queries while using space linear in the size of the *optimal* relative compression, for nearly all combinations of parameters. We also present solutions for restricted and extended sets of updates.

To achieve these results, we revisit the *dynamic partial sums problem* and the *substring concatenation problem*. We present new optimal or near optimal bounds for both of these problems (see detailed discussion below). Furthermore, plugging in our new results immediately leads to new bounds for the *string indexing for patterns with wildcards problem* [25, 5] and the *the dynamic text and static pattern matching problem* [2].

1.1 Dynamic Relative Compression

Given a *reference string* R and a *source string* S, a *relative compression of S with respect to R* is a sequence $C = (i_1, j_1), ..., (i_{|C|}, j_{|C|})$ such that $S = R[i_1, j_1] \cdots R[i_{|C|}, j_{|C|}]$. We call C a *substring cover* for S. The substring cover is *optimal* if $|C|$ is minimum over all relative compressions of S with respect to R. The *dynamic relative compression problem* is to maintain a relative compression of S under the following operations. Let i be a position in S and α be a character.

access(i): return the character $S[i]$,
replace(i, α): change $S[i]$ to character α,
insert(i, α): insert character α before position i in S,
delete(i): delete the character at position i in S.

Note that operations insert and delete change the length of S by a single character. In all bounds below, the access(i) operation extends to decompressing an arbitrary substring of length ℓ using only $O(\ell)$ additional time.

Our Results. Throughout the paper, let r be the length of the reference string R, N be the length of the (uncompressed) string S, and n be the size of an optimal relative compression of S with regards to R. All of the bounds mentioned below and presented in this paper hold for a standard unit-cost RAM with w-bit words with standard arithmetic and logical operations on a word. This means that the algorithms can be implemented directly in standard imperative programming languages such as C [22] or C++ [36]. An index into R or S can be stored in a single word and hence $w \geq \log(n + r)$.

▶ **Theorem 1.** *Let R and S be a reference and source string of lengths r and N, respectively, and let n be the length of the optimal substring cover of S by R. Then, we can solve the dynamic relative compression problem supporting* access, replace, insert, *and* delete

(i) *in $O(n + r)$ space and $O\left(\frac{\log n}{\log \log n} + \log \log r\right)$ time per operation, or*

(ii) *in $O(n + r \log^{\epsilon} r)$ space and $O\left(\frac{\log n}{\log \log n}\right)$ time per operation, for any constant $\epsilon > 0$.*

These are the first non-trivial bounds for the problem. Together, the bounds are optimal for most natural parameter combinations. In particular, any data structure for a string of length N supporting access, insert, and delete must use $\Omega(\log N/\log\log N)$ time in the worst-case regardless of the space [13] (this is called the *list representation problem*). Since $n \leq N$, we can view $O(\log n/\log\log n)$ as a compressed version of the optimal time bound that is always $O(\log N/\log\log N)$ and better when S is compressible. Hence, Theorem 1(i) provides a linear-space solution that achieves the compressed time bound except for an $O(\log\log r)$ additive term. Note that whenever $n \geq (\log r)^{\log^{\epsilon}\log r}$, for any $\epsilon > 0$, the $\log n/\log\log n$ term dominates the query time and we match the compressed time bound. Hence, Theorem 1(i) is only suboptimal in the special case when n is almost exponentially smaller than r. In this case, we can use Theorem 1(ii) which always provides a solution achieving the compressed time bound at the cost of increasing the space to $O(n + r\log^{\epsilon} r)$.

We note that dynamic compression under different models of compression has been studied extensively [17, 11, 10, 33, 16, 12, 21, 28]. However, all of these results require space dependent on the size of the original string and hence cannot take full advantage of highly-repetitive data.

1.2 Dynamic Partial Sums

The *partial sums problem* is to maintain an array $Z[1..s]$ under the following operations.

sum(i): return $\sum_{j=1}^{i} Z[j]$,
update(i, Δ): set $Z[i] = Z[i] + \Delta$,
search(t): return $1 \leq i \leq s$ such that $\mathsf{sum}(i-1) < t \leq \mathsf{sum}(i)$. To ensure well-defined answers, we require that $Z[i] \geq 0$ for all i.

The partial sums problem is a classic and well-studied problem, see e.g., [8, 32, 20, 13, 18, 30]. In our context, we consider the problem in the word RAM model, where each array entry stores a w-bit integer and the element of the array can be changed by δ-bit integers, i.e., the argument Δ can be stored in δ bits. In this setting, Pătraşcu and Demaine [30] gave a linear-space data structure with $\Theta(\log s/\log(w/\delta))$ time per operation. They also gave a matching lower bound.

We consider the following generalization supporting dynamic changes to the array. The *dynamic partial sums problems* is to additionally support the following operations.

insert(i, Δ): insert a new entry in Z with value Δ before $Z[i]$,
delete(i): delete the entry $Z[i]$ of value at most Δ.
merge(i): replace entry $Z[i]$ and $Z[i+1]$ with a new entry with value $Z[i] + Z[i+1]$.
divide(i, t): , where $0 \leq t \leq Z[i]$. Replace entry $Z[i]$ by two new consecutive entries with value t and $Z[i] - t$, respectively.

Hon et al. [18] and Navarro and Sadakane [29] presented optimal solutions for this problem in the case where the entries in Z are at most polylogarithmic in s (they did not explicitly consider the merge and divide operation).

Our Results. We show the following improved result.

▶ **Theorem 2.** *Given an array of length s storing w-bit integers and fixed parameter δ, such that $\Delta < 2^{\delta}$, we can solve the dynamic partial sums problem supporting* sum, update, search, insert, delete, merge, *and* divide *in linear space and $O(\log s/\log(w/\delta))$ time per operation.*

Note that this bound simultaneously matches the optimal time bound for the standard partial sums problem and supports storing arbitrary w-bit values in the entries of the array, i.e., the values we can handle in optimal time are exponentially larger than in the previous results.

To achieve our bounds we extend the static solution by Pătraşcu and Demaine [30]. Their solution is based on storing a sampled subset of *representative elements* of the array and difference encode the remaining elements. They pack multiple difference encoded elements in words and then apply word-level parallelism to speedup the operations. To support insert and delete the main challenge is to maintain the representative elements that now dynamically move within the array. We show how to efficiently do this by combining a new representation of representative elements with a recent result by Pătraşcu and Thorup [31]. Along the way we also slightly simplify the original construction by Pătraşcu and Demaine [30].

1.3 Substring Concatenation

Let R be a string of length r. A *substring concatenation query* on R takes two pairs of indices (i, j) and (i', j') and returns the start position in R of an occurrence of $R[i, j]R[i', j']$, or `NO` if the string is not a substring of R. The *substring concatenation problem* is to preprocess R into a data structure that supports substring concatenation queries.

Amir et al. [2] gave a solution using $O(r\sqrt{\log r})$ space with query time $O(\log \log r)$, and recently Gawrychowski et al. [15] showed how to solve the problem in $O(r \log r)$ space and $O(1)$ time.

Our Results. We give the following improved bounds.

▶ **Theorem 3.** *Given a string R of length r, the substring concatenation problem can be solved in either*

(i) *$O(r \log^\epsilon r)$ space and $O(1)$ time, for any constant $\epsilon > 0$, or*

(ii) *$O(r)$ space and $O(\log \log r)$ time.*

Hence, Theorem 3(i) matches the previous $O(1)$ time bound while reducing the space from $O(r \log r)$ to $O(r \log^\epsilon r)$ and Theorem 3(ii) achieves linear space while using $O(\log \log r)$ time. Plugging in the two solutions into our solution for dynamic relative compression leads to the two branches of Theorem 1.

To achieve the bound in (i), the main idea is a new construction that efficiently combines compact data structure for 1D range reporting [3] with the recent constant time weighted level ancestor data structure for suffix trees [15]. The bound in (ii) follows as a simple implication of another recent result for *unrooted LCP queries* [5] by some of the authors. Due to lack of space, we refer to the full version of the paper (see [4]) for the details of our solution.

The substring concatenation problem is a key component in several solutions to the *string indexing for patterns with wildcards problem* [5, 7, 25], where the goal is to preprocess a string T to support pattern matching queries for patterns with wildcards. Plugging in Theorem 3(i) we immediately obtain the following new bound for the problem.

▶ **Corollary 4.** *Let T be a string of length t. For any pattern string P of length p with k wildcards, we can support pattern matching queries on T using $O(t \log^\epsilon t)$ space and $O(p + \sigma^k)$ time for any constant $\epsilon > 0$.*

This improves the running time of fastest linear space solution by a factor $\log \log t$ at the cost of increasing the space slightly by a factor $\log^\epsilon t$. See [25] for detailed overview of the known results.

1.4 Extensions

Finally, we present two extensions of the dynamic relative compression problem. The proofs of these extensions are also omitted here and can be found in the full version of the paper.

1.4.1 Dynamic Relative Compression with Access and Replace

If we restrict the operations to access and replace we obtain the following improved bound.

▶ **Theorem 5.** *Let R and S be a reference and source string of lengths r and N, respectively, and let n be the length of the optimal substring cover of S by R. Then, we can solve the dynamic relative compression problem supporting* access *and* replace *in $O(n+r)$ space and $O(\log \log N)$ expected time.*

This version of dynamic relative compression is a key component in the *dynamic text and static pattern matching problem*, where the goal is to efficiently maintain a set of occurrences of a pattern P in a text T that is dynamically updated by changing individual characters. Let p and t denote the lengths of P and T, respectively. Amir et al. [2] gave a data structure using $O(t + p\sqrt{\log p})$ space which supports updates in $O(\log \log p)$ time. The computational bottleneck in the update operation is to update a substring cover of size $O(p)$. Plugging in the bounds from Theorem 5, we immediately obtain the following improved bound, matching the previous time bound while improving the space to linear.

▶ **Corollary 6.** *Given a pattern P and text T of lengths p and t, respectively, we can solve the dynamic text and static pattern matching problem in $O(t+p)$ space and $O(\log \log p)$ expected time per update.*

1.4.2 Dynamic Relative Compression with Split and Concatenate

We also consider maintaining a set of compressed strings under split and concatenate operations (as in Alstrup et al. [1]). Let R be a reference string and let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be a set of strings compressed relative to R. In addition to access, replace, insert and delete we also define the following operations.

concat(i, j): Add string $S_i \cdot S_j$ to $\mathcal{S}$ and remove S_i and S_j.

split(i, j): Remove S_i from $\mathcal{S}$ and add $S_i[1, j-1]$ and $S_i[j, |S_i|]$.

We obtain the following bounds.

▶ **Theorem 7.** *Let R be a reference string of length r, let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be a set of source strings of total length N, and let n be the total length of the optimal substring covers of the strings in $\mathcal{S}$. Then, we can solve the dynamic relative compression problem supporting* access, replace, insert, delete, split, *and* concat,

(i) *in space $O(n+r)$ and time $O(\log n)$ for* access *and time $O(\log n + \log \log r)$ for* replace, insert, delete, split, *and* concat, *or*

(ii) *in space $O(n + r \log^{\epsilon} r)$ and time $O(\log n)$ for all operations.*

Hence, compared to the bounds in Theorem 1 we only increase the time bounds by an additional $\log \log n$ factor.

2 Dynamic Relative Compression

In this section we show how Theorems 2 and 3 lead to Theorem 1.

Let $C = ((i_1, j_1), ..., (i_{|C|}, j_{|C|}))$ be the compressed representation of S. From now on, we refer to C as the *cover of* S, and call each element (i_l, j_l) in C a *block*. Recall that a block (i_l, j_l) refers to a substring $R[i_l, j_l]$ of R. A cover C is *maximal* if concatenating any two consecutive blocks $(i_l, j_l), (i_{l+1}, j_{l+1})$ in C yields a string that does not occur in R, i.e., the string $R[i_l, j_l]R[i_{l+1}, j_{l+1}]$ is not a substring of R. We need the following lemma.

▶ **Lemma 8.** *If C_{MAX} is a maximal cover and C is an arbitrary cover of S, then $|C_{\text{MAX}}| \leq 2|C| - 1$.*

Proof. In each block b of C there can start at most two blocks in C_{MAX}, because otherwise two adjacent blocks in C_{MAX} would be entirely contained in the block b, contradicting the maximality of C_{MAX}. Since the last block of both C and C_{MAX} end at the last position of S, a contradiction of the maximality is already obtained when more than one block of C_{MAX} start in the last block of C. Hence, $|C_{\text{MAX}}| \leq 2|C| - 1$. ◀

Recall that n is the size of an optimal cover of S with regards to R. The lemma implies that we can maintain a compression of size at most $2n - 1$ by maintaining a maximal cover of S. The remainder of this section describes our data structure for maintaining and accessing such a cover.

Initially, we can use the suffix tree of R to construct a maximal cover of S in $O(N + r)$ time by greedily matching the maximal prefix of the remaining part of S with any suffix of R. This guarantees that the blocks constitute a maximal cover of S.

2.1 Data Structure

The high level idea for supporting the operations on S is to store the sequence of block lengths $j_1 - i_1 + 1, \ldots, j_{|C|} - i_{|C|} + 1$ in a dynamic partial sums data structure. This allows us, for example, to identify the block that encodes the k^{th} character in S by performing a $\mathsf{search}(k)$ query.

Updates to S are implemented by splitting a block in C. This may break the maximality property so we use substring concatenation queries on R to detect if blocks can be merged. We only need a constant number of substring concatenation queries to restore maximality. To maintain the correct sequence of block lengths we use update, divide and merge operations on the dynamic partial sums data structure.

Our data structure consist of the string R, a substring concatenation data structure of Theorem 3 for R, a maximal cover C for S stored in a doubly linked list, and the dynamic partial sums data structure of Theorem 2 storing the block lengths of C. We also store auxiliary links between a block in the doubly linked list and the corresponding block length in the partial sums data structure, and a list of alphabet symbols in R with the location of an occurrence for each symbol. By Lemma 8 and since C is maximal we have $|C| \leq 2n - 1 = O(n)$. Hence, the total space for C and the partial sums data structure is $O(n)$. The space for R is $O(r)$ and the space for substring concatenation data structure is either $O(r)$ or $O(r \log^{\epsilon} r)$ depending on the choice in Lemma 3. Hence, in total we use either $O(n + r)$ or $O(n + r \log^{\epsilon} r)$ space.

2.2 Answering Queries

To answer access(i) queries we first compute search(i) in the dynamic partial sums structure to identify the block $b_l = (i_l, j_l)$ containing position i in S. The local index in $R[i_l, j_l]$ of the i^{th} character in R is $\ell = i - \mathsf{sum}(l-1)$, and thus the answer to the query is the character $R[i_l + \ell - 1]$.

We perform replace and delete by first identifying $b_l = (i_l, j_l)$ and ℓ as above. Then we partition b_l into three new blocks $b_l^1 = (i_l, i_l + \ell - 2)$, $b_l^2 = (i_l + \ell - 1, i_l + \ell - 1)$, $b_l^3 = (i_l + \ell, j_l)$ where b_l^2 is the single character block for index i in S that we must change. In replace we change b_l^2 to an index of an occurrence in R of the new character (which we can find from the list of alphabet symbols), while we remove b_l^2 in delete. The new blocks and their neighbors, that is, b_{l-1}, b_l^1, b_l^2, b_l^3, and b_{l+1} may now be non-maximal. To restore maximality we perform substring concatenation queries on each consecutive pair of these 5 blocks, and replace non-maximal blocks with merged maximal blocks. All other blocks are still maximal, since the strings obtained by concatenating $b_{l'}$ with $b_{l'+1}$, for all $l' < l - 1$ and all $l' > l$, was not present in R before the change and is not present afterwards. A similar idea is used by Amir et al. [2]. We perform update, divide and merge operations to maintain the corresponding lengths in the dynamic partial sums data structure. The insert operation is similar, but inserts a new single character block between two parts of b_l before restoring maximality. Observe that using $\delta = O(1)$ bits in update is sufficient to maintain the correct block lengths.

In total, each operation requires a constant number of substring concatenation queries and dynamic partial sums operations; the latter having time complexity $O(\log n / \log(w/\delta)) = O(\log n / \log\log n)$ as $w \geq \log n$ and $\delta = O(1)$. Hence, the total time for each access, replace, insert, and delete operation is either $O(\log n / \log\log n + \log\log r)$ or $O(\log n / \log\log n)$ depending on the substring concatenation data structure used. In summary, this proves Theorem 1.

3 Dynamic Partial Sums

In this section we prove Theorem 2. We support the operations insert(i, Δ) and delete(i) on a sequence of w-bit integer keys by implementing them using update and a divide or merge operation, respectively. This means that we support inserting or deleting keys with value at most 2^δ.

We first solve the problem for small sequences. The general solution uses a standard reduction, storing Z at the leaves of a B-tree of large outdegree. We use the solution for small sequences to navigate in the internal nodes of the B-tree.

We need the following recent result due to Pătraşcu and Thorup [31] on maintaining a set of integer keys X under insertions and deletions. The queries are as follows, where q is an integer. The membership query member(q) returns true if $q \in X$, predecessor $\mathsf{pred}_X(q)$ returns the largest key $x \in X$ where $x < q$, and successor $\mathsf{succ}_X(q)$ returns the smallest key $x \in X$ where $x \geq q$. The rank $\mathsf{rank}_X(q)$ returns the number of keys in X smaller than q, and select(i) returns the i^{th} smallest key in X.

▶ **Lemma 9** (Pătraşcu and Thorup [31]). *There is a data structure for maintaining a dynamic set of $w^{O(1)}$ w-bit integers that supports insert, delete, membership, predecessor, successor, rank and select in constant time per operation.*

3.1 Dynamic Partial Sums for Small Sequences

Let Z be a sequence of at most $B \leq w^{O(1)}$ integer keys. We will show how to store Z in linear space such that all dynamic partial sums operations can be performed in constant time. We let Y be the sequence of prefix sums of Z, defined such that each key $Y[i]$ is the sum of the first i keys in Z, i.e., $Y[i] = \sum_{j=1}^{i} Z[j]$. Observe that $\mathsf{sum}(i) = Y[i]$ and $\mathsf{search}(t)$ is the index of the successor of t in Y. Our goal is to store and maintain a representation of Y subject to the dynamic operations update, divide and merge in constant time per operation.

3.1.1 The Scheme by Pătraşcu and Demaine

We first review the solution to the static partial sums problem by Pătraşcu and Demaine [30], slightly simplified due to Lemma 9. Our dynamic solution builds on this.

The entire data structure is rebuilt every B operations as follows. We first partition Y greedily into *runs*. Two adjacent elements in Y are in the same run if their difference is at most $B2^\delta$, and we call the first element of each run a *representative* for all elements in the run. We use $\mathcal{R}$ to denote the sequence of representative values in Y and $\mathsf{rep}(i)$ to be the index of the representative for element $Y[i]$ among the elements in $\mathcal{R}$.

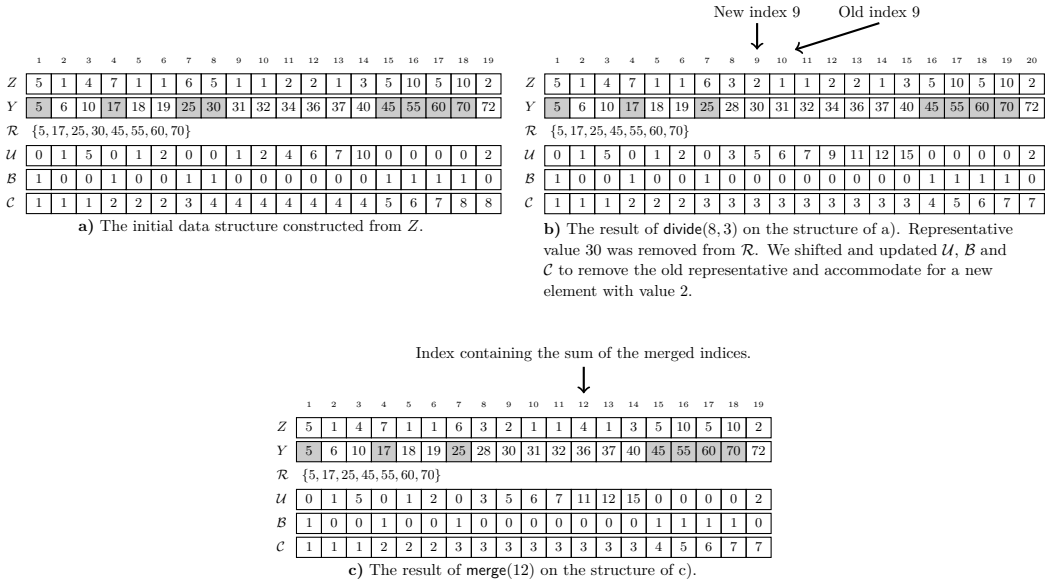
We store Y by splitting representatives and other elements into separate data structures: $\mathcal{I}$ and $\mathcal{R}$ store the representatives at the time of the last rebuild, while $\mathcal{U}$ stores each element in Y as an offset to its representative value as well as updates since the last rebuild. We ensure $Y[i] = \mathcal{R}[\mathsf{rep}(i)] + \mathcal{U}[i]$ for any i and can thus reconstruct the values of Y.

The representatives are stored as follows. $\mathcal{I}$ is the sequence of indices in Y of the representatives and $\mathcal{R}$ is the sequence of representative values in Y. Both $\mathcal{I}$ and $\mathcal{R}$ are stored using the data structure of Lemma 9. We can then define $\mathsf{rep}(i) = \mathsf{rank}_{\mathcal{I}}(\mathsf{pred}_{\mathcal{I}}(i))$ as the index of the representative for i among all representatives, and use $\mathcal{R}[\mathsf{rep}(i)] = \mathsf{select}_{\mathcal{R}}(\mathsf{rep}(i))$ to get the value of the representative for i.

We store in $\mathcal{U}$ the current difference from each element to its representative, $\mathcal{U}[i] = Y[i] - \mathcal{R}[\mathsf{rep}(i)]$ (i.e. updates between rebuilds are applied to $\mathcal{U}$). The idea is to pack $\mathcal{U}$ into a single word of B elements. Observe that $\mathsf{update}(i, \Delta)$ adds value Δ to all elements in Y with index at least i. We can support this operation in constant time by adding to $\mathcal{U}$ a word that encodes Δ for those elements. Since each difference between adjacent elements in a run is at most $B2^\delta$ and $|Y| = O(B)$, the maximum value in $\mathcal{U}$ after a rebuild is $O(B^2 2^\delta)$. As B updates of size 2^δ may be applied before a rebuild, the changed value at each element due to updates is $O(B2^\delta)$. So each element in $\mathcal{U}$ requires $O(\log B + \delta)$ bits (including an overflow bit per element). Thus, $\mathcal{U}$ requires $O(B(\log B + \delta))$ bits in total and can be packed in a single word for $B = O(\min\{w/\log w, w/\delta\})$.

Between rebuilds the stored representatives are potentially outdated because updates may have changed their values. However, observe that the values of two consecutive representatives differ by more than $B2^\delta$ at the time of a rebuild, so the gap between two representatives cannot be closed by B updates of δ bits each (before the structure is rebuilt again). Hence, an answer to $\mathsf{search}(t)$ cannot drift much from the values stored by the representatives; it can only be in a constant number of runs, namely those with a representative value $\mathsf{succ}_{\mathcal{R}}(t)$ and its two neighboring runs. In a run with representative value v, we find the smallest j (inside the run) such that $\mathcal{U}[j] + v - t > 0$. The smallest j found in all three runs is the answer to the $\mathsf{search}(t)$ query. Thus, by rebuilding periodically, we only need to check a constant number of runs when answering a $\mathsf{search}(t)$ query.

On this structure, Pătraşcu and Demaine [30] show that the operations sum, search and update can be supported in constant time each as follows:



a) The initial data structure constructed from Z.

b) The result of divide$(8, 3)$ on the structure of a). Representative value 30 was removed from $\mathcal{R}$. We shifted and updated $\mathcal{U}$, $\mathcal{B}$ and $\mathcal{C}$ to remove the old representative and accommodate for a new element with value 2.

c) The result of merge(12) on the structure of c).

Figure 1 Illustrating operations on the data structure with $B2^{\delta} = 4$. a) shows the data structure immediately after a rebuild, b) shows the result of performing divide$(8, 3)$ on the structure of a), and c) shows the result of performing merge(12) on the structure of b).

sum(i): return the sum of $\mathcal{R}[\mathsf{rep}(i)]$ and $\mathcal{U}[i]$. This takes constant time as $\mathcal{U}[i]$ is a field in a word and representatives are stored using Lemma 9.

search(t): let $r_0 = \mathsf{rank}_{\mathcal{R}}(\mathsf{succ}_{\mathcal{R}}(t))$. We must find the smallest j such that $\mathcal{U}[j]+R[r]-t > 0$ for $r \in \{r_0 - 1, r_0, r_0 + 1\}$, where j is in run r. We do this for each r using standard word operations in constant time by adding $R[r] - t$ to all elements in $\mathcal{U}$, masking elements not in the run (outside indices $\mathsf{select}_{\mathcal{I}}(r)$ to $\mathsf{select}_{\mathcal{I}}(r + 1) - 1$, and counting the number of negative elements.

update(i, Δ): we do this in constant time by copying Δ to all fields $j \geq i$ by a multiplication and adding the result to $\mathcal{U}$.

To count the number of negative elements or find the least significant bit in a word in constant time, we use the technique by Fredman and Willard [14].

Notice that rebuilding the data structure every B operations takes $O(B)$ time, resulting in amortized constant time per operation. We can instead do this incrementally by a standard approach by Dietz [8], reducing the time per operation to worst case constant. The idea is to construct the new replacement data structure incrementally while using the old and complete data structure.

3.1.2 Efficient Support for divide and merge

We now show how to maintain the structure described above while supporting operations divide(i, t) and merge(i). An example supporting the following explanation is provided in Figure 1.

Observe that the operations are only local: Splitting $Z[i]$ into two parts or merging $Z[i]$ and $Z[i + 1]$ does not influence the precomputed values in Y (besides adding/removing values for the divided/merged elements). We must update $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{U}$ to reflect these local changes accordingly. Because a divide or merge operation may create new representatives between rebuilds with values that do not fit in $\mathcal{U}$, we change $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{U}$ to reflect these new representatives by rebuilding the data structure locally. This is done as follows.

Consider the run representatives. Both $\mathsf{divide}(i,t)$ and $\mathsf{merge}(i)$ may require us to create a new run, combine two existing runs or remove a run. In any case, we can find a replacement representative for each run affected. As the operations are only local, the replacement is either a divided or merged element, or one of the neighbors of the replaced representative. Replacing representatives may cause both indices and values for the stored representatives to change. We use insertions and deletions on $\mathcal{R}$ to update representative values.

Since the new operations change the indices of the elements, these changes must also be reflected in $\mathcal{I}$. For example, a $\mathsf{merge}(i)$ operation decrements the indices of all elements with index larger than i compared to the indices stored at the time of the last rebuild We should in principle adjust the $O(B)$ changed indices stored in $\mathcal{I}$. The cost of adjusting the indices accordingly when using Lemma 9 to store $\mathcal{I}$ is $O(B)$. Instead, to get our desired constant time bounds, we represent $\mathcal{I}$ using a resizable data structure with the same number of elements as Y that supports this kind of update. We must support $\mathsf{select}_{\mathcal{I}}(i)$, $\mathsf{rank}_{\mathcal{I}}(q)$, and $\mathsf{pred}_{\mathcal{I}}(q)$ as well as inserting and deleting elements in constant time. Because $\mathcal{I}$ has few and small elements, we can support the operations in constant time by representing it using a bitstring $\mathcal{B}$ and a structure $\mathcal{C}$ which is the prefix sum over $\mathcal{B}$ as follows.

Let $\mathcal{B}$ be a bitstring of length $|Y| \leq B$, where $\mathcal{B}[i] = 1$ iff there is a representative at index i. $\mathcal{C}$ has $|Y|$ elements, where $\mathcal{C}[i]$ is the prefix sum of $\mathcal{B}$ including element i. Since $\mathcal{C}$ requires $O(B \log B)$ bits in total we can pack it in a single word. We answer queries as follows: $\mathsf{rank}_{\mathcal{I}}(q)$ equals $\mathcal{C}[q-1]$, we answer $\mathsf{select}_{\mathcal{I}}(i)$ by subtracting i from all elements in $\mathcal{C}$ and return one plus the number of elements smaller than 0 (as done in $\mathcal{U}$ when answering search), and we find $\mathsf{pred}_{\mathcal{I}}(q)$ as the index of the least significant bit in $\mathcal{B}$ after having masked all indices larger than q. Updates are performed as follows. Using mask, shift and concatenate operations, we can ensure that $\mathcal{B}$ and $\mathcal{C}$ have the same size as Y at all times (we extend and shrink them when performing divide and merge operations). Inserting or deleting a representative is to set a bit in $\mathcal{B}$, and to keep $\mathcal{C}$ up to date, we employ the same ± 1 update operation as used in $\mathcal{U}$.

We finally need to adjust the relative offsets of all elements with a changed representative in $\mathcal{U}$ (since they now belong to a representative with a different value). In particular, if the representative for $\mathcal{U}[j]$ changed value from v to v', we must subtract $v' - v$ from $\mathcal{U}[j]$. This can be done for all affected elements belonging to a single representative simultaneously in $\mathcal{U}$ by a single addition with an appropriate bitmask (update a range of $\mathcal{U}$). Note that we know the range of elements to update from the representative indices. Finally, we may need to insert or delete an element in $\mathcal{U}$, which can be done easily by mask, shift and concatenate operations on the word $\mathcal{U}$. This leads to Theorem 10.

▶ **Theorem 10.** *There is a linear space data structure for dynamic partial sums supporting each operation* search, sum, update, insert, delete, divide, *and* merge *on a sequence of length* $O(\min\{w/\log w, w/\delta\})$ *in worst-case constant time.*

3.2 Dynamic Partial Sums for Large Sequences

Willard [37] (and implicitly Dietz [8]) showed that a leaf-oriented B-tree with out-degree B of height h can be maintained in $O(h)$ worst-case time if: 1) searches, insertions and deletions take $O(1)$ time per node when no splits or merges occur, and 2) merging or splitting a node of size B requires $O(B)$ time.

We use this as follows, where Z is our integer sequence of length s. Create a leaf-oriented B-tree of degree $B = \Theta(\min\{w/\log w, w/\delta\})$ storing Z in the leaves, with height $h = O(\log_B n) = O(\log n/\log(w/\delta))$. Each node v uses Theorem 10 to store the $O(B)$

sums of leaves in each of the subtrees of its children. Searching for t in a node corresponds to finding the successor $Y[i]$ of t among these sums. Dividing or merging elements in Z corresponds to inserting or deleting a leaf. This concludes the proof of Theorem 2.

4 Conclusion

Our solution to DRC is built on data structures for the partial sums problem and the substring concatenation problem. Our partial sums-solution is optimal, but in order to get the desired constant query time for substring concatenation, our data structure uses $O(r \log^{\epsilon} r)$ space. Opposed to this, our linear space solution leads to $O(\log \log r)$ query time. We leave as an open problem if it is possible to get constant time substring concatenation queries using $O(r)$ space, which will also carry over to a stronger result for the DRC problem, and improved solutions for the string indexing for patterns with wildcards problem and the dynamic text and static pattern matching problem.

Currently we maintain a 2-approximation of the optimal cover. It would be useful to know if a better approximation ratio can be maintained under the same (or better) time and space bounds that we give.

Acknowledgments. We thank Pawel Gawrychowski for helpful discussions.

References

1. Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proc. 11th SODA*, pages 819–828, 2000.
2. Amihood Amir, Gad M Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM TALG*, 3(2):19, 2007.
3. Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *Proc. 18th ESA*, pages 427–438, 2010.
4. Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *CoRR*, abs/1504.07851, 2015. URL: `http://arxiv.org/abs/1504.07851`.
5. Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014.
6. B. G. Chern, Idoia Ochoa, Alexandros Manolakos, Albert No, Kartik Venkat, and Tsachy Weissman. Reference based genome compression. In *IEEE ITW*, pages 427–431, 2012.
7. Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th STOC*, pages 91–100, 2004.
8. Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1st WADS*, pages 39–46, 1989.
9. Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative Lempel–Ziv self-index for similar sequences. *TCS*, 532:14–30, 2014.
10. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
11. Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Succinct representation of sequences. Technical report, Università di Pisa, 2004.
12. Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *TCS*, 372(1):115–121, 2007.
13. Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.

14 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47(3):424–436, 1993.

15 Paweł Gawrychowski, Moshe Lewenstein, and Patrick K Nicholson. Weighted ancestors in suffix trees. In *Algorithms – ESA 2014*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

16 Rodrigo González and Gonzalo Navarro. Compressed text indexes with fast locate. In *Combinatorial Pattern Matching*, volume 4580, pages 216–227. Springer, 2007. doi:10.1007/978-3-540-73437-6_23.

17 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

18 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *TCS*, 412(39):5176–5186, 2011.

19 Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB*, 5(3):265–273, 2011.

20 Thore Husfeldt and Theis Rauhe. New lower bound techniques for dynamic partial sums and related problems. *SIAM J. Comput.*, 32(3):736–753, 2003.

21 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. CRAM: Compressed random access memory. In *Automata, Languages, and Programming*, pages 510–521. Springer, 2012. doi:10.1007/978-3-642-31594-7_43.

22 Brian Kernighan and Dennis Ritchie. *The C Programming Language (1st Ed.)*. Prentice-Hall, 1978.

23 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th SPIRE*, pages 201–206, 2010.

24 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proc. 34th ACSC*, pages 91–98, 2011.

25 Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. In *Proc. 31st STACS*, pages 506–517, 2014.

26 Stan Y. Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. Design Autom. Electr. Syst.*, 4(1):12–38, 1999.

27 Stan Y. Liao, Srinivas Devadas, Kurt Keutzer, Steven W. K. Tjiang, and Albert Wang. Code optimization techniques in embedded DSP microprocessors. *Design Autom. for Emb. Sys.*, 3(1):59–73, 1998.

28 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. In *Proc. 24th SODA*, pages 865–876, 2013.

29 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Alg.*, 10(3):16, 2014.

30 Mihai Pătraşcu and Erik D Demaine. Tight bounds for the partial-sums problem. In *Proc. 15th SODA*, pages 20–29, 2004.

31 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th FOCS*, pages 166–175, 2014.

32 Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct dynamic data structures. In *Algorithms and Data Structures*, pages 426–437. Springer, 2001. doi:10.1007/3-540-44634-6_39.

33 Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th SODA*, pages 1230–1239, 2006.

34 James A. Storer and Thomas G. Szymanski. The macro model for data compression. In *Proc. 10th STOC*, pages 30–39, 1978.

35 James A Storer and Thomas G Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

36 Bjarne Stroustrup. *The C++ Programming Language: Special Edition (3rd Edition)*. Addison-Wesley, 2000. First edition from 1985.

37 Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000.