Indexing Highly Repetitive String Collections, Part II: Compressed Indexes

GONZALO NAVARRO, University of Chile, Chile

Two decades ago, a breakthrough in indexing string collections made it possible to represent them within their compressed space while at the same time offering indexed search functionalities. As this new technology permeated through applications like bioinformatics, the string collections experienced a growth that outperforms Moore's Law and challenges our ability of handling them even in compressed form. It turns out, fortunately, that many of these rapidly growing string collections are highly repetitive, so that their information content is orders of magnitude lower than their plain size. The statistical compression methods used for classical collections, however, are blind to this repetitiveness, and therefore a new set of techniques has been developed to properly exploit it. The resulting indexes form a new generation of data structures able to handle the huge repetitive string collections that we are facing. In this survey, formed by two parts, we cover the algorithmic developments that have led to these data structures.

In this second part, we describe the fundamental algorithmic ideas and data structures that form the base of all the existing indexes, and the various concrete structures that have been proposed, comparing them both in theoretical and practical aspects, and uncovering some new combinations. We conclude with the current challenges in this fascinating field.

CCS Concepts: • **Theory of computation → Data structures design and analysis**;

Additional Key Words and Phrases: Text indexing, string searching, compressed data structures, repetitive string collections

ACM Reference format:
Gonzalo Navarro. 2021. Indexing Highly Repetitive String Collections, Part II: Compressed Indexes. *ACM Comput. Surv.* 54, 2, Article 26 (February 2021), 32 pages.
https://doi.org/10.1145/3432999

1 INTRODUCTION

This is the second part of a survey on how to index string collections that are highly repetitive, that is, where most documents can be obtained from chunks of other documents plus a comparatively small amount of new material. In the first part [Navarro 2020], we argued for the importance of this field of study and for the need to separate the concept of *data size* from its *actual information content* to handle the sharp growth of the data in many fields. The idea is to define

Funded by ANID Basal Funds FB0001, Millennium Science Initiative Program - Code ICN17_002, and Fondecyt Grant 1-200038, Chile.

Author's address: G. Navarro, Center for Biotechnology and Bioengineering (CeBiB) and Millennium Institute for Foundational Research on Data (IMFD), Department of Computer Science, University of Chile, Beauchef 851, Santiago, Chile; email: gnavarro@dcc.uchile.cl.

0360-0300/2021/02-ART26 $15.00
https://doi.org/10.1145/3432999

suitable measures of the repetitiveness of the data, which appropriately describe its information content, and to design data representations that fit in space proportional to that information content, rather than to the sheer data size. We gave examples suggesting that 100-fold space reductions could be achieved on various actual repetitive collections like Wikipedia, GitHub, and genome repositories.

We also discussed the need to go beyond mere compression if we want to work within compressed space. We need *compressed data structures* that allow us not only to directly access the data without decompressing it but also to perform sophisticated queries on it. *Compressed text indexing*, the topic of this second part of the survey, is one of the most mature applications of those compressed data structures. An *index* represents a collection of strings in a compressed format that allows fast *pattern matching* on it, that is, find all the places where a given short query string appears in the collection. Indexed pattern matching is at the core of areas like Information Retrieval [Büttcher et al. 2010; Baeza-Yates and Ribeiro-Neto 2011], Data Mining [Liu 2007; Linstead et al. 2009; Silvestri 2010], Bioinformatics [Gusfield 1997; Ohlebusch 2013; Mäkinen et al. 2015] Multimedia Retrieval [Typke et al. 2005; Su et al. 2010], and others.

The need of compressed data structures for pattern matching was recognized a couple of decades ago [Ferragina and Manzini 2000; Grossi and Vitter 2000], and there are nowadays mature and successful indexes [Navarro and Mäkinen 2007; Grossi 2011] that have made their way to applications; see for example bioinformatic software like `Bowtie`,[1] `BWA`,[2] or `Soap2`.[3] These indexes, however, build on statistical compression that, as we show in the first part of the survey, does not exploit the high degree of repetitiveness that arises in many applications.

This challenge was explicitly recognized almost a decade later [Sirén et al. 2008], in the same bioinformatic context. After about another decade, the specific challenges of text indexing on highly repetitive string collections have become apparent, but also there has been significant progress and important results have been reached.

Our aim in this second part of the survey is to give an exhaustive, yet friendly, coverage of the discoveries in the area of compressed text indexes for highly repetitive text collections. We build on the repetitiveness measures and access methods covered in the first part, which we summarize in Section 2, together with some notation remarks and basic concepts. The indexes are divided into those that build on "parsings" (i.e., partitions) of the string induced by the repetitiveness measures, in Section 3, and those that build on string suffixes, in Section 4. Finally, Section 5 discusses some open challenges in this fascinating area.

2 NOTATION AND BASIC CONCEPTS

We assume basic knowledge on algorithms, data structures, and algorithm analysis. In this section, we define some fundamental concepts on strings, preceded by a few more general concepts and notation remarks. To make this second part self-contained, we repeat some of the concepts given in Part I [Navarro 2020, Sec. 2] (sometimes with a different emphasis) and also summarize its main results.

Computation model. We use the RAM model of computation, where we assume the programs run on a random-access memory where words of $w = \Theta(\log n)$ bits are accessed and manipulated in constant time, where n is the input size. All the typical arithmetic and logical operations on the machine words are carried out in constant time, including multiplication and bit operations.

[1] `http://bowtie-bio.sourceforge.net.`
[2] `http://bio-bwa.sourceforge.net.`
[3] `http://soap.genomics.org.cn.`

Complexities. We will use big-O notation for the time complexities and in many cases for the space complexities as well. Space complexities are measured in amount of computer words, that is, $O(X)$ space means $O(X \log n)$ bits. By poly x we mean any polynomial in x, that is, $x^{O(1)}$, and polylog x denotes poly $(\log x)$. Logarithms will be to the base 2 by default. Within big-O complexities, $\log x$ must be understood as $\lceil \log(2+x) \rceil$, to avoid border cases.

2.1 Strings

A *string* $S = S[1\,..\,n]$ is a sequence of *symbols* drawn from a set Σ called the *alphabet*. We will assume $\Sigma = \{1, 2, \ldots, \sigma\}$. The *length* of $S[1\,..\,n]$ is n, also denoted $|S|$. We use $S[i]$ to denote the i-th symbol of S and $S[i\,..\,j] = S[i] \ldots S[j]$ to denote a *substring* of S. If $i > j$, then $S[i\,..\,j] = \varepsilon$, the empty string. A *prefix* of S is a substring of the form $S[1\,..\,j]$ and a *suffix* is a substring of the form $S[i\,..\,n] = S[i\,..\,]$. With SS' we denote the *concatenation* of the strings S and S', that is, the symbols of S' are appended after those of S. Sometimes we identify a single symbol with a string of length 1, so that aS and Sa, with $a \in \Sigma$, denote concatenations as well.

The *lexicographic order* among strings is defined as in a dictionary. Let $a, b \in \Sigma$ and let S and S' be strings. Then $aS \leq bS'$ if $a < b$ or if $a = b$ and $S \leq S'$, and $\varepsilon \leq S$ for every S.

For technical convenience, we will often assume that strings $S[1\,..\,n]$ are terminated with a special symbol $S[n] = \$$, which does not appear elsewhere in S nor in Σ. We assume that \$ is smaller than every symbol in Σ to be consistent with the lexicographic order. The string $S[1\,..\,n]$ read backwards is denoted $S^{rev} = S[n] \cdots S[1]$; note that in this case the terminator does not appear at the end of S^{rev}.

2.2 Pattern Matching

The *indexed pattern matching problem* consists in, given a sequence $S[1\,..\,n]$, build a data structure (called an *index*) so that, later, given a query string $P[1\,..\,m]$, one efficiently finds the *occ* places in S where P occurs, that is, one outputs the set $Occ = \{i,\ S[i\,..\,i+m-1] = P\}$.

With "efficiently," we mean that, in an indexed scenario, we expect the search times to be sublinear in n, typically of the form $O((\text{poly}\, m + occ)\ \text{polylog}\, n)$. The *optimal search time*, since we have to read the input and write the output, is $O(m + occ)$. Since P can be represented in $m \log \sigma$ bits, in a few cases we will go further and assume that P comes packed into $O(m/\log_\sigma n)$ consecutive machine words, in which case the RAM-optimal time is $O(m/\log_\sigma n + occ)$.

In general, we will handle a collection of \$-terminated strings, $S_1, \ldots, S_d$, but we model the collection by concatenating the strings into a single one, $S[1\,..\,n] = S_1 \cdots S_d$, and doing pattern matching on S.

2.3 Classic Text Indexes

Suffix trees, *suffix arrays*, and *CDAWGs* are the most classical pattern matching indexes. The *suffix tree* [Weiner 1973; McCreight 1976; Apostolico 1985] is a trie (or digital tree) containing all the suffixes of S. That is, every suffix of S labels a single root-to-leaf path in the suffix tree, and no node has two distinct children labeled by the same symbol. Further, the unary paths (i.e., paths of nodes with a single child) are compressed into single edges labeled by the concatenation of the contracted edge symbols. Every internal node in the suffix tree corresponds to a substring of S that appears more than once, and every leaf corresponds to a suffix. The leaves of the suffix tree indicate the position of S where their corresponding suffixes start. Since there are n suffixes in S, there are n leaves in the suffix tree, and since there are no nodes with a single child, it has less than n internal nodes. The suffix tree can then be represented within $O(n)$ space, for example by representing every string labeling edges with a couple of pointers to an occurrence of the label in S. The suffix tree can also be built in linear (i.e., $O(n)$) time [Weiner 1973; McCreight 1976; Ukkonen 1995; Farach-Colton et al. 2000].

The suffix tree is a very popular data structure in stringology and bioinformatics [Apostolico 1985; Crochemore and Rytter 2002; Gusfield 1997], supporting a large number of complex searches (by using extra information, such as suffix links, that we omit here). The most basic search is pattern matching: Since all the occurrences of P in S are prefixes of suffixes of S, we find them all by descending from the root following the successive symbols of P. If at some point we cannot descend by some $P[i]$, then P does not occur in S. Otherwise, we exhaust the symbols of P at some suffix tree node v or in the middle of some edge leading to v. We then say that v is the *locus* of P: Every leaf descending from v is a suffix starting with P. If the children $v_1, \ldots, v_k$ of every suffix tree node v are stored with perfect hashing (the keys being the first symbols of the strings labeling the edges (v, v_i)), then we reach the locus node in time $O(m)$. Further, since the suffix tree has no unary paths, the *occ* leaves with the occurrences of P are traversed from v in time $O(occ)$. In total, the suffix tree supports pattern matching in optimal time $O(m + occ)$. With more sophisticated structures, it supports RAM-optimal time search, $O(m/\log_\sigma n + occ)$ [Navarro and Nekrich 2017].

A convenient way to regard the suffix tree is as the *Patricia tree* [Morrison 1968] of all the suffixes of S. The Patricia tree, also known as *blind trie* [Ferragina and Grossi 1999] (their technical differences are not important here) is a trie where we compact the unary paths and retain only the first symbol and the length of the string labeling each edge. In this case we use the first symbols to choose the appropriate child, and simply trust that the omitted symbols match P. When arriving at the potential locus v of P, we jump to any leaf, where a potential occurrence $S[i\,..\,i+m-1]$ of P is pointed, and compare P with $S[i\,..\,i+m-1]$. If they match, then v is the correct locus of P and all its leaves match P; otherwise P does not occur in S. A pointer from each node v to a leaf descending from it is needed to maintain the verification within the optimal search time.

The suffix array [Manber and Myers 1993] of $S[1\,..\,n]$ is the array $A[1\,..\,n]$ of the positions of the suffixes of S in lexicographic order. If the children of the suffix tree nodes are lexicographically ordered by their first symbol, then the suffix array corresponds to the leaves of the suffix tree. The suffix array can be built directly, without building the suffix tree, in linear time [Kim et al. 2005; Ko and Aluru 2005; Kärkkäinen et al. 2006].

All the suffixes starting with P form a range in the suffix array $A[sp\,..\,ep]$. We can find the range with binary search in time $O(m \log n)$, by comparing P with the strings $S[A[i]\,..\,A[i]+m-1]$, so as to find the smallest and largest suffixes that start with P. The search time can be reduced to $O(m + \log n)$ by using further data structures [Manber and Myers 1993].

The Compact Deterministic Acyclic Word Graph (CDAWG) [Blumer et al. 1987] is obtained by merging all the identical subtrees of the suffix tree. The suffix trees of repetitive strings tend to have large isomorphic subtrees, which yields small CDAWGs. Every suffix of S corresponds to a distinct path from the root to the final node, and thus the search process is similar to that on suffix trees. Once the locus of P is found, each distinct path from it to the final node corresponds to an occurrence of P in S.

Example. Figure 1 (modified from Part I) shows the suffix tree, suffix array, and CDAWG of the string $S =$ alabaralalabarda\$. The search for $P =$ lab in the suffix tree leads to the grayed locus node: The search in fact falls in the middle of the edge from the parent to the locus node. The two leaves descending from the locus contain the positions 2 and 10, which is where P occurs in S. In the suffix array, we find with binary search the interval $A[13\,..\,14]$, where the answers lie. In the CDAWG, each path from the (grayed) locus to the final node corresponds to an occurrence.

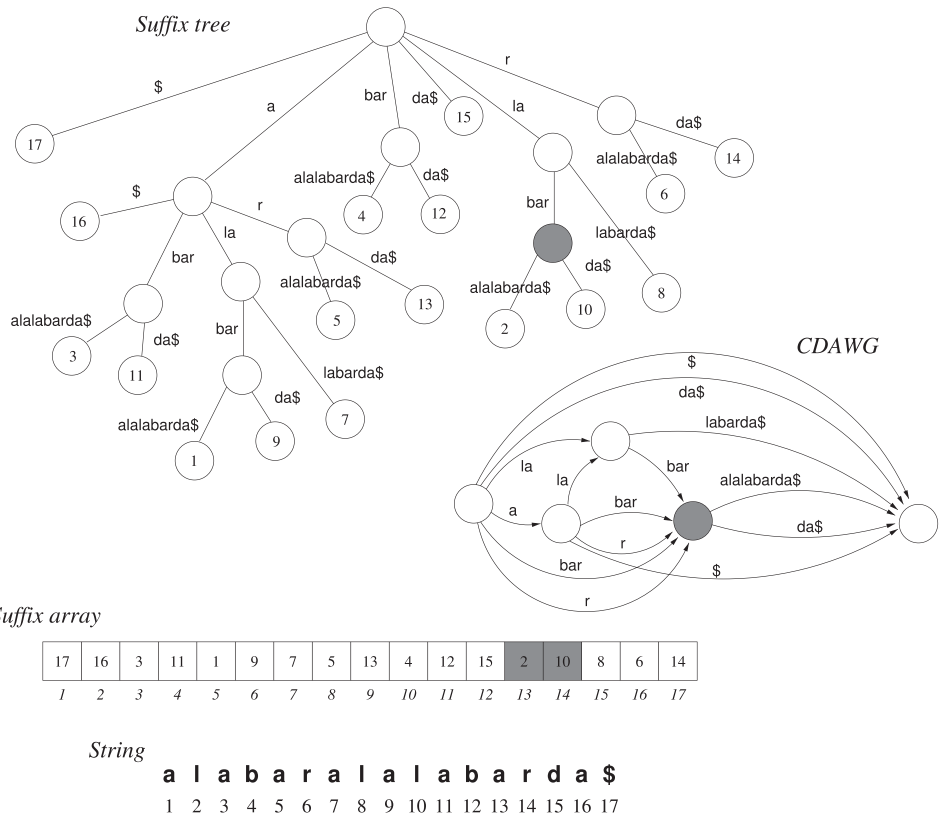

Fig. 1. The suffix tree, suffix array, and CDAWG of the string S = alabaralalabarda\$. The suffix tree leaves indicate the positions where the corresponding suffixes start, and those collected positions form the suffix array. The locus suffix tree and CDAWG node, and the suffix array interval, for P = lab, are grayed.

2.4 Karp-Rabin Fingerprints

Karp and Rabin [1987] proposed a technique to compute a *signature* or *fingerprint* of a string via hashing, in a way that enables (non-indexed) string matching in $O(n)$ average time. The signature $\kappa(Q)$ of a string $Q[1\,..\,q]$ is defined as

$$\kappa(Q) \;=\; \left(\sum_{i=1}^{q} Q[i]\cdot b^{i-1}\right) \bmod p,$$

where b is an integer and p a prime number. It is easy to maintain $\kappa(Q)$ in constant time upon symbol additions/removals at the front/rear of Q. Therefore, we can compute the signatures for all the prefixes or suffixes of Q in time $O(q)$.

By appropriately choosing b and p, the probability of two substrings having the same fingerprint is very low. Further, in $O(n \log n)$ expected time, we can find a function κ that ensures that no two different substrings of $S[1\,..\,n]$ have the same fingerprint [Bille et al. 2014]: They build fingerprints κ' that are collision-free only over substrings of lengths that are powers of two and then define $\kappa(Q) = \langle \kappa'(Q[1\,..\,2^{\lfloor \log_2 q \rfloor}]), \kappa'(Q[q - 2^{\lfloor \log_2 q \rfloor} + 1\,..\,q])\rangle$.

Table 1. The Measures of Repetitiveness Covered in Part I and Their Construction Cost

Measure	Meaning	Cost
z	Number of phrases in the Lempel-Ziv parse of the string	$O(n)$
z_{no}	Like z, but sources and phrases cannot overlap	$O(n)$
b	Number of phrases in the smallest bidirectional macro scheme	NP-hard
g	Size of the smallest context-free grammar generating the string	NP-hard
g_{rl}	Like g, but run-length rules are permitted	NP-hard
c	Size of the smallest collage system generating the string	Unknown
r	Number of runs in the Burrows-Wheeler Transform of the string	$O(n)$
v	Number of phrases in the lex-parse of the string	$O(n)$
e	Number of nodes plus edges of the CDAWG of the string	$O(n)$
γ	Minimum size of an attractor for the string	NP-hard
δ	Maximum $S(k)/k$, where S contains $S(k)$ distinct k-grams	$O(n)$

The NP-hard measures are all $O(\log n)$-approximable in $O(n)$ time.

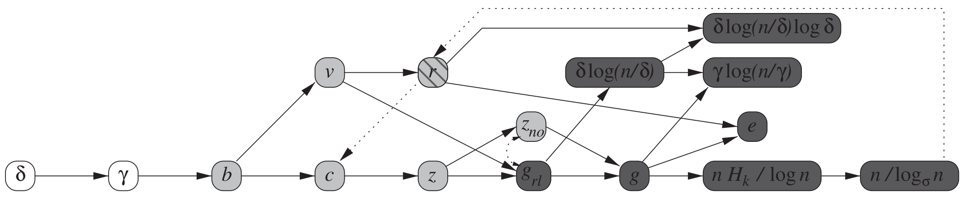

Fig. 2. Relations between the compressibility measures. A solid arrow from X to Y means that $X = O(Y)$ for all string families. For all solid and dotted arrows, there are string families where $X = o(Y)$, with the exceptions of $\gamma \to b$ and $c \to z$. Grayed measures X mean that we can encode every string in $O(X)$ space; darker gray means that we can also provide efficient access and indexed searches within $O(X)$ space; for r we can only provide indexed searches.

2.5 Measures of Repetitiveness and Access Methods

Table 1 recalls the repetitiveness measures covered in Part I of this survey [Navarro 2020], and Figure 2 (copied from Part I) summarizes their relations, as well as the ability to compress, provide direct access, and provide indexed searches within those spaces. It typically holds in practice that $b < z \approx v < g < r < e$, where "<" denotes a clear difference in magnitude.

In Part I, we also give the complexities within which we can extract an arbitrary substring of length ℓ and compute the Karp-Rabin signature of such a substring. The following space/time tradeoffs are obtained:

(1) Using *grammars*, within $O(g_{rl})$ space:
 —$O(\ell/\log_\sigma n + \log n)$ time for extracting a substring of length ℓ.
 —$O(\log n)$ time for computing a signature.
 —Real-time extraction of nonterminal prefixes or suffixes (i.e., $O(1)$ time for each new symbol extracted).

(2) Using *block trees*, within $O(\delta \log(n/\delta))$ space:
 —$O((1 + \ell/\log_\sigma n)\log(n/g))$ time for extraction.
 —$O(\log(n/g))$ time for computing a signature.

(3) Using *bookmarking*, with $O((p + \gamma)\log\log n)$ extra space, over any parse of p phrases:
 —$O(\ell)$ time for extracting a phrase prefix or suffix.
 —$O(\log \ell)$ time for computing the signature of a phrase prefix or suffix.

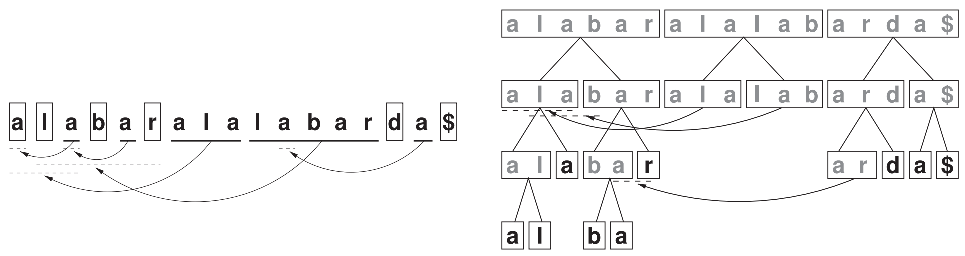

Fig. 3. On the left, the Lempel-Ziv parse of S = alabaralalabarda\$. Each phrase is either an underlined string, which appears before, or a boxed symbol. The arrows go from each underlined string to some of its occurrences to the left (its source, which is underlined with a dashed line). On the right, a block tree on the same string, which can be seen as a restricted Lempel-Ziv parse.

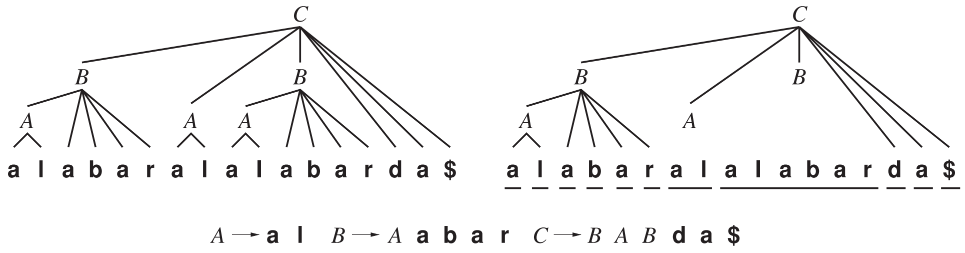
C
B
A
a l a b a r a l a l a b a r d a $
A → a l B → A a b a r C → B A B d a $

Fig. 4. A context-free grammar generating the string S = alabaralalabarda\$. The rules of the grammar are on the bottom. On the top we show the parse tree (left) and a grammar tree (right), emphasizing the parsing it induces on S with the phrases underlined.

To follow the next sections in a self-contained form, we recall that measures z, z_{no}, b, r, and v define, or at least induce, a parse of S into pieces.

Example. To define z via the Lempel-Ziv parse, each piece is the longest prefix of the rest of the string that appears earlier in S, or it is an explicit symbol if it appears for the first time. The left of Figure 3 (copied from Part I) shows how the string S = alabaralalabarda\$ is parsed into $z = 11$ phrases, a|l|a|b|a|r|ala|labar|d|a|\$.

Grammar-based methods, which are used to define the measures g and g_{rl}, define a *parse tree* on top of S, which starts with the initial symbol at the root and ends with the terminals that spell out S at the leaves. The *grammar tree* is defined by pruning all but one occurrence of each distinct nonterminal from the parse tree, and it has $g + 1$ or $g_{rl} + 1$ nodes. The segments of S covered by the leaves of the grammar tree also induce a parse on S, of size at most g or g_{rl}.

Example. Figure 4 (modified from Part I) shows a context-free grammar that generates only the string S = alabaralalabarda\$. The grammar has three rules, $A \rightarrow$ al, $B \rightarrow A$abar, and the initial rule $C \rightarrow BAB$da\$. The sum of the lengths of the right-hand sides of the rules is 13, the grammar size. On the right, a grammar tree is of size $13 + 1 = 14$, whose leaves induce a parsing of S of size $11 \leq 13$.

In turn, block trees can be seen as restrictions of parses where, at each level, the string is partitioned into blocks of a certain length and whole blocks S_v that appear elsewhere are replaced by a pointer inside a couple of consecutive blocks, $S_{v_1} \cdot S_{v_2}$, at the same level. The replaced blocks become the leaves of the block tree, and those leaves also induce a partition of S.

Example. The right part of Figure 3 (copied from Part I) shows a block tree for S = alabaralalabarda\$, which induces the partition S = a|l|a|b|a|r|ala|lab|ar|d|a|\$.

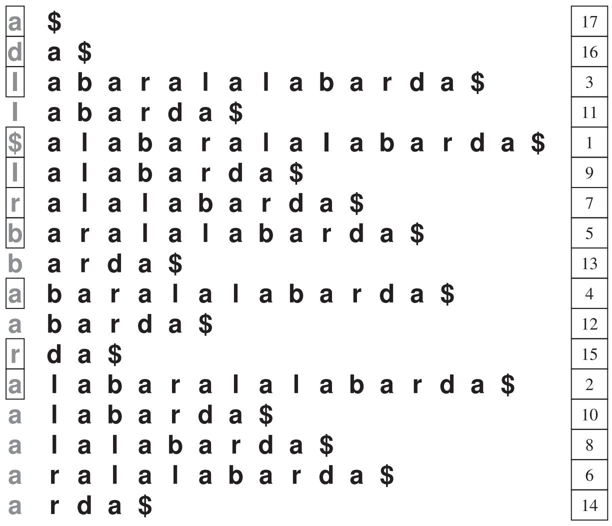
a $ 17
d a $ 16
l a b a r a l a l a b a r d a $ 3
l a b a r d a $ 11
$ a l a b a r a l a l a b a r d a $ 1
l a l a b a r d a $ 9
r a l a l a b a r d a $ 7
b a r a l a l a b a r d a $ 5
b a r d a $ 13
a b a r a l a l a b a r d a $ 4
a b a r d a $ 12
r d a $ 15
a l a b a r a l a l a b a r d a $ 2
a l a b a r d a $ 10
a l a l a b a r d a $ 8
a r a l a l a b a r d a $ 6
a r d a $ 14

Fig. 5. The list of suffixes of S = alabaralalabarda\$ in increasing lexicographic order. The sequence of preceding symbols (in gray) forms the BWT of S, S^{bwt} = adll\$lrbbaaraaaaa. The run heads are boxed. On the right we show the suffix array of S.

Bookmarking [Gagie et al. 2012] combines grammars with Lempel-Ziv parsing in order to speed up string extraction over (Lempel-Ziv) phrase prefixes and suffixes. The result extends to fingerprinting [Gagie et al. 2014]. The concepts are presented in Part I [Navarro 2020, Sec. 4.3] in simplified form and on attractors: Assume we have split S somehow into p phrases, and let there be an attractor on S, of size γ. To improve the extraction time from $O(\ell + \log n)$ to $O(\ell)$, we create a new string S' with the contexts of length $\log n$ centered around each attractor position. A particular type of run-length grammar built on S' then offers the described space and time complexities for extraction and fingerprint calculation.

Finally, the Burrows-Wheeler Transform (BWT) of $S[1\,..\,n]$ is another string $S^{bwt}[1\,..\,n]$ where the symbols of S are permuted: $S^{bwt}[i] = S[A[i] - 1]$, where A is the suffix array of S and we assume $S[0] = S[n]$. Measure r is the number of equal-symbol runs in S^{bwt}.

Example. The BWT of S = alabaralalabarda\$ is S^{bwt} = adll\$lrbbaaraaaaa, as shown in Figure 5 (modified from Part I). It has $r = 10$ runs.

3 PARSING-BASED INDEXING

In this section, we describe a common technique underlying a large class of indexes for repetitive string collections. The key idea, already devised by Kärkkäinen and Ukkonen [1996], builds on the parsing induced by a compression method, which divides $S[1\,..\,n]$ into p phrases, $S = S_1 \cdots S_p$. The parsing is used to classify the occurrences of any pattern $P[1\,..\,m]$ into two types:

- The *primary* occurrences are those that cross a phrase boundary.
- The *secondary* occurrences are those contained in a single phrase.

The main idea of parsing-based indexes is to first detect the primary occurrences with a structure using $O(p)$ space, and then obtain the secondary ones from those, also using $O(p)$ space. The key property that the parsing must hold is that it must allow finding the secondary occurrences from the primary ones within $O(p)$ space.

3.1 Geometric Structure to Track Primary Occurrences

Every primary occurrence of P in S can be uniquely described by $\langle i, j\rangle$, indicating:

(1) The leftmost phrase S_i it intersects.
(2) The position j of P that aligns at the end of that phrase.

A primary occurrence $\langle i, j\rangle$ then implies that

—$P[1\,..\,j]$ is a *suffix* of S_i, and
—$P[j+1\,..\,m]$ is a *prefix* of $S_{i+1}\cdots S_p$.

The idea is then to create two sets of strings:

—$\mathcal{X}$ is the set of all the *reversed* phrase contents, $X_i = S_i^{rev}$, for $1 \le i < p$, and
—$\mathcal{Y}$ is the set of all the suffixes $Y_i = S_{i+1}\cdots S_p$, for $1 \le i < p$.

If, for a given j, $P[1\,..\,j]^{rev}$ is a prefix of X_i (i.e., $P[1\,..\,j]$ is a suffix of S_i) and $P[j+1\,..\,m]$ is a prefix of Y_i, then $\langle i, j\rangle$ is a primary occurrence of P in S. To find them all, we lexicographically sort the strings in $\mathcal{X}$ and $\mathcal{Y}$, and set up a bidimensional grid of size $p \times p$. The grid has exactly p points, one per row and per column: If, for some i, the xth element of $\mathcal{X}$ in lexicographic order is X_i and the yth element of $\mathcal{Y}$ in lexicographic order is Y_i, then there is a point at (x, y) in the grid, which we label i.

The primary occurrences of P are then found with the following procedure:

—For each $1 \le j < m$

(1) Find the lexicographic range $[s_x, e_x]$ of $P[1\,..\,j]^{rev}$ in $\mathcal{X}$.
(2) Find the lexicographic range $[s_y, e_y]$ of $P[j+1\,..\,m]$ in $\mathcal{Y}$.
(3) Retrieve all the grid points $(x, y) \in [s_x, e_x] \times [s_y, e_y]$.
(4) For each retrieved point (x, y) labeled i, report the primary occurrence $\langle i, j\rangle$.

It is then sufficient to associate the end position $p(i) = |S_1\cdots S_i|$ with each phrase S_i, to know that the primary occurrence $\langle i, j\rangle$ must be reported at position $S[p(i) - j + 1\,..\,p(i) - j + m]$. Or we can simply store $p(i)$ instead of i in the grid.

Example. Figure 6 shows the grid built on a parsing of S = alabaralalabarda\$. Every reversed phrase appears on top, as an x-coordinate, and every suffix appears on the right, as a y-coordinate. Both sets of strings are lexicographically sorted and the points in the grid connect phrases (x) with their following suffix (y). Instead of points we draw the label, which is the number of the phrase in the x-coordinate.

A search for P = la finds its primary occurrences by searching $\mathcal{X}$ for $P[1]^{rev}$ = l, which yields the range $[x_s, x_e] = [7, 8]$, and searching $\mathcal{Y}$ for $P[2]$ = a, which gives the range $[y_s, y_e] = [2, 6]$. The search for the (grayed) zone $[7, 8] \times [2, 6]$ returns two points, with labels 2 and 7, meaning that $P[1]$ aligns at the end of those phrase numbers, precisely at positions $S[2]$ and $S[8]$.

Note that, by definition, there are no primary occurrences when $|P| = 1$. Still, it will be convenient to find all the occurrences of P that lie at the end of a phrase. To do this, we carry out the same steps above for $j = 1$, in the understanding that the lexicographic range on $\mathcal{Y}$ is $[s_y, e_y] = [1, p]$.

The challenges are then (1) how to find the intervals in $\mathcal{X}$ and $\mathcal{Y}$ and (2) how to find the points in the grid range.

3.1.1 *Finding the Intervals in $\mathcal{X}$ and $\mathcal{Y}$.*

A simple solution is to perform a binary search on the sets, which requires $O(\log p)$ comparisons of strings. The desired prefixes of X_i or Y_i to compare with, of length at most m, must be extracted from a compressed representation of S: we represent

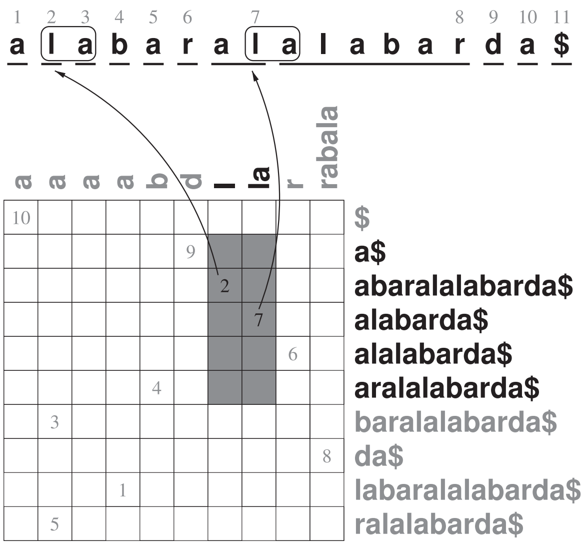

Fig. 6. A parse for S = alabaralalabarda\$ and the corresponding grid. We show the search process to find the primary occurrences of P = la.

X_i and Y_i just with the position where they appear in S. By using any of the techniques recalled in Section 2.5, we extract them in time $f_e = O(m/\log_\sigma n + \log n)$ or $f_e = O((1 + m/\log_\sigma n)\log(n/g))$. Since this is repeated for every $1 \le j < m$, all the intervals are found in time $O(f_e\ m \log p)$, which is in $O((m + \log n)m \log n)$ with the first tradeoff. This complexity can be reduced to $O(m^2 \log n)$ on grammars, by exploiting the fact that the phrases are defined as the leaves of the grammar tree, and therefore we always need to extract prefixes or suffixes of nonterminal expansions. The same time can be obtained with $O((p + \gamma)\log\log n)$ additional space by using bookmarking. In all cases, however, the complexity stays quadratic in m: We need to search for $m - 1$ prefixes/suffixes of P of length up to m.

The quadratic term can be removed by using a *batched search* for all the suffixes $P[j+1\,..\,m]$ together (or all the suffixes $P[1\,..\,j]^{rev}$). The technique is based on compact tries and Karp-Rabin fingerprints [Belazzougui et al. 2010; Bille et al. 2017a; Christiansen et al. 2019; Gagie et al. 2014, 2018]. The idea is to represent the sets $\mathcal{X}$ and $\mathcal{Y}$ with compact tries, storing fingerprints of the strings labeling selected paths, and then verifying the candidates to ensure they are actual matches. The fingerprints of all the suffixes sought are computed in time $O(m)$. The trie is actually a version called *z-fast trie* [Belazzougui et al. 2010, 2009], which allows searching for a string of length ℓ with $O(\log \ell)$ fingerprint comparisons. Once all the candidate results are found, a clever method does all the verifications with one single string extraction, exploiting the fact that we are looking for various suffixes of a single pattern. If a fingerprint $\kappa(S[i\,..\,j])$ is computed in time f_h, then the lexicographic ranges of any k suffixes of $P[1\,..\,m]$ can be found in time $O(m + k(f_h + \log m) + f_e)$. The structure uses $O(p)$ space, and is built in $O(n \log n)$ expected time to ensure that there are no fingerprint collisions in S.

As recalled in Section 2.5, we can compute Karp-Rabin fingerprints in time $O(\log n)$ or $O(\log(n/g))$ using grammars or block trees, respectively. To search for the $k = m - 1$ suffixes of P (or of its reverse) we then need time $O(m \log n)$.

The approach can be built on block trees that, as shown in Part I [Navarro 2020, Sec. 4.2], can be built on Lempel-Ziv ($O(z \log(n/z))$ space, and even $O(\delta \log(n/\delta))$), or built on attractors

($O(\gamma \log(n/\gamma))$ space). It can also be applied on grammars ($O(g)$ space), and even on run-length grammars, within space $O(g_{rl}) \subseteq O(\delta \log(n/\delta))$. Combined with bookmarking, the time can be reduced to $O(m \log m)$, because $f_h = O(\log m)$, yet we need $O(z \log\log n)$ or $O(g \log\log n)$ further space.

A recent twist [Christiansen and Ettienne 2018; Christiansen et al. 2019] is that a specific type of grammar, called *locally consistent grammar*,[4] can speed up the searches, because there are only $k = O(\log m)$ cuts of P that deserve consideration. In a locally consistent grammar, the subtrees of the parse tree expanding to two identical substrings $S[i\,..\,j] = S[i'\,..\,j']$ are identical except for $O(1)$ nodes in each level. Christiansen et al. [2019] show that locally consistent grammars of size $O(\gamma \log(n/\gamma))$ can be built without the need to compute γ (which would be NP-hard). Further, we can obtain $f_e = O(m)$ time with grammars, because, as explained, we extract only rule prefixes and suffixes when searching $\mathcal{X}$ or $\mathcal{Y}$. They also show how to compute Karp-Rabin fingerprints in time $O(\log^2 m)$ (without the extra space bookmarking uses) in their grammars. The time to find all the relevant intervals then decreases to $O(m + k(f_h + \log m) + f_e) \subseteq O(m)$.

3.1.2 Finding the Points in the Two-dimensional Range. This is a well-studied geometric problem [Chan et al. 2011]. We can represent p points on a $p \times p$ grid within $O(p)$ space, so that we can report all the t points within any given two-dimensional range in time $O((1+t)\log^\epsilon p)$, for any constant $\epsilon > 0$. By using slightly more space, $O(p \log\log p)$, the time drops to $O((1+t)\log\log p)$, and if we use $O(p \log^\epsilon p)$ space, the time drops to $O(\log\log p + t)$, thus enabling constant time per occurrence reported.

If we look for the $k = m - 1$ prefixes and suffixes of P with $O(p)$ space, then the total search time is $O(m \log^\epsilon p) \subseteq O(m \log n)$, plus $O(\log^\epsilon p)$ per primary occurrence. If we reduce k to $O(\log m)$, then the search time drops to $O(\log m \log^\epsilon p)$, plus $O(\log^\epsilon p)$ per primary occurrence. Christiansen et al. [2019] reduce this $O(\log m \log^\epsilon p)$ additive term to just $O(\log^\epsilon \gamma)$ by dealing with short patterns separately.

3.2 Tracking Secondary Occurrences

The parsing method must allow us infer all the secondary occurrences from the primary ones. The precise method for propagating primary to secondary occurrences depends on the underlying technique.

3.2.1 Lempel-Ziv Parsing and Macro Schemes. The idea of primary and secondary occurrences was first devised for the Lempel-Ziv parsing [Farach and Thorup 1995; Kärkkäinen and Ukkonen 1996], of size $p = z$. Note that the leftmost occurrence of any pattern P cannot be secondary, because then it would be inside a phrase that would occur earlier in S. Secondary occurrences can then be obtained by finding all the phrase sources that cover each primary occurrence. Each such source produces a secondary occurrence at the phrase that copies the source. In turn, one must find all the phrase sources that cover these secondary occurrences to find further secondary occurrences, and so on. All the occurrences are found in that way.

A simple technique [Kreft and Navarro 2013] is to maintain all the sources $[b_k, e_k]$ of the z phrases of S in arrays $B[1\,..\,z]$ (holding all b_ks) and $E[1\,..\,z]$ (holding all e_ks), both sorted by increasing endpoint e_k. Given an occurrence $S[i\,..\,j]$, a successor search on E finds (in $O(\log\log_w n)$ time [Belazzougui and Navarro 2015]) the smallest endpoint $e_k \geq j$, and therefore all the sources in $E[k\,..\,z]$ end at or after $S[j]$; those in $E[1\,..\,k-1]$ cannot cover $S[i\,..\,j]$, because they end before

[4]Built via various rounds of the better-known locally consistent parsings [Cole and Vishkin 1986; Sahinalp and Vishkin 1995; Mehlhorn et al. 1997; Batu et al. 2006].

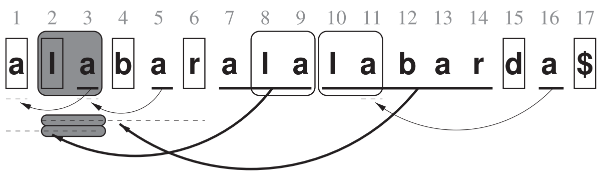

Fig. 7. Finding the secondary occurrences of $P =$ la on the Lempel-Ziv parse of Figure 3. The occurrences are marked with rounded boxes. The primary occurrence, which is grayed, is projected from sources to targets to find the secondary ones.

j. We then want to retrieve the values $B[l] \le i$ with $k \le l \le z$, that is, the sources that in addition start no later than i.

A technique to retrieve each source covering $S[i\,..\,j]$ in constant time is as follows. We build a Range Minimum Query (RMQ) data structure [Bender et al. 2005; Fischer and Heun 2011] on B, which uses $O(z)$ bits and returns, in constant time, the position of the minimum value in any range $B[k\,..\,k']$. We first query for $B[k\,..\,z]$. Let the minimum be at $B[l]$. If $B[l] > i$, then this source does not cover $S[i\,..\,j]$, and no other source does, because this is the one starting the earliest. We can therefore stop. If, instead, $B[l] \le i$, then we have a secondary occurrence in the target of $[b_l, e_l]$. We must report that occurrence and recursively look for other sources covering it. In addition, we must recursively look for sources that start early enough in $B[k\,..\,l-1]$ and $B[l+1\,..\,z]$. Since we get an occurrence each time we find a suitable value of B in the current range, and stop as soon as there are no further values, it is easy to see that we obtain each secondary occurrence in constant time.

Example. Figure 7 shows the search for $P =$ la on the Lempel-Ziv parse of Figure 3. There is only one primary occurrence at $S[2\,..\,3]$. The sources $[b_k, e_k]$ are, in increasing order of e_k, $[1,1], [1,3], [3,3], [2,6], [11,11]$, so we have the arrays $B = \langle 1,1,3,2,11\rangle$ and $E = \langle 1,3,3,6,11\rangle$. A successor search for 3 in E shows that $E[2\,..\,5]$ contains all the sources that finish at or after position 3. We now use RMQs on B to find those that start at or before position 2. The first candidate is $RMQ(2,5) = 2$, which identifies the valid source $[B[2], E[2]] = [1,3]$ covering our primary occurrence. The target of this source is $S[7\,..\,9]$, which contains a secondary occurrence at $S[8\,..\,9]$ (the offset of the occurrence within the target is the same as within the source). There is no other source covering $S[8\,..\,9]$, so that secondary occurrence does not propagate further. We continue with our RMQs, now on the remaining interval $B[3\,..\,5]$. The query $RMQ(3,5) = 4$ yields the source $[B[4], E[4]] = [2,6]$, which also covers the primary occurrence. Its target, $S[10\,..\,14]$, then contains a secondary occurrence at the same offset as in the source, in $S[10\,..\,11]$. Again, no source covers this secondary occurrence. Continuing, we must check the intervals $B[3\,..\,3]$ and $B[5\,..\,5]$. But since both are larger than 2, they do not cover the primary occurrence and we are done.

Thus, if we use a Lempel-Ziv parse, then we require $O(z)$ additional space to track the secondary occurrences. With occ occurrences in total, the time is $O(occ \log\log_w n)$, dominated by the successor searches. Note that the scheme works well also under bidirectional macro schemes, because it does not need that the targets be to the right of the sources. Thus, the space can be reduced to $O(b)$.

3.3 Block Trees

The sequence of leaves in any of the block tree variants we described partitions S into a sequence of p phrases (where p can be as small as $O(\delta \log(n/\delta))$, as recalled in Section 2.5). Each such phrase is either explicit (if it is at the last level) or it has another occurrence inside an internal node of the same level.

This parsing also permits applying the scheme of primary and secondary occurrences, if we use leaves of length 1 [Navarro and Prezza 2019]. It is not hard to see that every secondary occurrence $S[i\,..\,j]$, with $i < j$, has a copy that crosses a phrase boundary: $S[i\,..\,j]$ is inside a block S_v that is a leaf, thus it points to another occurrence of S_v inside $S_{v_1} \cdot S_{v_2}$. If the copy $S[i'\,..\,j']$ spans both S_{v_1} and S_{v_2}, then it is a primary occurrence. Otherwise it falls inside S_{v_1} or S_{v_2}, and then it is also inside a block of the next block tree level. At this next level, $S[i'\,..\,j']$ may fall inside a block $S_{v'}$ that is a leaf, and thus it points toward another occurrence of $S_{v'}$. In this case, $S[i'\,..\,j']$ is also a secondary occurrence and we will discover $S[i\,..\,j]$ from it; in turn $S[i'\,..\,j']$ will be discovered from its pointed occurrence $S[i''\,..\,j'']$, and so on. Instead, $S[i'\,..\,j']$ may fall inside an internal block $S_{v'} = S_{v_1'} \cdot S_{v_2'}$. If $S[i'\,..\,j']$ spans both $S_{v_1'}$ and $S_{v_2'}$, then it is a primary occurrence, otherwise it appears inside a block of the next level, and so on. We continue this process until we find an occurrence of $S[i\,..\,j]$ that crosses a block boundary, and thus it is primary. Our original occurrence $S[i\,..\,j]$ is then found from the primary occurrence, through a chain of zero or more intermediate secondary occurrences.

Example. Consider the parsing $S =$ a|l|a|b|a|r|a|a|l|ab|ar|d|a|\$ induced by the block tree of Figure 3, where the phrases of length 1 are the leaves of the last level. Then $P =$ al has two primary occurrences, at $S[1\,..\,2]$ and $S[9\,..\,10]$. The sources of blocks are $[1, 3]$, $[2, 4]$, and $[5, 6]$. Therefore the source $[1, 3]$ covers the first primary occurrence, which then has a secondary occurrence at the target, in $S[7, 8]$.

The process and data structures are then almost identical to those for the Lempel-Ziv parsing. We collect the sources of all the leaves at all the levels in arrays B and E, as for Lempel-Ziv, and use them to find, directly or transitively, all the secondary occurrences. In some variants, such as block trees built on attractors, the sources can be before or after the target in S, as in bidirectional macro schemes, and the scheme works equally well.

3.4 Grammars

In the case of context-free grammars [Claude and Navarro 2012] (and also run-length grammars), the partition of S induced by the leaves of the grammar tree induces a suitable parsing: a secondary occurrence $S[i\,..\,j]$ inside a leaf labeled by nonterminal A has another occurrence $S[i'\,..\,j']$ below the occurrence of A as an internal node of the grammar tree. If $S[i'\,..\,j']$ is inside a leaf labeled B (with $|B| < |A|$), then there is another occurrence $S[i''\,..\,j'']$ below the internal node labeled B, and so on. Eventually, we find a copy crossing a phrase boundary, and this is our primary occurrence.

The hierarchical structure of the grammar tree enables a simplified process to find the occurrences. Let $exp(X)$ be the string nonterminal X expands to. For each internal node v representing the rule $A \rightarrow X_1 \cdots X_k$, and for each $1 \leq i < k$, we insert $exp(X_i)^{rev}$ into $\mathcal{X}$ and $exp(X_{i+1}) \cdots exp(X_k)$ into $\mathcal{Y}$, associating the corresponding grid point to node v with offset $|exp(X_1) \cdots exp(X_i)|$. This ensures that every cutting point between consecutive grammar tree leaves X and Y is included and associated with the lowest common ancestor node $A = lca(X, Y)$ that covers both leaves. By associating the part of $exp(A)$ that follows $exp(X)$, instead of the full suffix, one ensures at construction time that A is the lowest nonterminal that covers the primary occurrence covering X and Y.

Once we establish that P occurs inside $exp(A)$ at position j, we must track j upwards in the grammar tree, adjusting it at each step, until locating the occurrence in the start symbol, which gives the position where P occurs in S. To support this upward traversal we store, in each grammar tree node A with parent C, the offset of $exp(A)$ inside $exp(C)$. This is added to j when we climb from A to C.

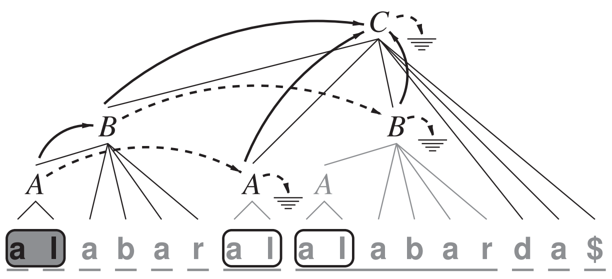

Fig. 8. Finding the secondary occurrences of P = al on the grammar-induced parse of Figure 4 (the removed edges of the parse tree are grayed). The occurrences are marked with rounded boxes; the primary one is grayed. The bold solid arrows translate each occurrence toward the root, and the bold dashed arrows toward the next occurrence of the same nonterminal.

In addition, every other occurrence of A in the grammar tree contains a secondary occurrence of P, with the same offset j. Note that all those other occurrences of A are leaves in the grammar tree. Each node labeled A has then a pointer to the next grammar tree node labeled A, forming a linked list that must be traversed to find all the secondary occurrences (in any desired order; the only restriction is that the list must start at the only internal node labeled A). Further, if C is the parent of A, then any other occurrence of C in the grammar tree (which is necessarily a leaf as well) also contains a new secondary occurrence of P.

The process then starts at each primary occurrence A and recursively moves to (1) the parent of A (adjusting the offset j), and (2) the next node labeled A. The recursive calls end when we reach the grammar tree root in step (1), which occurs once per distinct secondary occurrence of P, and when there is no next node to consider in step (2).

Example. Figure 8 shows how the only primary occurrence of P = al in S = alabaralalabarda\$, using the parse of Figure 4 (and Figure 6), are propagated using the grammar tree. The primary occurrence, $S[1..2]$ spans the first two leaves, and the pointer of the grid sends us to the internal node labeled A, which is the lowest common ancestor of those two leaves, with offset 1 (indeed, $exp(A)$ = al). To find its position in S, we go up to B, the parent of A, where the offset is still 1, because the offset of A within B is 0 ($exp(B)$ = alabar). Finally, we reach C, the parent of B and the tree root, where the offset is still 1 and thus we report the primary occurrence $S[1..2]$.

The secondary occurrences are found by recursively following the dashed arrows toward the other occurrences of the intermediate nonterminals. From the internal node A we reach the only other occurrence of A in the grammar tree (which is a leaf; remind that there is only one internal node per label). This leaf has offset 6 within its parent C, so the offset within C is $1 + 6 = 7$. We then move to C and report a secondary occurrence at $S[7..8]$. The list of the As ends there. Similarly, when we arrive at the internal node B, we follow the dashed arrow toward the only other occurrence of B in the grammar tree. This has offset 8 within its parent C, so when we move up to C we report the secondary occurrence $S[9..10]$.

Note that the bold arrows, solid and dashed, form a binary tree rooted at the primary occurrence. The leaves that are left children are list ends and those that are right children are secondary occurrences. Thus the total number of nodes (and the total cost of the tracking) is proportional to the number of occurrences reported.

Claude and Navarro [2012] show that the cost of the traversal indeed amortizes to constant time per secondary occurrence if we ensure that every nonterminal A occurs at least twice in the grammar tree (as in our example). Nonterminals appearing only once can be easily removed from the grammar. If we cannot modify the grammar, then we can instead make each node A point not to its parent, but to its lowest ancestor that appears at least twice in the grammar tree (or to the root, if no such ancestor exists) [Christiansen et al. 2019]. This ensures that we report the occ_s secondary occurrences in time $O(occ_p + occ_s)$.

Christiansen et al. [2019] show how the process is adapted to handle the special nodes induced by the rules $A \rightarrow X^k$ of run-length grammars.

3.5 Resulting Tradeoffs

By considering the cost to find the primary and secondary occurrences, and sticking to the best possible space in each case, it turns out that we can find all the occ occurrences of $P[1..m]$ in $S[1..n]$ either:

— in time $O(m \log n + occ \log^{\epsilon} n)$, within space $O(g_{rl}) \subseteq O(\delta \log(n/\delta))$.
— in time $O(m + (occ + 1) \log^{\epsilon} n)$, within space $O(\gamma \log(n/\gamma))$.

The first result is obtained by using a run-length grammar to define the parse of S (thus the grid is of size $g_{rl} \times g_{rl}$), to access nonterminal prefixes and suffixes and compute fingerprints, and to track the secondary occurrences. Note that finding the smallest grammar is NP-hard, but there are ways to build run-length grammars of size $O(\delta \log(n/\delta))$, as mentioned in Part I of this survey. The second result uses the improvement based on locally consistent parsing (end of Section 3.1.1); recall that we do not need to compute γ (which is NP-hard too) to obtain it.

3.5.1 Using More Space. We can combine the slightly larger grid representations (Section 3.1.2) with bookmarking in order to obtain improved times for the first result. We can use a bidirectional macro scheme to define the phrases, so that the grid is of size $b \times b$, and use the geometric data structure of size $O(b \log\log b)$ that reports the occ points in time $O((1 + occ) \log\log b)$. We then use a run-length grammar to provide direct access to S, and enrich it with bookmarking (Section 2.5) to provide substring extraction and Karp-Rabin hashes (to find the ranges in $\mathcal{X}$ and $\mathcal{Y}$) in time $f_e = O(m)$ and $f_h = O(\log m)$, respectively, at phrase boundaries. This adds $O((b + \gamma) \log\log n) = O(b \log\log n)$ space. The time to find the $m - 1$ ranges in $\mathcal{X}$ and $\mathcal{Y}$ is then $O(m + m(f_h + \log m) + f_e) = O(m \log m)$. The $m - 1$ geometric searches take time $O(m \log\log b + occ \log\log b)$, and the secondary occurrences are reported in time $O(occ \log\log_w n)$ (Section 3.2.1). Gagie et al. [2014] get rid of the $O(m \log\log b)$ term by dealing separately with short patterns (see their Section 4.2; it adapts to our combination of structures without any change).

Note again that it is NP-hard to find the smallest bidirectional macro scheme, but we can build suboptimal ones from the Lempel-Ziv parse, the lex-parse, or the BWT runs, for example (see Part I [Navarro 2020, Sec. 3]). Also, there are heuristics to build bidirectional macro schemes smaller than z [Nishimoto and Tabei 2019; Russo et al. 2020].

The larger grid representation, of size $O(p \log^{\epsilon} p)$ for p points, reports primary occurrences in constant time, but to maintain that constant time for secondary occurrences we need that the parse comes from a (run-length) grammar (Section 3.4). We must therefore use a grid of $g_{rl} \times g_{rl}$. The grammar already extracts phrase (i.e., nonterminal) prefixes and suffixes in constant time, yet bookmarking is still useful to compute fingerprints in $O(\log m)$ time. We can then search:

— in time $O(m \log m + occ \log\log n)$, within space $O(g_{rl} + b \log\log n)$.
— in time $O(m \log m + occ)$, within space $O(g_{rl} \log^{\epsilon} n)$.

Finally, using larger grids directly on the result that uses $O(\gamma \log(n/\gamma))$ space yields the first optimal-time index [Christiansen et al. 2019]. We can search:

— in time $O((m + occ) \log\log n)$, within space $O(\gamma \log(n/\gamma) \log\log n)$.
— in time $O(m + occ)$, within space $O(\gamma \log(n/\gamma) \log^{\epsilon} n)$.

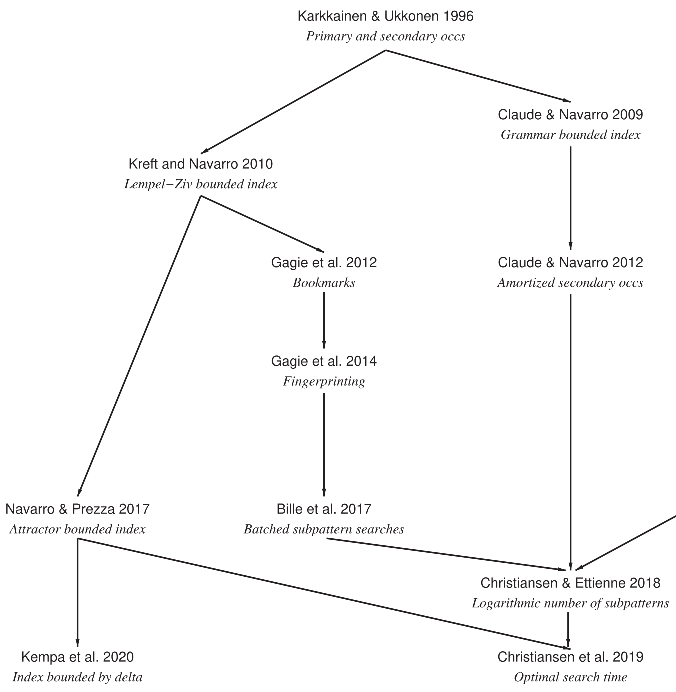

Fig. 9. Diagram of the influences and main ideas about parsing-based indexing.

3.5.2 History. The generic technique we have described encompasses a large number of indexes found in the literature. As said, Kärkkäinen and Ukkonen [1996] pioneered the idea of primary and secondary occurrences based on Lempel-Ziv for indexing. Their index is not properly a compressed index, because it stores S in plain form, and uses $O(z)$ additional space to store the grid and a mechanism of stratified lists of source areas to find the secondary occurrences.

Figure 9 shows a diagram with the main ideas that appeared along time and the influences between contributions. The Appendix gives a detailed account. The best results to date are those we have given explicitly, plus some intermediate tradeoffs given by Christiansen et al. [2019] (see their Table I).

4 SUFFIX-BASED INDEXING

Suffix arrays and suffix trees (Section 2.3) are data structures designed to support indexed searches. They are of size $O(n)$, but large in practice. We next describe how their search algorithms translate into structures of size $O(r)$ or $O(e)$, which are related to the regularities induced by repetitiveness on suffix arrays and trees.

4.1 Based on the BWT

The suffix array search based on the BWT dates back to Ferragina and Manzini [2000, 2005], who showed that, with appropriate data structures, S^{bwt} is sufficient to simulate a suffix array search and find the range $A[sp\,..\,ep]$ of the suffixes that start with a search pattern P. Their method, called *backward search*, consecutively finds the interval $A[sp_i\,..\,ep_i]$ of the suffixes starting with $P[i\,..\,m]$, by starting with $[sp_{m+1}\,..\,ep_{m+1}] = [1\,..\,n]$ and then computing, for $i = m$ to $i = 1$,

$$\begin{aligned} sp_i &= C[P[i]] + rank_{P[i]}(S^{bwt}, sp_{i+1} - 1) + 1, \\ ep_i &= C[P[i]] + rank_{P[i]}(S^{bwt}, ep_{i+1}), \end{aligned}$$

where $C[c]$ is the number of occurrences in S of symbols lexicographically smaller than c, and $rank_c(S^{bwt}, j)$ is the number of occurrences of c in $S^{bwt}[1..j]$.[5] Further, if $A[j] = i$, that is, the lexicographically jth smallest suffix of S is $S[i..]$, then $A[j'] = i - 1$ for $c = S^{bwt}[j]$ and

$$j' = LF(j) = C[c] + rank_c(S^{bwt}, j),$$

which is called an *LF-step* from j. By performing LF-steps on the BWT of S, we virtually traverse S backwards. Operation *rank* on S^{bwt} can be implemented in $n \log \sigma + o(n \log \sigma)$ bits (and even in statistically compressed space) and in the optimal time $O(\log \log_w \sigma)$ [Belazzougui and Navarro 2015], which yields $O(m \log \log_w \sigma)$ time for backward searching of P.

To understand the rationale of the backward search formula, let us start with the backward step. Recall from Section 2.5 that $c = S^{bwt}[j]$ is the symbol preceding the suffix $A[j]$, $c = S^{bwt}[j] = S[A[j] - 1]$. The function $j' = LF(j)$ computes where is in A the suffix that points to $A[j] - 1$. First, all the $C[c]$ suffixes that start with symbols less than c precede $A[j']$. Second, the suffixes $S[A[j] - 1..]$ are stably sorted by $\langle S[A[j] - 1], A[j]\rangle = \langle S^{bwt}[j], A[j]\rangle$, that is, by their first symbol and breaking ties with the rank of the suffix that follows. Therefore, $LF(j)$ adds the number $C[c]$ of suffixes starting with symbols less than c and the number $rank_c(S^{bwt}, j)$ of suffixes that start with c up to the one we want to translate, $A[j]$.

Example. Consider S = alabaralalabarda\$, with S^{bwt} = adll\$lrbbaaraaaaa, in Figure 5. From $A[14] = 10$ and $S^{bwt}[14]$ = a (which correspond to the suffix $S[10..]$ = labarda\$), we compute $LF(14) = C[\mathrm{a}] + rank_{\mathrm{a}}(S^{bwt}, 14) = 1 + 5 = 6$. Indeed $A[6] = 9 = A[14] - 1$, corresponding to the suffix alabarda\$.

Let us now consider the backward search steps. Note that we know that the suffixes in $A[sp_{i+1}..ep_{i+1}]$ start with $P[i+1..m]$. The range $A[sp_i..ep_i]$ lists the suffixes that start with $P[i..m]$, that is, they start with $P[i]$ and then continue with a suffix in $A[sp_{i+1}..ep_{i+1}]$. We then want to capture all the suffixes in $A[sp_{i+1}..ep_{i+1}]$ that are preceded by $c = P[i]$ and map them to their corresponding position in A. Since they will be mapped to a range, the backward search formula is a way to perform all those LF-steps in one shot.

Example. Consider again S = alabaralalabarda\$, with S^{bwt} = adll\$lrbbaaraaaaa, in Figure 5. To search for P = la, we start with the range $A[sp_3..ep_3] = [1..17]$. The first backward step, for $P[2]$ = a, gives $sp_2 = C[\mathrm{a}] + rank_{\mathrm{a}}(S^{bwt}, 0) + 1 = 1 + 1 = 2$ and $ep_2 = C[\mathrm{a}] + rank_{\mathrm{a}}(S^{bwt}, 17) = 1 + 8 = 9$. Indeed, $A[2..9]$ is the range of all the suffixes starting with $P[2..2]$ = a. The second and final backward step, for $P[1]$ = l, gives $sp_1 = C[\mathrm{l}] + rank_{\mathrm{l}}(S^{bwt}, 1) + 1 = 12 + 1 = 13$ and $ep_2 = C[\mathrm{l}] + rank_{\mathrm{l}}(S^{bwt}, 9) = 12 + 3 = 15$. Indeed, $A[13..15]$ is the range of the suffixes starting with P = la, and thus the occurrences of P are at $A[13] = 2$, $A[14] = 10$, and $A[15] = 8$. Note that, if we knew that the suffixes in $A[2..9]$ preceded by l were at positions 3, 4, and 6, and we had computed $LF(3)$, $LF(4)$, and $LF(6)$, we would also have obtained the interval $A[13..15]$.

Ferragina and Manzini [2005] and Ferragina et al. [2007] show how to represent S^{bwt} within $nH_k(S) + o(n \log \sigma)$ bits of space, that is, asymptotically within the kth order empirical entropy of S, while supporting pattern searches in time $O(m \log \sigma + occ \log^{1+\epsilon} n)$ for any constant $\epsilon > 0$. These concepts are well covered in other surveys [Navarro and Mäkinen 2007], so we will not develop them further here; we will jump directly to how to implement them when S is highly repetitive.

[5]If $sp_i > ep_i$, then P does not occur in S and we must not continue the backward search.

4.1.1 Finding the Interval. Mäkinen and Navarro [2005] showed how to compute *rank* on S^{bwt} when it is represented in run-length form (i.e., as a sequence of r runs). We present the results in a more recent setup [Gagie et al. 2020; Mäkinen et al. 2010] that ensures $O(r)$ space. The positions that start runs in S^{bwt} are stored in a predecessor data structure that also tells the number of the corresponding runs. A string $S'[1..r]$ stores the symbol corresponding to each run of S^{bwt}, in the same order of S^{bwt}. The run lengths are also stored in another array, $R[1..r]$, but they are stably sorted lexicographically by the associated symbol. More precisely, if $R[t]$ is associated with symbol c, it stores the cumulative length of the runs associated with c in $R[1..t]$. Finally, $C'[c]$ tells the number of runs of symbols d for all $d < c$. Then, to compute $rank_c(S^{bwt}, j)$, we:

(1) Find the predecessor j' of j, so that we know that j belongs to the kth run in S^{bwt}, which starts at position $j' \leq j$.
(2) Determine that the symbol of the current run is $c' = S'[k]$.
(3) Compute $p = rank_c(S', k-1)$ to determine that there are p runs of c *before* the current run.
(4) The position of the run $k-1$ in R is $C'[c] + p$: R lists the $C'[c]$ runs of symbols less than c, and then the p runs of c preceding our run k (because R is stably sorted, upon ties it retains the order of the runs in S^{bwt}).
(5) We then know that $rank_c(S^{bwt}, j'-1) = R[C'[c] + p]$.
(6) This is the final answer if $c \neq c'$. If $c = c'$, then j is within a run of cs and thus we must add $j - j' + 1$ to the answer.

Example. The BWT of $S =$ alabaralalabarda\$ has $r(S) = 10$ runs (recall Figure 5), $S^{bwt} =$ a|d|ll|\$|l|r|bb|aa|r|aaaaa. The predecessor data structure then contains the run start positions, $\langle 1, 2, 3, 5, 6, 7, 8, 10, 12, 13 \rangle$. The string of distinct run symbols is $S'[1..10] =$ adl\$lrbara. Stably sorting the runs $\langle 1..10 \rangle$ by symbol we obtain $\langle 4, 1, 8, 10, 7, 2, 3, 5, 6, 9 \rangle$ (e.g., we first list the fourth run, because its symbol is the smallest, \$, then we list the 3 positions of 'a' in S', 1,8,10, and so on), and therefore $R = \langle 1, 1, 3, 8, 2, 1, 2, 3, 1, 2 \rangle$ (e.g., $R[2..4] = \langle 1, 3, 8 \rangle$, because the runs of 'a' are of lengths 1, 2, and 5, which cumulate to 1, 3, and 8). Finally, $C'[\$] = 0, C'[\mathrm{a}] = 1, C'[\mathrm{b}] = 4, C'[\mathrm{d}] = 5$, $C'[\mathrm{l}] = 6$, and $C'[\mathrm{r}] = 8$ precede the positions where the runs of each symbol start in R.

To compute $rank_a(S^{bwt}, 15)$ we find the predecessor $j' = 13$ of $j = 15$ and from the same structure learn that it is the run number $k = 10$. We then know that it is a run of 'a's, because $S'[10] = \mathrm{a}$. We then find out that there are $p = 2$ runs of 'a's preceding it, because $rank_\mathrm{a}(S', 9) = 2$. Further, there are $C'[\mathrm{a}] = 1$ runs of symbols smaller than 'a' in S^{bwt}. This means that the runs of 'a's start in R after position $C'[\mathrm{a}] = 1$, and that the run $k - 1 = 9$ is, precisely, at $C'[\mathrm{a}] + p = 3$. With $R[3] = 3$ we learn that there are 3 'a's in $S^{bwt}[1..12]$. Finally, since we are counting 'a's and j is in a run of 'a's, we must add the $j - j' + 1 = 15 - 13 + 1 = 3$ as in our current run. The final answer is then $rank_a(S^{bwt}, 15) = 3 + 3 = 6$.

The cost of the above procedure is dominated by the time to find the predecessor of j, and the time to compute *rank* on $S'[1..r]$. Using only structures of size $O(r)$ [Gagie et al. 2020], the predecessor can be computed in time $O(\log\log_w(n/r))$ if there are r elements in a universe $[1..n]$ [Belazzougui and Navarro 2015], and *rank* on S' can be computed in time $O(\log\log_w \sigma)$ as explained. This yields a total time of $O(m \log\log_w(\sigma + n/r))$ to determine the range $A[sp..ep]$ using backward search for P, in space $O(r)$ [Gagie et al. 2020]. Recently, Nishimoto and Tabei [2020] showed that, by adding some artificial cuts to the runs, it is possible to avoid the $O(\log\log_w(n/r))$-time predecessor searches by always maintaining the run to which the position j belongs in constant time. This reduces the time to $O(m \log\log_w \sigma)$, still within $O(r)$ space.

4.1.2 Locating the Occurrences. Once we have determined the interval $[sp\,..\,ep]$ where the answers lie in the suffix array, we must output the positions $A[sp], \ldots, A[ep]$ to complete the query. We do not have, however, the suffix array in explicit form. The classical procedure [Ferragina and Manzini 2005; Mäkinen et al. 2010] is to choose a sampling step t and sample the suffix array entries that point to all the positions of the form $S[i \cdot t + 1]$, for all $0 \leq i < n/t$. Then, if $A[j]$ is not sampled, we compute $LF(j)$ and see if $A[LF(j)]$ is not sampled, and so on. Since we sample S regularly, some $A[LF^s(j)]$ must be sampled for $0 \leq s < t$. Since function LF implicitly moves us one position backward in S, it holds that $A[j] = A[LF^s(j)] + s$. Therefore, within $O(n/t)$ extra space, we can report each of the occurrence positions in time $O(t \log\log_w(n/r))$ (the LF-steps do not require $O(\log\log_w \sigma)$ time for the *rank* on S', because its queries are of the form $rank_{S'[i]}(S', i)$, for which we can just store the answers to the r distinct queries).

Though this procedure is reasonable for statistical compression, the extra space $O(n/t)$ is usually much larger than r, unless we accept a significantly high time, $O((n/r) \log\log_w(n/r))$, to report each occurrence. This had been a challenge for BWT-based indexes on repetitive collections until very recently [Gagie et al. 2020], where a way to efficiently locate the occurrences within $O(r)$ space was devised.

Gagie et al. [2020] solve the problem of reporting $A[sp\,..\,ep]$ in two steps, in their so-called r-index (we present the simplified version of Bannai et al. [2020]). First, they show that the backward search can be modified so that, at the end, we know the value of $A[ep]$ (in turn, this simplifies a previous solution [Policriti and Prezza 2018]). Second, they show how to find $A[j-1]$ given the value of $A[j]$.

The first part is not difficult. When we start with $[sp\,..\,ep] = [1\,..\,n]$, we just need the value of $A[n]$ stored. Now, assume we know $[sp_{i+1}\,..\,ep_{i+1}]$ and $A[ep_{i+1}]$, and compute $[sp_i\,..\,ep_i]$ using the backward search formula. If the last suffix, $A[ep_{i+1}]$, is preceded by $P[i]$ (i.e., if $S^{bwt}[ep_{i+1}] = P[i]$), then the last suffix of $A[sp_i\,..\,ep_i]$ will be precisely $A[ep_i] = A[LF(ep_{i+1})] = A[ep_{i+1}] - 1$, and thus we know it. Otherwise, we must find the last occurrence of $P[i]$ in $S^{bwt}[sp_{i+1}\,..\,ep_{i+1}]$, because this is the one that will be mapped to $A[ep_i]$. This can be done by storing an array $L[1\,..\,r]$ parallel to R, so that $L[t]$ is the value of A for the last entry of the run $R[t]$ refers to. Once we determine p using the backward search step described above, we have that $A[ep_i] = L[C'[c] + p] - 1$.

Example. For $S = \mathsf{alabaralalabarda\$}$ and $S^{bwt} = \mathsf{adll\$lrbbaaraaaaa}$, we have $L = \langle 1, 17, 12, 14, 13, 16, 11, 9, 7, 15 \rangle$. For example, $L[3]$ refers to the end of the second run of 'a's, as seen in the previous example for $R[3]$. This ends at $S^{bwt}[11]$, and $A[11] = 12 = L[3]$. In the backward search for $P = \mathsf{la}$, we start with $[sp_3\,..\,ep_3] = [1\,..\,17]$, and know that $A[17] = 14$. The backward search computation then yields $[sp_2\,..\,ep_2] = [2\,..\,9]$. Since $S^{bwt}[17] = \mathsf{a} = P[2]$, we deduce that $A[9] = 14 - 1 = 13$. A new backward step yields $[sp_1\,..\,ep_1] = [13\,..\,15]$. Since $S^{bwt}[9] = \mathsf{b} \neq P[1]$, we must consult L. The desired position of L is obtained with the same process to find $rank_{\mathsf{l}}(S^{bwt}, 9)$: $k = 7$, $p = rank_{\mathsf{l}}(S', 6) = 2$, $t = C'[\mathsf{l}] + p = 6 + 2 = 8$, from where we obtain that $A[15] = L[8] - 1 = 9 - 1 = 8$.

For the second part, finding $A[j-1]$ from $A[j]$, let us define $d = A[j-1] - A[j]$, and assume both positions are in the same run, that is, $S^{bwt}[j-1] = S^{bwt}[j] = c$ for some c. By the LF-step formula, it is not hard to see that $LF(j-1) = LF(j) - 1$, and thus $A[LF(j-1)] - A[LF(j)] = (A[j-1] - 1) - (A[j] - 1) = d = A[LF(j) - 1] - A[LF(j)]$.[6] This means that, as we perform LF-steps from j and $j - 1$ and both stay in the same run, their difference d stays the same. After performing s LF-steps, for some s, $S^{bwt}[j'] = S^{bwt}[LF^s(j)]$ will be a run head and $S^{bwt}[j'-1] = S^{bwt}[LF^s(j-1)]$ will

[6]With the possible exception of $A[j-1]$ or $A[j]$ being 1, but in this case the BWT symbol is \$, and thus they cannot be in the same run.

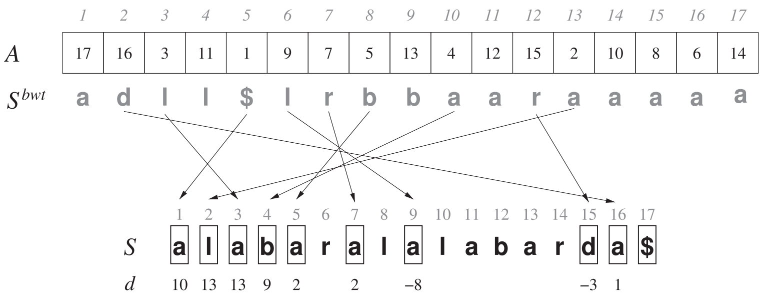

Fig. 10. The sampling on S = alabaralalabarda$ induced by the runs of its BWT with the purpose of locating occurrences. We omit the sample of $A[1] = 17$, because no pattern can occur at $S[17]$.

belong to another run. If we store $d = A[j' - 1] - A[j']$ for the run head j', then we can compute $A[j - 1] = A[j] + d$.

The key to find the proper run head is to note that $A[j'] = A[j] - s$ is the only position of a run head mapped to S in $A[j] - s, \ldots, A[j]$. We then store another predecessor data structure with the positions $A[j']$ in S that correspond to run heads in S^{bwt}, $S^{bwt}[j' - 1] \neq S^{bwt}[j']$. To the position $t = A[j']$ we associate $d(t) = A[j' - 1] - A[j']$. To compute $A[j - 1]$ from $A[j]$, we simply find the predecessor $t = A[j']$ of $A[j]$ and then know that $A[j - 1] = A[j] + d(t)$.[7]

Example. Figure 10 shows the run heads projected to S, and the associated values d. Once we find the interval $A[13\,..\,15]$ for P = la in the previous example, and since we know that $A[15] = 8$, we can compute $A[14]$ as follows. The boxed predecessor of $S[8]$ is $S[7]$. Since $A[7] = 7$, we stored $d(7) = A[6] - A[7] = 2$ associated with $S[7]$, and thus we know that $A[14] = A[15] + d(7) = 10$. Now, the boxed predecessor of $S[10]$ is $S[9]$. Since $A[6] = 9$, we stored $d(9) = A[5] - A[6] = -8$ associated with $S[9]$, and thus we know that $A[13] = A[14] + d(9) = 2$.

Each new position is then found with a predecessor search, yielding total search time $O(m \log\log_w(\sigma + n/r) + occ \log\log_w(n/r))$, and $O(r)$ space [Gagie et al. 2020]. Nishimoto and Tabei [2020] manage to compute these predecessors in constant time as well, thus reducing the search time to of $O(m \log\log_w \sigma + occ)$ in $O(r)$ space.

This index was implemented and shown to be 1–2 orders of magnitude faster than parsing-based indexes, though up to twice as large [Gagie et al. 2020]. When the collections are very repetitive, its size is still small enough, but the index (as well as measure r) degrades faster than z or g when repetitiveness starts to decrease.

Note that the index does not provide direct access to S within $O(r)$ space, only within $O(r \log(n/r))$ space (and $O(\ell + \log(n/r))$ time), and this is provided through a run-length grammar of that size and height $O(\log(n/r))$. What is more interesting is that they also build grammars of size $O(r \log(n/r))$ that provide access in time $O(\log(n/r))$ to any cell of A or A^{-1}.

A previous attempt to provide fast searches on top of the BWT [Belazzougui et al. 2015a] combines it with Lempel-Ziv parsing: it uses $O(z(S) + r(S) + r(S^{rev}))$ space and searches in time $O(m(\log\log n + \log z) + occ \log^{\epsilon} z)$. A careful implementation [Belazzougui et al. 2017] shows to be relevant, for example it uses about 3 times more space and is faster than the index of Kreft and Navarro [2013].

4.1.3 *Optimal Search Time.*

Kempa [2019] generalizes the concept of BWT runs to s-runs, where the s symbols preceding each suffix $A[j]$ must coincide. He shows that, if S has r runs,

[7]Gagie et al. [2020] store $A[j' - 1]$ instead of $d(t)$, and thus add $s = A[j] - t$ to return $A[j - 1] = A[j' - 1] + s$.

then it has $O(rs)$ s-runs. Gagie et al. [2020] use this idea to define a new string $S^* = S^0 \cdot S^1 \cdots S^{s-1}$, where S^k is formed by discarding the first k symbols of S and then packing it into "metasymbols" of length s. The metasymbols are lexicographically compared in the same way as the string that composes them. They show that the suffix array interval for P in S and for P^0 in S^* are the same. Since the length of P^0 is $m' = m/s$, one can choose $s = \log\log_w(\sigma + n/r)$ to represent S^* within $O(rs) = O(r \log\log_w(\sigma + n/r))$ space and find the interval $A[sp\,..\,ep]$ of P in S by searching for P^0 in S^*, in time $O(m' \log\log_w(\sigma^s + n/r)) = O(m)$.

In turn, the occurrences are located also in chunks of s, by storing in $d(t)$ not only the information on $A[j'-1]$ but also on $A[j'-2], \ldots, A[j'-s]$, in space $O(rs)$ as well. Thus, we invest a predecessor search, time $O(\log\log_w(n/r))$, but in exchange retrieve s occurrences. The resulting time is $O(m + \log\log_w(n/r) + occ)$, which is converted into the optimal $O(m + occ)$ by handling short patterns separately.

With the new technique of Nishimoto and Tabei [2020], we can obtain $O(m + occ)$ time within $O(r \log\log_w \sigma)$ space, because it is sufficient to set $s = \log\log_w \sigma$ to find the interval $A[sp\,..\,ep]$ in time $O(m)$ and the occurrences are already located in time $O(occ)$. Note that this space is $O(r)$ if $\sigma = O(\text{poly}\, w)$.

RAM-optimal search time is also possible with this index, within $O(rw \log_\sigma \log_w n)$ space [Gagie et al. 2020]. Interestingly, RAM-optimal search time had been obtained only in the classical scenario, using $O(n)$ words of space.

4.2 Based on the CDAWG

In principle, searching the CDAWG as easy as searching a suffix tree [Crochemore and Hancart 1997]: Since any suffix can be read from the root node, we simply move from the root using P until finding its locus node (like on the suffix tree, if we end in the middle of an edge, we move to its target node).

A first problem is that we do not store the strings that label the edges of the CDAWG. Instead, we may store only the first symbols and the lengths of those strings, as done for suffix trees in Section 2.3. Once we reach the locus node, we must verify that all the skipped symbols coincide with P [Belazzougui and Cunial 2017a; Crochemore and Hancart 1997]. The problem is that the string S is not directly available for verification. Since $e = \Omega(g)$, however, we can in principle build a grammar of size $O(e)$ so that we can extract a substring of S of length m, and thus verify the skipped symbols, in time $O(m + \log n)$; recall Section 2.5.

To determine which position of S to extract, we use the property that all the strings arriving at a CDAWG node are suffixes of one another [Blumer et al. 1987, Lem. 1]. Thus, we store the final position in S of the longest string arriving at each node from the root. If the longest string arriving at the locus node v ends at position $t(v)$, and we skipped the last l symbols of the last edge, then P should be equal to $S[t(v) - l - m + 1\,..\,t(v) - l]$, so we extract that substring and compare it with P. If they coincide, then every distinct path from v to the final node, of total length L, represents an occurrence at $S[n - L - l - m + 1\,..\,n - L - l]$. Since every node has at least two outgoing edges, we spend $O(1)$ amortized time per occurrence reported [Blumer et al. 1987]. The total search time is then $O(m + \log n + occ)$.

Example. Let us search for $P = \mathsf{la}$ in the CDAWG of Figure 1. We leave the root by the arrow whose string starts with l, and arrive at the target node v (grayed) with $l = 0$ (because the edge consumes the $m = 2$ symbols of P). The node v can be associated with position $t(v) = 3$, which ends an occurrence of the longest string arriving at v, ala. We then extract $S[t(v) - l - m + 1\,..\,t(v) - l] = S[2\,..\,3] = \mathsf{la}$ and verify that the skipped symbols match P. We then traverse all the forward paths from v, reaching the final node in three ways, with total lengths $L = 8, 14, 6$. Therefore, P occurs in S at positions $n - L - l - m + 1 = 8, 2, 10$.

Another alternative, exploiting the fact that $e = \Omega(r)$, is to enrich the CDAWG with the BWT-based index of size $O(r)$: As shown in Section 4.1.1, we can determine in time $O(m \log\log_w \sigma)$ whether P occurs in S or not, that is, if $sp \le ep$ in the interval $A[sp\,..\,ep]$ we compute. If P occurs in S, then we do not need the grammar to extract and verify the skipped symbols; we can just proceed to output all the occurrences [Belazzougui et al. 2015a]. The total time is then $O(m \log\log_w \sigma + occ)$. This variant is carefully implemented by Belazzougui et al. [2017], who show that the structure is about two orders of magnitude faster than the index of Kreft and Navarro [2013], though it uses an order of magnitude more space.

It is even possible to reach optimal $O(m + occ)$ time with the CDAWG, by exploiting the fact that the CDAWG induces a particular grammar of size $O(e)$ where there is a distinct nonterminal per string labeling a CDAWG edge [Belazzougui and Cunial 2017b]. Since we need to extract a prefix of the string leading to the locus of P, and this is the concatenation of several edges plus a prefix of the last edge, the technique recalled in Section 2.5 allows us to extract the string to verify in time $O(m)$. Variants of this idea are given by Belazzougui and Cunial [2017a] (using $O(e)$ space) and Takagi et al. [2017] (using $O(e(S) + e(S^{rev}))$ space).

5 CURRENT CHALLENGES

In the final section of this survey, we consider the most important open challenges in this area: (1) obtaining practical implementations and (2) being able to build the indexes for very large text collections.

5.1 Practicality and Implementations

There is usually a long way between a theoretical finding and its practical deployment. Many decisions that are made for the sake of obtaining good worst-case complexities, or for simplicity of presentation, are not good in practice. Algorithm engineering is the process of modifying a theoretically appealing idea into a competitive implementation, involving knowledge of the detailed cost model of computers (caching, prefetching, multithreading, etc.). Further, big-O space figures ignore constants, which must be carefully considered to obtain competitive space for the indexes. In practice variables like z, g, r, and so on, are a hundredth or a thousandth of n, and therefore using space like $10z$ bytes may yield a large index in practice.

While competitive implementations have been developed for indexes based on Lempel-Ziv [Kreft and Navarro 2013; Ferrada et al. 2014; Claude et al. 2016, 2018], grammars [Maruyama et al. 2013a; Takabatake et al. 2014; Claude et al. 2016, 2020], the BWT [Mäkinen et al. 2010; Belazzougui et al. 2017; Gagie et al. 2020; Kuhnle et al. 2020], and CDAWGs [Belazzougui et al. 2017], the most recent and promising theoretical developments [Bille et al. 2018; Navarro and Prezza 2019; Christiansen et al. 2019; Kociumaka et al. 2020] are yet to be implemented and tested. It is unknown how much these improvements will impact in practice.

Figure 11 shows, in very broad terms, the space/time tradeoffs obtained by the implementations built on the different repetitiveness concepts. It is made by taking the most representative values obtained across the different experiments of the publications mentioned above, discarding too repetitive and not repetitive enough collections. The black dots represent the run-length BWT built on regular sampling [Mäkinen et al. 2010], which has been a baseline for most comparisons. Though r seems to dominate e, the latter (represented by CDAWG indexes) is implemented only on DNA, where it is nearly twice as fast as r (represented by the r-index).

5.2 Construction and Dynamism

An important obstacle for the practical adoption of the indexes we covered is how to build them on huge datasets. Once built, the indexes are orders of magnitude smaller than the input and

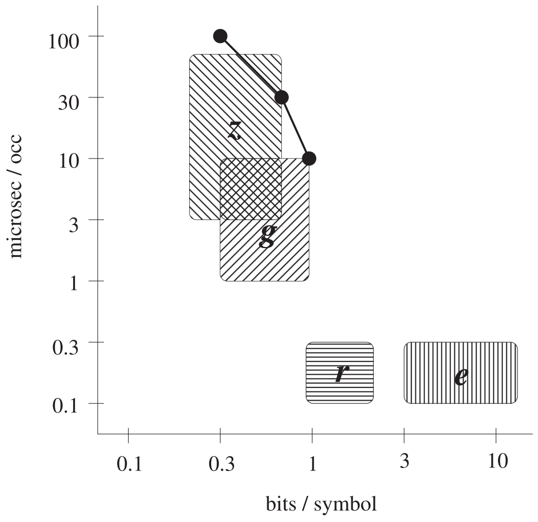

Fig. 11. Space/time tradeoffs of the indexes building on different repetitiveness measures. Both axes are logarithmic: bits per symbol (x) and search time per occurrence in microseconds (y).

one hopes to handle them in main memory. However, the initial step of computing the parsing, the run-length BWT, or another small representation of a very large text collection, even if it can be generally performed in the optimal $O(n)$ time, usually requires $O(n)$ main memory space with a significant constant. There are various approaches that aim to reduce those main memory requirements and/or read the text in streaming mode, but some are still incipient.

Burrows-Wheeler Transform. The BWT is easily obtained from the suffix array, which in turn can be built in $O(n)$ time and space [Kim et al. 2005; Ko and Aluru 2005; Kärkkäinen et al. 2006]. However, the constant associated with the space is large. Kärkkäinen et al. [2006] allow using $O(nv)$ time and $O(n/\sqrt{v})$ space for a parameter v, but they still need to store the $n \log n + n \log \sigma$ bits of the suffix array and the text. External-memory suffix array construction requires optimal $O(Sort(n))$ I/Os and time [Farach-Colton et al. 2000; Kärkkäinen et al. 2006].

There are various algorithms to build the BWT directly using compact space or in external memory [Belazzougui et al. 2020; Beller et al. 2013; Ferragina et al. 2012; Fuentes-Sepúlveda et al. 2020; Hon et al. 2007, 2009; Kärkkäinen 2007; Kempa and Kociumaka 2019; Munro et al. 2017; Okanohara and Sadakane 2009], but they do not produce it in run-length compressed form. Recently, Kempa [2019] showed how to build the run-length BWT in $O(n/\log_\sigma n + r \operatorname{polylog} n)$ time and working space.

With a dynamic representation of sequences that supports insertions of symbols [Munro and Nekrich 2015] one can build the run-length encoded BWT incrementally by traversing the text in reversed form; the LF-mapping formula given in Section 4.1 shows where to insert the next text symbol. This idea is used by Policriti and Prezza [2018] to build the run-length compressed BWT directly, in streaming mode, in $O(n \log r)$ time and within $O(r)$ main memory space. Ohno et al. [2018] improve their practical performace by a factor of 50 (using just twice the space).

Sirén [2016] presents a practical technique to build very large BWTs (i.e., for terabytes of data) in run-length compressed form. It splits the collection into subcollections, builds the individual BWTs, and then merges them into a global one.

Boucher et al. [2019] propose a practical method called "prefix free parsing," which first parses the text using a rolling hash (a Karp-Rabin-like hash that depends on the last ℓ symbols read; a phrase ends whenever the hash modulo a parameter value is zero). The result is a dictionary of phrases and a sequence of phrase identifiers; both are generally much smaller than n when the text is repetitive. They then build the BWT from those elements. Their experiments show that they can build BWTs of very large collections in reasonable time. Kuhnle et al. [2020] show how to add in this construction the sampling used by the run-length BWT [Gagie et al. 2020]; recall Section 4.1.2.

Lempel-Ziv parsing. While it has been known for decades how to obtain this parse in $O(n)$ time [Rodeh et al. 1981; Storer and Szymanski 1982], these algorithms use $O(n)$ space (i.e., $O(n \log n)$ bits) with a significant constant. A long line of research [Chen et al. 2008; Ohlebusch and Gog 2011; Kempa and Puglisi 2013; Kärkkäinen et al. 2013, 2014, 2016; Goto and Bannai 2013, 2014; Yamamoto et al. 2014; Fischer et al. 2015b, 2018; Köppl and Sadakane 2016; Belazzougui and Puglisi 2016; Munro et al. 2017; Kempa 2019] has focused on using little space, reducing the constant associated with the $O(n \log n)$ bits and even reaching $O(n \log \sigma)$ bits [Kärkkäinen et al. 2013; Belazzougui and Puglisi 2016; Munro et al. 2017; Kempa 2019] and statistically compressed space [Policriti and Prezza 2015]. This is still unaware of repetitiveness, however.

Interestingly, the only known method to build the Lempel-Ziv parsing in repetition-aware space ($O(z + r)$) is to build the run-length BWT first and then derive the Lempel-Ziv parse from it [Policriti and Prezza 2018; Ohno et al. 2018; Bannai et al. 2020]. These methods use up to 3 orders of magnitude less space (and are just 2–8 times slower) than the previous approaches. Another interesting development [Kärkkäinen et al. 2014] uses external memory: with a RAM of size M, it performs $O(n^2/M)$ I/Os and requires $2n$ bytes of disk working space. Despite this quadratic complexity, their experiments show that this technique can handle, in practice, much larger texts than previous approaches.

Other approaches aim at approximating the Lempel-Ziv parse. Fischer et al. [2015a] build an $(1 + \epsilon)$-approximation in $O((n/\epsilon) \log n)$ time and $O(z)$ space. Kempa and Kosolobov [2017] build the LZ-End variant [Kreft and Navarro 2013] in streaming mode and $O(z + \ell)$ main memory space, where ℓ is the length of the longest phrase. Valenzuela et al. [2020] use Relative Lempel-Ziv as a building block.

Grammar construction. RePair [Larsson and Moffat 2000] is the heuristic that obtains the best grammars in practice. While it computes the grammar in $O(n)$ time and space, the constant associated with the space is significant and prevents using it on large texts. Attempts to reduce this space have paid a significant price in time [Bille et al. 2017b; Sakai et al. 2019; Köppl et al. 2020]. A recent heuristic [Gagie et al. 2019] obtains space close to that of RePair using a semi-streaming algorithm, but it degrades quickly as the repetitiveness decreases. Various other grammar construction algorithms, for example Sakamoto [2005] and Jeż [2015, 2016], build a balanced grammar that approximates the smallest grammar within an $O(\log n)$ factor by performing a logarithmic number of passes on the text (which halves at each pass), and could be amenable to a semi-streamed construction.

An important line of research in this regard are the online grammar construction algorithms [Maruyama et al. 2012, 2013b; Takabatake et al. 2017]. In OLCA [Maruyama et al. 2012], the authors build a grammar in $O(n)$ time by reading the text in streaming mode. They obtain an $O(\log^2 n)$ approximation to the smallest grammar using $O(g \log^2 n)$ space. In SOLCA [Maruyama et al. 2013b] they reduce the space to $O(g)$. FOLCA [Takabatake et al. 2017] improves the space to $g \log g + o(g \log g)$ bits; the authors also prove that the grammar built by SOLCA and FOLCA is an $O(\log n \log^* n)$ approximation. Their experiments show that the grammar is built very efficiently in time and main memory space, though the resulting grammar is 4–5 times larger than the one generated by RePair.

Dynamism. A related challenge is dynamism, that is, the ability to modify the index when the text changes. Although a dynamic index is clearly more practical than one that has to be rebuilt every time, this is a difficult topic on which little progress has been made. The online construction methods for run-length BWTs [Policriti and Prezza 2018; Ohno et al. 2018; Bannai et al. 2020] and grammars [Maruyama et al. 2013b; Takabatake et al. 2017] naturally allow us adding new text at the beginning or at the end of the string. A dynamic BWT representation allows adding or removing arbitrary documents from a text collection [Mäkinen and Navarro 2008]. Supporting arbitrary modifications to the text is much more difficult, however. We are only aware of two words. One [Nishimoto et al. 2020] builds on edit-sensitive parsing to maintain a grammar under arbitrary substring insertions and deletions. They use $O(z \log n \log^* n)$ space and search in time $O(m(\log\log n)^2 + \log n \log^* n(\log n + \log m \log^* n) + occ \log n)$ (simplified, see the precise formula in the appendix). A substring of length ℓ is inserted/deleted in time $O((\ell + \log n \log^* n) \log^2 n \log^* n)$. In practice, the search is fast for long patterns only; Nishimoto et al. [2018] improved their search time on short patterns. A second work [Gawrychowski et al. 2015] uses another kind of locally-consistent grammar to insert substrings of length ℓ in time $O(\ell \log n)$, split and concatenate represented strings in time $O(\log^2 n)$, and search for patterns in those strings in time $O(m + occ \log n)$; all the operations succeed with high probability.

APPENDIX

A HISTORY OF THE CONTRIBUTIONS TO PARSING-BASED INDEXING

We cover only the developments related to the repetitiveness measures we have considered. Other parsed-based indexes, such as those building on the LZ78 compression format [Ziv and Lempel 1978], are omitted, because they are not competitive on highly repetitive text collections.

Claude and Navarro [2009, 2011] proposed the first compressed index based on grammar compression. Given any grammar of size g, their index uses $O(g)$ space to implement the grid and the tracking of occurrences over the grammar tree, but not yet the amortization mechanism we described. On a grammar tree of height h, the index searches in time $O(m(m+h)\log n + occ \cdot h \log n)$ and extracts a substring of length ℓ in time $O((\ell + h)\log n)$. The terms $O(\log n)$ can be reduced by using more advanced data structures, but the index was designed with practice in mind and it was actually implemented [Claude et al. 2016], using a RePair construction [Larsson and Moffat 2000] that is heuristically balanced.

Kreft and Navarro [2011, 2013] proposed the first compressed index based on Lempel-Ziv, and the only one so far of size $O(z)$. Within this size, they cannot provide access to S with good time guarantees: Each accessed symbol must be traced through the chain of target-to-source dependencies. If the maximum length of such a chain is $h \le z$, then their search time is $O(m^2 h + (m + occ)\log z)$. The term $\log z$ could be $\log^\epsilon z$ by using the geometric structure we have described but, again, they opt for a practical version. Binary searches in $\mathcal{X}$ and $\mathcal{Y}$ are sped up with Patricia trees [Morrison 1968]. A substring of length ℓ is extracted in time $O(\ell h)$. This is the smallest implemented index; it is rather efficient unless the patterns are too long [Claude et al. 2016; Kreft and Navarro 2013]. Interestingly, it outperforms the previous index [Claude and Navarro 2011] both in space (as expected) and time (not expected).

Maruyama et al. [2011, 2013a], and Takabatake et al. [2014] propose another grammar index based on "edit-sensitive parsing," which is related to locally consistent parsing (see the end of Section 3.1.1). This ensures that the parsing of P and of any of its occurrences in S will differ only by a few ($O(\log n \log^* n)$) symbols in the extremes of the respective parse trees, and therefore the internal symbols are consistent. By looking for those one captures all the occ_c *potential* occurrences, which however can be more than the actual occurrences. Given a grammar of size $g_e \geq g$ built using edit-sensitive parsing, their index takes $O(g_e)$ space and searches in time $O(m \log\log n \log^* n + occ_c \log m \log n \log\log n \log^* n)$. Substrings of length ℓ are extracted in time $O((\ell + \log n) \log\log g_e)$. Their index is implemented, and outperforms that of Kreft and Navarro [2013] for $m \geq 100$.

Claude and Navarro [2012] and Claude et al. [2020] improved the proposal of Claude and Navarro [2011] by introducing the amortization mechanism and also using the mechanism to extract phrase prefixes and suffixes in optimal time. The result is an index of size $O(g)$ built on any grammar of size g, which searches in time $O(m^2 \log\log_g n + (m + occ) \log n)$. Again, this index is described with practicality in mind; they show that with larger data structures of size $O(g)$ one can reach search time $O(m^2 + (m + occ) \log^\epsilon n)$. Any substring of size ℓ can be extracted in time $O(\ell + \log n)$ with the mechanisms seen in Part I [Navarro 2020, Sec. 4.1]. An implementation of this index [Claude et al. 2020] outperforms the Lempel-Ziv based index [Kreft and Navarro 2013] in time, while using somewhat more space. The optimal-time extraction of prefixes and suffixes is shown to have no practical impact on balanced grammars.

Gagie et al. [2012] invented bookmarking to speed up substring extraction in the structure of Kreft and Navarro [2013]. They use bookmarking on a Lempel-Ziv parse, of size $O(z \log\log z)$, which is added to a grammar of size $O(g)$ to provide direct access to S. As a result, their index is of size $O(g + z \log\log z)$ and searches in time $O(m^2 + (m + occ) \log\log n)$. Their technique is more sophisticated than the one we present in Part I [Navarro 2020, Sec. 4.3], but it would not improve the tradeoffs we obtained.

Ferrada et al. [2014, 2018] proposed the so-called *hybrid indexing*. Given a maximum pattern length M that can be sought, and a suitable parse (Lempel-Ziv, in their case) of size z, they form a string S' of size $< 2Mz$ by collecting the symbols at distance at most M from a phrase boundary and separating disjoint areas with \$s. Any primary occurrence in S is then found in S', and any occurrence in S' is a distinct occurrence in S. They then index S' using any compact index and search it for P. The occurrences of P in S' that cross the middle of a piece are the primary occurrences of P in S; the other occurrences in S' are discarded, but these are at most occ. The mechanism of Section 3.2.1 to propagate primary to secondary occurrences is then used. Patterns longer than M are searched for by cutting them into chunks of length M and assembling their occurrences. Within space $O(Mz)$, they can search in time $O((m + occ) \log\log n)$ if $m \leq M$. Though they offer no guarantees for longer patterns, their implementation outperforms other classical indexes [Kreft and Navarro 2013; Mäkinen et al. 2010] when m is up to a few times M. The weak point of this index shows up when m is much smaller or much larger than the value M chosen at index construction time. Gagie and Puglisi [2015] relate this technique with earlier more specific developments, and call *kernelization* the general technique to solve string matching problems on repetitive sequences by working on the texts surrounding phrases.

Gagie et al. [2014] extended bookmarking to include fingerprinting as well (Part I [Navarro 2020, Sec. 4.3], again more sophisticated than our presentation), and invented the technique of using fingerprinting to remove the $O(m^2)$ term that appeared in all previous indexes. In the way they present their index, the space is $O(z \log n)$ and the search time is $O(m \log m + occ \log\log n)$.[8]

[8]Their actual space is $O(z(\log^* n + \log(n/z) + \log\log z))$, which they convert to $O(z \log(n/z))$ by assuming a small enough alphabet and using $z = O(n/\log_\sigma n)$.

Nishimoto et al. [2015, 2020] propose the first dynamic compressed index (i.e., one can modify S without rebuilding the index from scratch). It is based on edit-sensitive parsing, and they manage to remove the term occ_c in the previous index [Takabatake et al. 2014] by finding stronger properties of the encoding of P via its parse tree. Their search time is $O(m \min(\log\log n \log\log g_e / \log\log\log n, \sqrt{\log g_e / \log\log g_e}) + \log m \log n \log^* n(\log n + \log m \log^* n) + occ \log n)$.

Bille et al. [2017a, 2018] improve upon the result of Gagie et al. [2014]. They propose for the first time the batched search for the pattern prefixes and suffixes, recall Section 3.1.1. They also speed up the searches by storing more points in the grid: If we store the points $S[i], \ldots, S[i+\tau-1]$ for every phrase starting at $S[i]$, then we need to check only one out of τ partitions of P, that is, we check m/τ partitions. This leads to various tradeoffs, which in simplified form are $O(z\log(n/z)\log\log z)$ space and $O((m+occ)\log\log n)$ time, $O(z(\log(n/z)+\log\log z))$ space and $O((m+occ)\log^\epsilon n)$ time, $O(z(\log(n/z)+\log\log z)\log\log z)$ space and $O(m+occ\log\log n)$ time, and $O(z(\log(n/z)+\log^\epsilon n))$ space and $O(m+occ\log^\epsilon n)$ time. The last two reach for the first time linear complexity in m. They also show how to extract a substring of length ℓ in time $O(\ell+\log(n/z))$.

Navarro [2017] and Navarro and Prezza [2019] build a compressed index based on block trees, which are used to provide both access and a suitable parse of S. They reuse the idea of the grid and the mechanism to propagate secondary occurrences. By using a block tree built on attractors [Navarro and Prezza 2019], they obtain $O(\gamma\log(n/\gamma))$ space and $O(m\log n + occ\log^\epsilon n)$ search time. They called this index "universal," because it was the first one built on a general measure of compressibility (attractors) instead of on specific compressors like grammars or Lempel-Ziv. For example, if one builds a bidirectional macro scheme of size b (on which no index has been proposed), then one can use it as an upper bound to γ and have a functional index of size $O(b\log(n/b))$.

Christiansen and Ettienne [2018] were the first to show that, using a locally consistent parsing, only $O(\log m)$ partitions of P need be considered (see the end of Section 3.1.1). Building on the grammar-based index of Claude and Navarro [2012] and on batched pattern searches (Section 3.1.1), they obtain an index using $O(z(\log(n/z)+\log\log z))$ space and $O(m+\log^\epsilon(z\log(n/z))+occ(\log\log n+\log^\epsilon z)) \subseteq O(m+(1+occ)\log^\epsilon n)$ time,[9] thus offering another tradeoff with time linear in m.

Christiansen et al. [2019] rebuild the result of Christiansen and Ettienne [2018] on top of attractors, like Navarro and Prezza [2019]. They use a slightly different run-length grammar, which is proved to be of size $O(\gamma\log(n/\gamma))$, and a better mechanism to track secondary occurrences within constant amortized time [Claude and Navarro 2012]. Their index, of size $O(\gamma\log(n/\gamma))$, then searches in time $O(m+\log^\epsilon\gamma+occ\log^\epsilon(\gamma\log(n/\gamma))) \subseteq O(m+(1+occ)\log^\epsilon n)$. By enlarging the index to size $O(\gamma\log(n/\gamma)\log^\epsilon n)$, they reach for the first time optimal time in parsing-based indexes, $O(m+occ)$. Several other intermediate tradeoffs are obtained too. Interestingly, they obtain this space in terms of γ without the need to find the smallest attractor, which makes the index implementable (they use measure δ, see Part I [Navarro 2020, Sec. 3.10], to approximate γ). Finally, they extend the current results on indexes based on grammars to run-length grammars, thus reaching an index of size $O(g_{rl})$ that searches in time $O(m\log n + occ\log^\epsilon n)$.

Kociumaka et al. [2020] prove that the original block trees [Belazzougui et al. 2015b] are not only of size $O(z\log(n/z))$, but also $O(\delta\log(n/\delta))$. They then show that the universal index

[9]This corrected time is given in the journal version [Christiansen et al. 2019].

of Navarro and Prezza [2019] can also be represented in space $O(\delta \log(n/\delta))$ and is directly implementable within this space. The search time, $O(m \log n + occ \log^{\epsilon} n)$, is also obtained in space $O(g_{rl})$ [Christiansen et al. 2019], which as explained can be proved to be in $O(\delta \log(n/\delta))$, though there is no efficient way to obtain a run-length grammar of the optimal size g_{rl}.

Tsuruta et al. [2020] use the grammar-based techniques we have presented to build an index that searches in time $O(m + \log n \log m + occ \log n)$, by exploiting the properties of a particular grammar: they decompose S recursively into Lyndon words (a Lyndon word is lexicographically smaller than its suffixes), and then build a binary grammar that follows the decomposition, in the hope that the same nonterminals are generated many times. They empirically show that the resulting grammar is only 1.5–2.0 times larger than that of RePair.

ACKNOWLEDGMENTS

We thank Travis Gagie, Nicola Prezza, and the reviewers for useful comments.

REFERENCES

A. Apostolico. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words (NATO ISI Series)*. Springer-Verlag, 85–96.

R. Baeza-Yates and B. Ribeiro-Neto. 2011. *Modern Information Retrieval* (2nd ed.). Addison-Wesley.

H. Bannai, T. Gagie, and T. I. 2020. Refining the r-index. *Theor. Comput. Sci.* 812 (2020), 96–108.

T. Batu, F. Ergün, and S. C. Sahinalp. 2006. Oblivious string embeddings and edit distance approximations. In *Proceedings of the 17th Symposium on Discrete Algorithms (SODA'06)*. 792–801.

D. Belazzougui, Paolo B., R. Pagh, and S. Vigna. 2010. Fast prefix search in little space, with applications. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. 427–438.

D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. 2009. Monotone minimal perfect hashing: Searching a sorted table with O(1) accesses. In *Proceedings of the 20th Annual Symposium on Discrete Mathematics (SODA'09)*. 785–794.

D. Belazzougui and F. Cunial. 2017a. Fast label extraction in the CDAWG. In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE'17)*. 161–175.

D. Belazzougui and F. Cunial. 2017b. Representing the suffix tree with the CDAWG. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM'17)*. 7:1–7:13.

D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. 2015a. Composite repetition-aware data structures. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM'15)*. 26–39.

D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. 2017. Flexible indexing of repetitive collections. In *Proceedings of the 13th Conference on Computability in Europe (CiE'17)*. 162–174.

D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. 2020. Linear-time string indexing and analysis in small space. *ACM Trans. Algor.* 16, 2 (2020), article 17.

D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. Ordóñez, S. J. Puglisi, and Y. Tabei. 2015b. Queries on LZ-bounded encodings. In *Proceedings of the 25th Data Compression Conference (DCC'15)*. 83–92.

D. Belazzougui and G. Navarro. 2015. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algor.* 11, 4 (2015), article 31.

D. Belazzougui and S. J. Puglisi. 2016. Range predecessor and lempel-ziv parsing. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'16)*. 2053–2071.

T. Beller, M. Zwerger, S. Gog, and E. Ohlebusch. 2013. Space-efficient construction of the burrows-wheeler transform. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE'13)*. 5–16.

M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. 2005. Lowest common ancestors in trees and directed acyclic graphs. *J. Algor.* 57, 2 (2005), 75–94.

P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. 2017a. Time-space trade-offs for lempel-ziv compressed indexing. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM'17)*. 16:1–16:17.

P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. 2018. Time-space trade-offs for lempel-ziv compressed indexing. *Theor. Comput. Sci.* 713 (2018), 66–77.

P. Bille, I. L. Gørtz, and N. Prezza. 2017b. Space-efficient re-pair compression. In *Proceedings of the 27th Data Compression Conference (DCC'17)*. 171–180.

P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. 2014. Time-space trade-offs for longest common extensions. *J. Discr. Algor.* 25 (2014), 42–50.

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

C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, and T. Mun. 2019. Prefix-free parsing for building big BWTs. *Algor. Molec. Biol.* 14, 1 (2019), 13:1–13:15.

S. Büttcher, C. L. A. Clarke, and G. V. Cormack. 2010. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press.

T. M. Chan, K. G. Larsen, and M. Pătraşcu. 2011. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry (SoCG'11).* 1–10.

G. Chen, S. J. Puglisi, and W. F. Smyth. 2008. Lempel-ziv factorization using less time & space. *Math. Comput. Sci.* 1 (2008), 605–623.

A. R. Christiansen and M. B. Ettienne. 2018. Compressed indexing with signature grammars. In *Proceedings of the13th Latin American Symposium on Theoretical Informatics (LATIN'18).* 331–345.

A. R. Christiansen, M. B. Ettienne, T. Kociumaka, G. Navarro, and N. Prezza. 2020. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms* 17, 1, Article 8 (2020), 207–219.

F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. 2016. Universal indexes for highly repetitive document collections. *Inf. Syst.* 61 (2016), 1–23.

F. Claude and G. Navarro. 2009. Self-indexed text compression using straight-line programs. In *Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS'09).* 235–246.

F. Claude and G. Navarro. 2011. Self-indexed grammar-based compression. *Fundam. Inf.* 111, 3 (2011), 313–337.

F. Claude and G. Navarro. 2012. Improved grammar-based compressed indexes. In *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE'12).* 180–192.

F. Claude, G. Navarro, and A. Pacheco. 2021. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences* 118 (2021), 53–74.

R. Cole and U. Vishkin. 1986. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Contr.* 70, 1 (1986), 32–53.

M. Crochemore and C. Hancart. 1997. Automata for matching patterns. In *Handbook of Formal Languages.* Springer, 399–462.

M. Crochemore and W. Rytter. 2002. *Jewels of Stringology.* World Scientific.

M. Farach and M. Thorup. 1995. String matching in lempel-ziv compressed strings. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC'95).* 703–712.

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

H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. 2014. Hybrid indexes for repetitive datasets. *Philos. Trans. Roy. Soc. A* 372, 2016 (2014), article 20130137.

H. Ferrada, D. Kempa, and S. J. Puglisi. 2018. Hybrid indexing revisited. In *Proceedings of the 20th Workshop on Algorithm Engineering and Experiments (ALENEX'18).* 1–8.

P. Ferragina, T. Gagie, and G. Manzini. 2012. Lightweight data indexing and compression in external memory. *Algorithmica* 63, 3 (2012), 707–730.

P. Ferragina and R. Grossi. 1999. The string b-tree: A new data structure for string search in external memory and its applications. *J. ACM* 46, 2 (1999), 236–280.

P. Ferragina and G. Manzini. 2000. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS'00).* 390–398.

P. Ferragina and G. Manzini. 2005. Indexing compressed texts. *J. ACM* 52, 4 (2005), 552–581.

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

J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka. 2015a. Approximating LZ77 via small-space multiple-pattern matching. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA).* 533–544.

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

J. Fischer, T. I, and D. Köppl. 2015b. Lempel ziv computation in small space (LZ-CISS). In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM'15).* 172–184.

J. Fischer, T. I. D. Köppl, and K. Sadakane. 2018. Lempel-ziv factorization powered by space efficient suffix trees. *Algorithmica* 80, 7 (2018), 2048–2081.

J. Fuentes-Sepúlveda, G. Navarro, and Y. Nekrich. 2020. Parallel computation of the burrows wheeler transform in compact space. *Theor. Comput. Sci.* 812 (2020), 123–136.

T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. 2012. A faster grammar-based self-index. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA'12)*. 240–251.

T. Gagie, P Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. 2014. LZ77-based self-indexing with faster pattern matching. In *Proceedings of the 11th Latin American Symposium on Theoretical Informatics (LATIN'14)*. 731–742.

T. Gagie, T. I, G. Manzini, G. Navarro, H. Sakamoto, and Y. Takabatake. 2019. Rpair: Scaling up repair with rsync. In *Proceedings of the 26th International Symposium on String Processing and Information Retrieval (SPIRE'19)*. 35–44.

T. Gagie, G. Navarro, and N. Prezza. 2018. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'18)*. 1459–1477.

T. Gagie, G. Navarro, and N. Prezza. 2020. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM* 67, 1 (2020), article 2.

T. Gagie and S. J. Puglisi. 2015. Searching and indexing genomic databases via kernelization. *Front. Bioeng. Biotechnol.* 3 (2015), article 12.

K. Goto and H. Bannai. 2013. Simpler and faster lempel ziv factorization. In *Proceedings of the 23rd Data Compression Conference (DCC'13)*. 133–142.

K. Goto and H. Bannai. 2014. Space efficient linear time lempel-ziv Factorization for Small Alphabets. In *Proceedings of the 24th Data Compression Conference (DCC'14)*. 163–172.

R. Grossi. 2011. A quick tour on suffix arrays and compressed suffix arrays. *Theor. Comput. Sci.* 412, 27 (2011), 2964–2973.

R. Grossi and J. S. Vitter. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC'00)*. 397–406.

D. Gusfield. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Lacki, and P. Sankowski. 2015. Optimal dynamic strings. *CoRR* 1511.02612 (2015).

W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. 2007. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica* 48, 1 (2007), 23–36.

W.-K. Hon, K. Sadakane, and W.-K. Sung. 2009. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.* 38, 6 (2009), 2162–2178.

A. Jeż. 2015. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.* 592 (2015), 115–134.

A. Jeż. 2016. A really simple approximation of smallest grammar. *Theor. Comput. Sci.* 616 (2016), 141–150.

J. Kärkkäinen. 2007. Fast BWT in small space by blockwise suffix sorting. *Theor. Comput. Sci.* 387, 3 (2007), 249–257.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. 2013. Lightweight lempel-ziv parsing. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. 139–150.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. 2014. Lempel-ziv parsing in external memory. In *Proceedings of the 24th Data Compression Conference (DCC'14)*. 153–162.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. 2016. Lazy lempel-ziv factorization algorithms. *ACM J. Exp. Algor.* 21, 1 (2016), 2.4:1–2.4:19.

J. Kärkkäinen, P. Sanders, and S. Burkhardt. 2006. Linear work suffix array construction. *J. ACM* 53, 6 (2006), 918–936.

J. Kärkkäinen and E. Ukkonen. 1996. Lempel-ziv parsing and sublinear-size index structures for string matching. In *Proceedings of the 3rd South American Workshop on String Processing (WSP'96)*. 141–155.

R. M. Karp and M. O. Rabin. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 2 (1987), 249–260.

D. Kempa. 2019. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'19)*. 1344–1357.

D. Kempa and T. Kociumaka. 2019. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC'19)*. 756–767.

D. Kempa and D. Kosolobov. 2017. LZ-end parsing in compressed space. In *Proceedings of the 27th Data Compression Conference (DCC'17)*. 350–359.

D. Kempa and S. J. Puglisi. 2013. Lempel-ziv factorization: Simple, fast, practical. In *Proceedings of the 15th Workshop on Algorithm Engineering and Experiments (ALENEX'13)*. 103–112.

D. K. Kim, J. S. Sim, H. Park, and K. Park. 2005. Constructing suffix arrays in linear time. *J. Discr. Algor.* 3, 2–4 (2005), 126–142.

P. Ko and S. Aluru. 2005. Space efficient linear time construction of suffix arrays. *J. Discr. Algor.* 3, 2–4 (2005), 143–156.

T. Kociumaka, G. Navarro, and N. Prezza. 2020. Towards a definitive measure of repetitiveness. In *Proceedings of the 14th Latin American Symposium on Theoretical Informatics (LATIN'20)*.

D. Köppl, T. I. I. Furuya, Y. Takabatake, K. Sakai, and K. Goto. 2020. Re-pair in small space. In *Proceedings of the 30th Data Compression Conference (DCC'20)*. 377.

D. Köppl and K. Sadakane. 2016. Lempel-ziv computation in compressed space (LZ-CICS). In *Proceedings of the 26th Data Compression Conference (DCC'16)*. 3–12.

S. Kreft and G. Navarro. 2011. Self-indexing based on LZ77. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM'11)*. 41–54.

S. Kreft and G. Navarro. 2013. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.* 483 (2013), 115–133.

A. Kuhnle, T. Mun, C. Boucher, T. Gagie, B. Langmead, and G. Manzini. 2020. Efficient construction of a complete index for pan-genomics read alignment. *J. Comput. Biol.* 27, 4 (2020), 500–513.

J. Larsson and A. Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. 2009. Sourcerer: Mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.* 18, 2 (2009), 300–336.

B. Liu. 2007. *Web Data Mining: Exploring Hyperlinks, Contents and Usage Data.* Springer.

V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. 2015. *Genome-Scale Algorithm Design.* Cambridge University Press.

V. Mäkinen and G. Navarro. 2005. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.* 12, 1 (2005), 40–66.

V. Mäkinen and G. Navarro. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algor.* 4, 3 (2008), article 32.

V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. 2010. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.* 17, 3 (2010), 281–308.

U. Manber and G. Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* 22, 5 (1993), 935–948.

S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. 2011. ESP-Index: A compressed index based on edit-sensitive parsing. In *Proceedings of the 18th International Symposium on String Processing and Information Retrieval (SPIRE'11)*. 398–409.

S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. 2013a. ESP-index: A compressed index based on edit-sensitive parsing. *J. Discr. Algor.* 18 (2013), 100–112.

S. Maruyama, H. Sakamoto, and M. Takeda. 2012. An online algorithm for lightweight grammar-based compression. *Algorithms* 5, 2 (2012), 213–235.

S. Maruyama, Y. Tabei, H. Sakamoto, and K. Sadakane. 2013b. Fully-online grammar compression. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE'13)*. 218–â229.

E. McCreight. 1976. A space-economical suffix tree construction algorithm. *J. ACM* 23, 2 (1976), 262–272.

K. Mehlhorn, R. Sundar, and C. Uhrig. 1997. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica* 17, 2 (1997), 183–198.

D. Morrison. 1968. PATRICIA—Practical algorithm to retrieve information coded in alphanumeric. *J. ACM* 15, 4 (1968), 514–534.

J. I. Munro, G. Navarro, and Y. Nekrich. 2017. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'17)*. 408–424.

J. I. Munro and Y. Nekrich. 2015. Compressed data structures for dynamic sequences. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*. 891–902.

G. Navarro. 2017. A self-index on block trees. In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE'17)*. 278–289.

G. Navarro. 2020. Indexing highly repetitive string collections, Part I: Repetitiveness measures *CoRR* 2004.02781 (2020).

G. Navarro and V. Mäkinen. 2007. Compressed full-text indexes. *Comput. Surv.* 39, 1 (2007), article 2.

G. Navarro and Y. Nekrich. 2017. Time-optimal top-k document retrieval. *SIAM J. Comput.* 46, 1 (2017), 89–113.

G. Navarro and N. Prezza. 2019. Universal compressed text indexing. *Theor. Comput. Sci.* 762 (2019), 41–50.

T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. 2015. Dynamic index, LZ factorization, and LCE queries in compressed space. *CoRR* 1504.06954 (2015).

T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. 2020. Dynamic index and LZ factorization in compressed space. *Discr. Appl. Math.* 274 (2020), 116–129.

T. Nishimoto and Y. Tabei. 2019. LZRR: LZ77 parsing with right reference. In *Proceedings of the 29th Data Compression Conference (DCC'19)*. 211–220.

T. Nishimoto and Y. Tabei. 2020. Faster queries on BWT-runs compressed indexes. *CoRR* 2006.05104 (2020).

T. Nishimoto, Y. Takabatake, and Y. Tabei. 2018. A dynamic compressed self-index for highly repetitive text collections. In *Proceedings of the 28th Data Compression Conference (DCC'18)*. 287–296.

E. Ohlebusch. 2013. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag.

E. Ohlebusch and S. Gog. 2011. Lempel-ziv factorization revisited. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM'11)*. 15–26.

T. Ohno, K. Sakai, Y. Takabatake, T. I, and H. Sakamoto. 2018. A faster implementation of online RLBWT and its application to LZ77 parsing. *J. Discr. Algor.* 52–53 (2018), 18–28.

D. Okanohara and K. Sadakane. 2009. A linear-time burrows-wheeler transform using induced sorting. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE'09),* Lecture Notes in Computer Science, Vol. 5721. 90–101.

A. Policriti and N. Prezza. 2015. Fast online lempel-ziv factorization in compressed space. In *Proceedings of the 22nd String Processing and Information Retrieval (SPIRE'15).* 13–20.

A. Policriti and N. Prezza. 2018. LZ77 computation based on the run-length encoded BWT. *Algorithmica* 80, 7 (2018), 1986–2011.

M. Rodeh, V. R. Pratt, and S. Even. 1981. Linear algorithm for data compression via string matching. *J. ACM* 28, 1 (1981), 16–24.

L. M. S. Russo, A. Correia, G. Navarro, and A. P. Francisco. 2020. Approximating optimal bidirectional macro schemes. In *Proceedings of the 30th Data Compression Conference (DCC'20).* 153–162.

S. C. Sahinalp and U. Vishkin. 1995. *Data Compression Using Locally Consistent Parsing.* Technical Report. Department of Computer Science, University of Maryland.

K. Sakai, T. Ohno, K. Goto, Y. Takabatake, T. I, and H. Sakamoto. 2019. RePair in compressed space and time. In *Proceedings of the 29th Data Compression Conference (DCC'19).* 518–527.

H. Sakamoto. 2005. A fully linear-time approximation algorithm for grammar-based compression. *J. Discr. Algor.* 3, 2â4 (2005), 416–430.

F. Silvestri. 2010. Mining query logs: Turning search usage data into knowledge. *Found. Trends Inf. Retriev.* 4, 1–2 (2010), 1–174.

J. Sirén. 2016. Burrows-wheeler transform for terabases. In *Proceedings of the 26th Data Compression Conference (DCC'16).* 211–220.

J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. 2008. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE'08).* 164–175.

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

J.-H. Su, Y.-T. Huang, H.-H. Yeh, and V. S. Tseng. 2010. Effective content-based video retrieval using pattern-indexing and matching techniques. *Expert Syst. Appl.* 37, 7 (2010), 5068–5085.

Y. Takabatake, T. I, and H. Sakamoto. 2017. A space-optimal grammar compression. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA'17).* 67:1–67:15.

Y. Takabatake, Y. Tabei, and H. Sakamoto. 2014. Improved ESP-index: A practical self-index for highly repetitive texts. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14).* 338–350.

T. Takagi, K. Goto, Y. Fujishige, S. Inenaga, and H. Arimura. 2017. Linear-size CDAWG: New repetition-aware indexing and grammar compression. In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE'17).* 304–316.

K. Tsuruta, D. Köppl, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. 2020. Grammar-compressed Self-index with lyndon words. *CoRR* 2004.05309 (2020).

R. Typke, F. Wiering, and R. Veltkamp. 2005. A survey of music information retrieval systems. In *Proceedings of the 6th International Conference on Music Information Retrieval (ISMIR'05).* 153–160.

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

D. Valenzuela, D. Kosolobov, G. Navarro, and S. J. Puglisi. 2020. Lempel-Ziv like parsing in small space. *Algorithmica* 82, 11 (2020), 3195–3215.

P. Weiner. 1973. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory (FOCS'73).* 1–11.

J. Yamamoto, T. I, H. Bannai, S. Inenaga, and M. Takeda. 2014. Faster compact on-line lempel-ziv factorization. In *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS'14).* 675–686.

J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory* 24, 5 (1978), 530–536.

Received April 2020; revised October 2020; accepted October 2020