Optimal-Time Text Indexing in BWT-runs Bounded Space *

Travis Gagie† Gonzalo Navarro‡ Nicola Prezza§

Abstract

Indexing highly repetitive texts — such as genomic databases, software repositories and versioned text collections — has become an important problem since the turn of the millennium. A relevant compressibility measure for repetitive texts is r, the number of runs in their Burrows-Wheeler Transform (BWT). One of the earliest indexes for repetitive collections, the Run-Length FM-index, used $O(r)$ space and was able to efficiently count the number of occurrences of a pattern of length m in the text (in loglogarithmic time per pattern symbol, with current techniques). However, it was unable to locate the positions of those occurrences efficiently within a space bounded in terms of r. Since then, a number of other indexes with space bounded by other measures of repetitiveness — the number of phrases in the Lempel-Ziv parse, the size of the smallest grammar generating the text, the size of the smallest automaton recognizing the text factors — have been proposed for efficiently locating, but not directly counting, the occurrences of a pattern. In this paper we close this long-standing problem, showing how to extend the Run-Length FM-index so that it can locate the *occ* occurrences efficiently within $O(r)$ space (in loglogarithmic time each), and reaching optimal time $O(m + occ)$ within $O(r \log(n/r))$ space, on a RAM machine with words of $w = \Omega(\log n)$ bits. Raising the space to $O(rw \log_\sigma(n/r))$, we support locate in $O(m \log(\sigma)/w + occ)$ time, which is optimal in the packed setting and had not been obtained before in compressed space. We also describe a structure using $O(r \log(n/r))$ space that replaces the text and efficiently extracts any text substring, with an $O(\log(n/r))$ additive time penalty over the optimum. Preliminary experiments show that our new structure outperforms the alternatives by orders of magnitude in the space/time tradeoff map.

*Partially funded by Basal Funds FB0001, Conicyt, by Fondecyt Grants 1-170048 and 1-171058, Chile, and by the Danish Research Council DFF-4005-00267.

†EIT, Diego Portales University, and Center for Biotechnology and Bioengineering (CeBiB), Chile, travis.gagie@gmail.com

‡Department of Computer Science, University of Chile, and Center for Biotechnology and Bioengineering (CeBiB), Chile, gnavarro@dcc.uchile.cl

§DTU Compute, Technical University of Denmark, Denmark, npre@dtu.dk

1 Introduction

The data deluge has become a routine problem in most organizations that aim to collect and process data, even in relatively modest and focused scenarios. We are concerned about string (or text, or sequence) data, formed by collections of symbol sequences. This includes natural language text collections, DNA and protein sequences, source code repositories, digitalized music, and many others. The rate at which those sequence collections are growing is daunting, and outpaces Moore's Law by a significant margin [74]. One of the key technologies to handle those growing datasets is *compact data structures*, which aim to handle the data directly in compressed form, without ever decompressing it [63]. In general, however, compact data structures do not compress the data by orders of magnitude, but rather offer complex functionality within the space required by the raw data, or a moderate fraction of it. As such, they do not seem to offer the significant space reductions that are required to curb the sharply growing sizes of today's datasets.

What makes a fundamental difference, however, is that the fastest-growing string collections are in many cases *highly repetitive*, that is, most of the strings can be obtained from others with a few modifications. For example, most genome sequence collections store many genomes from the same species, which in the case of, say, humans differ by 0.1% [69] (there is some discussion about the exact percentage). The 1000-genomes project[1] uses a Lempel-Ziv-like compression mechanism that reports compression ratios around 1% [29] (i.e., the compressed space is about two orders of magnitude less than the uncompressed space). Versioned document collections and software repositories are another natural source of repetitiveness. For example, Wikipedia reports that, by June 2015, there were over 20 revisions (i.e., versions) per article in its 10 TB content, and that `p7zip` compressed it to about 1%. They also report that what grows the fastest today are the revisions rather than the new articles, which increases repetitiveness.[2] A study of GitHub (which surpassed 20 TB in 2016)[3] re-

[1] www.internationalgenome.org

[2] en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

[3] blog.sourced.tech/post/tab_vs_spaces

ports a ratio of *commit* (new versions) over *create* (brand new projects) around 20.[4] Repetitiveness also arises in other less obvious scenarios: it is estimated that about 50% of (non-versioned) software sources [40], 40% of the Web pages [37], 50% of emails [22], and 80% of tweets [77], are near-duplicates.

When the versioning structure is explicit, version management systems are able to factor out repetitiveness efficiently while providing access to any version. The idea is simply to store the first version of a document in plain form and then the edits of each version of it, so as to reconstruct any version efficiently. This becomes much harder when there is not a clear versioning structure (as in genomic databases) or when we want to provide more advanced functionalities, such as counting or locating the positions where a string pattern occurs across the collection. In this case, the problem is how to reduce the size of classical data structures for indexed pattern matching, like suffix trees [80] or suffix arrays [54], so that they become proportional to the amount of distinct material in the collection. It should be noted that all the work on *statistical* compression of suffix trees and arrays [61] is not very useful for this purpose, as it does not capture well this kind of repetitiveness [48, Lem. 2.6].

Mäkinen et al. [51, 73, 52, 53] pioneered the research on searching repetitive collections. They regard the collection as a single concatenated text $T[1..n]$ with separator symbols, and note that the number r of *runs* (i.e., maximal substrings formed by a single symbol) in the *Burrows-Wheeler Transform* [14] of the text is relatively very low on repetitive texts. Their index, *Run-Length FM-Index (RLFM-index)*, uses $O(r)$ words and can *count* the number of occurrences of a pattern $P[1..m]$ in time $O(m \log n)$ and even less. However, they are unable to *locate* where those positions are in T unless they add a set of samples that require $O(n/s)$ words in order to offer $O(s \log n)$ time to locate each occurrence. On repetitive texts, either this sampled structure is orders of magnitude larger than the $O(r)$-size basic index, or the locating time is unacceptably high.

Many proposals since then aimed at reducing the locating time by building on other measures related to repetitiveness: indexes based on the Lempel-Ziv parse [50] of T, with size bounded in terms of the number z of phrases [48, 31, 64, 4]; indexes based on the smallest context-free grammar [45, 15] that generates T and only T, with size bounded in terms of the size g of the grammar [19, 20, 30]; and indexes based on the size e of the smallest automaton (CDAWG) [13] recognizing the substrings of T [4, 76, 2]. The achievements are summarized in Table 1; note that none of those later approaches can count the occurrences without enumerating them all. We are not considering in this paper indexes based on other measures of repetitiveness that only apply in restricted scenarios, such as based on Relative Lempel-Ziv [49, 21, 6, 24] or on alignments [59, 60].

There are a few known asymptotic bounds between the repetitiveness measures r, z, g, and e: $z \leq g = O(z \log(n/z))$ [71, 15, 39] and $e = \Omega(\max(r, z, g))$ [4, 3]. Several examples of string families are known that show that r is not comparable with z and g [4, 68]. Experimental results [53, 48, 4, 18], on the other hand, suggest that in typical repetitive texts it holds $z < r \approx g \ll e$.

For highly repetitive texts, one hopes to have a compressed index not only able to count and locate pattern occurrences, but also to *replace* the text with a compressed version that nonetheless can efficiently *extract* any substring $T[i..i+\ell]$. Indexes that, implicitly or not, contain a replacement of T, are called *self-indexes*. As can be seen in Table 1, self-indexes with $O(z)$ space require up to $O(n)$ time per extracted character, and none exists within $O(r)$ space. Good extraction times are instead obtained with $O(g)$, $O(z \log(n/z))$, or $O(e)$ space. A lower bound [79] shows that $\Omega((\log n)^{1-\epsilon}/\log g)$ time, for any constant $\epsilon > 0$, is needed to access one random position within $O(\text{poly}(g))$ space. This bound shows that various current techniques using structures bounded in terms of g or z [12, 9, 32, 5] are nearly optimal (note that $g = \Omega(\log n)$, so the space of all these structures is $O(\text{poly}(g))$). In an extended article [16], the authors give a lower bound in terms of r, but only for binary texts and $\log r = o(w)$: $\Omega\left(\frac{\log n}{w^{\epsilon/(1-\epsilon)} \log r}\right)$ for any constant $\epsilon > 0$, where $w = \Omega(\log n)$ is the number of bits in the RAM word. In fact, since there are string families where $z = \Omega(r \log n)$ [68], no extraction mechanism in space $O(\text{poly}(r))$ can escape in general from the lower bound [79].

Summarizing Table 1 and our discussion, the situation on repetitive text indexing is as follows.

1. The RLFM-index is the only structure able to efficiently count the occurrences of P in T without having to enumerate them all. However, it does not offer efficient locating within $O(r)$ space.

2. The only structure clearly smaller than the RLFM-index, using $O(z)$ space [48], has unbounded locate time. Structures using about the same space, $O(g)$, have a one-time overhead quadratic in m in the locate time.

[4] `http://blog.coderstats.net/github/2013/event-types`, see the ratios of *push* per *create* and *commit* per *push*.

Index	Space	**Count time**
Mäkinen et al. [53, Thm. 17]	$O(r)$	$O(m(\frac{\log\sigma}{\log\log r} + (\log\log n)^2))$
This paper (Lem. 2.1)	$O(r)$	$O(m \log\log_w(\sigma + n/r))$

Index	Space	**Locate time**
Kreft and Navarro [48, Thm. 4.11]	$O(z)$	$O(m^2h + (m + occ)\log z)$
Gagie et al. [31, Thm. 4]	$O(z \log(n/z))$	$O(m \log m + occ \log\log n)$
Bille et al. [10, Thm. 1]	$O(z \log(n/z))$	$O(m(1 + \log^\epsilon z/\log(n/z)) + occ(\log^\epsilon z + \log\log n))$
Nishimoto et al. [64, Thm. 1]	$O(z \log n \log^* n)$	$O(m \log\log n \log\log z + \log z \log m \log n(\log^* n)^2$ $+occ \log n)$
Bille et al. [10, Thm. 1]	$O(z \log(n/z) \log\log z)$	$O(m + occ \log\log n)$
Claude and Navarro [19, Thm. 4]	$O(g)$	$O(m(m + \log n)\log n + occ \log^2 n)$
Claude and Navarro [20, Thm. 1]	$O(g)$	$O(m^2 \log\log_g n + (m + occ)\log g)$
Gagie et al. [30, Thm. 4]	$O(g + z \log\log z)$	$O(m^2 + (m + occ)\log\log n)$
Mäkinen et al. [53, Thm. 20]	$O(r + n/s)$	$O((m + s \cdot occ)(\frac{\log\sigma}{\log\log r} + (\log\log n)^2))$
Belazzougui et al. [4, Thm. 3]	$O(\overline{r} + z)$	$O(m(\log z + \log\log n) + occ(\log^\epsilon z + \log\log n))$
This paper (Thm. 3.1)	$O(r)$	$O(m \log\log_w(\sigma + n/r) + occ \log\log_w(n/r))$
This paper (Thm. 3.1)	$O(r \log\log_w(n/r))$	$O(m \log\log_w(\sigma + n/r) + occ)$
This paper (Thm. 5.1)	$O(r \log(n/r))$	$O(m + occ)$
This paper (Thm. 5.2)	$O(rw \log_\sigma(n/r))$	$O(m \log(\sigma)/w + occ)$
Belazzougui et al. [4, Thm. 4]	$O(e)$	$O(m \log\log n + occ)$
Takagi et al. [76, Thm. 9]	$O(\overline{e})$	$O(m + occ)$
Belazzougui and Cunial [2, Thm. 1]	$O(e)$	$O(m + occ)$

Structure	Space	**Extract time**
Kreft and Navarro [48, Thm. 4.11]	$O(z)$	$O(\ell\, h)$
Gagie et al. [32, Thm. 1–2]	$O(z \log n)$	$O(\ell + \log n)$
Rytter [71], Charikar et al. [15]	$O(z \log(n/z))$	$O(\ell + \log n)$
Bille et al. [10, Lem. 5]	$O(z \log(n/z))$	$O(\ell + \log(n/z))$
Gagie et al. [5, Thm. 2]	$O(z \log(n/z))$	$O((1 + \ell/\log_\sigma n)\log(n/z))$
Bille et al. [12, Thm. 1.1]	$O(g)$	$O(\ell + \log n)$
Belazzougui et al. [9, Thm. 1]	$O(g)$	$O(\log n + \ell/\log_\sigma n)$
Belazzougui et al. [9, Thm. 2]	$O(g \log^\epsilon n \log(n/g))$	$O(\log n/\log\log n + \ell/\log_\sigma n)$
Mäkinen et al. [53, Thm. 20]	$O(r + n/s)$	$O((\ell + s)(\frac{\log\sigma}{\log\log r} + (\log\log n)^2))$
This paper (Thm. 4.1)	$O(r \log(n/r))$	$O(\log(n/r) + \ell \log(\sigma)/w)$
Takagi et al. [76, Thm. 9]	$O(\overline{e})$	$O(\log n + \ell)$
Belazzougui and Cunial [2, Thm. 1]	$O(e)$	$O(\log n + \ell)$

Table 1: Previous and our new results on counting, locating, and extracting. We simplified some formulas with tight upper bounds. The main variables are the text size n, pattern length m, number of occurrences occ of the pattern, alphabet size σ, Lempel-Ziv parsing size z, smallest grammar size g, BWT runs r, CDAWG size e, and machine word length in bits w. Variable $h \leq n$ is the depth of the dependency chain in the Lempel-Ziv parse, and $\epsilon > 0$ is an arbitrarily small constant. Symbols $\overline{r}$ or $\overline{e}$ mean r or e of T plus r or e of its reverse. The z in Nishimoto et al. [64] refers to the Lempel-Ziv variant that does not allow overlaps between sources and targets (Kreft and Navarro [48] claim the same but their index actually works in either variant). Rytter [71] and Charikar et al. [15] enable the given extraction time because they produce balanced grammars of the given size (as several others that came later). Takagi et al. [76] claim time $O(m \log\sigma + occ)$ but they can reach $O(m + occ)$ by using perfect hashing.

3. Structures offering lower locate time require $O(z \log(n/z))$ space or more [31, 64, 10], $O(\overline{r} + z)$ space [4] (where $\overline{r}$ is the sum of r for T and its reverse), or $O(e)$ space or more [4, 76, 2].

4. Self-indexes with efficient extraction require $O(z \log(n/z))$ space or more [32, 5], $O(g)$ space [12, 9], or $O(e)$ space or more [76, 2].

1.1 Contributions Efficiently locating the occurrences of P in T within $O(r)$ space has been a bottleneck and an open problem for almost a decade. In this paper we give the first solution to this problem. Our precise contributions, largely detailed in Tables 1 and 2, are the following:

1. We improve the counting time of the RLFM-index to $O(m \log \log_w(\sigma + n/r))$, where $\sigma \leq r$ is the alphabet size of T.

2. We show how to locate each occurrence in time $O(\log \log_w(n/r))$, within $O(r)$ space. We reduce the locate time to $O(1)$ per occurrence by using slightly more space, $O(r \log \log_w(n/r))$.

3. We give the first structure built on BWT runs that replaces T while retaining direct access. It extracts any substring of length ℓ in time $O(\log(n/r) + \ell \log(\sigma)/w)$, using $O(r \log(n/r))$ space. As discussed, even the additive penalty is near-optimal [79].

4. By using $O(r \log(n/r))$ space, we obtain optimal locate time in the general setting, $O(m + occ)$. This had been obtained before only with space bounds $O(e)$ [2] or $O(\overline{e})$ [76]. By increasing the space to $O(rw \log_\sigma(n/r))$, we obtain optimal locate time $O(m \log(\sigma)/w + occ)$ in the packed setting (i.e., the pattern symbols come packed in blocks of $w/\log \sigma$ symbols per word). This had not been achieved so far by any compressed index, but only by uncompressed ones [62].

Contribution 1 is a simple update of the RLFM-index [53] with newer data structures for rank and predecessor queries [8]. We present it in Section 2, together with a review of the basic concepts needed to follow the paper.

Contribution 2 is one of the central parts of the paper, and is obtained in Section 3 in two steps. The first uses the fact that we can carry out the classical RLFM counting process for P in a way that we always know the position of one occurrence in T [68, 66]; we give a simpler proof of this fact in Lemma 3.1. The second shows that, if we know the position in T of one occurrence of BWT, then we can quickly obtain the preceding and following ones with an $O(r)$-size sampling. This is achieved by using the BWT runs to induce *phrases* in T (which are somewhat analogous to the Lempel-Ziv phrases [50]) and showing that the positions of occurrences within phrases can be obtained from the positions of their preceding phrase beginning. The time $O(1)$ is obtained by using an extended sampling.

In Section 4, Contribution 3 uses an analogue of the Block Tree [5] built on the BWT-induced phrases, which satisfies the same property that any distinct string has an occurrence overlapping a border between phrases.

For Contribution 4, we discard in Section 5 the RLFM-index and use a mechanism similar to the one used in Lempel-Ziv or grammar indexes [19, 20, 48] to find one *primary* occurrence, that is, one that overlaps phrase borders; then the others are found with the mechanism to obtain neighboring occurrences already described. Here we use a stronger property of primary occurrences that does not hold on those of Lempel-Ziv or grammars, and that might have independent interest. Further, to avoid time quadratic in m to explore all the suffixes of P, we use a (deterministic) mechanism based on Karp-Rabin signatures [1, 31], which we show how to compute (Las Vegas) from a variant of the structure we create for extracting text substrings. The optimal packed time is obtained by enlarging samplings.

We report preliminary experimental results in Section 6, showing that our new structure outperforms the alternatives by orders of magnitude in the space/time tradeoff map, and conclude in Section 7.

2 Basic Concepts

A string is a sequence $S[1..\ell] = S[1]S[2]\dots S[\ell]$, of length $\ell = |S|$, of symbols (or characters, or letters) chosen from an alphabet $[1..\sigma] = \{1, 2, \dots, \sigma\}$, that is, $S[i] \in [1..\sigma]$ for all $1 \leq i \leq \ell$. We use $S[i..j] = S[i]\dots S[j]$, with $1 \leq i, j \leq \ell$, to denote a substring of S, which is the empty string ε if $i > j$. A prefix of S is a substring of the form $S[1..i]$ and a suffix is a substring of the form $S[i..\ell]$. The juxtaposition of strings and/or symbols represents their concatenation.

We will consider indexing a *text* $T[1..n]$, which is a string over alphabet $[1..\sigma]$ terminated by the special symbol $\$ = 1$, that is, the lexicographically smallest one, which appears only at $T[n] = \$$. This makes any lexicographic comparison between suffixes well defined.

Our computation model is the transdichotomous RAM, with a word of $w = \Omega(\log n)$ bits, where all the standard arithmetic and logic operations can be carried out in constant time. In this article we generally measure space in words.

Functionality	Space (words)	Time
Count (Lem. 2.1)	$O(r)$	$O(m \log \log_w(\sigma + n/r))$
Count + Locate (Thm. 3.1)	$O(r)$	$O(m \log \log_w(\sigma + n/r) + occ \log \log_w(n/r))$
Count + Locate (Thm. 3.1)	$O(r \log \log_w(n/r))$	$O(m \log \log_w(\sigma + n/r) + occ)$
Count + Locate (Thm. 5.1)	$O(r \log(n/r))$	$O(m + occ)$
Count + Locate (Thm. 5.2)	$O(rw \log_\sigma(n/r))$	$O(m \log(\sigma)/w + occ)$
Extract (Thm. 4.1)	$O(r \log(n/r))$	$O(\log(n/r) + \ell \log(\sigma)/w)$

Table 2: Our contributions.

2.1 Suffix Trees and Arrays The *suffix tree* [80] of $T[1..n]$ is a compacted trie where all the n suffixes of T have been inserted. By compacted we mean that chains of degree-1 nodes are collapsed into a single edge that is labeled with the concatenation of the individual symbols labeling the collapsed edges. The suffix tree has n leaves and less than n internal nodes. By representing edge labels with pointers to T, the suffix tree uses $O(n)$ space, and can be built in $O(n)$ time [80, 56, 78, 23].

The *suffix array* [54] of $T[1..n]$ is an array $SA[1..n]$ storing a permutation of $[1..n]$ so that, for all $1 \leq i < n$, the suffix $T[SA[i]..]$ is lexicographically smaller than the suffix $T[SA[i+1]..]$. Thus $SA[i]$ is the starting position in T of the ith smallest suffix of T in lexicographic order. This can be regarded as an array collecting the leaves of the suffix tree. The suffix array uses n words and can be built in $O(n)$ time [46, 47, 41].

All the occurrences of a pattern string $P[1..m]$ in T can be easily spotted in the suffix tree or array. In the suffix tree, we descend from the root matching the successive symbols of P with the strings labeling the edges. If P is in T, the symbols of P will be exhausted at a node v or inside an edge leading to a node v; this node is called the *locus* of P, and all the occ leaves descending from v are the suffixes starting with P, that is, the starting positions of the occurrences of P in T. By using perfect hashing to store the first characters of the edge labels descending from each node of v, we reach the locus in optimal time $O(m)$ and the space is still $O(n)$. If P comes packed using $w/\log \sigma$ symbols per computer word, we can descend in time $O(m \log(\sigma)/w)$ [62], which is optimal in the packed model. In the suffix array, all the suffixes starting with P form a range $SA[sp..ep]$, which can be binary searched in time $O(m \log n)$, or $O(m + \log n)$ with additional structures [54].

The inverse permutation of SA, $ISA[1..n]$, is called the *inverse suffix array*, so that $ISA[j]$ is the lexicographical position of the suffix $T[j..n]$ among the suffixes of T.

Another important concept related to suffix arrays and trees is the longest common prefix array. Let $lcp(S, S')$ be the length of the longest common prefix between strings S and S', that is, $S[1..lcp(S, S')] = S'[1..lcp(S, S')]$ but $S[lcp(S, S') + 1] \neq S'[lcp(S, S') + 1]$. Then we define the *longest common prefix array* $LCP[1..n]$ as $LCP[1] = 0$ and $LCP[i] = lcp(T[SA[i-1]..], T[SA[i]..])$. The LCP array uses n words and can be built in $O(n)$ time [44].

2.2 Self-indexes A *self-index* is a data structure built on $T[1..n]$ that provides at least the following functionality:

Count: Given a pattern $P[1..m]$, count the number of occurrences of P in T.

Locate: Given a pattern $P[1..m]$, return the positions where P occurs in T.

Extract: Given a range $[i..i+\ell-1]$, return $T[i..i+\ell-1]$.

The last operation allows a self-index to act as a replacement of T, that is, it is not necessary to store T since any desired substring can be extracted from the self-index. This can be trivially obtained by including a copy of T as a part of the self-index, but it is challenging when the self-index uses less space than a plain representation of T.

In principle, suffix trees and arrays can be regarded as self-indexes that can count in time $O(m)$ or $O(m \log(\sigma)/w)$ (suffix tree, by storing occ in each node v) and $O(m \log n)$ or $O(m + \log n)$ (suffix array, with $occ = ep - sp + 1$), locate each occurrence in $O(1)$ time, and extract in time $O(1 + \ell \log(\sigma)/w)$. However, they use $O(n \log n)$ bits, much more than the $n \log \sigma$ bits needed to represent T in plain form. We are interested in *compressed self-indexes* [61, 63], which use the space required by a compressed representation of T (under some entropy model) plus some redundancy (at worst $o(n \log \sigma)$ bits). We describe later the FM-index, a particular self-index of interest to us.

2.3 Burrows-Wheeler Transform The *Burrows-Wheeler Transform* of $T[1..n]$, $BWT[1..n]$ [14], is a string defined as $BWT[i] = T[SA[i] - 1]$ if $SA[i] > 1$, and $BWT[i] = T[n] = \$$ if $SA[i] = 1$. That is, BWT

has the same symbols of T in a different order, and is a reversible transform.

The array BWT is obtained from T by first building SA, although it can be built directly, in $O(n)$ time and within $O(n \log \sigma)$ bits of space [57]. To obtain T from BWT [14], one considers two arrays, $L[1..n] = BWT$ and $F[1..n]$, the latter of which contains all the symbols of L (or T) in ascending order. Alternatively, $F[i] = T[SA[i]]$, so $F[i]$ follows $L[i]$ in T. We need a function that maps any $L[i]$ to the position j of that same character in F. The formula is $LF(i) = C[c] + \text{rank}[i]$, where $c = L[i]$, $C[c]$ is the number of occurrences of symbols less than c in L, and $\text{rank}[i]$ is the number of occurrences of symbol $L[i]$ in $L[1..i]$. A simple $O(n)$-time pass on L suffices to compute arrays $C[i]$ and $\text{rank}[i]$ using $O(n \log \sigma)$ bits of space. Once they are computed, we reconstruct $T[n] = \$$ and $T[n-k] \leftarrow L[LF^{k-1}(1)]$ for $k = 1, \ldots, n-1$, in $O(n)$ time as well.

2.4 Compressed Suffix Arrays and FM-indexes Compressed suffix arrays [61] are a particular case of self-indexes that simulate SA in compressed form. Therefore, they aim to obtain the suffix array range $[sp..ep]$ of P, which is sufficient to count since P then appears $occ = ep - sp + 1$ times in T. For locating, they need to access the content of cells $SA[sp], \ldots, SA[ep]$, without having SA stored.

The FM-index [25, 26] is a compressed suffix array that exploits the relation between the string $L = BWT$ and the suffix array SA. It stores L in compressed form (as it can be easily compressed to the high-order empirical entropy of T [55]) and adds sublinear-size data structures to compute (i) any desired position $L[i]$, (ii) the generalized *rank function* $\text{rank}_c(L, i)$, which is the number of times symbol c appears in $L[1..i]$. Note that these two operations permit, in particular, computing $\text{rank}[i] = \text{rank}_{L[i]}(L, i)$, which is called *partial rank*. Therefore, they compute

$$LF(i) \;=\; C[i] + \text{rank}_{L[i]}(L, i).$$

For counting, the FM-index resorts to *backward search*. This procedure reads P backwards and at any step knows the range $[sp_i, ep_i]$ of $P[i..m]$ in T. Initially, we have the range $[sp_{m+1}..ep_{m+1}] = [1..n]$ for $P[m+1..m] = \varepsilon$. Given the range $[sp_{i+1}..ep_{i+1}]$, one obtains the range $[sp_i..ep_i]$ from $c = P[i]$ with the operations

$$\begin{aligned} sp_i &= C[c] + \text{rank}_c(L, sp_{i+1} - 1) + 1, \\ ep_i &= C[c] + \text{rank}_c(L, ep_{i+1}). \end{aligned}$$

Thus the range $[sp..ep] = [sp_1..ep_1]$ is obtained with $O(m)$ computations of rank, which dominates the counting complexity.

For locating, the FM-index (and most compressed suffix arrays) stores sampled values of SA at regularly spaced text positions, say multiples of s. Thus, to retrieve SA, we find the smallest k for which $SA[LF^k(i)]$ is sampled, and then the answer is $SA[i] = SA[LF^k(i)] + k$. This is because function LF virtually traverses the text backwards, that is, it drives us from $L[i]$, which points to some $SA[i]$, to its corresponding position $F[j]$, which is preceded by $L[j]$, that is, $SA[j] = SA[i] - 1$:

$$SA[LF(i)] \;=\; SA[i] - 1.$$

Since it is guaranteed that $k < s$, each occurrence is located with s accesses to L and computations of LF, and the extra space for the sampling is $O((n \log n)/s)$ bits, or $O(n/s)$ words.

For extracting, a similar sampling is used on ISA, that is, we sample the positions of ISA that are multiples of s. To extract $T[i..i + \ell - 1]$ we find the smallest multiple of s in $[i + \ell..n]$, $j = s \cdot \lceil (i + \ell)/s \rceil$, and extract $T[i..j]$. Since $ISA[j] = p$ is sampled, we know that $T[j-1] = L[p]$, $T[j-2] = L[LF(p)]$, and so on. In total we require at most $\ell + s$ accesses to L and computations of LF to extract $T[i..i + \ell - 1]$. The extra space is also $O(n/s)$ words.

For example, using a representation [8] that accesses L and computes partial ranks in constant time (so LF is computed in $O(1)$ time), and computes rank in the optimal $O(\log \log_w \sigma)$ time, an FM-index can count in time $O(m \log \log_w \sigma)$, locate each occurrence in $O(s)$ time, and extract ℓ symbols of T in time $O(s + \ell)$, by using $O(n/s)$ space on top of the empirical entropy of T [8]. There exist even faster variants [7], but they do not rely on backward search.

2.5 Run-Length FM-index One of the sources of the compressibility of BWT is that symbols are clustered into $r \leq n$ *runs*, which are maximal substrings formed by the same symbol. Mäkinen and Navarro [51] proved a (relatively weak) bound on r in terms of the high-order empirical entropy of T and, more importantly, designed an FM-index variant that uses $O(r)$ words of space, called *Run-Length FM-index* or *RLFM-index*. They later experimented with several variants of the RLFM-index, where the variant RLFM+ [53, Thm. 17] corresponds to the original one [51].

The structure stores the *run heads*, that is, the first positions of the runs in BWT, in a data structure $E = \{1\} \cup \{1 < i \leq n, BWT[i] \neq BWT[i-1]\}$ that supports predecessor searches. Each element $e \in E$ has associated the value $e.p = |\{e' \in E, e' \leq e\}|$, which is its position in a string $L'[1..r]$ that stores the run symbols. Another array, $D[0..r]$, stores the cumulative lengths of the runs after sorting them lexicographically by their

symbols (with $D[0] = 0$). Let array $C'[1..\sigma]$ count the number of *runs* of symbols smaller than c in L. One can then simulate

$$\begin{aligned}\text{rank}_c(L,i) \;&=\; D[C'[c] + \text{rank}_c(L', pred(i).p - 1)] \\ &+\; [\text{if } L'[pred(i).p] = c \text{ then } i - pred(i) + 1 \text{ else } 0]\end{aligned}$$

at the cost of a predecessor search ($pred$) in E and a rank on L'. By using up-to-date data structures, the counting performance of the RLFM-index can be stated as follows.

LEMMA 2.1. *The Run-Length FM-index of a text $T[1..n]$ whose BWT has r runs can occupy $O(r)$ words and count the number of occurrences of a pattern $P[1..m]$ in time $O(m \log\log_w(\sigma + n/r))$. It also computes LF and access to any $BWT[p]$ in time $O(\log\log_w(n/r))$.*

Proof. We use the RLFM+ [53, Thm. 17], using the structure of Belazzougui and Navarro [8, Thm. 10] for the sequence L' (with constant access time) and the predecessor data structure described by Belazzougui and Navarro [8, Thm. 14] to implement E (instead of the bitvector they originally used). They also implement D with a bitvector, but we use a plain array. The sum of both operation times is $O(\log\log_w \sigma + \log\log_w(n/r))$, which can be written as $O(\log\log_w(\sigma + n/r))$. To access $BWT[p] = L[p]$ we only need a predecessor search on E, which takes time $O(\log\log_w(n/r))$. Finally, we compute LF faster than a general *rank* query, as we only need the partial rank query $rank_{L[i]}(L, i)$. This can be supported in constant time on L' using $O(r)$ space, by just recording all the answers, and therefore the time for LF on L is also dominated by the predecessor search on E, with $O(\log\log_w(n/r))$ time. □

We will generally assume that σ is the *effective* alphabet of T, that is, the σ symbols appear in T. This implies that $\sigma \le r \le n$. If this is not the case, we can map T to an effective alphabet $[1..\sigma']$ before indexing it. A mapping of $\sigma' \le r$ words then stores the actual symbols when extracting a substring of T is necessary. For searches, we have to map the m positions of P to the effective alphabet. By storing a predecessor structure of $O(\sigma') = O(r)$ words, we map each symbol of P in time $O(\log\log_w(\sigma/\sigma'))$ [8, Thm. 14]. This is within the bounds given in Lemma 2.1, which therefore holds for any alphabet size.

To provide locating and extracting functionality, Mäkinen et al. [53] use the sampling mechanism we described for the FM-index. Therefore, although they can efficiently count within $O(r)$ space, they need a much larger $O(n/s)$ space to support these operations in time proportional to $O(s)$. Despite various efforts [53], this has been a bottleneck in theory and in practice since then.

3 Locating Occurrences

In this section we show that, if the BWT of a text $T[1..n]$ has r runs, we can have an index using $O(r)$ space that not only efficiently finds the interval $SA[sp..ep]$ of the occurrences of a pattern $P[1..m]$ (as was already known in the literature, see previous sections) but that can locate each such occurrence in time $O(\log\log_w(n/r))$ on a RAM machine of w bits. Further, the time per occurrence may become constant if the space is raised to $O(r \log\log_w(n/r))$.

We start with Lemma 3.1, which shows that the typical backward search process can be enhanced so that we always know the position of one of the values in $SA[sp..ep]$. We give a simplification of the previous proof [68, 66]. Lemma 3.2 then shows how to efficiently obtain the two neighboring cells of SA if we know the value of one. This allows us to extend the first known cell in both directions, until obtaining the whole interval $SA[sp..ep]$. In Lemma 3.3 we show how this process can be sped up by using more space. Theorem 3.1 then summarizes the main result of this section.

We then extend the idea in order to obtain LCP values analogously to how we obtain SA values. While not of immediate use for locating, this result is useful later in the article and also has independent interest.

LEMMA 3.1. ([68, 66]) *We can store $O(r)$ words such that, given $P[1..m]$, in time $O(m \log\log_w(\sigma + n/r))$ we can compute the interval $SA[sp, ep]$ of the occurrences of P in T, and also return the position j and contents $SA[j]$ of at least one cell in the interval $[sp, ep]$.*

Proof. We store a RLFM-index and predecessor structures R_c storing the position in BWT of the right and left endpoints of each run of copies of c. Each element in R_c is associated to its corresponding text position, that is, we store pairs $\langle i, SA[i] - 1\rangle$ sorted by their first component (equivalently, we store in the predecessor structures their concatenated binary representation). These structures take a total of $O(r)$ words.

The interval of characters immediately preceding occurrences of the empty string is the entire $BWT[1..n]$, which clearly includes $P[m]$ as the last character in some run (unless P does not occur in T). It follows that we find an occurrence of $P[m]$ in predecessor time by querying $pred(R_{P[m]}, n)$.

Assume we have found the interval $BWT[sp, ep]$ containing the characters immediately preceding all the occurrences of some (possibly empty) suffix $P[i + 1..m]$ of P, and we know the position and contents of some

cell $SA[j]$ in the corresponding interval, $sp \le j \le ep$. Since $SA[LF(j)] = SA[j] - 1$, if $BWT[j] = P[i]$ then, after the next application of LF-mapping, we still know the position and value of some cell $SA[j']$ corresponding to the interval $BWT[sp', ep']$ for $P[i..m]$, namely $j' = LF(j)$ and $SA[j'] = SA[j] - 1$.

On the other hand, if $BWT[j] \neq P[i]$ but P still occurs somewhere in T (i.e., $sp' \le ep'$), then there is at least one $P[i]$ and one non-$P[i]$ in $BWT[sp, ep]$, and therefore the interval intersects an extreme of a run of copies of $P[i]$. Then, a predecessor query $pred(R_{P[i]}, ep)$ gives us the desired pair $\langle j', SA[j']-1\rangle$ with $sp \le j' \le ep$ and $BWT[j'] = P[i]$.

Therefore, by induction, when we have computed the BWT interval for P, we know the position and contents of at least one cell in the corresponding interval in SA.

To obtain the desired time bounds, we concatenate all the universes of the R_c structures into a single one of size σn, and use a single structure R on that universe: each $\langle x, SA[x-1]\rangle \in R_c$ becomes $\langle x+(c-1)n, SA[x]-1\rangle$ in R, and a search $pred(R_c, y)$ becomes $pred(R, (c-1)n + y) - (c-1)n$. Since R contains $2r$ elements on a universe of size σn, we can have predecessor searches in time $O(\log\log_w(n\sigma/r))$ and $O(r)$ space [8, Thm. 14]. This is the same $O(\log\log_w(\sigma + n/r))$ time we obtained in Lemma 2.1 to carry out the normal backward search operations on the RLFM-index. □

Lemma 3.1 gives us a toehold in the suffix array, and we show in this section that a toehold is all we need. We first show that, given the position and contents of one cell of the suffix array SA of a text T, we can compute the contents of the neighbouring cells in $O(\log\log_w(n/r))$ time. It follows that, once we have counted the occurrences of a pattern in T, we can locate all the occurrences in $O(\log\log_w(n/r))$ time each.

LEMMA 3.2. *We can store $O(r)$ words such that, given p and $SA[p]$, we can compute $SA[p-1]$ and $SA[p+1]$ in $O(\log\log_w(n/r))$ time.*

Proof. We parse T into phrases such that $T[i]$ is the first character in a phrase if and only if $i = 1$ or $q = SA^{-1}[i+1]$ is the first or last position of a run in BWT (i.e., $BWT[q] = T[i]$ starts or ends a run). We store an $O(r)$-space predecessor data structure with $O(\log\log_w(n/r))$ query time [8, Thm. 14] for the starting phrase positions in T (i.e., the values i just mentioned). We also store, associated with such values i in the predecessor structure, the positions in T of the characters immediately preceding and following q in BWT, that is, $N[i] = \langle SA[q-1], SA[q+1]\rangle$.

Suppose we know $SA[p] = k+1$ and want to know $SA[p-1]$ and $SA[p+1]$. This is equivalent to knowing the position $BWT[p] = T[k]$ and wanting to know the positions in T of $BWT[p-1]$ and $BWT[p+1]$. To compute these positions, we find with the predecessor data structure the position i in T of the first character of the phrase containing $T[k]$, take the associated positions $N[i] = \langle x, y\rangle$, and return $SA[p-1] = x + k - i$ and $SA[p+1] = y + k - i$.

To see why this works, let $SA[p-1] = j+1$ and $SA[p+1] = \ell+1$, that is, j and ℓ are the positions in T of $BWT[p-1] = T[j]$ and $BWT[p+1] = T[\ell]$. Note that, for all $0 \le t < k - i$, $T[k-t]$ is not the first nor the last character of a run in BWT. Thus, by definition of LF, $LF^t(p-1)$, $LF^t(p)$, and $LF^t(p+1)$, that is, the BWT positions of $T[j-t]$, $T[k-t]$, and $T[\ell-t]$, are contiguous and within a single run, thus $T[j-t] = T[k-t] = T[\ell-t]$. Therefore, for $t = k-i-1$, $T[j-(k-i-1)] = T[i+1] = T[\ell-(k-i+1)]$ are contiguous in BWT, and thus a further LF step yields that $BWT[q] = T[i]$ is immediately preceded and followed by $BWT[q-1] = T[j-(k-i)]$ and $BWT[q+1] = T[\ell-(k-i)]$. That is, $N[i] = \langle SA[q-1], SA[q+1]\rangle = \langle j-(k-i)+1, \ell-(k-i)+1\rangle$ and our answer is correct. □

The following lemma shows that the above technique can be generalized. The result is a space-time trade-off allowing us to list each occurrence in constant time at the expense of a slight increase in space usage.

LEMMA 3.3. *Let $s > 0$. We can store a data structure of $O(rs)$ words such that, given $SA[p]$, we can compute $SA[p-i]$ and $SA[p+i]$ for $i = 1, \ldots, s'$ and any $s' \le s$, in $O(\log\log_w(n/r) + s')$ time.*

Proof. Consider all BWT positions $j_1 < \cdots < j_t$ that are at distance at most s from a run border (we say that characters on run borders are at distance 1), and let $W[1..t]$ be an array such that $W[k]$ is the text position corresponding to j_k, for $k = 1, \ldots, t$. Let now $j_1^+ < \cdots < j_{t^+}^+$ be the BWT positions having a run border at most s positions after them, and $j_1^- < \cdots < j_{t^-}^-$ be the BWT positions having a run border at most s positions before them. We store the text positions corresponding to $j_1^+ < \cdots < j_{t^+}^+$ and $j_1^- < \cdots < j_{t^-}^-$ in two predecessor structures P^+ and P^-, respectively, of size $O(rs)$. We store, for each $i \in P^+ \cup P^-$, its position in W, that is, $W[f(i)] = i$.

To answer queries given $SA[p]$, we first compute its P^+-predecessor $i < SA[p]$ in $O(\log\log_w(n/r))$ time, and retrieve $f(i)$. Then, it holds that $SA[p+j] = W[f(i)+j] + (SA[p] - i)$, for $j = 0, \ldots, s$. Computing $SA[p-j]$ is symmetric (just use P^- instead of P^+).

To see why this procedure is correct, consider the range $SA[p..p+s]$. We distinguish two cases.

(i) $BWT[p..p+s]$ contains at least two distinct characters. Then, $SA[p]-1$ is inside P^+ (because p is

followed by a run break at most s positions away), and is therefore the immediate predecessor of $SA[p]$. Moreover, all BWT positions $[p, p+s]$ are in $j_1, \ldots, j_t$ (since they are at distance at most s from a run break), and their corresponding text positions are therefore contained in a contiguous range of W (i.e., $W[f(SA[p]-1)..f(SA[p]-1)+s]$). The claim follows.

(ii) $BWT[p..p+s]$ contains a single character; we say it is unary. Then $SA[p]-1 \notin P^+$, since there are no run breaks in $BWT[p..p+s]$. Moreover, by the LF formula, the LF mapping applied on the unary range $BWT[p..p+s]$ gives a contiguous range $BWT[LF(p)..LF(p+s)] = BWT[LF(p)..LF(p)+s]$. Note that this corresponds to a parallel backward step on text positions $SA[p] \to SA[p]-1, \ldots, SA[p+s] \to SA[p+s]-1$. We iterate the application of LF until we end up in a range $BWT[LF^\delta(p)..LF^\delta(p+s)]$ that is not unary. Then, $SA[LF^\delta(p)]-1$ is the immediate predecessor of $SA[p]$ in P^+, and δ is their distance (minus one). This means that with a single predecessor query on P^+ we "skip" all the unary BWT ranges $BWT[LF^i(p)..LF^i(p+s)]$ for $i = 1, \ldots, \delta-1$ and, as in case (i), retrieve the contiguous range in W containing the values $SA[p]-\delta, \ldots, SA[p+s]-\delta$ and add δ to obtain the desired SA values. □

Combining Lemmas 3.1 and 3.3, we obtain the main result of this section. The $O(\log\log_w(n/\sigma))$ additional time spent at locating is absorbed by the counting time.

THEOREM 3.1. *Let $s > 0$. We can store a text $T[1..n]$, over alphabet $[1..\sigma]$, in $O(rs)$ words, where r is the number of runs in the BWT of T, such that later, given a pattern $P[1..m]$, we can count the occurrences of P in T in $O(m\log\log_w(\sigma + n/r))$ time and (after counting) report their occ locations in overall time $O((1+\log\log_w(n/r)/s)\cdot occ)$.*

In particular, we can locate in $O(m\log\log_w(\sigma + n/r) + occ\log\log_w(n/r))$ time and $O(r)$ space or, alternatively, in $O(m\log\log_w(\sigma+n/r)+occ)$ time and $O(r\log\log_w(n/r))$ space.

Lemma 3.3 can be further extended to entries of the LCP array, which we will use later in the article. That is, given $SA[p]$, we compute $LCP[p]$ and its adjacent entries (note that we do not need to know p, but just $SA[p]$). The result is also an extension of a representation by Fischer et al. [28].

LEMMA 3.4. *Let $s > 0$. We can store a data structure of $O(rs)$ words such that, given $SA[p]$, we can compute $LCP[p-i+1]$ and $LCP[p+i]$, for $i = 1, \ldots, s'$ and any $s' \le s$, in $O(\log\log_w(n/r)+s')$ time.*

Proof. The proof follows closely that of Lemma 3.3, except that now we sample LCP entries corresponding to suffixes following sampled BWT positions. Let us define $j_1 < \cdots < j_t$, $j_1^+ < \cdots < j_{t^+}^+$, and $j_1^- < \cdots < j_{t^-}^-$, as well as the predecessor structures P^+ and P^-, exactly as in the proof of Lemma 3.3. We store $LCP'[1..t] = LCP[j_1], \ldots, LCP[j_t]$. We also store, for each $i \in P^+ \cup P^-$, its corresponding position $f(i)$ in LCP', that is, $LCP'[f(i)] = LCP[ISA[i+1]]$.

To answer queries given $SA[p]$, we first compute its P^+-predecessor $i < SA[p]$ in $O(\log\log_w(n/r))$ time, and retrieve $f(i)$. Then, it holds that $LCP[p+j] = LCP'[f(i)+j] - (SA[p]-i-1)$, for $j = 1, \ldots, s$. Computing $LCP[p-j]$ for $j = 0, \ldots, s-1$ is symmetric (just use P^- instead of P^+).

To see why this procedure is correct, consider the range $SA[p..p+s]$. We distinguish two cases.

(i) $BWT[p..p+s]$ contains at least two distinct characters. Then, as in case (i) of Lemma 3.3, $SA[p]-1$ is inside P^+ and is therefore the immediate predecessor $i = SA[p]-1$ of $SA[p]$. Moreover, all BWT positions $[p, p+s]$ are in $j_1, \ldots, j_t$, and therefore values $LCP[p..p+s]$ are explicitly stored in a contiguous range in LCP' (i.e., $LCP'[f(i)..f(i)+s]$). Note that $(SA[p]-i) = 1$, so $LCP'[f(i)+j] - (SA[p]-i-1) = LCP'[f(i)+j]$ for $j = 0, \ldots, s$. The claim follows.

(ii) $BWT[p..p+s]$ contains a single character; we say it is unary. Then we reason exactly as in case (ii) of Lemma 3.3 to define δ so that $i' = SA[LF^\delta(p)]-1$ is the immediate predecessor of $SA[p]$ in P^+ and, as in case (i) of this proof, retrieve the contiguous range $LCP'[f(i')..f(i')+s]$ containing the values $LCP[LF^\delta(p)..LF^\delta(p+s)]$. Since the skipped BWT ranges are unary, it is then not hard to see that $LCP[LF^\delta(p+j)] = LCP[p+j]+\delta$ for $j = 1, \ldots, s$ (note that we do not include $s = 0$ since we cannot exclude that, for some $i < \delta$, $LF^i(p)$ is the first position in its run). From the equality $\delta = SA[p] - i' - 1 = SA[p] - SA[LF^\delta(p)]$ (that is, δ is the distance between $SA[p]$ and its predecessor minus one or, equivalently, the number of LF steps virtually performed), we then compute $LCP[p+j] = LCP'[f(i')+j] - \delta$ for $j = 1, \ldots, s$. □

4 Extracting Substrings and Computing Fingerprints

In this section we consider the problem of extracting arbitrary substrings of $T[1..n]$. Though an obvious solution is to store a grammar-compressed version of T [12], little is known about the relation between the size g of the smallest grammar that generates T (which nevertheless is NP-hard to find [15]) and the number of runs r in its BWT. Another choice is to use block trees [5], which require $O(z\log(n/z))$ space, where z is the size of the Lempel-Ziv parse [50] of T. Again, z can be

larger or smaller than r [68].

Instead, we introduce a novel representation that uses $O(r\log(n/r))$ space and can retrieve any substring of length ℓ from T in time $O(\log(n/r) + \ell\log(\sigma)/w)$. This is similar (though incomparable) with the $O(\log(n/g) + \ell/\log_\sigma n)$ time that could be obtained with grammar compression [12, 9], and with the $O(\log(n/z) + \ell/\log_\sigma n)$ that could be obtained with block trees. Also, as explained in the Introduction, the $O(\log(n/r))$ additive penalty is near-optimal in general.

We first give an important result in Lemma 4.1: any desired substring of T has a *primary* occurrence, that is, one overlapping a border between phrases. The property is indeed stronger than in alternative formulations that hold for Lempel-Ziv parses [42] or grammars [19]: If we choose a primary occurrence overlapping at its leftmost position, then all the other occurrences of the string suffix must be preceded by the same prefix. This stronger property is crucial to design an optimal locating procedure in Section 5. The weaker property, instead, is sufficient to design in Theorem 4.1 a data structure reminiscent of block trees [5] for extracting substrings of T, which needs to store only some text around phrase borders. Finally, in Lemma 4.2, we show that a Karp-Rabin fingerprint [43, 31] of any substring of T can be obtained in time $O(\log(n/r))$, which will also be used in Section 5.

DEFINITION 1. *We say that a text character $T[i]$ is* sampled *if and only if $T[i]$ is the first or last character in its BWT run.*

DEFINITION 2. *We say that a text substring $T[i..j]$ is* primary *if and only if it contains at least one sampled character.*

LEMMA 4.1. *Every text substring $T[i..j]$ has a primary occurrence $T[i'..j'] = T[i..j]$ such that the following hold for some $i' \le p \le j'$:*

1. *$T[p]$ is sampled*
2. *$T[i'], \ldots, T[p-1]$ are not sampled*
3. *every text occurrence of $T[p..j']$ is always preceded by the string $T[i'..p-1]$*

Proof. We prove the lemma by induction on $j - i$. If $j - i = 0$, then $T[i..j]$ is a single character. Every character has a sampled occurrence i' in the text, therefore the three properties trivially hold for $p = i'$.

Let $j - i > 0$. By the inductive hypothesis, $T[i+1..j]$ has an occurrence $T[i'+1..j']$ satisfying the three properties for some $i'+1 \le p \le j'$. Let $[sp, ep]$ be the BWT range of $T[i+1..j]$. We distinguish three cases.

(i) All characters in $BWT[sp, ep]$ are equal to $T[i] = T[i']$ and are not the first or last in their run. Then, we leave p unchanged. $T[p]$ is sampled by the inductive hypothesis, so Property 1 still holds. Also, $T[i'+1], \ldots, T[p-1]$ are not sampled by the inductive hypothesis, and $T[i']$ is not sampled by assumption, so Property 2 still holds. By the inductive hypothesis, every text occurrence of $T[p..j']$ is always preceded by the string $T[i'+1..p-1]$. Since all characters in $BWT[sp, ep]$ are equal to $T[i] = T[i']$, Property 3 also holds for $T[i..j]$ and p.

(ii) All characters in $BWT[sp, ep]$ are equal to $T[i]$ and either $BWT[sp]$ is the first character in its run, or $BWT[ep]$ is the last character in its run (or both). Then, we set p to the text position corresponding to sp or ep, depending on which one is sampled (if both are sampled, choose sp). The three properties then hold trivially for $T[i..j]$ and p.

(iii) $BWT[sp, ep]$ contains at least one character $c \neq T[i]$. Then, there must be a run of $T[i]$'s ending or beginning in $BWT[sp, ep]$, meaning that there is a $sp \le q \le ep$ such that $BWT[q] = T[i]$ and the text position i' corresponding to q is sampled. We then set $p = i'$. Again, the three properties hold trivially for $T[i..j]$ and p. □

Lemma 4.1 has several important implications. We start by using it to build a data structure supporting efficient text extraction queries. In Section 5 we will use it to locate pattern occurrences in optimal time.

THEOREM 4.1. *Let $T[1..n]$ be a text over alphabet $[1..\sigma]$. We can store a data structure of $O(r\log(n/r))$ words supporting the extraction of any length-ℓ substring of T in $O(\log(n/r) + \ell\log(\sigma)/w)$ time.*

Proof. We describe a data structure supporting the extraction of $\alpha = \frac{w\log(n/r)}{\log\sigma}$ packed characters in $O(\log(n/r))$ time. To extract a text substring of length ℓ we divide it into $\lceil \ell/\alpha \rceil$ blocks and extract each block with the proposed data structure. Overall, this will take $O((\ell/\alpha + 1)\log(n/r)) = O(\log(n/r) + \ell\log(\sigma)/w)$ time.

Our data structure is stored in $O(\log(n/r))$ levels. For simplicity, we assume that r divides n and that n/r is a power of two. The top level (level 0) is special: we divide the text into r blocks $T[1..n/r]\,T[n/r + 1..2n/r] \ldots T[n - n/r + 1..n]$ of size n/r. For levels $i > 0$, we let $s_i = n/(r \cdot 2^{i-1})$ and, for every sampled position j (Definition 1), we consider the two non-overlapping blocks of length s_i: $X^1_{i,j} = T[j - s_i..j-1]$ and $X^2_{i,j} = T[j..j + s_i - 1]$. Each such block $X^k_{i,j}$, for $k = 1, 2$, is composed of two half-blocks, $X^k_{i,j} = X^k_{i,j}[1..s_i/2]\,X^k_{i,j}[s_i/2 + 1..s_i]$. We moreover consider

three additional consecutive and non-overlapping half-blocks, starting in the middle of the first, $X^1_{i,j}[1..s_i/2]$, and ending in the middle of the last, $X^2_{i,j}[s_i/2+1..s_i]$, of the 4 half-blocks just described: $T[j-s_i+s_i/4..j-s_i/4-1]$, $T[j-s_i/4..j+s_i/4-1]$, and $T[j+s_i/4..j+s_i-s_i/4-1]$.

From Lemma 4.1, blocks at level 0 and each half-block at level $i > 0$ have a primary occurrence at level $i+1$. Such an occurrence can be fully identified by the coordinate $\langle \mathit{off}, j' \rangle$, for $0 < \mathit{off} \leq s_{i+1}$ and j' sampled position, indicating that the occurrence starts at position $j' - s_{i+1} + \mathit{off}$.

Let i^* be the smallest number such that $s_{i^*} < 4\alpha = \frac{4w\log(n/r)}{\log\sigma}$. Then i^* is the last level of our structure. At this level, we explicitly store a packed string with the characters of the blocks. This uses in total $O(r \cdot s_{i^*}\log(\sigma)/w) = O(r\log(n/r))$ words of space. All the blocks at level 0 and half-block at levels $0 < i < i^*$ store instead the coordinates $\langle \mathit{off}, j' \rangle$ of their primary occurrence in the next level. At level $i^* - 1$, these coordinates point inside the strings of explicitly stored characters.

Let $S = T[i..i+\alpha-1]$ be the text substring to be extracted. Note that we can assume $n/r \geq \alpha$; otherwise all the text can be stored in plain packed form using $n\log(\sigma)/w < \alpha r\log(\sigma)/w \in O(r\log(n/r))$ words and we do not need any data structure. It follows that S either spans two blocks at level 0, or it is contained in a single block. The former case can be solved with two queries of the latter, so we assume, without losing generality, that S is fully contained inside a block at level 0. To retrieve S, we map it down to the next levels (using the stored coordinates of primary occurrences of half-blocks) as a contiguous text substring as long as this is possible, that is, as long as it fits inside a single half-block. Note that, thanks to the way half-blocks overlap, this is always possible as long as $\alpha \leq s_i/4$. By definition, then, we arrive in this way precisely to level i^*, where characters are stored explicitly and we can return the packed text substring. □

Using a similar idea, we can compute the Karp-Rabin fingerprint of any text substring in just $O(\log(n/r))$ time. This will be used in Section 5 to obtain our optimal-time locate solution.

LEMMA 4.2. *We can store a data structure of $O(r\log(n/r))$ words supporting computation of the Karp-Rabin fingerprint of any text substring in $O(\log(n/r))$ time.*

Proof. We store a data structure with $O(\log(n/r))$ levels, similar to the one of Theorem 4.1 but with two non-overlapping children blocks. Assume again that r divides n and that n/r is a power of two. The top level 0 divides the text into r blocks $T[1..n/r]\,T[n/r+1..2n/r]\ldots T[n-n/r+1..n]$ of size n/r. For levels $i > 0$, we let $s_i = n/(r\cdot 2^{i-1})$ and, for every sampled position j, we consider the two non-overlapping blocks of length s_i: $X^1_{i,j} = T[j-s_i..j-1]$ and $X^2_{i,j} = T[j..j+s_i-1]$. Each such block $X^k_{i,j}$ is composed of two half-blocks, $X^k_{i,j} = X^k_{i,j}[1..s_i/2]\,X^k_{i,j}[s_i/2+1..s_i]$. As in Theorem 4.1, blocks at level 0 and each half-block at level $i > 0$ have a primary occurrence at level $i+1$, meaning that such an occurrence can be written as $X^1_{i+1,j'}[L..s_{i+1}]\,X^2_{i+1,j'}[1..R]$ for some $1 \leq L, R \leq s_{i+1}$, and some sampled position j' (the special case where the half-block is equal to $X^2_{i+1,j'}$ is expressed as $L = s_{i+1}+1$ and $R = s_{i+1}$).

We associate with every block at level 0 and every half-block at level $i > 0$ the following information: its Karp-Rabin fingerprint κ, the coordinates $\langle j', L\rangle$ of its primary occurrence in the next level, and the Karp-Rabin fingerprints $\kappa(X^1_{i+1,j'}[L..s_{i+1}])$ and $\kappa(X^2_{i+1,j'}[1..R])$ of (the two pieces of) its occurrence. At level 0, we also store the Karp-Rabin fingerprint of every text prefix ending at block boundaries, $\kappa(T[1..jr])$ for $j = 1,\ldots,n/r$. At the last level, where blocks are of length 1, we only store their Karp-Rabin fingerprint (or we may compute them on the fly).

To answer queries $\kappa(T[i..j])$ quickly, the key point is to show that computing the Karp-Rabin fingerprint of a prefix or a suffix of a block translates into the same problem (prefix/suffix of a block) in the next level, and therefore leads to a single-path descent in the block structure. To prove this, consider computing the fingerprint of the prefix $\kappa(X^k_{i,j}[1..R'])$ of some block (computing suffixes is symmetric). Note that we explicitly store $\kappa(X^k_{i,j}[1..s_i/2])$, so we can consider only the problem of computing the fingerprint of a prefix of a half-block, that is, we assume $R' \leq s_i/2 = s_{i+1}$ (the proof is the same for the right half of $X^k_{i,j}$). Let $X^1_{i+1,j'}[L..s_{i+1}]\,X^2_{i+1,j'}[1..R]$ be the occurrence of the half-block in the next level. We have two cases. (i) $R' \geq s_{i+1}-L+1$. Then, $X^k_{i,j}[1..R'] = X^1_{i+1,j'}[L..s_{i+1}]\,X^2_{i+1,j'}[1..R'-(s_{i+1}-L+1)]$. Since we explicitly store the fingerprint $\kappa(X^1_{i+1,j'}[L..s_{i+1}])$, the problem reduces to computing the fingerprint of the block prefix $X^2_{i+1,j'}[1..R'-(s_{i+1}-L+1)]$. (ii) $R' < s_{i+1}-L+1$. Then, $X^k_{i,j}[1..R'] = X^1_{i+1,j'}[L..L+R'-1]$. Even though this is not a prefix nor a suffix of a block, note that $X^1_{i+1,j'}[L..s_{i+1}] = X^1_{i+1,j'}[L..L+R'-1]\,X^1_{i+1,j'}[L+R'..s_{i+1}]$. We explicitly store the fingerprint of the left-hand side of this equation, so the problem reduces to finding the fingerprint of $X^k_{i+1,j'}[L+R'..s_{i+1}]$, which is a suffix of a block. From both fingerprints we can compute $\kappa(X^1_{i+1,j'}[L..L+R'-1])$.

Note that, in order to combine fingerprints, we also

need the corresponding exponents and their inverses (i.e., $\sigma^{\pm\ell} \bmod q$, where ℓ is the string length and q is the prime used in κ). We store the exponents associated with the lengths of the explicitly stored fingerprints at all levels. The remaining exponents needed for the calculations can be retrieved by combining exponents from the next level (with a plain modular multiplication) in the same way we retrieve fingerprints by combining partial results from next levels.

To find the fingerprint of any text substring $T[i..j]$, we proceed as follows. If $T[i..j]$ spans at least two blocks at level 0, then $T[i..j]$ can be factored into (a) a suffix of a block, (b) a central part (possibly empty) of full blocks, and (c) a prefix of a block. Since at level 0 we store the Karp-Rabin fingerprint of every text prefix ending at block boundaries, the fingerprint of (b) can be found in constant time. Computing the fingerprints of (a) and (c), as proved above, requires only a single-path descent in the block structure, taking $O(\log(n/r))$ time each. If $T[i..j]$ is fully contained in a block at level 0, then we map it down to the next levels until it spans two blocks. From this point, the problem translates into a prefix/suffix problem, which can be solved in $O(\log(n/r))$ time. □

5 Locating in Optimal Time

In this section we show how to obtain optimal locating time in the unpacked — $O(m + occ)$ — and packed — $O(m \log(\sigma)/w + occ)$ — scenarios, by using $O(r \log(n/r))$ and $O(rw \log_\sigma(n/r))$ space, respectively. To improve upon the times of Theorem 3.1 we must abandon the idea of using the RLFM-index to find the toehold suffix array entry, as counting on the RLFM-index takes $\omega(m)$ time. We will use a different machinery that, albeit conceptually based on the BWT properties, does not use it at all. We exploit the idea that some pattern occurrence must cross a run boundary to build a structure that only finds pattern suffixes starting at a run boundary. By sampling more text suffixes around those run boundaries, we manage to find one pattern occurrence in time $O(m + \log(n/r))$, Lemma 5.3. We then show how the LCP information we obtained in Section 3 can be used to extract all the occurrences in time $O(m + occ + \log(n/r))$, in Lemma 5.5. Finally, by adding a structure that finds faster the patterns shorter than $\log(n/r)$, we obtain the unpacked result in Theorem 5.1. We use the same techniques, but with larger structures, in the packed setting, Theorem 5.2.

We make use of Lemma 4.1: if the pattern $P[1..m]$ occurs in the text then there must exist an integer $1 \le p \le m$ such that (1) $P[p..m]$ prefixes a text suffix $T[i+p-1..]$, where $T[i+p-1]$ is sampled, (2) none of the characters $T[i], \ldots, T[i+p-2]$ are sampled, and (3) $P[p..m]$ is always preceded by $P[1..p-1]$ in the text. It follows that $T[i..i+m-1] = P$. This implies that we can locate a pattern occurrence by finding the longest pattern suffix prefixing some text suffix that starts with a sampled character. Indeed, those properties are preserved if we enlarge the sampling, as proved next.

LEMMA 5.1. *Lemma 4.1 still holds if we add arbitrary sampled positions to the original sampling.*

Proof. If the leftmost sampled position $T[i+p-1]$ in the pattern occurrence belongs to the original sampling, then the properties hold by Lemma 4.1. If, on the other hand, $T[i+p-1]$ is one of the extra samples we added, then let $i'+p'-1$ be the position of the original sampling satisfying the three properties, with $p' > p$. Properties (1) and (2) hold for the sampled position $i+p-1$ by definition. By property (3) applied to $i'+p'-1$, we have that $P[p'..m]$ is always preceded by $P[1..p'-1]$ in the text. Since $p' > p$, it follows that also $P[p..m]$ is always preceded by $P[1..p-1]$ in the text, that is, property (3) holds for position $i+p-1$ as well. □

We therefore add to the sampling the r equally-spaced extra text positions $i \cdot (n/r)+1$, for $i = 0, \ldots, r-1$; we now have at most $3r$ sampled positions. This will be used later to efficiently locate patterns longer than n/r. The task of finding a pattern occurrence satisfying properties (1)–(3) on the extended sampling can be efficiently solved by inserting all the text suffixes starting with a sampled character in a data structure supporting fast prefix search operations and taking $O(r)$ words (e.g., a z-fast trie [1]). We make use of the following lemma.

LEMMA 5.2. ([31, 1]) *Let $\mathcal{S}$ be a set of strings and assume we have some data structure supporting extraction of any length-ℓ substring of strings in $\mathcal{S}$ in time $f_e(\ell)$ and computation of the Karp-Rabin fingerprint of any substring of strings in $\mathcal{S}$ in time f_h. We can build a data structure of $O(|\mathcal{S}|)$ words such that, later, we can solve the following problem in $O(m \log(\sigma)/w + t(f_h + \log m) + f_e(m))$ time: given a pattern $P[1..m]$ and $t > 0$ suffixes $Q_1, \ldots, Q_t$ of P, discover the ranges of strings in (the lexicographically-sorted) $\mathcal{S}$ prefixed by $Q_1, \ldots, Q_t$.*

Proof. Z-fast tries [1, App. H.3] already solve the *weak* part of the lemma in $O(m \log(\sigma)/w + t \log m)$ time. By *weak* we mean that the returned answer for suffix Q_i is not guaranteed to be correct if Q_i does not prefix any string in $\mathcal{S}$: we could therefore have false positives among the answers, but false negatives cannot occur. A procedure for deterministically discarding false positives has already been proposed [31] and requires extracting

substrings and their fingerprints from $\mathcal{S}$. We describe this strategy in detail in order to analyze its time complexity in our scenario.

First, we require the Karp-Rabin function κ to be collision-free between equal-length text substrings whose length is a power of two. We can find such a function at index-construction time in $O(n \log n)$ expected time and $O(n)$ space [11]. We extend the collision-free property to pairs of equal-letter strings of general length switching to the hash function κ' defined as $\kappa'(T[i..i+\ell-1]) = \langle \kappa(T[i..i+2^{\lfloor \log_2 \ell \rfloor} - 1]), \kappa(T[i+\ell-2^{\lfloor \log_2 \ell \rfloor}..i+\ell-1]) \rangle$. Let $Q_1, \ldots, Q_j$ be the pattern suffixes for which the prefix search found a candidate node. Order the pattern suffixes so that $|Q_1| < \cdots < |Q_j|$, that is, Q_i is a suffix of $Q_{i'}$ whenever $i < i'$. Let moreover $v_1, \ldots, v_j$ be the candidate nodes (explicit or implicit) of the z-fast trie such all substrings below them are prefixed by $Q_1, \ldots, Q_j$ (modulo false positives), respectively, and let $t_i = \mathit{string}(v_i)$ be the substring read from the root of the trie to v_i. Our goal is to discard all nodes v_k such that $t_k \neq Q_k$.

We compute the κ'-signatures of all candidate pattern suffixes $Q_1, \ldots, Q_t$ in $O(m \log(\sigma)/w + t)$ time. We proceed in rounds. At the beginning, let $a = 1$ and $b = 2$. At each round, we perform the following checks:

1. If $\kappa'(Q_a) \neq \kappa'(t_a)$: discard v_a and set $a \leftarrow a+1$ and $b \leftarrow b+1$.

2. If $\kappa'(Q_a) = \kappa'(t_a)$: let R be the length-$|t_a|$ suffix of t_b, i.e. $R = t_b[|t_b| - |t_a| + 1..|t_b|]$. We have two sub-cases:

   (a) $\kappa'(Q_a) = \kappa'(R)$. Then, we set $b \leftarrow b+1$ and a to the next integer a' such that $v_{a'}$ has not been discarded.

   (b) $\kappa'(Q_a) \neq \kappa'(R)$. Then, discard v_b and set $b \leftarrow b+1$.

3. If $b = j+1$: let v_f be the last node that was not discarded. Note that Q_f is the longest pattern suffix that was not discarded; other non-discarded pattern suffixes are suffixes of Q_f. We extract t_f. Let s be the length of the longest common suffix between Q_f and t_f. We report as a true match all nodes v_i that were not discarded in the above procedure and such that $|Q_i| \leq s$.

Intuitively, the above procedure is correct because we deterministically check that text substrings read from the root to the candidate nodes form a monotonically increasing sequence according to the suffix relation: $t_i \subseteq_{suf} t_{i'}$ for $i < i'$ (if the relation fails at some step, we discard the failing node). Comparisons to the pattern are delegated to the last step, where we explicitly compare the longest matching pattern suffix with t_f. For a full formal proof, see Gagie et al. [31].

For every candidate node we compute a κ'-signature from the set of strings ($O(f_h)$ time). For the last candidate, we extract a substring of length at most m ($O(f_e(m))$ time) and compare it with the longest candidate pattern suffix ($O(m \log(\sigma)/w)$ time). There are at most t candidates, so the verification process takes $O(m \log(\sigma)/w + t \cdot f_h + f_e(m))$. Added to the time spent to find the candidates in the z-fast trie, we obtain the claimed bounds. □

In our case, we use the results stated in Theorem 4.1 and Lemma 4.2 to extract text substrings and their fingerprints, so we get $f_e(m) = O(\log(n/r) + m \log(\sigma)/w)$ and $f_h = O(\log(n/r))$. Moreover note that, by the way we added the r equally-spaced extra text samples, if $m \geq n/r$ then the position p satisfying Lemma 5.1 must occur in the prefix of length n/r of the pattern. It follows that, for long patterns, it is sufficient to search the prefix data structure for only the $t = n/r$ longest pattern suffixes. We can therefore solve the problem stated in Lemma 5.2 in time $O(m \log(\sigma)/w + \min(m, n/r)(\log(n/r) + \log m))$. Note that, while the fingerprints are obtained with a randomized method, the resulting data structure offers deterministic worst-case query times and cannot fail.

To further speed up operations, for every sampled character $T[i]$ we insert in $\mathcal{S}$ the text suffixes $T[i-j..]$ for $j = 0, \ldots, \tau-1$, for some parameter τ that we determine later. This increases the size of the prefix-search structure to $O(r\,\tau)$ (excluding the components for extracting substrings and fingerprints), but in exchange it is sufficient to search only for aligned pattern suffixes of the form $P[\ell \cdot \tau + 1..m]$, for $\ell = 0, \ldots, \lceil \min(m, n/r)/\tau \rceil - 1$, to find any primary occurrence: to find the longest suffix of P that prefixes a string in $\mathcal{S}$, we keep an array $B[1..|\mathcal{S}|]$ storing the shift relative to each element in $\mathcal{S}$; for every sampled $T[i]$ and $j = 0, \ldots, \tau-1$, if k is the rank of $T[i-j..]$ among all suffixes in $\mathcal{S}$, then $B[k] = j$. We build a constant-time range minimum data structure on B, which requires only $O(|\mathcal{S}|) = O(r\,\tau)$ bits [27]. Let $[L, R]$ be the lexicographical range of suffixes in $\mathcal{S}$ prefixed by the longest aligned suffix of P that has occurrences in $\mathcal{S}$. With a range minimum query on B in the interval $[L, R]$ we find a text suffix with minimum shift, thereby matching the longest suffix of P. By Lemma 5.1, if P occurs in T then the remaining prefix of P appears to the left of the longest suffix found. However, if P does not occur in T this is not the case. We therefore verify the candidate occurrence of P using Theorem 4.1 in time $f_e(m) = O(\log(n/r) + m \log(\sigma)/w)$.

Overall, we find one pattern occurrence in

$O(m\log(\sigma)/w+(\min(m,n/r)/\tau+1)(\log m+\log(n/r)))$ time. By setting $\tau=\log(n/r)$, we obtain the following result.

LEMMA 5.3. *We can find one pattern occurrence in time $O(m+\log(n/r))$ with a structure using $O(r\log(n/r))$ space.*

Proof. If $m < n/r$, then it is easy to verify that $O(m\log(\sigma)/w + (\min(m,n/r)/\tau + 1)(\log m + \log(n/r))) = O(m+\log(n/r))$. If $m \geq n/r$, the running time is $O(m\log(\sigma)/w+((n/r)/\log(n/r)+1)\log m)$. The claim follows by noticing that $(n/r)/\log(n/r) = O(m/\log m)$, as $x/\log x = \omega(1)$. □

If we choose, instead, $\tau = w\log_\sigma(n/r)$, we can approach optimal-time locate in the packed setting.

LEMMA 5.4. *We can find one pattern occurrence in time $O(m\log(\sigma)/w+\log(n/r))$ with a structure using $O(rw\log_\sigma(n/r))$ space.*

Proof. We follow the proof of Lemma 5.3. The main difference is that, when $m \geq n/r$, we end up with time $O(m\log(\sigma)/w+\log m)$, which is $O(m)$ but not necessarily $O(m\log(\sigma)/w)$. However, if $m\log(\sigma)/w = o(\log m)$, then $m/\log m = o(w/\log\sigma)$ and thus $(n/r)/\log(n/r) = o(w/\log\sigma)$. The space we use, $O(rw\log_\sigma(n/r))$, is therefore $\omega(n)$, within which we can include a classical structure that finds one occurrence in $O(m\log(\sigma)/w)$ time (see, e.g., Belazzougui et al. [1, Sec. 7.1]). □

Let us now consider how to find the other occurrences. Note that, differently from Section 3, at this point we know the position of one pattern occurrence but we do not know its relative position in the suffix array nor the BWT range of the pattern. In other words, we can extract adjacent suffix array entries using Lemma 3.3, but we do not know where we are in the suffix array. More critically, we do not know when to stop extracting adjacent suffix array entries. We can solve this problem using LCP information extracted with Lemma 3.4: it is sufficient to continue extraction of candidate occurrences and corresponding LCP values (in both directions) as long as the LCP is greater than or equal to m. It follows that, after finding the first occurrence of P, we can locate the remaining ones in $O(occ+\log\log_w(n/r))$ time using Lemmas 3.3 and 3.4 (with $s=\log\log_w(n/r)$). This yields two first results with a logarithmic additive term over the optimal time.

LEMMA 5.5. *We can find all the occ pattern occurrences in time $O(m+occ+\log(n/r))$ with a structure using $O(r\log(n/r))$ space.*

LEMMA 5.6. *We can find all the occ pattern occurrences in time $O(m\log(\sigma)/w+occ+\log(n/r))$ with a structure using $O(rw\log_\sigma(n/r))$ space.*

To achieve the optimal running time, we must speed up the search for patterns that are shorter than $\log(n/r)$ (Lemma 5.5) and $w\log_\sigma n$ (Lemma 5.6). We index all the possible short patterns by exploiting the following property.

LEMMA 5.7. *There are at most $2rk$ distinct k-mers in the text, for any $1 \leq k \leq n$.*

Proof. From Lemma 4.1, every distinct k-mer appearing in the text has a primary occurrence. It follows that, in order to count the number of distinct k-mers, we can restrict our attention to the regions of size $2k-1$ overlapping the at most $2r$ sampled positions (Definition 1). The claim easily follows. □

Note that, without Lemma 5.7, we would only be able to bound the number of distinct k-mers by σ^k. We first consider achieving optimal locate time in the unpacked setting.

THEOREM 5.1. *We can store a text $T[1..n]$ in $O(r\log(n/r))$ words, where r is the number of runs in the BWT of T, such that later, given a pattern $P[1..m]$, we can report the occ occurrences of P in optimal $O(m+occ)$ time.*

Proof. We store in a path-compressed trie $\mathcal{T}$ all the strings of length $\log(n/r)$ occurring in the text. By Lemma 5.7, $\mathcal{T}$ has $O(r\log(n/r))$ leaves, and since it is path-compressed, it has $O(r\log(n/r))$ nodes. The texts labeling the edges are represented with offsets pointing inside r strings of length $2\log(n/r)$ extracted around each run boundary and stored in plain form (taking care of possible overlaps). Child operations on the trie are implemented with perfect hashing to support constant-time traversal.

In addition, we use the sampling structure of Lemma 3.3 with $s = \log\log_w(n/r)$. Recall from Lemma 3.3 that we store an array W such that, given any range $SA[sp..sp+s-1]$, there exists a range $W[i..i+s-1]$ and an integer δ such that $SA[sp+j] = W[i+j]+\delta$, for $j = 0,\ldots,s-1$. We store this information on $\mathcal{T}$ nodes: for each node $v \in \mathcal{T}$, whose string prefixes the range of suffixes $SA[sp..ep]$, we store in v the triple $\langle ep-sp+1, i, \delta\rangle$ such that $SA[sp+j] = W[i+j]+\delta$, for $j = 0,\ldots,s-1$.

Our complete locate strategy is as follows. If $m > \log(n/r)$, then we use the structures of Lemma 5.5, which already gives us $O(m+occ)$ time. Otherwise,

we search for the pattern in $\mathcal{T}$. If P does not occur in $\mathcal{T}$, then its number of occurrences must be zero, and we stop. If it occurs and the locus node of P is v, let $\langle occ, i, \delta\rangle$ be the triple associated with v. If $occ \leq s = \log\log_w(n/r)$, then we obtain the whole interval $SA[sp..ep]$ in time $O(occ)$ by accessing $W[i, i + occ - 1]$ and adding δ to the results. Otherwise, if $occ > s$, a plain application of Lemma 3.3 starting from the pattern occurrence $W[i] + \delta$ yields time $O(\log\log_w(n/r) + occ) = O(occ)$. Thus we obtain $O(m + occ)$ time and the trie uses $O(r\log(n/r))$ space. Considering all the structures, we obtain the main result. □

With more space, we can achieve optimal locate time in the packed setting.

THEOREM 5.2. *We can store a text $T[1..n]$ over alphabet $[1..\sigma]$ in $O(rw\log_\sigma(n/r))$ words, where r is the number of runs in the BWT of T, such that later, given a packed pattern $P[1..m]$, we can report the occ occurrences of P in optimal $O(m\log(\sigma)/w + occ)$ time.*

Proof. As in the proof of Theorem 5.1 we need to index all the short patterns, in this case of length at most $\ell = w\log_\sigma(n/r)$. We insert all the short text substrings in a z-fast trie to achieve optimal-time navigation. As opposed to the z-fast trie used in Lemma 5.2, now we need to perform the trie navigation (i.e., a prefix search) in only $O(m\log(\sigma)/w)$ time, that is, we need to avoid the additive term $O(\log m)$ that was instead allowed in Lemma 5.2, as it could be larger than $m\log(\sigma)/w$ for very short patterns. We exploit a result by Belazzougui et al. [1, Sec. H.2]: letting n' be the number of indexed strings of average length ℓ, we can support weak prefix search in optimal $O(m\log(\sigma)/w)$ time with a data structure of size $O(n'\ell^{1/c}(\log\ell + \log\log n))$ bits, for any constant c. Note that, since $\ell = O(w^2)$, this is $O(n')$ space for any $c > 2$. We insert in this structure all n' text ℓ-mers. For Lemma 5.7, $n' = O(r\ell)$. It follows that the prefix-search data structure takes space $O(r\ell) = O(rw\log_\sigma(n/r))$. This space is asymptotically the same of Lemma 5.6, which we use to find long patterns. We store in a packed string V the contexts of length ℓ surrounding sampled text positions ($O(rw\log_\sigma(n/r))$ space); z-fast trie nodes point to their corresponding substrings in V. After finding the candidate node on the z-fast trie, we verify it in $O(m\log(\sigma)/w)$ time by extracting a substring from V. We augment each trie node as done in Theorem 5.1 with triples $\langle occ, i, \delta\rangle$. The locate procedure works as for Theorem 5.1, except that now we use the z-fast trie mechanism to navigate the trie of all short patterns. □

6 Experimental Results

We implemented our simplest scheme and compared it with the state of the art.

6.1 Implementation We implemented the structure of Theorem 3.1 with $s = 1$ using the `sdsl` library [35]. For the run-length FM-index, we used the implementation described by Prezza [68, Thm. 28] (suffix array sampling excluded), taking $(1+\epsilon)r\log(n/r)+r(\log\sigma+2)$ bits of space for any constant $\epsilon > 0$ (in our implementation, $\epsilon = 0.5$) and supporting $O(\log(n/r) + \log\sigma)$-time LF mapping. This structure employs Huffman-compressed wavelet trees (`sdsl`'s `wt_huff`) to represent run heads, as in our experiments they turned out to be comparable in size and faster than Golynski et al.'s structure [36], which is implemented in `sdsl`'s `wt_gmr`. Our `locate` machinery is implemented as follows. We store two gap-encoded bitvectors `U` ("Up") and `D` ("Down") marking with a bit set text positions that are the last and first in their BWT run, respectively (we use these names because, in the BWT, each last run position is above—i.e. "Up"—the first run position in the following run). These bitvectors are implemented using `sdsl`'s `sd_vector`, take overall $2r(\log(n/r) + 2)$ bits of space, and answer queries in $O(\log(n/r))$ time. We moreover store two permutations, `DU` and `RD`. `DU` connects bits set in `D` and `U`: `DU`$[i] = j$ iff $T[$`D`$.select_1(i)]$ and $T[$`U`$.select_1(j)]$ are adjacent in the BWT and belong to two different runs (with $T[$`U`$.select_1(j)]$ being the last position of a run and $T[$`D`$.select_1(i)]$ the first position of the following run). Intuitively, `DU` permits moving bottom-up (from "Down" to "Up") in the BWT during the locate process. `RD` connects (ranks of) BWT runs and bits set in `D`: `RD`$[i] = j$ iff $T[$`D`$.select_1(j)]$ is the first character in the i-th BWT run. `DU` and `RD` are implemented using Munro et al.'s representation [58], take $(1 + \epsilon')r\log r$ bits each for any constant $\epsilon' > 0$, and support map and inverse in $O(1)$ time. It can easily be shown that `U`, `D`, `UD`, and `RD` are sufficient to locate each pattern occurrence in $O(\log(n/r))$ time with the strategy of Theorem 3.1. We choose $\epsilon' = \epsilon/2$. Overall, our index takes at most $r\log(n/r)+r\log\sigma+6r+(2+\epsilon)r\log n$ bits of space for any constant $\epsilon > 0$ and, after counting, locates each pattern occurrence in $O(\log(n/r))$ time. Note that this space is $(2 + \epsilon)r\log n + O(r)$ bits larger than an optimal run-length BWT representation. Since we store $2r$ suffix array samples, this is just $\epsilon r\log n + O(r)$ bits larger than the optimum (i.e., RLBWT + samples). In the following, we refer to our index as `r-index`. The code is publicly available [67].

6.2 Experimental setup We compared `r-index` with the state-of-the-art index for each compressibility

measure: `lzi` [17] (z), `slp` [17] (g), `rlcsa` [72] (r), and `cdawg` [70] (e). We tested `rlcsa` using three suffix array sample rates per dataset: the rate X resulting in the same size for `rlcsa` and `r-index`, plus rates $X/2$ and $X/4$. We measured memory usage and `locate` times per occurrence of all indexes on 1000 patterns of length 8 extracted from four repetitive datasets, which are also published with our implementation:

`DNA`, an artificial dataset of 629145 copies of a DNA sequence of length 1000 (Human genome) where each character was mutated with probability 10^{-3};

`boost`, a dataset consisting of concatenated versions of the `GitHub`'s `boost` library;

`einstein`, a dataset consisting of concatenated versions of `Wikipedia`'s English `Einstein` page;

`world_leaders`, a collection of all pdf files of CIA World Leaders from January 2003 to December 2009 downloaded from the `Pizza&Chili` corpus.

Memory usage (Resident Set Size, RSS) was measured using `/usr/bin/time` between index loading time and query time. This choice was motivated by the fact that, due to the datasets' high repetitiveness, the number *occ* of pattern occurrences was very large. This impacts sharply on the working space of indexes such as `lzi` and `slp`, which report the occurrences in a recursive fashion. When considering this extra space, these indexes always use more space than the `r-index`, but we prefer to emphasize the relation between the index sizes and their associated compressibility measure. The only existing implementation of `cdawg` works only on DNA files, so we tested it only on the `DNA` dataset.

6.3 Results The results of our experiments are summarized in Figure 1. On all datasets, the `r-index` significantly deviates from the space-time curve on which all other indexes are aligned. We locate occurrences one to three orders of magnitude faster than all other indexes except `cdawg`, which is however 35 times larger (and only twice as fast). It is also clear that `r-index` dominates all practical space-time tradeoffs of `rlcsa`: when the latter uses 30% more space than `r-index`, it is 7–150 times slower. The smallest indexes use 40%–80% (`lzi`) or 50%–100% (`slp`) of the space of `r-index`, at the expense of being orders of magnitude slower: 20–125 (`lzi`) or 8–40 (`slp`) times.

7 Conclusion

We have closed the long-standing problem of efficiently locating the occurrences of a pattern in a text using an index whose space is bounded by the number of equal-letter runs of the Burrows-Wheeler transform (BWT) of the text. The *occ* occurrences of a pattern $P[1..m]$ in a text $T[1..n]$ over alphabet $[1..\sigma]$ whose BWT has r runs are located in $O(m \log \log_w(\sigma + n/r) + occ \log \log_w(n/r))$ time, on a w-bit RAM machine, using an $O(r)$-space index. This near-optimal time was then reduced to the optimal $O(m + occ)$ using slightly more space, $O(r \log(n/r))$, and to the optimal in the packed setting, $O(m \log(\sigma)/w + occ)$ using $O(rw \log_\sigma(n/r))$ space.

The number of runs in the BWT is an important measure of the compressibility of highly repetitive text collections, which can be compressed by orders of magnitude by exploiting the repetitiveness. While the first index of this type [52, 53] managed to exploit the BWT runs, it was not able to locate occurrences efficiently. This gave rise to many other indexes based on other measures, like the size of a Lempel-Ziv parse [50], the size of a context-free grammar [45], the size of the smallest compact automaton recognizing the text substrings [13], etc. While the complexities are not always comparable, our experimental results show that our new index is either smaller or faster (or both) than all those (implemented) alternatives, by orders of magnitude.

Our index can be easily built in $O(r)$ space. We can start from any $O(r)$-space construction of the run-length BWT [53, 68, 65]. Once built, we scan it left-to-right collecting the $O(r)$ starts and ends of runs, and then scan it again in text order (using LF-steps from the position of the \$ symbol) so as to find the text positions of the run starts and ends. Finally, we build the predecessor data structures on the $O(r)$ samples.

Our results also yield a much deeper understanding of the properties of r as a measure of repetitiveness. In an extended version of this paper [33], we obtain several additional results:

1. Using a variant of locally consistent parsing [38], we build a run-length context free grammar of size $O(r \log(n/r))$ over the arrays SA, its inverse ISA, and LCP. This yields a data structure of that size giving access to any range of length ℓ in those arrays in time $O(\ell + \log(n/r))$.

2. We extend those representations to provide the richer functionality of a full-fledged compressed suffix tree, using $O(r \log(n/r))$ space and implementing most of the query and navigational operations in time $O(\log(n/r))$. We know of only one comparable alternative [3].

3. Building on the suffix tree functionality, we manage to count in $O(m)$ time with $O(r \log(n/r))$ space, or $O(m \log(\sigma)/w)$ time with $O(rw \log_\sigma(n/r))$ space, which is also unprecedented.

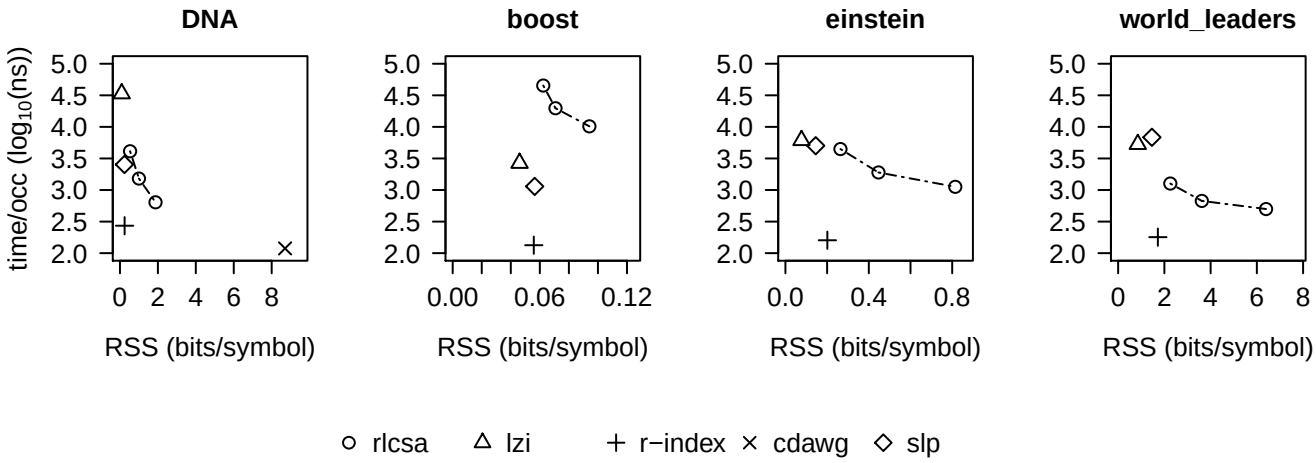

Figure 1: Locate time per occurrence and working space (in bits per symbol) of the indexes. The y-scale measures nanoseconds per occurrence reported and is logarithmic. `rlcsa` is tested on three different space-time tradeoffs.

4. We prove that the phrases in T we have defined in this paper are a particular case of a *bidirectional macro scheme* [75], which is arguably the most general technique to exploit repetitiveness. Using the results developed here, we are able to close a long-standing conjecture on the size z of the Lempel-Ziv parsing with respect to the size b of the smallest bidirectional scheme (which is NP-hard to find [34]): we show that $z = O(b \log(n/b))$ and that this is tight in terms of n.

References

[1] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*, pages 427–438, 2010.

[2] D. Belazzougui and F. Cunial. Fast label extraction in the CDAWG. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 161–175, 2017.

[3] D. Belazzougui and F. Cunial. Representing the suffix tree with the CDAWG. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LIPIcs 78, page article 7, 2017.

[4] D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Composite repetition-aware data structures. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 26–39, 2015.

[5] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. Ordóñez, S. J. Puglisi, and Y. Tabei. Queries on LZ-bounded encodings. In *Proc. 25th Data Compression Conference (DCC)*, pages 83–92, 2015.

[6] D. Belazzougui, T. Gagie, S. Gog, G. Manzini, and J. Sirén. Relative FM-indexes. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 52–64, 2014.

[7] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):article 23, 2014.

[8] D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.

[9] D. Belazzougui, S. J. Puglisi, and Y. Tabei. Access, rank, select in grammar-compressed strings. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*, pages 142–154, 2015.

[10] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LIPIcs 78, page article 16, 2017.

[11] P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time–space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014.

[12] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[13] A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

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

[15] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[16] S. Chen, E. Verbin, and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. *CoRR*, abs/1203.1080, 2012.

[17] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Suite of universal indexes for highly repetitive document collections. `https://github.com/migumar2/uiHRDC`. Accessed: 2017-06-08.

[18] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016.

[19] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2010.

[20] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 180–192, 2012.

[21] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative Lempel-Ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014.

[22] T. Elsayed and D. W. Oard. Modeling identity in archival collections of email: A preliminary study. In *Proc. 3rd Conference on Email and Anti-Spam (CEAS)*, 2006.

[23] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[24] A. Farruggia, T. Gagie, G. Navarro, S. J. Puglisi, and J. Sirén. Relative suffix trees. *CoRR*, abs/1508.02550, 2017.

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

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

[27] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[28] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[29] M. H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, pages 734–740, 2011.

[30] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*, pages 240–251, 2012.

[31] T. Gagie, P Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 731–742, 2014.

[32] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Approximate pattern matching in LZ77-compressed texts. *Journal of Discrete Algorithms*, 32:64–68, 2015.

[33] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in bwt-runs bounded space. *CoRR*, abs/1705.10382, 2017.

[34] J. K. Gallant. *String Compression Algorithms*. PhD thesis, Princeton University, 1982.

[35] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

[36] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 368–373. Society for Industrial and Applied Mathematics, 2006.

[37] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proc. 29th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 284–291, 2006.

[38] A. Jez. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.

[39] A. Jez. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016.

[40] C. Kapser and M. W. Godfrey. Improved tool support for the investigation of duplication in software. In *Proc. 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 305–314, 2005.

[41] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[42] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.

[43] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 2:249–260, 1987.

[44] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001.

[45] J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[46] D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2-4):126–142, 2005.

[47] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[48] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[49] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206, 2010.

[50] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

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

[52] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of individual genomes. In *Proc. 13th Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 121–137, 2009.

[53] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[54] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

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

[56] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[57] J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 408–424, 2017.

[58] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *ICALP*, volume 2033, pages 345–356. Springer, 2003.

[59] J. C. Na, H. Park, M. Crochemore, J. Holub, C. S. Iliopoulos, L. Mouchard, and K. Park. Suffix tree of alignment: An efficient index for similar data. In *Proc. 24th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 337–348, 2013.

[60] J. C. Na, H. Park, S. Lee, M. Hong, T. Lecroq, L. Mouchard, and K. Park. Suffix array of alignment: A practical index for similar data. In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 243–254, 2013.

[61] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[62] G. Navarro and Y. Nekrich. Time-optimal top-k document retrieval. *SIAM Journal on Computing*, 46(1):89–113, 2017.

[63] Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

[64] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and LZ factorization in compressed space. In *Proc. Prague Stringology Conference (PSC)*, pages 158–170, 2015.

[65] T. Ohno, Y. Takabatake, T. I, and H. Sakamoto. A faster implementation of online run-length Burrows-Wheeler Transform. *CoRR*, abs/1704.05233, 2017.

[66] A. Policriti and N. Prezza. Computing LZ77 in run-compressed space. In *Proc. 26th Data Compression Conference (DCC)*, pages 23–32, 2016.

[67] N. Prezza. r-index: the run-length BWT index. `https://github.com/nicolaprezza/r-index`. Accessed: 2017-06-08.

[68] N. Prezza. *Compressed Computation for Text Indexing*. PhD thesis, University of Udine, 2016.

[69] M. Przeworski, R. R. Hudson, and A. Di Rienzo. Adjusting the focus on human variation. *Trends in Genetics*, 16(7):296–302, 2000.

[70] M. Raffinot. Locate-cdawg: replacing sampling by CDAWG localisation in BWT indexing approaches. `https://github.com/mathieuraffinot/locate-cdawg`. Accessed: 2017-06-08.

[71] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[72] J. Sirén. Convenient repository for/fork of the RLCSA library. `https://github.com/adamnovak/rlcsa`. Accessed: 2017-06-08.

[73] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 164–175, 2008.

[74] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, Z. Chenxiang, M. J. Efron, R. Iyer, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical? *PLoS Biology*, 17(7):e1002195, 2015.

[75] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[76] T. Takagi, K. Goto, Y. Fujishige, S. Inenaga, and H. Arimura. Linear-size CDAWG: New repetition-aware indexing and grammar compression. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 304–316, 2017.

[77] K. Tao, F. Abel, C. Hauff, G.-J. Houben, and U. Gadiraju. Groundhog day: Near-duplicate detection on twitter. In *Proc. 22nd International World Wide Web Conference (WWW)*, pages 1273–1284, 2013.

[78] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[79] E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 247–258, 2013.

[80] P. Weiner. Linear Pattern Matching Algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory (FOCS)*, pages 1–11, 1973.