Optimal Rank and Select Queries on Dictionary-Compressed Text

Nicola Prezza
Department of Computer Science, University of Pisa, Italy
nicola.prezza@di.unipi.it

Abstract

We study the problem of supporting queries on a string S of length n within a space bounded by the size γ of a string attractor for S. In the paper introducing string attractors it was shown that random access on S can be supported in optimal $O(\log(n/\gamma)/\log\log n)$ time within $O(\gamma \text{ polylog } n)$ space. In this paper, we extend this result to *rank* and *select* queries and provide lower bounds matching our upper bounds on alphabets of polylogarithmic size. Our solutions are given in the form of a space-time trade-off that is more general than the one previously known for grammars and that improves existing bounds on LZ77-compressed text by a $\log\log n$ time-factor in *select* queries. We also provide matching lower and upper bounds for *partial sum* and *predecessor* queries within attractor-bounded space, and extend our lower bounds to encompass navigation of dictionary-compressed tree representations.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Cell probe models and lower bounds

Keywords and phrases Rank, Select, Dictionary compression, String Attractors

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.4

Related Version A full version of the paper is available at `https://arxiv.org/abs/1811.01209`.

Funding The author is supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

1 Related Work

Access, rank, and select queries stand at the core of many tasks on compact and compressed data structures, including compressed indexes, graphs, trees, sets of points, etc. Given a string $S[1..n]$ over an integer alphabet $\Sigma = [0, \sigma - 1]$, these queries are defined as:

- $S.access(i)$, $i = 1, \ldots, n$: return the i-th symbol of S.
- $S.rank_c(i)$, $i = 1, \ldots, n$: return the number of occurrences of symbol c in $S[1..i]$.
- $S.select_c(i)$, $i = 1, \ldots, S.rank_c(n)$: return the position of the i-th occurrence of c in S.

While the problem is essentially solved within entropy-compressed space bounds [2], the increasingly growing production of repetitive datasets in fields such as biology, physics, and storage of software web repositories is raising the problem of extending such functionalities (as well as more complex queries such as indexing) to dictionary-compressed representations. This problem has lately received some attention in the literature, and solutions are known for some (but not all) compression schemes. Belazzougui et al. in [5] provided near-optimal bounds on LZ77-compressed text: letting z be the number of LZ77 phrases, they show how to perform access in $O(z \log^{\epsilon} n \log(n/z)/\log\log n)$ space and $O(\log(n/z)/\log\log n)$ time. *rank* and *select* require a σ-factor more space and are supported in $O(\log(n/z)/\log\log n)$ and $O(\log(n/z))$ time, respectively. Ordóñez et al. [23] and Belazzougui at al. [4] solved the problem on grammar-compressed strings in $O(\sigma g)$ space and $O(\log n)$ time for rank and select queries. The latter paper also provides a trade-off using $O(\tau\sigma g \log_{\tau}(n/g))$ words and supporting queries in $O(\log_{\tau}(n/g))$ time, for $2 \leq \tau \leq \log^{\epsilon} n$ and any constant $\epsilon > 0$. For $\tau = \log^{\epsilon} n$, this

solution yields $O(\sigma g \log^{\epsilon} n \log(n/g)/\log\log n)$ space and $O(\log(n/g)/\log\log n)$ query time. No solutions are known for other dictionary compression schemes such as Macro Schemes [24], Collage Systems [17], or the run-length Burrows-Wheeler transform [9].

In this paper, we use the newborn theory of string attractors [15, 16] to provide a universal solution working simultaneously on all known dictionary compressors. We moreover provide the first lower bounds for these queries and describe matching upper bounds on polylogarithmic alphabets. Our solutions are the first for the run-length Burrows-Wheeler transform, Macro Schemes, Run-length SLPs, and Collage Systems. We obtain the full-spectrum trade-off for grammars, generalizing the result of Belazzougui at al. [4] for all $2 \leq \tau \leq n$, and improve the existing bounds on LZ77-compressed text [5] by a $\log\log n$ time-factor in *select* queries.

We note that the reason why Belazzougui et al. [5, 4] cannot reach the optimal time for *select* queries on LZ77 and cannot get the full spectrum $2 \leq \tau \leq n$ on grammars is the same: they need to perform a predecessor query at each level of their structure. In the first case, they use z-fast tries [3], which introduce a $O(\log\log n)$ multiplicative factor in query times. In the second case, they use fusion trees [12], which support predecessor queries in constant time only for $\tau = O(\log^{\epsilon} n)$. We solve this problem, and obtain optimal query times also for *select* queries, by reducing both *rank* and *select* to partial sum queries:

- $S.psum(i) = \sum_{j=1}^{i} S[j],\ i = 1, \ldots, n.$

We show how to support partial sums in optimal time within attractor-bounded space (optimality follows from a straightforward reduction from *access* queries) by generalizing the strategy used by Belazzougui et al. [5] to solve *rank* queries. Importantly, our reductions to *partial sum* preserve asymptotically the attractor size. This solution allows us to avoid performing expensive predecessor queries at each level of our structure, thus obtaining constant time per level on the whole range $2 \leq \tau \leq n$.

On our way, we extend our lower bounds to operations on dictionary-compressed sequences of balanced parentheses, typically used to support navigation on (compressed) trees. Our lower bounds show that existing solutions [7, Lem. 8.2, 8.3] on trees represented using grammar-compressed sequences of balanced parentheses are far from the optimum by a $O(\log\log n)$-factor.

We also note that our lower- and upper- bounds easily extend to the well-studied *predecessor* problem, which asks to find the largest element x not larger than a given y in an opportunely-encoded set $\{x_1, \ldots, x_m\} \subseteq [1, n]$. A classic result from Beame and Fich [1, Cor 3.10] states that, using words of size $\log^{O(1)} n$, no static data structure of size $m^{O(1)}$ can answer predecessor queries in time $o(\sqrt{\log m/\log\log m})$. First, note that a dictionary compressed representation of the sequence $x_1, x_2 - x_1, \ldots, x_m - x_{m-1}$ always takes at most $O(m)$ words of space, and could take much less if the sequence is repetitive. Indeed, we show that Beame and Fich's lower bound can be improved to $\Omega(\log n/\log\log n)$ when the sequence is dictionary-compressed, and provide a data structure matching this lower bound.

1.1 String Attractors

Let S be a string of length n. Informally, a string attractor [16] for S is a set $\Gamma \subseteq [1..n]$ with the following property: any substring of S has at least one occurrence in S crossing at least one position in Γ. The following definition formalizes this concept.

▶ **Definition 1** (String attractor [16]). *A* string attractor *of a string $S \in \Sigma^n$ is a set of positions $\Gamma \subseteq [1..n]$ such that every substring $S[i..j]$ has at least one occurrence $S[i'..j'] = S[i..j]$ with $j'' \in [i'..j']$ for some $j'' \in \Gamma$.*

String attractors were originally introduced as a unifying framework for known dictionary compressors: Straight-Line programs [18] (context-free grammars generating the string), Collage Systems [17], Macro schemes [24] (a set of substring equations having the string as unique solution; this includes Lempel-Ziv 77 [19]), the run-length Burrows-Wheeler transform [9] (a string permutation whose number of equal-letter runs decreases as the string's repetitiveness increases), and the compact directed acyclic word graph [8, 11] (the minimization of the suffix tree). As shown in [16], any of the above compressed representations induces a string attractor of the same asymptotic size, which for most compressors is also a polylogarithmic approximation to the smallest attractor (NP-hard to find [15, 16]). The other way round also holds for a subset of the above compressors: given a string attractor of size γ, one can build a compressed representation of size $O(\gamma \log(n/\gamma))$. These reductions imply that we can design universal compressed data structures (i.e. working on top of any of the above compressors). In particular, it can be shown that optimal-time random access can be supported within $O(\gamma \text{ polylog } n)$ space [16]. Similarly, optimal text indexing can be achieved within the same space [21, 22].

2 Lower Bounds on Dictionary-Compressed Strings

We start by providing lower bounds for *rank, select* and *partial sum* queries on dictionary-compressed text. We also consider *predecessor* queries on sets represented by dictionary-compressed binary strings. Our lower bounds are shown using grammar compression (Straight-Line Programs, SLPs [18]), and therefore automatically extend to any compression scheme more powerful than SLPs. In the following, we only consider grammars whose right-hand side has size two, and define the grammar's size to be the number of nonterminals. Our lower bounds are proved in Yao's cell-probe model [26] (inherited from the lower bounds [25, 16] that we use as building blocks).

We make an important clarification. Our lower bounds will state that certain operations cannot be done in less than $O(\log n/\log\log n)$ time within $O(g \text{ polylog } n)$ space. This, of course, does not mean that the lower bounds hold whenever g is the size of *any* grammar representation of the sequence (otherwise, the lower bounds would hold for $g = \Theta(n)$ since we can always build a grammar of size $g = \Theta(n)$). What the lower bounds state is that there cannot exist a structure that, given *any* grammar G for the sequence, can *always* answer queries faster than $O(\log n/\log\log n)$ time within $O(g \text{ polylog } n)$ space, where g is the size of G. This means that any data structure of that size could answer queries faster than $O(\log n/\log\log n)$ time for *some* grammar G, but not for *all* of them.

Our starting point is the following theorem from Verbin and Yu [25], in the variant revisited by Kempa and Prezza [16, Thm 5.1]:

▶ **Theorem 2** (Verbin and Yu [25]). *Let g be the size of any Straight-Line Program for a string S of length n over a binary alphabet. Any static data structure taking $O(g \text{ polylog } n)$ space cannot answer random access queries on S in $o(\log n/\log\log n)$ time.*

The idea is to show a reduction from *access* queries to *partial sum*, *predecessor*, and *rank* and from *rank* to *select*, while asymptotically preserving the grammar size. For *rank* and *partial sum* the reduction is straightforward. Let S be a binary string. Then, $S.access(i) = S.rank_1(i) - S.rank_1(i-1) = S.psum(i) - S.psum(i-1)$, where $S.rank(0) = S.psum(0) = 0$ for convenience. It follows that also $rank_1$ and *partial sum* queries cannot break the lower bound of Theorem 2 (note that, since this is a lower bound, we can remove the restriction on the alphabet size).

To extend the lower bound to *select* queries, we build a string $\delta(S)$ such that (i) $\delta(S)$ has a SLP of size at most $g+1$, and (ii) *rank* queries on S can be simulated with a constant number of *select* queries on $\delta(S)$. Then, the result follows from the hardness of *rank*.

▶ **Definition 3.** *Let S be a binary string of length n. With $\delta : \{0,1\} \to \{0,1\}^*$ we denote the function defined as $\delta(0) = 1$ and $\delta(1) = 01$. With $\delta(S)$ we denote the string $\delta(S[1])\delta(S[2])\dots\delta(S[n])$.*

▶ **Lemma 4.** *If S has a Straight Line Program of size g, then $\delta(S)$ has a Straight Line Program of size at most $g+1$.*

Proof. It is sufficient to modify the Straight Line Program G for S as follows. First, we create a new nonterminal X expanding to 01 (this is not necessary if such a nonterminal already exists). Then, in the rules of G, we replace each terminal 1 with X and each terminal 0 with 1. It is easy to see that the resulting SLP – of size at most $g+1$ – generates $\delta(S)$. ◀

▶ **Lemma 5.** $S.rank_1(i) = \delta(S).select_1(i) - i$.

Proof. First, note that each character of S generates exactly one bit set in $\delta(S)$. Then, $\delta(S).select_1(i)$ is the position of the last bit of the encoding of $S[i]$ in $\delta(S)$.

Moreover, each bit equal to 1 in S generates exactly one 0-bit in $\delta(S)$, while 0-bits in S do not generate 0-bits in $\delta(S)$. Then, the number of 0's before position t in $\delta(S)$ – which is $t - i = \delta(S).select_1(i) - i$ – corresponds to $S.rank_1(i)$, i.e. our claim. ◀

The above lemmas imply that also $select_1$ queries cannot break the lower bound within grammar-compressed space. Note that our lower bounds can trivially be extended to $rank_0$ and $select_0$ by simply flipping all bits (this operation does not increase the grammar size as it is sufficient to flip the two grammar's terminals).

▶ **Theorem 6.** *Let S be a string of length n, and let g be the size of a Straight-Line Program for S. Then, $\Omega(\log n / \log\log n)$ time is needed to perform partial sum, rank, and select queries on S within $O(g \text{ polylog } n)$ space.*

We now move to the well-studied *predecessor* problem. Let $U \subseteq [1,n]$ be a set of integers of cardinality m. Beame and Fich [1, Cor 3.10] proved that, using words of size $\log^{O(1)} n$, no static data structure of size $m^{O(1)}$ can answer predecessor queries in time $o(\sqrt{\log m / \log\log m})$. Note that the size g of a straight-line program expanding to the distances between elements of U could be much smaller than m; one of the consequences of this increased compression power is that we can improve Beame and Fich's lower bound within space bounded by g:

▶ **Theorem 7.** *Let $U \subseteq [1,n]$ be a set of size m, and let $S \in \{0,1\}^n$ be the bit-string representing U: $S[i] = 1$ iff $i \in U$. Let moreover g be the size of a Straight-Line Program for S. Then, $\Omega(\log n / \log\log n)$ time is needed to perform predecessor queries on U within $O(g \text{ polylog } n)$ space.*

Proof. The proof is a straightforward reduction from *access* queries. Let S be a binary string, and U be the set defined as $i \in U$ iff $S[i] = 1$. Then, for $i < n$, $S[i] = 1$ if and only if the predecessor of $i+1$ in U is i (we take $i+1$ because, by definition, the predecessor of i is the largest j such that $j < i$). The lower bound follows from Theorem 2. ◀

As noted in [16], all the above lower bounds immediately extend to any compression scheme more powerful than grammars (including string attractors).

▶ **Corollary 8.** *Let S be a string of length n, and let α be any of these measures of repetitivity of S: the size γ of a string attractor, the size g of a SLP, the size g_{rl} of a RLSLP, the size c of a collage system, the size z of the LZ77 parse, the size b of a macro scheme. Then, $\Omega(\log n/\log\log n)$ time is needed to perform partial sum, rank, and select queries on S within $O(\alpha \text{ polylog } n)$ space. The same lower bound holds for* predecessor *queries when S is binary and represents a set of integers.*

3 Lower bounds on Dictionary-Compressed Trees

A space-efficient way to represent trees, that also supports fast navigational operations, is to encode their topology with a (binary) string. There are three main ways to do this: LOUDS [14], DFUDS [6], and BP [14]. LOUDS requires just *rank/select* support on a binary string, for which we already provided lower bounds. We now extend our lower bounds to operations on balanced parentheses (DFUDS/BP), which will show the hardness of navigating a dictionary-compressed tree representation. DFUDS and BP require additional primitives to be supported on the underlying string of balanced parentheses:

(1) `excess(`i`)`: the difference between open and closed parentheses before position i (included).
(2) `findopen(`i`)`: the position j of the open parenthesis matching the close parenthesis in position i.
(3) `findclose(`i`)`: the inverse of `findopen`.
(4) `fwd_search(`i,δ`)`: the first position $j > i$ such that `excess(`j`)` $=$ `excess(`i`)`$+\delta$.
(5) `bwd_search(`i,δ`)`: as `fwd_search`, but looking backwards.
(6) `rmq(`i,j`)`/`RMQ(`i,j`)`: minimum/maximum in `excess(`$i..j$`)`.
(7) `rmqi(`i,j`)`/`RMQi(`i,j`)`: leftmost position of a minimum/maximum in `excess(`$i..j$`)`.

For more details, Navarro [20] gives a complete description of the above operations and explains how tree navigation queries can be reduced to them. Alternative ways to compress trees include *Tree Straight-Line Programs* (TSLPs), which are not covered here[1].

We start by showing a reduction from $rank_1$ to *excess*. The reduction is actually not new, as it already appeared in Chan et al. [10, Sec. 5.3]. The new ingredient we introduce is the hardness of performing *excess* on grammar-compressed strings. Given any binary string S, we show how to build a string $\Delta(S)$ of balanced parentheses such that (i) the grammar-compressed representation of $\Delta(S)$ is not much larger than that of S and (ii) $S.rank_1$ can be solved with a constant number of $\Delta(S).excess$ queries. Note that, in the definition below, we add an extra pair of enclosing parentheses in order to make the sequence a tree (otherwise, the transformed string could represent a forest).

▶ **Definition 9** ([10])**.** *Let $\delta(0) =$ `()` and $\delta(1) =$ `((`. When S is a binary string of length n, we define $\Delta(S) =$ `(` $\cdot\ \delta(S[1])\ \cdots\ \delta(S[n])\ \cdot$ `)`$^{k+1}$, where $k = 2 \cdot S.rank_1(n)$.*

▶ **Example 10.** If $S = 00101$, then $\Delta(S) =$ `(()()(((()(()))))).`

Note that $\Delta(S)$ is always balanced: first, we introduce an open parenthesis, then terms $\delta(0)$ are balanced and terms $\delta(1)$ introduce two unbalanced open parentheses each. Those $k = 2 \cdot S.rank_1(n)$ parentheses, plus the first open parenthesis are balanced in the final suffix `)`$^{k+1}$ of $\Delta(S)$.

[1] It is worth to note, however, that the compression ratio of these representations is not as good as that of an SLP for the DFUDS string of small-degree trees [13, Thms 2, 4].

▶ **Lemma 11.** *If $S \in \{0,1\}^n$ has a SLP of size g, then $\Delta(S)$ has a SLP of size $O(g + \log n)$.*

Proof. Let G be a SLP for S. We replace the terminal '0' with a nonterminal expanding to `()`, and the terminal '1' with a nonterminal expanding to `((`. We finally create at most $O(\log k) = O(\log n)$ new nonterminals to generate the final suffix $)^{k+1}$, and add two more rules to concatenate the resulting SLPs to the additional open parenthesis prefixing $\Delta(S)$. ◀

▶ **Lemma 12.** $S.rank_1(i) = (\Delta(S).excess(2i+1) - 1)/2$.

Proof. Assume $S.rank_1(i) = t$. Then, $S.rank_0(i) = i - t$. Since each '0' in S generates one open and one close parenthesis in $\Delta(S)$, and each '1' in S generates two open parentheses in $\Delta(S)$ and taking into account the extra open parenthesis at the beginning of $\Delta(S)$, the number of open parentheses before $\Delta(S)[2i+1]$ is $1 + 2 \cdot t + 1 \cdot (i - t)$. Similarly, the number of close parentheses before $\Delta(S)[2i+1]$ is $i - t$. Then, by definition *excess* is precisely the difference between these two values: $\Delta(S).excess(2i+1) = 1 + 2 \cdot t + 1 \cdot (i-t) - (i-t) = 2 \cdot t + 1 = 2 \cdot S.rank_1(i) + 1$. Our claim follows. ◀

Suppose, by contradiction, that there is a structure supporting $o(\log n'/\log\log n')$-time *excess* queries on a length-n' sequence B within $O(g'\text{polylog n}')$ space, where g' is the size of any SLP compressing B. Then, given any binary string $S \in \{0,1\}^n$ with SLP of size g, we can build $\Delta(S)$ of length $n' = \Theta(n)$, which by Lemma 11 has a SLP of size $g' = O(g + \log n') = O(g + \log n)$. By Lemma 12 we can use our hypothetical *excess* structure to answer $rank_1$ queries on S in $o(\log n'/\log\log n') = o(\log n/\log\log n)$ time and $O(g'\text{polylog n}') = \text{O(g polylog n)}$ space, which by Theorem 8 is a contradiction. This completes our hardness proof for *excess*.

We now reduce $rank_1$ to *findclose*. Given any binary string S, we show how to build a string $\Delta(S)$ of balanced parentheses such that (i) the grammar-compressed representation of $\Delta(S)$ is not much larger than that of S and (ii) $S.rank_1$ can be solved with a constant number of $\Delta(S).findclose$ queries. The solution for *findopen* is symmetric and is not considered here.

▶ **Definition 13.** *Let $\delta(0) =$ `)` and $\delta(1) =$ `())`. When S is a binary string of length n, we define $\Delta(S) = (^n \cdot \delta(S[1]) \cdots \delta(S[n])$.*

Note that $\Delta(S)$ is always balanced: we first open n parentheses, and then each term $\delta(S[i])$ adds an unmatched closed parenthesis.

▶ **Example 14.** If $S = 00101$, then $\Delta(S) =$ `((((())()))())`.

The proof of the following lemma is analogous to that of Lemma 11.

▶ **Lemma 15.** *If $S \in \{0,1\}^n$ has a SLP of size g, then $\Delta(S)$ has a SLP of size $O(g + \log n)$.*

We obtain the following reduction:

▶ **Lemma 16.** $S.rank_1(i) = \big(\Delta(S).findclose(n - i + 1) - n - i\big)/2$.

Proof. The idea behind the proof is the following. To solve rank, we first "jump" on the i-th last open parentheses $\Delta(S)[n-i+1]$ in the prefix of length n of $\Delta(S)$. Then, the corresponding closed parentheses $\Delta(S).findclose(n-i+1)$ is the last parenthesis of $\delta(S[i])$ (note that each character of S generates exactly one locally-unmatched closed parenthesis in $\Delta(S)$, which is matched in the prefix $(^n$). Let $S.rank_1(i) = t$. It follows that before (and including) position $\Delta(S).findclose(n-i+1)$ there are: n parentheses (the prefix $(^n$ of $\Delta(S)$), plus $3t$ parentheses (three for each '1' in $S[1..i]$), plus $i - t$ parentheses (one for each '0' in $S[1..i]$). We conclude that $\Delta(S).findclose(n-i+1) = n + 3t + (i-t) = n + 2t + i$. Our claim follows. ◀

Note that operation *findopen* is symmetric to *findclose*, so its hardness is immediate. Finally, the lower bounds automatically transfer to `fwd_search` and `bwd_search` since they can be used to implement *findopen* and *findclose* (see also [20]): `findclose(`i`)` = `fwd_search(`$i, -1$`)` and `findopen(`i`)` = `bwd_search(`$i, 0$`)`$+1$.

To prove the hardness of range queries, consider the string $\Delta(S)$ of Definition 9. Clearly, the maximum excess in the length-2 string $\delta(0) =$ `()` is reached in the first parenthesis. On the other hand, the maximum excess in $\delta(1) =$ `((` is reached in the second parenthesis. This shows that `RMQi` and `rmqi` can be used to answer *access* queries: the maximum (resp. minimum) excess in the length-2 substring $\Delta(S)[k, k+1]$ corresponding to $\delta(S[i])$ is in position k (resp. $k+1$) if and only if $S[i] = 0$ (resp. 1). Similarly, we can solve `RMQi` (resp. `rmqi`) by issuing two `RMQ` (resp. `rmq`) in the unary ranges $\Delta(S)[k]$ and $\Delta(S)[k+1]$, and taking the position with the maximum (resp. the minimum) excess. We finally obtain our result, stated in the most general form:

▶ **Theorem 17.** *Let S be a balanced parentheses sequence of length n, and let α be any of these measures of repetitivity of S: the size γ of a string attractor, the size g of a SLP, the size g_{rl} of a RLSLP, the size c of a collage system, the size z of the LZ77 parse, the size b of a macro scheme. Then, $\Omega(\log n/\log\log n)$ time is needed to perform operations (1-7) on S within $O(\alpha \text{ polylog } n)$ space.*

4 Upper Bounds

In this section we provide upper bounds matching our lower bounds on known dictionary-compressed string representations.

4.1 Partial Sums

We support *partial sum* queries by generalizing the *rank* solution presented by Belazzougui et al. [5] (designed on block trees) as follows. We divide the text in γ blocks of length n/γ. We call this the *level 0* of our structure. We keep $\log_\tau(n/\gamma)$ further levels: for each attractor position i, at level $j \geq 1$ we store some information about 2τ non-overlapping and equally-spaced substrings of S (we call them blocks) centered on i and whose length exponentially decreases with the level j (i.e. at level $j = 0$ the block length is n/γ, at level 1 it is $n/(\tau\gamma)$, at level 2 it is $n/(\tau^2\gamma)$, and so on). For each block, we store some partial sum information and a pointer to one of its occurrences crossing an attractor position (which exist by definition of attractor). Then, a *partial sum* query is answered by navigating the structure from the first to last level. All details are reported in the following Theorem.

▶ **Theorem 18.** *Let γ be the size of an attractor for a string S of length n over an integer alphabet. Then, for all $2 \leq \tau \leq n/\gamma$, we can store a data structure of size $O(\tau\gamma\log_\tau(n/\gamma))$ supporting `partial sum` queries on S in $O(\log_\tau(n/\gamma))$ time.*

Proof. For simplicity, we assume that γ divides n and that $\log_\tau(n/\gamma)$ is an integer. Our structure is composed of $\log_\tau(n/\gamma)+1$ levels. Level $j \geq 0$ contains a set of blocks (i.e. text substrings), each of length $\ell_j = n/(\gamma\cdot\tau^j)$ and defined as follows. In the first level, number 0, our blocks are the γ contiguous and non-overlapping S-substrings of length n/γ: $B_{0,k} = S[(k-1)\cdot(n/\gamma)+1..k\cdot(n/\gamma)]$, for $k = 1,\ldots,\gamma$. At level $j \geq 1$, blocks are instead centered around attractor elements. Let i be an attractor position. Then, at level $j \geq 1$ we store the 2τ blocks $\overleftarrow{B}_{i,j,k} = S[i-\ell_j\cdot k..i-\ell_j\cdot(k-1)-1]$, for $k = 1,\ldots,\tau$, and

$\overrightarrow{B}_{i,j,k} = S[i+1+\ell_j\cdot(k-1)..i+\ell_j\cdot k]$, for $k=1,\ldots,\tau$ (note: $\overleftarrow{B}_{i,j,k}$ are on the left of attractor position i, blocks $\overrightarrow{B}_{i,j,k}$ are on its right, and none of the blocks intersects i).

Note: clearly, we do not explicitly store the text substrings associated with each block. Each block will store just a constant amount of information (detailed below) that will be used to answer partial sum queries. However, letting B be a block, to simplify notation in the following we will also use the symbol B to indicate the substring represented by the block B. In this sense, $|B|$ will refer to the substring's length. The use will be clear from the context.

Each block B at level $j\geq 0$ stores a pointer to one of its occurrences (as a string) at level $j+1$ crossing an attractor position (at least one occurrence of this kind exists by definition of 1-adjacent attractor). This pointer is simply a pair (i,v), where i is the attractor position and v is such that $S[i-v..i-v+|B|-1]=B$. Crucially, note that the substring $S[i-v..i-v+|B|-1]$ is completely covered by contiguous and non-overlapping blocks of length $|B|/\tau$ at level $j+1$, except possibly position $S[i]$ that is not included in any of those blocks (this will be used to devise a recursive strategy).

In the last level $j=\log_\tau(n/\gamma)$ (where the block size is $\ell_{\log_\tau(n/\gamma)}=1$) we explicitly store in $2\gamma\tau$ words the strings representing the blocks. We also store the character $S[i]$ under each attractor position i. Note that $S[i]$ can be retrieved in constant time from i using, e.g. perfect hashing.

We associate to each block some partial sum information. To simplify notation, let $sum(B)$ denote the sum of all integers in the string B.

- **(a)** For each block B at any level, let (i,v) be the pointer associated with it. Then, we associate to B the value $sum(B[1..v])$.
- **(b)** At level 0, each block $B_{0,k}$ with $k=1,\ldots,\gamma$, stores $sum(B_{0,1}\cdots B_{0,k-1})$, i.e. the sum of all integers preceding the block in the input string (this value is 0 for $B_{0,1}$).
- **(c)** At levels $j\geq 1$, each block B stores $sum(B)$.
- **(d)** Blocks $\overleftarrow{B}_{i,j,k}$ moreover store the partial sum $sum(\overleftarrow{B}_{i,j,k}\cdots\overleftarrow{B}_{i,j,1})$, for $k=1,\ldots,\tau$, while blocks $\overrightarrow{B}_{i,j,k}$ store the partial sum $sum(\overrightarrow{B}_{i,j,1}\cdots\overrightarrow{B}_{i,j,k})$, for $k=1,\ldots,\tau$.

Overall, we store $O(1)$ words per block. It follows that our structure fits in $O(\tau\gamma\log_\tau(n/\gamma))$ words. We now show how to efficiently answer partial sum queries using this information.

To answer $S.psum(v)$ we proceed as follows. Let $k=\lfloor(v-1)/(n/\gamma)\rfloor+1$ and $v'=((v-1) \bmod (n/\gamma))+1$. Then, $S.psum(v)=sum(B_{0,1},\ldots,B_{0,k-1})+B_{0,k}.psum(v')$. The first term $sum(B_{0,1},\ldots,B_{0,k-1})$ is explicitly stored (read point (b) above). To compute $B_{0,k}.psum(v')$, note that this is a block prefix; we now show how to compute the sum of the integers in the prefix of any block at level $j\geq 0$ by reducing the problem to that of computing the sum of the integers in a prefix of a block at level $j+1$. The answer in the last level $j=\log_\tau(n/\gamma)$ (where the block size is $\ell_{\log_\tau(n/\gamma)}=1$) can be obtained in constant time since we explicitly store the integers contained in the blocks.

Let us show how to compute $sum(B[1..t])$ at level $j\geq 0$, for some $t\leq\ell_j$ and some level-j block B. First, we map $B[1..t]$ to level $j+1$ using its pointer (i,v). We distinguish two cases.

- **(1)** If $t>v$, then $sum(B[1..t])=sum(B[1..v])+sum(B[v+1..t])$. The first term, $sum(B[1..v])$ is explicitly stored (read point (a) above). Note that the string appearing in the second term, $B[v+1..t]$, prefixes $S[i]\cdot\overrightarrow{B}_{i,j+1,1}\cdots\overrightarrow{B}_{i,j+1,\tau}$. We can therefore decompose $B[v+1..t]$ into $S[i]$, followed by a (possibly empty) prefix of $d=\lfloor(t-v-1)/\ell_{j+1}\rfloor$ blocks of length ℓ_{j+1} – i.e. the prefix $\overrightarrow{B}_{i,j+1,1},\ldots,\overrightarrow{B}_{i,j+1,d}$ –, followed by a (possibly empty) suffix of length $l=(t-v-1) \bmod \ell_{j+1}$. We can retrieve in constant time the

sum (explicitly stored, see point (d) above) of the integers contained in the prefix of full blocks, as well as the value $S[i]$. As far as the remaining suffix of $B[v+1..t]$ is concerned, note that it coincides with the block prefix $\overrightarrow{B}_{i,j+1,d+1}[1..l]$ and we can thus compute the corresponding partial sum by recursing our strategy.

(2) If $t \leq v$, then $sum(B[1..t]) = sum(B[1..v]) - sum(B[t+1..v])$. The first term, $sum(B[1..v])$ is explicitly stored (read point (a) above). The second term can be computed with a strategy completely symmetric to that described in point (1). Let $d = \lfloor (v-t)/\ell_{j+1} \rfloor$ and $l = (v-t) \bmod \ell_{j+1}$. We decompose $B[t+1..v]$ into the prefix $\overleftarrow{B}_{i,j+1,d+1}[(\ell_{j+1}-l+1)..\ell_{j+1}]$ (note: this is a block suffix) followed by the suffix $\overleftarrow{B}_{i,j+1,d} \cdots \overleftarrow{B}_{i,j+1,1}$. The sum of integers in the suffix of full blocks is explicitly stored (point (d) above), so we are left with the problem of computing the sum in $\overleftarrow{B}_{i,j+1,d+1}[(\ell_{j+1}-l+1)..\ell_{j+1}]$, which is a block suffix. Since we explicitly store $sum(\overleftarrow{B}_{i,j+1,d+1})$ (point (c) above), we can, also in this case, reduce the problem to that of computing the sum in a prefix of a block at level $j+1$ (i.e. the block prefix $\overleftarrow{B}_{i,j+1,d+1}[1..\ell_{j+1}-l]$) with a simple subtraction.

The strategy above described allows us to compute $S.psum(v)$ with a single descent from the first to last level. At each level we spend constant time. It follows that the overall procedure terminates in $O(\log_\tau(n/\gamma))$ time. ◀

For $\tau = \log^\epsilon n$ and any constant $\epsilon > 0$ our structure takes $O(\gamma \log^\epsilon n \log(n/\gamma)/\log\log n)$ words of space and answers queries in $O(\log(n/\gamma)/\log\log n)$ time. This matches the lower bound stated in Theorem 8.

4.2 Rank

Clearly, on binary strings it holds that $S.rank_1(i) = S.psum(i)$ so our problem is already solved in this case by Theorem 18. Given a string S over a generic alphabet of size σ, we can solve $S.rank_c()$ as follows. We build σ bit-strings S_c, one for each alphabet character c, defined as $S_c[i] = 1$ if and only if $S[i] = c$. It is easy to verify that, if S has an attractor Γ, then Γ is an attractor also for S_c (repetitions are preserved). Then, we build our structure of Theorem 18 on each S_c using Γ as attractor and compute $S.rank_c(i) = S_c.rank_1(i)$ (we can associate each c to its structure on S_c in constant time by using perfect hashing). We obtain:

▶ **Theorem 19.** *Let γ be the size of a string attractor for a string S of length n over an alphabet of size σ. Then, for all $2 \leq \tau \leq n/\gamma$, we can store a data structure of size $O(\tau\sigma\gamma\log_\tau(n/\gamma))$ supporting* rank *queries on S in $O(\log_\tau(n/\gamma))$ time.*

For $\tau = \log^\epsilon n$ and any constant $\epsilon > 0$, our structure takes $O(\sigma\gamma \log^\epsilon n \log(n/\gamma)/\log\log n)$ words of space and answers queries in $O(\log(n/\gamma)/\log\log n)$ time. This running time matches the lower bound stated in Theorem 8 when $\sigma \in O(\text{polylog n})$.

4.3 Select

We first consider the binary case and $select_1$ queries. We use a straightforward reduction from $select_1$ to *partial sum* queries that blows up the attractor size only by a constant factor.

We assume for simplicity that our input bit-vector S ends with bit 1. If this is not the case, removing the trailing zeros from S does not change the answer of any $select_1$ query and does not increase the attractor's size. Let $S = 0^{x_1-1}1\ldots0^{x_m-1}1$, with $x_i \geq 1$ for $i = 1,\ldots,m$, and define $S' = x_1 \ldots x_m$.

▶ **Lemma 20.** *If S has an attractor of size γ, then S' has an attractor of size at most $2\gamma+1$.*

Proof. Let Γ be an attractor for S. We build an attractor Γ' for S' as follows. Note that, to build S', we partition S in blocks: each block is formed by a sequence of zeros terminated by a bit set. Given a position $i \in [1, |S|]$, we say that i' is the *corresponding position* of i in S' iff i belongs to the i'-th block. Starting with $\Gamma' = \{1\}$, for every $i \in \Gamma$ we insert in Γ' the positions i' and $i'+1$ (unless they fall outside the range $[1, |S'|]$), where i' is the corresponding position of i in S'. More formally, if $S[i] = 0$ then we insert $S.rank_1(i)+1$ and $S.rank_1(i)+2$ in Γ'. Otherwise, if $S[i] = 1$ then we insert $S.rank_1(i)$ and $S.rank_1(i)+1$ in Γ'. Clearly, $|\Gamma'| \leq 2|\Gamma| + 1 = 2\gamma + 1$.

Consider any substring $S'[i..j]$. We prove that $S'[i..j]$ has an occurrence $S'[i'..j']$ (possibly, $i' = i$ and $j' = j$) such that $S'[i'..j']$ crosses an element of Γ'. This will prove our claim.

If $i = 1$, then $S'[i..j]$ crosses position $1 \in \Gamma'$ and we are done. Otherwise, we focus on sequence $S'[i-1..j] = y_1 y_2 \dots y_{j-i+2}$. Note that this is the sequence of exponents of zeros in a subsequence of S entirely covered by blocks: $S[S.select_1(i-2)+1..S.select_1(j)] = 0^{y_1}1 \dots 0^{y_{j-i+2}}1$, where we take $S.select_1(0) = 0$. By definition of Γ, this substring of S has an occurrence $S[i'', j'']$ crossing an element of Γ. However, note that this occurrence is not necessarily preceded by a bit set. As a consequence, we can only state that the corresponding subsequence of S', i.e. $S'[t..S.rank_1(j'')]$ with $t = S.rank_1(i'')+1$ if $S[i''] = 0$ and $t = S.rank_1(i'')$ otherwise, is such that $S'[t..S.rank_1(j'')] = w y_2 \dots y_{j-i+2}$, with $w \geq y_1$. Since $S[i'', j'']$ crosses an element of Γ then, by the way we defined Γ', either $S'[S.rank_1(j'')]$ or two adjacent characters $S'[k, k+1]$, with $t \leq k < S.rank_1(j'')$, cross an element of Γ'. Then, this means that $S'[t+1..S.rank_1(j'')] = y_2 \dots y_{j-i+2} = S'[i..j]$ crosses an element of Γ'. This concludes our proof. ◀

At this point, the solution for select is immediate: we build the structure of Theorem 18 on the sequence $S' = x_1 \dots x_m$ using the attractor of Lemma 20, and simply note that $S.select_1(i) = S'.psum(i)$.

Given a string S over a generic alphabet of size σ, we can solve $S.select_c()$ as follows. We build σ bit-strings S_c, one for each alphabet character c, defined as $S_c[i] = 1$ if and only if $S[i] = c$. As seen in the previous section, an attractor for S is also an attractor for S_c. We build our structure solving $select_1$ on each S_c and compute $S.select_c(i) = S_c.select_1(i)$.

▶ **Theorem 21.** *Let γ be the size of a string attractor for a string S of length n over an alphabet of size σ. Then, for all $2 \leq \tau \leq n/\gamma$, we can store a data structure of size $O(\tau\sigma\gamma \log_\tau(n/\gamma))$ supporting* select *queries on S in $O(\log_\tau(n/\gamma))$ time.*

For $\tau = \log^\epsilon n$ and any constant $\epsilon > 0$, our structure takes $O\left(\sigma\gamma \log^\epsilon n \log(n/\gamma)/\log\log n\right)$ words of space and answers queries in $O(\log(n/\gamma)/\log\log n)$ time. This running time matches the lower bound stated in Theorem 8 when $\sigma \in O(\text{polylog n})$.

4.4 Predecessor and Tree Navigation

Let $U \subseteq [1, n]$ be a set of size m, and let S be a binary string of length n such that $S[i] = 1$ iff $i \in U$ with attractor of size γ. Using the solutions for *rank* and *select* seen in the previous sections, we can easily support *predecessor* on U in $O(\gamma$ polylog n) space and $O(\log(n/\gamma)/\log\log n)$ time (optimal by Corollary 8). In an extended version of this paper we will show that we can improve this upper bound (both in space and time) on sparse sets, achieving $O(\gamma \log^{1+\epsilon} m/\log\log m) = O(\gamma$ polylog m) space and $O(\log m/\log\log m)$ query time. This solution is analogous to that used in Theorem 18 but, in addition, uses fusion trees to accelerate local predecessor queries.

To conclude, one can obtain fast navigational queries on attractor-compressed trees by combining the SLP-based implementation of balanced-parentheses operations (1-7) (see Section 3) described by Bille et al. [7, Lem. 8.2, 8.3] with the SLP of [16, Thm. 3.14], built on the balanced-parentheses representation of the tree. This immediately yields $O(\log n)$-time navigation within $O(\gamma \log^2(n/\gamma))$ words of space. To reduce this running time to the optimal $O(\log n/\log\log n)$, we note that one could increase the arity of the SLP to $\log^\epsilon n$ as described in [4, Thm. 2]. Space could be further reduced by employing the RLSLP – of size $O(\gamma \log(n/\gamma))$ – described in [21] and adapting the algorithms to work on run-length SLPs. We will cover these improvements in an extended version of this paper.

References

1 Paul Beame and Faith E Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

2 D. Belazzougui and G. Navarro. Optimal Lower and Upper Bounds for Representing Sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.

3 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with O (1) accesses. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 785–794. SIAM, 2009.

4 Djamal Belazzougui, Patrick Hagge Cording, Simon J Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Algorithms-ESA 2015*, pages 142–154. Springer, 2015.

5 Djamal Belazzougui, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Alberto Ordónez, Simon J Puglisi, and Yasuo Tabei. Queries on LZ-bounded encodings. In *Data Compression Conference (DCC), 2015*, pages 83–92. IEEE, 2015.

6 David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

7 Philip Bille, Gad M Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

8 Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

9 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

10 Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms (TALG)*, 3(2):21, 2007.

11 Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching (CPM)*, pages 116–129. Springer, 1997.

12 Michael L Fredman and Dan E Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7. ACM, 1990.

13 Moses Ganardi, Danny Hucke, Markus Lohrey, and Eric Noeth. Tree compression using string grammars. *Algorithmica*, 80(3):885–917, 2018.

14 Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

15 Dominik Kempa, Alberto Policriti, Nicola Prezza, and Eva Rotenberg. String Attractors: Verification and Optimization. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

16 Dominik Kempa and Nicola Prezza. At the Roots of Dictionary Compression: String Attractors. In *Annual Symposium on Theory of Computing (STOC)*, pages 827–840. ACM, 2018.

17 T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: A unifying framework for compressed pattern matching. *Theor. Comput. Sci.*, 298(1):253–272, 2003.

18 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

19 A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Trans. Information Theory*, 22(1):75–81, 1976.

20 Gonzalo Navarro. *Compact data structures: A practical approach.* Cambridge University Press, 2016.

21 Gonzalo Navarro and Nicola Prezza. Faster Attractor-Based Indexes. *arXiv preprint*, 2018. `arXiv:1811.12779`.

22 Gonzalo Navarro and Nicola Prezza. Universal Compressed Text Indexing. *Theoretical Computer Science*, 2018.

23 Alberto Ordóñez, Gonzalo Navarro, and Nieves R Brisaboa. Grammar compressed sequences with rank/select support. *Journal of Discrete Algorithms*, 43:54–71, 2017.

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

25 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Annual Symposium on Combinatorial Pattern Matching*, pages 247–258. Springer, 2013.

26 Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM (JACM)*, 28(3):615–628, 1981.