Document Retrieval on Repetitive String Collections

Travis Gagie · Aleksi Hartikainen · Kalle Karhu · Juha Kärkkäinen · Gonzalo Navarro · Simon J. Puglisi · Jouni Sirén ·

Received: date / Accepted: date

Abstract Most of the fastest-growing string collections today are repetitive, that is, most of the constituent documents are similar to many others. As these collections keep growing, a key approach to handling them is to exploit their repetitiveness, which can reduce their space usage by orders of magnitude.

Preliminary partial versions of this paper appeared in Proc. CPM 2013, Proc. ESA 2014, and Proc. DCC 2015. This work was done while the third author was at Department of Computer Science and Engineering, Aalto University, Finland.

Travis Gagie
Helsinki Institute of Information Technology, Department of Computer Science, University of Helsinki, Finland
E-mail: travis.gagie@gmail.com

Aleksi Hartikainen
Google Inc, USA
E-mail: ahartik@gmail.com

Kalle Karhu
Research and Technology, Planmeca Oy, Finland
E-mail: kalle.karhu@iki.fi

Juha Kärkkäinen
Helsinki Institute of Information Technology, Department of Computer Science, University of Helsinki, Finland
E-mail: tpkarkka@cs.helsinki.fi

Gonzalo Navarro
CeBiB — Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile
E-mail: gnavarro@dcc.uchile.cl

Simon J. Puglisi
Helsinki Institute of Information Technology, Department of Computer Science, University of Helsinki, Finland
E-mail: puglisi@cs.helsinki.fi

Jouni Sirén
Wellcome Trust Sanger Institute, UK
E-mail: jouni.siren@iki.fi
We study the problem of indexing repetitive string collections in order to perform efficient document retrieval operations on them. Document retrieval problems are routinely solved by search engines on large natural language collections, but the techniques are less developed on generic string collections. The case of repetitive string collections is even less understood, and there are very few existing solutions. We develop two novel ideas, interleaved LCPs and precomputed document lists, that yield highly compressed indexes solving the problem of document listing (find all the documents where a string appears), top-\(k\) document retrieval (find the \(k\) documents where a string appears most often), and document counting (count the number of documents where a string appears). We also show that a classical data structure supporting the latter query becomes highly compressible on repetitive data. We thoroughly evaluate the resulting techniques in various real-life repetitiveness scenarios, and also show how they are combined to solve ranked conjunctive and disjunctive queries under the simple tf-idf model of relevance.

**Keywords** Repetitive string collections · Document retrieval on strings · Suffix trees and arrays

1 Introduction

Document retrieval on natural language text collections is a routine activity in web and enterprise search engines. It is solved with variants of the inverted index (Büttcher et al. 2010; Baeza-Yates and Ribeiro-Neto 2011), a technology that can by now be considered mature. The inverted index has limitations, however, that hamper its use in other kinds of string collections, such as DNA and protein sequences, source code, music streams, and even some East Asian languages. Document retrieval queries are of interest in those string collections, but the state of the art is much less developed (Hon et al. 2013; Navarro 2014).

In this article we focus on **repetitive string collections**, where most of the strings are very similar to many others. These types of collections arise naturally in scenarios like versioned document collections (such as Wikipedia\(^1\) or the Wayback Machine\(^2\), versioned software repositories, periodical data publications in text form (where very similar data is published over and over), sequence databases with genomes of individuals of the same species (which differ at relatively few positions), and so on. Such collections are the fastest-growing ones today. For example, genome sequencing data is expected to grow at least as fast as astronomical, YouTube, or Twitter data by 2025, outperforming Moore’s Law by a significant margin (Stephens et al. 2015). This growth brings new scientific opportunities but also new computational problems.

A key tool for handling this kind of growth is to exploit repetitiveness to obtain size reductions of orders of magnitude. An appropriate Lempel-Ziv compressor\(^3\) can successfully capture such repetitiveness, and version control

---

1. [www.wikipedia.org](http://www.wikipedia.org)
2. From the Internet Archive, [www.archive.org/web/web.php](http://www.archive.org/web/web.php)
3. Such as p7zip, [http://p7zip.sourceforge.net](http://p7zip.sourceforge.net)
Document Retrieval on Repetitive String Collections

systems have offered direct access to any version since their beginnings, by means of storing the edits of a version with respect to some other version that is stored in full (Rockkind, 1975). However, document retrieval requires much more than retrieving individual documents. In this article we focus on three basic document retrieval problems on string collections:

**Document Listing:** Given a string $P$, list the identifiers of all the $df$ documents where $P$ appears.

**Top-k Retrieval:** Given a string $P$ and $k$, list $k$ documents where $P$ appears most often.

**Document Counting:** Given a string $P$, return the number $df$ of documents where $P$ appears.

Apart from the obvious case of Information Retrieval on East Asian and other languages where separating words is difficult, these queries are relevant in many other applications where string collections are maintained. For example, in pan-genomics (Marschall et al, 2016) we index the genomes of all the strains of an organism. The index can be either a specialized data structure, such as a colored de Bruijn graph, or a text index over the concatenation of the individual genomes. The parts of the genome common to all strains are called core; the parts common to several strains are called peripheral; and the parts in only one strain are called unique. Given a set of DNA reads from an unidentified strain, we may want to identify it (if it is known) or find the closest strain in our database (if it is not), by identifying reads from unique or peripheral genomes (i.e., those that occur rarely) and listing the corresponding strains. This boils down to document listing and counting problems. In turn, top-$k$ retrieval is at the core of Information Retrieval systems, since the term frequency $tf$ (i.e., the number of times a pattern appears in a document) is a basic criterion to establish the relevance of a document for a query (Büttcher et al, 2010; Baeza-Yates and Ribeiro-Neto, 2011). On multi-term queries, it is usually combined with the document frequency, $df$. Document counting is also important for data mining applications on strings (or string mining (Dhaliwal et al, 2012), where the value $df/d$ of a given pattern, $d$ being the total number of documents, is its support in the collection. Finally, we will show that the best choice of document listing and top-$k$ retrieval algorithms in practice strongly depends on the $df/occ$ ratio, where $occ$ is the number of times the pattern appears in the collection, and thus the ability to compute $df$ quickly allows for the efficient selection of an appropriate listing or top-$k$ algorithm at query time. Navarro (2014) lists several other applications of these queries.

In the case of natural language, there exist some successful attempts at reducing the inverted index size by exploiting the text repetitiveness (Anick and Flynn, 1992; Broder et al, 2006; He et al, 2009, 2010; He and Suel, 2012; Claude et al, 2016). For general string collections, the situation is much worse. Most of the indexing structures designed for repetitive string collections (Mäkinen et al, 2010; Claude et al, 2010; Claude and Navarro, 2010; 2012; Kreft and Navarro, 2013; Gagie et al, 2012a, 2014; Do et al, 2014; Beffazzougui et al, 2015) are oriented toward pattern matching, that is, given $P$,
they can count or list the occurrences of \( P \) in the whole collection. Of course one can retrieve the occurrences and then answer any of our three document retrieval queries, but the time will be \( \Omega(\text{occ}) \). Instead, there exist optimal-time indexes for string collections that solve document listing in time \( O(|P| + df) \) (Muthukrishnan, 2002), top-\( k \) retrieval in time \( O(|P| + k) \) (Navarro and Nekrich, 2012), and document counting in time \( O(|P|) \) (Sadakane, 2007). The first two solutions, however, use a great deal of space even for classical, non-repetitive collections. While several more compact representations have been studied (Hon et al, 2013; Navarro, 2014), none of those is tailored to the repetitive scenario, except for a grammar-based index that solves document listing (Claude and Munro, 2013).

In this article we develop several novel solutions for the three document retrieval queries of interest, tailored to repetitive string collections. Our first idea, called interleaved LCPs (ILCP) stores the longest common prefix (LCP) array of the documents, interleaved in the order of the global LCP array. The ILCP turns out to have a number of interesting properties that make it compressible on repetitive collections, and useful for document listing and counting. Our second idea, precomputed document lists (PDL), samples some nodes in the global suffix tree of the collection and stores precomputed answers on those. Then it applies grammar compression on the stored answers, which is effective when the collection is repetitive. PDL yields very efficient solutions for document listing and top-\( k \) retrieval. Finally, we show that a solution for document counting (Sadakane, 2007) that uses just two bits per symbol in the worst case (which is unacceptably high in the repetitive scenario) turns out to be highly compressible when the collection is repetitive, and becomes the most attractive solution for document counting.

We implement and experimentally compare several variants of our solutions with the state of the art, including the solution for repetitive string collections (Claude and Munro, 2013) and some relevant solutions for general string collections (Ferrada and Navarro, 2013; Gog and Navarro, 2015a). We consider various kinds of real-life repetitiveness scenarios, and show which solutions are the best depending on the kind and amount of repetitiveness, and the space reduction that can be achieved. For example, on very repetitive collections we perform document listing and top-\( k \) retrieval in 10–100 microseconds per result and using 1–2 bits per character. For counting, we use as little as 0.1 bits per character and answer queries in less than a microsecond. Finally, we show how the different components of our solutions can be assembled to offer tf-idf ranked conjunctive and disjunctive queries on repetitive string collections.

2 Preliminaries

2.1 Suffix Trees and Arrays

A large number of solutions for pattern matching or document retrieval on string collections rely on the suffix tree (Weiner, 1973) or the suffix array
Assume that we have a collection of \( d \) strings, each terminated with a special character “$” (which we consider to be lexicographically smaller than any other character), and let \( T[1..n] \) be their concatenation. The suffix tree of \( T \) is a compacted digital tree where all the suffixes \( T[i..n] \) are inserted. Collecting the leaves of the suffix tree yields the suffix array, \( SA[1..n] \), which is an array of pointers to all the suffixes sorted in increasing lexicographic order, that is, \( T[SA[i]..n] < T[SA[i+1]..n] \) for all \( 1 \leq i < n \). To find all the \( \text{occ} \) occurrences of a string \( P[1..m] \) in the collection, we traverse the suffix tree following the symbols of \( P \) and output the leaves of the node we arrive at, called the locus of \( P \), in time \( O(m + \text{occ}) \).

On a suffix array, we obtain the range \( SA[\ell..r] \) of the leaves (i.e., of the suffixes prefixed by \( P \)) by binary search, and then list the contents of the range, in total time \( O(m \log n + \text{occ}) \).

We will make use of compressed suffix arrays (Navarro and Mäkinen, 2007), which we will call generically CSAs. Their size in bits is denoted \(|CSA|\), their time to find \( \ell \) and \( r \) is denoted \( \text{search}(m) \), and their time to access any cell \( SA[i] \) is denoted \( \text{lookup}(n) \). As it is generally the case, we will assume \( \text{lookup}(n) = \Omega(\log n) \) for simplicity. A particular version of the CSA that is tailored for repetitive collections is the Run-Length Compressed Suffix Array (RLCSA) (Mäkinen et al., 2010).

### 2.2 Rank and Select on Sequences

Let \( S[1..n] \) be a sequence over an alphabet \([1..\sigma]\). When \( \sigma = 2 \) we use 0 and 1 as the two symbols, and the sequence is called a bitvector. Two operations of interest on \( S \) are \( \text{rank}_c(S, i) \), which counts the number of occurrences of symbol \( c \) in \( S[1..i] \), and \( \text{select}_c(S, j) \), which gives the position of the \( j \)th occurrence of symbol \( c \) in \( S \). For bitvectors, one can compute both functions in \( O(1) \) time using \( o(n) \) bits on top of \( S \) (Clark, 1996). If \( S \) contains \( m \) 1s, we can also represent it using \( m \log \frac{n}{m} + O(m) \) bits, so that \( \text{rank} \) takes \( O(\log \frac{n}{m}) \) time and \( \text{select} \) takes \( O(1) \) (Okano and Sadakane, 2007).

The wavelet tree (Grossi et al., 2003) is a tool for extending bitvector representations to sequences. It is a binary tree where the alphabet \([1..\sigma]\) is recursively partitioned in some way. The root represents \( S \) and stores a bitvector \( W[1..n] \) where \( W[i] = 0 \) iff symbol \( S[i] \) belongs to the left child. Left and right children represent a subsequence of \( S \) formed by the symbols of \([1..\sigma]\) they handle, so they recursively store a bitvector and so on until reaching the leaves, which represent a single symbol. By giving constant-time \( \text{rank} \) and \( \text{select} \) capabilities to the bitvectors associated with the nodes, the wavelet tree can compute any \( S[i] = c \), \( \text{rank}_c(S, i) \), or \( \text{select}_c(S, j) \) in time proportional to the depth of the leaf of \( c \). If the bitvectors are represented in a certain compressed form (Raman et al., 2007), then the total space is at most \( n \log \sigma + o(nh) \), where \( h \) is the wavelet tree height, independent of the way the alphabet is partitioned (Grossi et al., 2003).

---

4 This is achieved by using a constant-time solution (Clark, 1996) on their internal bitvector \( H \).
2.3 Document Listing

Let us now describe the optimal-time algorithm of Muthukrishnan (2002) for document listing. Muthukrishnan stores the suffix tree of $T$; a so-called document array $DA[1..n]$ of $T$, in which each cell $DA[i]$ stores the identifier of the document containing $T[SA[i]]$; an array $C[1..n]$, in which each cell $C[i]$ stores the largest value $h < i$ such that $DA[h] = DA[i]$, or 0 if there is no such value $h$; and a data structure supporting range-minimum queries (RMQs) over $C$, \( \text{rmq}_{C}(i,j) = \arg \min_{i \leq k \leq j} C[k] \). These data structures take a total of $O(n \lg n)$ bits. Given a pattern $P[1..m]$, the suffix tree is used to find the interval $SA[\ell..r]$ that contains the starting positions of the suffixes prefixed by $P$. It follows that every value $C[i] < \ell$ in $C[\ell..r]$ corresponds to a distinct document in $DA[i]$. Thus a recursive algorithm finding all those positions $i$ starts with $k = \text{rmq}_{C}(\ell, r)$. If $C[k] \geq \ell$ it stops. Otherwise it reports document $DA[k]$ and continues recursively with the ranges $C[\ell..k-1]$ and $C[k+1..r]$ (the condition $C[k] \geq \ell$ always uses the original $\ell$ value). In total, the algorithm uses $O(m + df)$ time, where $df$ is the number of documents returned.

Sadakane (2007) proposed a space-efficient version of this algorithm. The suffix tree is replaced with a CSA. The array $DA$ is replaced with a bitvector $B[1..n]$ such that $B[i] = 1$ iff $i$ is the first symbol of a document in $T$. Therefore $DA[i] = \text{rank}_1(B, SA[i])$ can be computed in constant time (Clark, 1996). The RMQ data structure is replaced with a variant (Fischer and Heun, 2011) that uses just $2n + o(n)$ bits and answers queries in constant time without accessing $C$. Finally, the tests $C[k] \geq \ell$ are replaced by marking the documents already reported in a bitvector $V[1..d]$ (initially all 0s), so that $V[DA[i]] = 1$ iff document $DA[i]$ has already been reported. If $V[DA[i]] = 1$ the recursion stops, otherwise it sets $V[DA[i]]$, reports $DA[i]$, and continues. This is correct as long as the RMQ structure returns the leftmost minimum in the range, and the range $[\ell..k-1]$ is processed before the range $[k+1..r]$ (Navarro, 2014). The total time is then $O(\text{search}(m) + df \cdot \text{lookup}(n))$.

3 Interleaved LCP

We introduce our first structure, the Interleaved LCP (ILCP). The main idea is to interleave the longest-common-prefix (LCP) arrays of the documents, in the order given by the global LCP of the collection. This yields long runs of equal values on repetitive collections, making the ILCP structure run-length compressible. Then, we show that a classical document listing technique (Muthukrishnan, 2002), designed to work on a completely different array, works almost verbatim over the ILCP array, and this yields a new document listing technique of independent interest for string collections. Finally, we show that a particular representation of the ILCP array allows us to count the number of documents where a string appears without having to list them one by one.
3.1 The ILCP Array

The longest-common-prefix array $\text{LCP}_S[i..|S|]$ of a string $S$ is defined such that $\text{LCP}_S[1] = 0$ and, for $2 \leq i \leq |S|$, $\text{LCP}_S[i]$ is the length of the longest common prefix of the lexicographically $(i-1)$th and $i$th suffixes of $S$, that is, of $S[\text{SA}_S[i-1]..|S|]$ and $S[\text{SA}_S[i]..|S|]$, where $\text{SA}_S$ is the suffix array of $S$. We define the interleaved LCP array of $T$, ILCP, to be the interleaving of the LCP arrays of the individual documents according to the document array.

**Definition 1** Let $T[1..n] = S_1 \cdot S_2 \cdot \ldots \cdot S_d$ be the concatenation of documents $S_j$, $\text{DA}$ the document array of $T$, and $\text{LCP}_{S_j}$ the longest-common-prefix array of string $S_j$. Then the *interleaved LCP array* of $T$ is defined, for all $1 \leq i \leq n$, as

$$\text{ILCP}[i] = \text{LCP}_{\text{SA}_{\text{DA}[i]}(\text{DA}, i)}.$$ 

**Example** Consider the documents $S_1 = "\text{TATĄS}"$, $S_2 = "\text{LATAŚ}"$, and $S_3 = "\text{AAAŚS}"$. Their concatenation is $T = "\text{TATĄSLATAŚAAAŚS}"$, its suffix array is $\text{SA} = \langle 15, 10, 5, 14, 9, 4, 13, 12, 11, 7, 2, 6, 8, 3, 1 \rangle$ and its document array is $\text{DA} = \langle 3, 2, 1, 3, 2, 1, 3, 3, 2, 1, 2, 2, 1, 1 \rangle$. The LCP arrays of the documents are $\text{LCP}_{S_1} = \langle 0, 0, 1, 0, 2 \rangle$, $\text{LCP}_{S_2} = \langle 0, 0, 1, 0, 0 \rangle$, and $\text{LCP}_{S_3} = \langle 0, 0, 1, 2, 3 \rangle$. Therefore, $\text{ILCP} = \langle 0, 0, 0, 0, 0, 1, 2, 3, 1, 1, 0, 0, 0, 2 \rangle$.

The following property of ILCP makes it suitable for document retrieval.

**Lemma 1** Let $T[1..n] = S_1 \cdot S_2 \cdot \ldots \cdot S_d$ be the concatenation of documents $S_j$, $\text{SA}$ its suffix array and $\text{DA}$ its document array. Let $\text{SA}[\ell..r]$ be the interval that contains the starting positions of suffixes prefixed by a pattern $P[1..m]$. Then the leftmost occurrences of the distinct document identifiers in $\text{DA}[\ell..r]$ are in the same positions as the values strictly less than $m$ in $\text{ILCP}[\ell..r]$.

**Proof** Let $\text{SA}_{S_j}[\ell_j..r_j]$ be the interval of all the suffixes of $S_j$ starting with $P[1..m]$. Then $\text{LCP}_{S_j}[\ell_j] < m$, as otherwise $S_j[\text{SA}_{S_j}[\ell_j-1]..\text{SA}_{S_j}[\ell_j-1]+m-1] = S_j[\text{SA}_{S_j}[\ell_j]..\text{SA}_{S_j}[\ell_j]+m-1] = P$ as well, contradicting the definition of $\ell_j$. For the same reason, it holds that $\text{LCP}_{S_j}[\ell_j+k] \geq m$ for all $1 \leq k \leq r_j - \ell_j$.

Now let $S_j$ start at position $p_j + 1$ in $T$, where $p_j = |S_1 \cdot S_{j-1}|$. Because each $S_j$ is terminated by "$\$", the lexicographic ordering between the suffixes $S_j[|..]$ in $\text{SA}_{S_j}$ is the same as that of the corresponding suffixes $T[p_j+|..]$ in $\text{SA}$. Hence $\langle \text{SA}[i] \mid \text{DA}[i] \rangle = j$, $1 \leq i \leq n = \langle p_j + \text{SA}_{S_j}[i] \mid 1 \leq i \leq |S_j| \rangle$. Or, put another way, $\langle \text{SA}[i] = p_j + \text{SA}_{S_j}[\text{rank}_{\ell}(\text{DA}, i)] \rangle$ whenever $\text{DA}[i] = j$.

Now let $f_j$ be the leftmost occurrence of $j$ in $\text{DA}[\ell..r]$. This means that $\text{SA}_{S_j}[f_j]$ is the lexicographically first suffix of $S_j$ that starts with $P$. By the definition of $f_j$, it holds that $\ell_j = \text{rank}_{\ell}(\text{DA}, f_j)$. Thus, by definition of ILCP, it holds that $\text{ILCP}[f_j] = \text{LCP}_{S_j}[\text{rank}_{\ell}(\text{DA}, f_j)] = \text{LCP}_{S_j}[\ell_j] < m$, whereas all the other $\text{ILCP}[k]$ values, for $\ell \leq k \leq r$, where $\text{DA}[k] = j$, must be $\geq m$. ⊓⊔
Example In the example above, if we search for \( P[1..2] = "TA" \), the resulting range is \( SA[13..15] = (8, 3, 1) \). The corresponding range \( DA[13..15] = (2, 1, 1) \) indicates that the occurrence at \( SA[13] \) is in \( S_2 \) and those in \( SA[14..15] \) are in \( S_1 \). According to the lemma, it is sufficient to report the documents \( DA[13] = 2 \) and \( DA[14] = 1 \), as those are the positions in \( ILCP[13..15] = (0, 0, 2) \) with values less than \( |P| = 2 \).

Therefore, for the purposes of document listing, we can replace the \( C \) array by \( ILCP \) in Muthukrishnan’s algorithm: instead of recursing until we have listed all the positions \( k \) such that \( C[k] < \epsilon \), we recurse until we list all the positions \( k \) such that \( ILCP[k] < m \). Instead of using it directly, however, we will design a variant that exploits repetitiveness in the string collection.

3.2 ILCP on Repetitive Collections

The array \( ILCP \) has yet another property, which also makes it attractive for repetitive collections: it contains long runs of equal values. We give an analytic proof of this fact under a model where a base document \( S \) is generated at random under the very general A2 probabilistic model of Szpankowski [1993] (which includes Bernoulli and Markov chains of fixed memory), and the collection is formed by performing some edits on \( d \) copies of \( S \).

**Lemma 2** Let \( S[1..r] \) be a string generated under Szpankowski’s A2 model. Let \( T \) be formed by concatenating \( d \) copies of \( S \), each terminated with the special symbol “$”, and then carrying out \( s \) edits (symbol insertions, deletions, or substitutions) at arbitrary positions in \( T \) (excluding the “$’s”). Then, almost surely (a.s.\(^5\)), the ILCP array of \( T \) is formed by \( \rho \leq r + \mathcal{O}(s \log(r + s)) \) runs of equal values.

**Proof** Before applying the edit operations, we have \( T = S_1 \cdots S_d \) and \( S_j = SS \) for all \( j \). At this point, ILCP is formed by at most \( r + 1 \) runs of equal values, since the \( d \) equal suffixes \( S_j[S_1][i..r + 1] \) must be contiguous in the suffix array \( SA \) of \( T \), in the area \( SA[(i - 1)d + 1..id] \). Since the values \( l = \text{LCP}_{S_j}[i] \) are also equal, and ILCP values are the \( \text{LCP}_{S_j} \) values listed in the order of \( SA \), it follows that \( ILCP[(i - 1)d + 1..id] = l \) forms a run, and thus there are \( r + 1 = n/d \) runs in ILCP. Now, if we carry out \( s \) edit operations on \( T \), any \( S_j \) will be of length at most \( r + s + 1 \). Consider an arbitrary edit operation at \( T[k] \). It changes all the suffixes \( T[k-h..n] \) for all \( 0 \leq h < k \). However, since a.s. the string depth of a leaf in the suffix tree of \( S \) is \( \mathcal{O}(\log(r + s)) \) (Szpankowski, 1993), the suffix will possibly be moved in \( SA \) only for \( h = \mathcal{O}(\log(r + s)) \). Thus, a.s., only \( \mathcal{O}(\log(r + s)) \) suffixes are moved in \( SA \), and possibly the corresponding runs in ILCP are broken. Hence \( \rho \leq r + \mathcal{O}(s \log(r + s)) \) a.s. □

\(^5\) This is a very strong kind of convergence. A sequence \( X_n \) tends to a value \( \beta \) almost surely if, for every \( \epsilon > 0 \), the probability that \( |X_N/\beta - 1| > \epsilon \) for some \( N > n \) tends to zero as \( n \) tends to infinity, \( \lim_{n \to \infty} \sup_{N > n} \Pr(|X_N/\beta - 1| > \epsilon) = 0 \).
Therefore, the number of runs depends linearly on the size of the base
document and the number of edits, not on the total collection size. The proof
generalizes the arguments of Mäkinen et al (2010), which hold for uniformly
distributed strings $S$. There is also experimental evidence (Mäkinen et al 2010)
that, in real-life text collections, a small change to a string usually causes only
a small change to its LCP array. Next we design a document listing data
structure whose size is bounded in terms of $\rho$.

### 3.3 Document Listing

Let $\text{LILCP}[1..\rho]$ be the array containing the partial sums of the lengths of
the $\rho$ runs in ILCP, and let $\text{VILCP}[1..\rho]$ be the array containing the values
in those runs. We can store $\text{LILCP}$ as a bitvector $L[1..n]$ with $\rho$ 1s, so that
$L[i] = \text{select}(L, i)$. Then $L$ can be stored using the structure of Okanohara
and Sadakane (2007) that requires $\rho \lg(n/\rho) + O(\rho)$ bits.

With this representation, it holds that $\text{ILCP}[i] = \text{VILCP}[\text{rank}(L, i)]$. We can
map from any position $i$ to its run $i' = \text{rank}(L, i)$ in time $O(\lg(n/\rho))$, and from
any run $i'$ to its starting position in $\text{ILCP}$, $i = \text{select}(L, i')$, in constant time.

**Example.** Consider the array $\text{ILCP}[1..15] = (0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 1, 1, 0, 0, 0, 2)$
of our running example. It has $\rho = 7$ runs, so we represent it with $\text{VILCP}[1..7] =
(0, 1, 2, 3, 1, 0, 2)$ and $L[1..15] = 100000111101001$.

This is sufficient to emulate the document listing algorithm of Sadakane
(2007) on a repetitive collection. We will use RLCSA as the CSA. The sparse
bitvector $B[1..n]$ marking the document beginnings in $T$ will be represented
just like $L$, so that it requires $d \lg(n/d) + O(d)$ bits and lets us compute
any value $\text{DA}[i] = \text{rank}(B, \text{SA}[i])$ in time $O(\text{lookup}(n))$. Finally, we build the
compact RMQ data structure (Fischer and Heun 2011) on $\text{VILCP}$, requiring
$2\rho + o(\rho)$ bits. We note that this RMQ structure does not need access to $\text{VILCP}$
to answer queries.

Assume that we have already found the range $\text{SA}[\ell..r]$ in $O(\text{search}(m))$
time. We compute $\ell' = \text{rank}(L, \ell)$ and $r' = \text{rank}(L, r)$, which are the endpoints
of the interval $\text{VILCP}[\ell'..r']$ containing the values in the runs in $\text{ILCP}[\ell..r]$. Now
we run Sadakane’s algorithm on $\text{VILCP}[\ell'..r']$. Each time we find a minimum
at $\text{VILCP}[i']$, we remap it to the run $\text{ILCP}[i..j]$, where $i = \max(\ell, \text{select}(L, i'))$
and $j = \min(r, \text{select}(L, i' + 1) - 1)$. For each $i \leq k \leq j$, we compute $\text{DA}[k]$
using $B$ and RLCSA as explained, mark it in $\text{V}[\text{DA}[k]] \leftarrow 1$, and report it. If,
however, it already holds that $\text{V}[\text{DA}[k]] = 1$, we stop the recursion. Figure 1

gives the pseudocode.

We show next that this is correct as long as RMQ returns the leftmost
minimum in the range and that we recurse first to the left and then to the
right of each minimum $\text{VILCP}[i']$ found.

**Lemma 3** Using the procedure described, we correctly find all the positions
$\ell \leq k \leq r$ such that $\text{ILCP}[k] < m$. 

function listDocuments(ℓ, r)
    \((\ell', r') \leftarrow \text{rank}(L, \ell, \text{rank}(L, r))\)
    return list(\(\ell', r'\))

function list(\(\ell', r'\))
    if \(\ell' > r'\): return \(\emptyset\)
    \(i' \leftarrow \text{rmq}_{\text{ILCP}}(\ell', r')\)
    \(i \leftarrow \max(\ell, \text{select}(L, i'))\)
    \(j \leftarrow \min(r, \text{select}(L, i' + 1) - 1)\)
    \(\text{res} \leftarrow \emptyset\)
    for \(k \leftarrow i \ldots j\):
        \(g \leftarrow \text{rank}(B, SA[k])\)
        if \(V[g] = 1\): return \(\text{res}\)
    \(V[g] \leftarrow 1\)
    \(\text{res} \leftarrow \text{res} \cup \{g\}\)
    return \(\text{res} \cup \text{list}(\ell', i' - 1) \cup \text{list}(i' + 1, r')\)

Fig. 1 Pseudocode for document listing using the ILCP array. Function listDocuments(\(\ell, r\)) lists the documents from interval \(SA[\ell..r]\); list(\(\ell', r'\)) returns the distinct documents mentioned in the runs \(\ell'\) to \(r'\) that also belong to \(DA[\ell..r]\). We assume that in the beginning it holds \(V[k] = 0\) for all \(k\); this can be arranged by resetting to 0 the same positions after the query or by using initializable arrays. All the unions on \(\text{res}\) are known to be disjoint.

Proof Let \(j = DA[k]\) be the leftmost occurrence of document \(j\) in \(DA[\ell..r]\). By Lemma 1, among all the positions where \(DA[k'] = j\) in \(DA[\ell..r]\), \(k\) is the only one where ILCP\([k]\) < \(m\). Since we find a minimum ILCP value in the range, and then explore the left subrange before the right subrange, it is not possible to find first another occurrence \(DA[k'] = j\), since it has a larger ILCP value and is to the right of \(k\). Therefore, when \(V[DA[k]] = 0\), that is, the first time we find a \(DA[k] = j\), it must hold that ILCP\([k]\) < \(m\), and the same is true for all the other ILCP values in the run. Hence it is correct to list all those documents and mark them in \(V\). Conversely, whenever we find a \(V[DA[k']] = 1\), the document has already been reported. Thus this is not its leftmost occurrence and then ILCP\([k]\) ≥ \(m\) holds, as well as for the whole run. Hence it is correct to avoid reporting the whole run and to stop the recursion in the range, as the minimum value is already at least \(m\).

Note that we are not storing \(VILCP\) at all. We have obtained our first result for document listing, where we recall that \(\rho\) is small on repetitive collections (Lemma 2):

Theorem 1 Let \(T = S_1 \cdot S_2 \cdot \ldots \cdot S_d\) be the concatenation of \(d\) documents \(S_j\), and CSA be a compressed suffix array on \(T\), searching for any pattern \(P[1..m]\) in time search\((m)\) and accessing \(SA[i]\) in time lookup\((n) = \Omega(\lg n)\). Let \(\rho\) be the number of runs in the ILCP array of \(T\). We can store \(T\) in \(|CSA| + \rho \cdot \log(n/\rho) + O(\rho) + d \cdot \lg(n/d) + O(d)\) bits such that document listing takes \(O(\text{search}(m) + df \cdot \text{lookup}(n))\) time.
3.4 Document Counting

Array ILCP also allows us to efficiently count the number of distinct documents where \( P \) appears, without listing them all. This time we will explicitly represent VILCP, in the following convenient way: consider a skewed wavelet tree, where the leftmost leaf is at depth 1, the next 2 leaves are at depth 3, the next 4 leaves are at depth 5, and in general the \( 2^{d-1} \)th to \( 2^d - 1 \)th leftmost leaves are at depth \( 2d - 1 \). Then the \( i \)th leftmost leaf is at depth \( 1 + 2 \lfloor \log i \rfloor = O(\log i) \). The number of wavelet tree nodes up to depth \( d \) is \( \sum_{i=1}^{2^d-1} \frac{1}{2} = 2(2^{d+1}/2 - 1) \). The number of nodes up to the depth of the \( m \)th leftmost leaf is maximized when \( m \) is of the form \( 2^d - 1 \), reaching \( 2(2^d - 1) = 4m - 2 = O(m) \). See Figure 2.

Let \( \lambda \) be the maximum value in the ILCP array. Then the height of the wavelet tree is \( O(\log \lambda) \) and the representation of VILCP takes at most \( \rho \log \lambda + o(\rho \log \lambda) \) bits. If the documents \( S \) are generated using the A2 probabilistic model of Szpankowski (1993), then \( \lambda = O(\log |S|) = O(\log n) \), and VILCP uses \( \rho \log \log n(1 + o(1)) \) bits. The same happens under the model used in Section 3.2.

The number of documents where \( P \) appears, \( df \), is the number of times a value smaller than \( m \) occurs in ILCP[\( \ell..r \)]. An algorithm to find all those values in a wavelet tree of ILCP is as follows (Gagie et al., 2012b). Start at the root with the range [\( \ell..r \)] and its bitvector \( W \). Go to the left child with the interval \([\text{rank}_0(W, \ell - 1) + 1..\text{rank}_0(W, r)] \) and to the right child with the interval \([\text{rank}_1(W, \ell - 1) + 1..\text{rank}_1(W, r)] \), stopping the recursion on empty intervals. This method arrives at all the wavelet tree leaves corresponding to the distinct values in ILCP[\( \ell..r \)]. Moreover, when it arrives at a leaf \( l \) with interval [\( \ell_l..r_l \)], then there are \( r_l - \ell_l + 1 \) occurrences of the symbol of that leaf in ILCP[\( \ell..r \)]. Now, in the skewed wavelet tree of VILCP, we are interested in the occurrences of symbols 0 to \( m - 1 \). Thus we apply the above algorithm but we do not enter into subtrees handling an interval of values that is disjoint with \([0..m - 1] \). Therefore, we only arrive at the \( m \) leftmost leaves of the wavelet tree, and thus traverse only \( O(m) \) wavelet tree nodes, in time \( O(m) \).

A complication is that VILCP is the array of run length heads, so when we start at VILCP[\( \ell'..r' \)] and arrive at each leaf \( l \) with interval [\( \ell_l'..r_l' \)], we only know that VILCP[\( \ell'..r' \)] contains from the \( \ell_l' \)th to the \( r_l' \)th occurrences of value \( l \) in VILCP[\( \ell'..r' \)]. We store a reordering of the run lengths so that the runs...
function countDocuments(ℓ, r)
(ℓ', r') ← (rank(L, ℓ), rank(L, r))
l ← m
c ← count(root, ℓ', r')
if VILCP[ℓ'] < m: c ← c - (ℓ - select(L, ℓ'))
if VILCP[r'] < m: c ← c - (select(L, r' + 1) - 1 - r)
return c

function count(v, ℓ', r')
if l = 0: return 0
if ℓ' > r': return 0
if v is a leaf:
l ← l - 1
return select(L', r' + 1) - select(L', ℓ')
(ℓ1, r1) ← (rank(v.W, ℓ' - 1) + 1, rank(v.W, r'))
return count(v.left, ℓ' - ℓ1 + 1, r' - r1) + count(v.right, ℓ1, r1)

Fig. 3 Document counting with the ILCP array. Function countDocuments(ℓ, r) counts the distinct documents from interval SA[ℓ..r]; count(v, ℓ', r') returns the number of documents mentioned in the runs ℓ' to r' under wavelet tree node v that also belong to DA[ℓ..r]. We assume that the wavelet tree root node is root, and that any internal wavelet tree node v has fields v.W (bitvector), v.left (left child), and v.right (right child). Global variable l is used to traverse the first m leaves. The access to VILCP is also done with the wavelet tree.

corresponding to each value l are collected left to right in ILCP and stored aligned to the wavelet tree leaf l. Those are concatenated into another bitmap L'[1..n] with ρ 1s, similar to L, which allows us, using select(L', ·), to count the total length spanned by the ℓ' th to r' th runs in leaf l. By adding the areas spanned over the m leaves, we count the total number of documents where P occurs. Note that we need to correct the lengths of runs ℓ' and r', as they may overlap the original interval ILCP[ℓ..r]. Figure 3 gives the pseudocode.

Theorem 2 Let T = S1 · S2 · · · Sd be the concatenation of d documents Sj, and CSA a compressed suffix array on T that searches for any pattern P[1..m] in time search(m) = \(\Omega(m)\). Let ρ be the number of runs in the ILCP array of T and λ be the maximum length of a repeated substring inside any Sj. Then we can store T in \(|CSA| + \rho \lg \lambda + 2 \lg (n/\rho) + O(1)) bits such that the number of documents where a pattern P[1..m] occurs can be computed in time \(O(\text{search}(m))\).

4 Precomputed Document Lists

In this section we introduce the idea of precomputing the answers of document retrieval queries for a sample of suffix tree nodes, and then exploit repetitiveness by grammar-compressing the resulting sets of answers. Such grammar compression is effective when the underlying collection is repetitive. The queries are then extremely fast on the sampled nodes, whereas on the others we have a way to bound the amount of work performed. The resulting structure is called PDL (Precomputed Document Lists), for which we develop a variant for document listing and another for top-k retrieval queries.
4.1 Document Listing

Let \( v \) be a suffix tree node. We write \( \text{SA}_v \) to denote the interval of the suffix array covered by node \( v \), and \( D_v \) to denote the set of distinct document identifiers occurring in the same interval of the document array. Given block size \( b \) and a constant \( \beta \geq 1 \), we build a sparse suffix tree that allows us to answer document listing queries efficiently. For any suffix tree node \( v \), it holds that:

1. \( v \) is in the sparse suffix tree and thus \( D_v \) is directly stored; or
2. \( |\text{SA}_v| < b \), and thus documents can be listed in time \( \mathcal{O}(b \cdot \text{lookup}(n)) \) by using a CSA and bitvector \( B \); or
3. we can compute the set \( D_v \) as the union of stored sets \( D_{u_1}, \ldots, D_{u_k} \) of total size at most \( \beta \cdot |D_v| \), where nodes \( u_1, \ldots, u_k \) are the children of \( v \) in the sparse suffix tree.

We start by selecting suffix tree nodes \( v_1, \ldots, v_L \), so that no selected node is an ancestor of another, and the intervals \( \text{SA}_{v_i} \) of the selected nodes cover the entire suffix array. Given node \( v \) and its parent \( w \), we select \( v \) if \( |\text{SA}_w| \leq b \) and \( |\text{SA}_w| > b \), and store \( D_v \) with the node. These nodes \( v \) become the leaves of the sparse suffix tree, and we assume that they are numbered from left to right. Next we proceed upward in the suffix tree. Let \( v \) be an internal node, \( u_1, \ldots, u_k \) its children, and \( w \) its parent. If the total size of sets \( D_{u_1}, \ldots, D_{u_k} \) is at most \( \beta \cdot |D_v| \), we remove node \( v \) from the tree, and add nodes \( u_1, \ldots, u_k \) to the children of node \( w \). Otherwise we keep node \( v \) in the sparse suffix tree, and store \( D_v \) there.

When the document collection is repetitive, the document array \( \text{DA}[1..n] \) is also repetitive. This property has been used in the past to compress it using grammars \([\text{Navarro et al.} 2014]\). We can apply a similar idea on the \( D_v \) sets stored at the sampled suffix tree nodes, since \( D_v \) is a function of the range \( \text{DA}[\ell..r] \) that corresponds to node \( v \).

Let \( v_1, \ldots, v_L \) be the leaf nodes and \( v_{L+1}, \ldots, v_{L+T} \) the internal nodes of the sparse suffix tree. We use grammar-based compression to replace frequent subsets in sets \( D_{v_1}, \ldots, D_{v_{L+T}} \) with grammar rules expanding to those subsets. Given a set \( Z \) and a grammar rule \( X \rightarrow Y \), where \( Y \subseteq \{1, \ldots, d\} \), we can replace \( Z \) with \( (Z \cup \{X\}) \setminus Y \), if \( Y \subseteq Z \). As long as \( |Y| \geq 2 \) for all grammar rules \( X \rightarrow Y \), each set \( D_{v_i} \) can be decompressed in \( \mathcal{O}(|D_{v_i}|) \) time.

To choose the replacements, consider the bipartite graph with vertex sets \( \{v_1, \ldots, v_{L+T}\} \) and \( \{1, \ldots, d\} \), with an edge from \( v_i \) to \( j \) if \( j \in D_{v_i} \). Let \( X \rightarrow Y \) be a grammar rule, and let \( V \) be the set of nodes \( v_i \) such that rule \( X \rightarrow Y \) can be applied to set \( D_{v_i} \). As \( Y \subseteq D_{v_i} \) for all \( v_i \in V \), the induced subgraph with vertex sets \( V \) and \( Y \) is a complete bipartite graph or a \emph{biclique}. Many Web graph compression algorithms are based on finding bicliques or other dense subgraphs \([\text{Hernández and Navarro} 2014]\), and we can use these algorithms to find a good grammar compressing the precomputed document lists.

When all rules have been applied, we store the reduced sets \( D_{v_1}, \ldots, D_{v_{L+T}} \) as an array \( A \) of document and rule identifiers. The array takes \( |A| \lg (d + n_R) \) bits of space, where \( n_R \) is the total number of rules. We mark the first cell
function listDocuments(ℓ, r)
(res, ln) ← (∅, rank(B_L, ℓ))

if select(B_L, ln) < ℓ:
    ℓ′ ← min(select(B_L, ln + 1) − 1, r)
    (res, ln) ← (list(ℓ, ℓ′), ln + 1)
if ℓ′ = r:
    return res
rn ← rank(B_L, r + 1) − 1
if select(B_L, ln + 1) ≤ r:
    ℓ′ ← select(B_L, ln + 1)
    res ← res ∪ list(ℓ′, r)
return res ∪ decompress(ln, rn)

function decompress(ℓ, r)
(res, i) ← (∅, ℓ)
while ℓ ≤ r:
    next ← i + 1
    while B_R[i] = 1:
        (i′, next) ← parent(i)
        if next > r + 1:
            break
        (i, next) ← (i′, next')
        res ← res ∪ list(i)
    i ← next
return res

Fig. 4 Document listing using precomputed answers. Function listDocuments(ℓ, r) lists the documents from interval SA[ℓ..r]; decompress(ℓ, r) decompresses the sets stored in nodes v_1, ..., v_N; parent(i) returns the parent node and the leaf node following it for a first child v_i; set(i) decompresses the set stored in v_i; rule(i) expands the i-th grammar rule; and list(ℓ, r) lists the documents from interval SA[ℓ..r] by using CSA and bitvector B.

in the encoding of each set with a 1 in a bitvector B_A[1..|A|], so that set D_{v_i} can be retrieved by decompressing A[select(B_A, i)..select(B_A, i + 1) − 1]. The bitvector takes |A|(1 + o(1)) bits of space and answers select queries in O(1) time (Okanohara and Sadakane 2007). The grammar rules are stored similarly, in an array G taking |G| lg d bits, with a bitvector B_G[1..|G|] of |G|(1 + o(1)) bits separating the array into rules (note that right hand sides of rules are formed only by terminals).

In addition to the sets and the grammar, we must also store the sparse suffix tree. A bitvector B_L[1..n] marks the first cell of interval SA_{v_i} for all leaf nodes v_i, allowing us to convert interval SA[ℓ..r] into a range of nodes [ln..rn] = [rank(B_L, ℓ), rank(B_L, r + 1) − 1]. Using again the format of Okanohara and Sadakane (2007) for B_L, it uses ⌈lg(n/L)⌉ + O(L) bits, and answers rank queries in O(lg(n/L)) time and select queries in constant time. A second bitvector B_R[1..L + 1], using (L + 1)(1 + o(1)) bits and supporting rank queries in constant time (Clark 1996), marks the nodes that are the first children of their parents. An array F[1..I] of I lg I bits stores pointers from first children to their parent nodes, so that if node v_i is a first child, its parent node is v_{j}, where j = L + F[rank(B_R, i)]. Finally, array N[1..I] of I lg L bits stores a pointer to the leaf node following those below each internal node.

Figure 4 gives the pseudocode for document listing using the precomputed answers. Function list(ℓ, r) takes O((r + 1 − ℓ) lookup(n)) time, set(i) takes
\(O(|D_v|)\) time, and parent\((i)\) takes \(O(1)\) time. Function decompress\((\ell, r)\) produces set \(res\) in time \(O(|res| \cdot \beta h)\), where \(h\) is the height of the sparse suffix tree: at each level, we may find the same document up to \(\beta\) times to produce the set \(D_v\) of its parent, which has been omitted in the sparse suffix tree. Hence the total time for listDocuments\((\ell, r)\) is \(O(df \cdot \beta h + \log n)\) for unions of precomputed answers, and \(O(b \cdot \text{lookup}(n))\) otherwise. We do not write the result as a theorem because we cannot upper bound the space used by the structure in terms of \(b\) and \(\beta\).

### 4.2 Top-\(k\) Retrieval

Since we have the freedom to represent the documents in sets \(D_v\) in any order, we can in particular sort the document identifiers by their frequencies in decreasing order, with ties broken by document identifiers in increasing order. Then a top-\(k\) query on a node \(v\) that stores its list \(D_v\) boils down to listing the first \(k\) elements of \(D_v\).

This time we cannot use the set-based grammar compressor, but we rather need a compressor that preserves the order. We use Re-Pair [Larsson and Moffat 2000], which produces a grammar where each nonterminal produces two new symbols, terminal or nonterminal. As Re-Pair decompression is recursive, decompression can be slower than in document listing, although it is still fast in practice and takes linear time in the length of the decompressed sequence.

In order to merge the results from multiple nodes in the sampled suffix tree, we need to store the frequency of each document. These are stored in the same order as the identifiers. Since the frequencies are nonincreasing, with potentially long runs of small values, we can represent them space-efficiently by run-length encoding the sequences and using differential encoding for the run heads. A node containing \(s\) suffixes in its subtree has at most \(O(\sqrt{s})\) distinct frequencies, and the frequencies can be encoded in \(O(\sqrt{s} \log s)\) bits.

There are two basic approaches to using the PDL structure for top-\(k\) document retrieval. We can store the document lists for all suffix tree nodes above the leaf blocks, producing a structure that is essentially an inverted index for all frequent substrings. This approach is very fast, as we need only decompress the first \(k\) document identifiers from the stored sequence. It works well with repetitive collections thanks to the grammar-compression of the lists.

Alternatively, we can build the PDL structure normally with some parameter \(\beta\), achieving better space usage. Answering queries is now slower, as we have to decompress up to \(\beta\) document sets, merge the sets, and determine the top \(k\) documents. We tried different heuristics for merging prefixes of the document sequences, stopping when a correct answer to the top-\(k\) query could be guaranteed. The heuristics did not generally work well, making brute-force merging the fastest alternative.
5 Engineering a Document Counting Structure

In this section we revisit a generic document counting structure by Sadakane (2007), which uses $2^n$ bits and answers counting queries in constant time. We show that the structure inherits the repetitiveness present in the text collection, which can then be exploited to reduce its space occupancy. Surprisingly, the structure also becomes repetitive with random and near-random data, such as DNA sequences, which is a result of interest for general string collections. We show how to take advantage of this redundancy in a number of different ways, leading to different time/space trade-offs.

5.1 The Basic Bitvector

We describe the original document structure of Sadakane (2007), which computes $df$ in constant time given the locus of the pattern $P$, while using just $2n + o(n)$ bits of space.

We start with the suffix tree of the text, and add new internal nodes to it to make it a binary tree. For each internal node $v$ of the binary suffix tree, let $list(v)$ be the set of distinct document identifiers in the corresponding range $DA[\ell..r]$, and let $count(v)$ be the size of that set. If node $v$ has children $u$ and $w$, we define the number of redundant suffixes as

$$h(v) = |list(u) \cap list(w)|.$$

This allows us to compute $df$ recursively:

$$count(v) = count(u) + count(w) - h(v).$$

By using the leaf nodes descending from $v$, $[\ell..r]$, as base cases, we can solve the recurrence:

$$count(\ell, r) = (r + 1 - \ell) - \sum_u h(u),$$

where the summation goes over the internal nodes of the subtree rooted at $v$.

We form an array $H[1..n-1]$ by traversing the internal nodes in inorder and listing the $h(v)$ values. As the nodes are listed in inorder, subtrees form contiguous ranges in the array. We can therefore rewrite the solution as

$$count(\ell, r) = (r + 1 - \ell) - \sum_{i=\ell}^{r-1} H[i].$$

To speed up the computation, we encode the array in unary as bitvector $H'$. Each cell $H'[i]$ is encoded as a 1-bit, followed by $H'[i]$ 0s. We can now compute the sum by counting the number of 0s between the 1s of ranks $\ell$ and $r$:

$$count(\ell, r) = 2(r - \ell) - (select_1(H', r) - select_1(H', \ell)) + 1.$$

As there are $n-1$ 1s and $n-d$ 0s, bitvector $H'$ takes at most $2n + o(n)$ bits.
5.2 Compressing the Bitvector

The original bitvector requires $2n + o(n)$ bits, regardless of the underlying data. This can be a considerable overhead with highly compressible collections, taking significantly more space than the CSA (on top of which the structure operates). Fortunately, as we now show, the bitvector $H'$ used in Sadakane’s method is highly compressible. There are five main ways of compressing the bitvector, with different combinations of them working better with different datasets.

1. Let $V_v$ be the set of nodes of the binary suffix tree corresponding to node $v$ of the original suffix tree. As we only need to compute $\text{count}(\cdot)$ for the nodes of the original suffix tree, the individual values of $h(u), u \in V_v$, do not matter, as long as the sum $\sum_{u \in V_v} h(u)$ remains the same. We can therefore make bitvector $H'$ more compressible by setting $H'[i] = \sum_{u \in V_v} h(u)$, where $i$ is the inorder rank of node $v$, and $H[j] = 0$ for the rest of the nodes. As there are no real drawbacks in this reordering, we will use it with all of our variants of Sadakane’s method.

2. Run-length encoding works well with versioned collections and collections of random documents. When a pattern occurs in many documents, but no more than once in each, the corresponding subtree will be encoded as a run of 1s in $H'$.

3. When the documents in the collection have a versioned structure, we can reasonably expect grammar compression to be effective. To see this, consider a substring $x$ that occurs in many documents, but at most once in each document. If each occurrence of substring $x$ is preceded by character $a$, the subtrees of the binary suffix tree corresponding to patterns $x$ and $ax$ have identical structure, and the corresponding areas in $D$ are identical. Hence the subtrees are encoded identically in bitvector $H'$.

4. If the documents are internally repetitive but unrelated to each other, the suffix tree has many subtrees with suffixes from just one document. We can prune these subtrees into leaves in the binary suffix tree, using a filter bitvector $F[1..n-1]$ to mark the remaining nodes. Let $v$ be a node of the binary suffix tree with inorder rank $i$. We will set $F[i] = 1$ iff $\text{count}(v) > 1$. Given a range $[\ell..r-1]$ of nodes in the binary suffix tree, the corresponding subtree of the pruned tree is $[\text{rank}_1(F,\ell)..<\text{rank}_1(F,r-1)]$. The filtered structure consists of bitvector $H'$ for the pruned tree and a compressed encoding of $F$.

5. We can also use filters based on the values in array $H$ instead of the sizes of the document sets. If $H[i] = 0$ for the most cells, we can use a sparse filter $F_S[1..n-1]$, where $F_S[i] = 1$ iff $H[i] > 0$, and build bitvector $H'$ only for those nodes. We can also encode positions with $H[i] = 1$ separately with a 1-filter $F_1[1..n-1]$, where $F_1[i] = 1$ iff $H[i] = 1$. With a 1-filter, we do not write 0s in $H'$ for nodes with $H[i] = 1$, but subtract the number of 1s in $F_1[\ell..r-1]$ from the result of the query instead. It is also possible to use a
sparse filter and a 1-filter simultaneously. In that case, we set \( F_S[i] = 1 \) iff \( H[i] > 1 \).

5.3 Analysis

We analyze the number of runs of 1s in bitvector \( H' \) in the expected case. Assume that our document collection consists of \( d \) documents, each of length \( m \), over an alphabet of size \( \sigma \). We call string \( S \) unique, if it occurs at most once in every document. The subtree of the sparse suffix tree corresponding to a unique string is encoded as a run of 1s in bitvector \( H' \). If we can cover all leaves of the tree with \( u \) unique substrings, bitvector \( H' \) has at most \( 2u \) runs of 1s.

Consider a random string of length \( k \). Suppose the probability that the string occurs at least twice in any given document is at most \( m^2/(2\sigma^{2k}) \), which is the case if, e.g., we choose each document randomly or we choose one document randomly and generate the others by copying it and randomly substituting some characters. By the union bound, the probability the string is non-unique is at most \( dm^2/(2\sigma^{2k}) \). Let \( N(i) \) be the number of non-unique strings of length \( k_i = \log_\sigma (m\sqrt{d}) + i \). As there are \( \sigma^{k_i} \) strings of length \( k_i \), the expected value of \( N(i) \) is at most \( m\sqrt{d}/(2\sigma^i) \). The expected size of the smallest cover of unique strings is therefore at most

\[
(\sigma^{k_0} - N(0)) + \sum_{i=1}^{\infty} (\sigma N(i-1) - N(i)) = m\sqrt{d} + (\sigma - 1) \sum_{i=0}^{\infty} N(i) \leq \left( \frac{\sigma}{2} + 1 \right) m\sqrt{d},
\]

where \( \sigma N(i-1) - N(i) \) is the number of strings that become unique at length \( k_i \). The number of runs of 1s in \( H' \) is therefore sublinear in the size of the collection \((dm)\). See Figure 5 for an experimental confirmation of this analysis.

6 Experiments and Discussion

6.1 Experimental Setup

6.1.1 Document Collections

We performed extensive experiments with both real and synthetic collections. Most of our document collections were relatively small, around 100 MB in size, as some of the implementations (Navarro et al, 2014) use 32-bit libraries. We also used larger versions of some collections, up to 1 GB in size, to see how the collection size affects the results. In general, collection size is more important in top-k document retrieval. Increasing the number of documents generally increases the \( df/k \) ratio, and thus makes brute-force solutions based on document listing less appealing. In document listing, the size of the documents is

\footnote{See \url{http://jltsiren.kapsi.fi/rlcsa} for the datasets and full results.}
Fig. 5 The number of runs of 1-bits in Sadakane’s bitvector $H'$ on synthetic collections of DNA sequences ($\sigma = 4$). Each collection has been generated by taking a random sequence of length $m = 2^7$ to $2^{17}$, duplicating it $d = 2^{17}$ to $2^7$ times (making the total size of the collection $2^{24}$), and mutating the sequences with random point mutations at probability $p = 0.001$ to 1. The dashed line represents the expected case upper bound for $p = 1$.

more important than collection size, as large $occ/df$ ratio makes brute-force solutions based on pattern matching less appealing.

The performance of various solutions depends both on the repetitiveness of the collection and the type of the repetitiveness. Hence we used a large number of real and synthetic collections with different characteristics for our experiments. We describe them next, and summarize their statistics in Table 1.

Real collections. We use various document collections from real-life repetitive scenarios. Some collections come in small, medium, and large variants. Page and Revision are repetitive collections generated from a Finnish-language Wikipedia archive with full version history. There are 60 (small), 190 (medium), or 280 (large) pages with a total of 8,834, 31,208, or 65,565 revisions. In Page, all the revisions of a page form a single document, while each revision becomes a separate document in Revision. Enwiki is a non-repetitive collection of 7,000, 44,000, or 90,000 pages from a snapshot of the English-language Wikipedia. Influenza is a repetitive collection containing 100,000 or 227,356 sequences from influenza virus genomes (we only have small and large variants). Swissprot is a non-repetitive collection of 143,244 protein sequences used in many document retrieval papers (e.g., [Navarro et al (2014)]). As the full collection is only 54 MB, only the small version of Swissprot exists. Wiki is a repetitive collection similar to Revision. It is generated by sampling all revisions of 1% of pages from the English-language versions of Wikibooks, Wikinews, Wikiquote, and Wikivoyage.

Synthetic collections. To explore the effect of collection repetitiveness on document retrieval performance in more detail, we generated three types of synthetic collections, using files from the Pizza & Chilli corpusDNA is similar

7 http://pizzachilli.dcc.uchile.cl
Table 1  Statistics for document collections (small, medium, and large variants). Collection size, RLCSA size without suffix array samples, number of documents, average document length, number of patterns, average number of occurrences and document occurrences, and the ratio of occurrences to document occurrences. For the synthetic collections (second group), most of the statistics vary greatly.

| Collection | Size  | CSA size (RLCSA) | Documents | Avg. doc size (n/d) | Patterns | Occurrences (occ) | Document occs (df) | Occs per doc (occ/df) |
|------------|-------|------------------|-----------|---------------------|----------|------------------|-----------------|----------------------|
| Page       | 110 MB| 2.58 MB          | 60        | 1,919,382           | 7,658    | 781              | 3               | 242.75               |
|            | 641 MB| 9.00 MB          | 190       | 3,534,921           | 14,286   | 2,601            | 6               | 444.79               |
|            | 1,037 MB| 17.45 MB      | 280       | 3,883,145           | 20,536   | 2,889            | 7               | 429.04               |
| Revision   | 110 MB| 2.59 MB          | 8,834     | 13,005              | 7,658    | 776              | 371             | 2.09                 |
|            | 640 MB| 9.04 MB          | 31,208    | 21,490              | 14,284   | 2,592            | 1,065           | 2.43                 |
|            | 1,035 MB| 17.55 MB       | 65,565    | 16,552              | 20,536   | 2,876            | 1,188           | 2.42                 |
| Enwiki     | 113 MB| 49.44 MB         | 7,000     | 16,932              | 18,935   | 1,904            | 505             | 3.77                 |
|            | 639 MB| 309.31 MB        | 44,000    | 15,236              | 19,628   | 10,316           | 2,856           | 3.61                 |
|            | 1,034 MB| 482.16 MB      | 90,000    | 12,050              | 19,805   | 17,092           | 4,976           | 3.44                 |
| Influenza  | 137 MB| 5.52 MB          | 100,000   | 1,436               | 1,000    | 24,975           | 18,547          | 1.35                 |
|            | 321 MB| 10.53 MB         | 227,356   | 1,480               | 1,000    | 59,997           | 44,012          | 1.36                 |
| Swissprot  | 54 MB | 25.19 MB         | 143,244   | 398                 | 10,000   | 160              | 121             | 1.33                 |
| Wiki       | 14.32 MB| 42.90 MB        | 103,190   | 14,540              |          |                  |                 |                      |
| DNA        | 95 MB |                  | 100,000   |                     |          |                  |                 |                      |
| Concat     | 95 MB |                  | 10–1,000  |                     |          |                  |                 |                      |
| Version    | 95 MB |                  | 10,000    |                     |          |                  |                 |                      |
to Influenza. Each collection has $d = 1, 10, 100, \text{ or } 1,000$ base documents, $100,000/d$ variants of each base document, and mutation rate $p = 0.001, 0.003, 0.01, 0.03, \text{ or } 0.1$. We take a prefix of length 1,000 from the Pizza & Chili DNA file and generate the base documents by mutating the prefix with zero-order entropy preserving point mutations at probability $10p$. We then generate the variants in the same way with mutation rate $p$. Concat and Version are similar to Page and Revision, respectively. We read $d = 10, 100, \text{ or } 1,000$ base documents of length 10,000 from the Pizza & Chili English file, and generate $10,000/d$ variants of each base document with mutation rates $0.001, 0.003, 0.01, 0.03, \text{ and } 0.1$, as above. Each variant becomes a separate document in Version, while all variants of the same base document are concatenated into a single document in Concat.

6.1.2 Queries

Real collections. For Page and Revision, we downloaded a list of Finnish words from the Institute for the Languages in Finland, and chose all words of length $\geq 5$ that occur in the collection. For Enwiki, we used search terms from an MSN query log with stopwords filtered out. We generated 20,000 patterns according to term frequencies, and selected those that occur in the collection. For Influenza, we extracted 100,000 random substrings of length 7, filtered out duplicates, and kept the 1,000 patterns with the largest occ/df ratios. For Swissprot, we extracted 200,000 random substrings of length 5, filtered out duplicates, and kept the 10,000 patterns with the largest occ/df ratios. For Wiki, we used the TREC 2006 Terabyte Track efficiency queries\footnote{http://trec.nist.gov/data/terabyte06.html} consisting of 411,394 terms in 100,000 queries.

Synthetic collections. We generated the patterns for DNA with a similar process as for Influenza and Swissprot. We extracted 100,000 substrings of length 7, filter out duplicates, and chose the 1,000 with the largest occ/df ratios. For Concat and Version, patterns were generated from the MSN query log in the same way as for Enwiki.

6.1.3 Test Environment

We used two separate systems for the experiments. For document listing and document counting, our test environment had two 2.40 GHz quad-core Intel Xeon E5620 processors and 96 GB memory. Only one core was used for the queries. The operating system was Ubuntu 12.04 with Linux kernel 3.2.0. All code was written in C++. We used g++ version 4.6.3 for the document listing experiments and version 4.8.1 for the document counting experiments.

For the top-$k$ retrieval and TF-IDF experiments, we used another system with two 16-core AMD Opteron 6378 processors and 256 GB memory. We used only a single core for the single-term queries and up to 32 cores for the
multi-term queries. The operating system was Ubuntu 12.04 with Linux kernel 3.2.0. All code was written in C++ and compiled with g++ version 4.9.2.

6.2 Document Listing

We compare our new proposals from Sections 3.3 and 4.1 to the existing document listing solutions. We also aim to determine when these sophisticated approaches are better than brute-force solutions based on pattern matching.

6.2.1 Indexes

**Brute force.** These algorithms simply sort the document identifiers in the range $DA[\ell..r]$ and report each of them once. Brute-D stores $DA$ in $n \lg d$ bits, while Brute-L retrieves the range $SA[\ell..r]$ with the locate functionality of the CSA and uses bitvector $B$ to convert it to $DA[\ell..r]$.

**Sadakane.** This family of algorithms is based on the improvements of Sadakane (2007) to the algorithm of Muthukrishnan (2002). Sada-C-L is the original algorithm, while Sada-C-D uses an explicit document array $DA$ instead of retrieving the document identifiers with locate. Sada-I-L and Sada-I-D are otherwise the same, respectively, except that they build the RMQ over the run-length encoded ILCP (Section 3.3) instead of $C$.

**Wavelet tree.** This index stores the document array in a wavelet tree to efficiently find the distinct elements in $DA[\ell..r]$ (Välimäki and Mäkinen, 2007). The best known implementation of this idea (Navarro et al, 2014) uses plain, entropy-compressed, and grammar-compressed bitvectors in the wavelet tree, depending on the level. Our WT implementation uses a heuristic similar to the original WT-alpha (Navarro et al, 2014), multiplying the size of the plain bitvector by 0.81 and the size of the entropy-compressed bitvector by 0.9, before choosing the smallest one for each level of the tree. These constants were determined by experimental tuning.

**Precomputed document lists.** This is our proposal in Section 4.1. Our implementation resorts to Brute-L to handle the short regions that the index does not cover. The variant PDL-BC compresses sets of equal documents using a Web graph compressor (Hernández and Navarro, 2014). PDL-RP uses Re-Pair compression (Larsson and Moffat, 2000) as implemented by Navarro and stores the dictionary in plain form. We use block size $b = 256$ and storing factor $\beta = 16$, which proved to be good general-purpose parameter values.

9 [http://www.dcc.uchile.cl/gnavarro/software](http://www.dcc.uchile.cl/gnavarro/software)
Grammar-based. Grammar (Claude and Munro, 2013) is an adaptation of a grammar-compressed self-index (Claude and Navarro, 2012) to document listing. Conceptually similar to PDL, Grammar uses Re-Pair to parse the collection. For each nonterminal symbol in the grammar, it stores the set of identifiers of the documents whose encoding contains the symbol. A second round of Re-Pair is used to compress the sets. Unlike most of the other solutions, Grammar is an independent index and needs no CSA to operate.

Lempel-Ziv. LZ (Ferrada and Navarro, 2013) is an adaptation of a pattern-matching index based on LZ78 parsing (Navarro, 2004) to document listing. Like Grammar, LZ does not need a CSA.

We implemented Brute, Sada, and the PDL variants ourselves and modified existing implementations of WT, Grammar, and LZ for our purposes. We always used the RLCSA (Mäkinen et al., 2010) as the CSA, as it performs well on repetitive collections. The locate support in RLCSA includes optimizations for long query ranges and repetitive collections, which is important for Brute-L and Sada-I-L. We used suffix array sample periods 8, 16, 32, 64, 128 for non-repetitive collections and 32, 64, 128, 256, 512 for repetitive ones.

When a document listing solution uses a CSA, we start the queries from the lexicographic range \([\ell..r]\) instead of the pattern \(P\). This allows us to see the performance differences between the fastest solutions better. The total time required for obtaining the ranges was 0.04 to 0.15 seconds, depending on the collection, which is negligible compared to the total time used by Grammar and LZ.

6.2.2 Results

Real collections. Figure 6 contains the results for document listing with real collections. For most of the indexes, the time/space trade-off is given by the RLCSA sample period. The trade-off of LZ comes from a parameter specific to that structure involving RMQs (Ferrada and Navarro, 2013). Grammar has no trade-off.

Brute-L always uses the least amount of space, but it is also the slowest solution. In collections with many short documents (i.e., all except Page), we have \(\text{occ/df} < 4\) on the average. The additional effort done by Sada-C-L and Sada-I-L to report each document only once does not pay off, and the space used by the RMQ structure is better spent on increasing the number of suffix array samples for Brute-L. The difference is, however, very noticeable on Page, where the documents are large and there are hundreds of occurrences of the pattern in each document. Sada-I-L uses less space than Sada-C-L when the collection is repetitive and contains many similar documents (i.e., on Revision and Influenza); otherwise Sada-C-L is slightly smaller.

The two PDL alternatives usually have similar performance, but in some cases PDL-BC uses much less space. PDL-BC, in turn, can use significantly more
space than \textbf{Brute-L}, \textbf{Sada-C-L}, and \textbf{Sada-I-L}, but is always orders of magnitude faster. The document sets of versioned collections such as \textbf{Page} and \textbf{Revision} are very compressible, making the collections very suitable for \textbf{PDL}. On the other hand, grammar-based compression cannot reduce the size of the stored document sets enough when the collections are non-repetitive. Repetitive but unstructured collections like \textbf{Influenza} represent an interesting special case. When the number of revisions of each base document is much larger than the block size $b$, each leaf block stores an essentially random subset of the revisions, which cannot be compressed very well.

Among the other indexes, \textbf{Sada-C-D} and \textbf{Sada-I-D} can be significantly faster than \textbf{PDL-BC}, but they also use much more space. From the non-CSA-based indexes, \textbf{Grammar} reaches the Pareto-optimal curve on \textbf{Revision} and \textbf{Influenza}()

In general, we can recommend \textbf{PDL-BC} as a medium-space alternative for document listing. When less space is desired, we can use \textbf{Brute-L} if the documents are small. A more robust alternative using almost the same space is \textbf{Sada-I-L}. Note that we can use fast document counting to compare $df = r - \ell + 1$, and choose between \textbf{Sada-I-L} and \textbf{Brute-L} according to the results.

\textit{Synthetic collections.} Figures 7 and 8 shows our document listing results with synthetic collections. Due to the large number of collections, the results for a given collection type and number of base documents are combined in a single plot, showing the fastest algorithm for a given amount of space and a mutation rate. Solid lines connect measurements that are the fastest for their size, while dashed lines are rough interpolations.

The plots were simplified in two ways. Algorithms providing a marginal and/or inconsistent improvement in speed in a very narrow region (mainly \textbf{Sada-C-L} and \textbf{Sada-I-L}) were left out. When \textbf{PDL-BC} and \textbf{PDL-RP} had very similar performance, only one of them was chosen for the plot.

On \textbf{DNA}, \textbf{Grammar} was a good solution for small mutation rates, while \textbf{LZ} was good with larger mutation rates. With more space available, \textbf{PDL-BC} became the fastest algorithm. \textbf{Brute-D} and \textbf{Sada-I-D} were often slightly faster than \textbf{PDL}, when there was enough space available to store the document array. On \textbf{Concat} and \textbf{Version}, \textbf{PDL} was usually a good mid-range solution, with \textbf{PDL-RP} being usually smaller than \textbf{PDL-BC}. The exceptions were the collections with 10 base documents, where the number of variants (1,000) was clearly larger than the block size (256). With no other structure in the collection, \textbf{PDL} was unable to find a good grammar to compress the sets. At the large end of the size scale, algorithms using an explicit document array $D$ were usually the fastest choice.
Fig. 6  Document listing on small (left) and large (right) real collections. The total size of the index in bits per character ($x$) and the time required to run the queries in seconds ($y$).
Fig. 7 Document listing on synthetic collections. The fastest solution for a given size in bits per character and a mutation rate. From top to bottom: 10, 100, and 1,000 base documents with Concat (left) and Version (right). None denotes that no solution can achieve that size.

6.3 Single-Term Top-k Retrieval

6.3.1 Indexes

We compare the following top-k retrieval algorithms. Many of them share names with the corresponding document listing structures described in Section 6.2.1.

Brute force. These algorithms correspond to the document listing algorithms Brute-D and Brute-L. To perform top-k retrieval, we not only collect the distinct document identifiers after sorting DA[ℓ..r], but also record the number of times each one appears. Then k identifiers appearing most frequently are reported.
Precomputed document lists. We use the variant of PDL-RP modified for top-k retrieval, as described in Section 4.2. PDL-b denotes PDL with block size b and with document sets for all suffix tree nodes above the leaf blocks, while PDL-b+F is the same with term frequencies. PDL-b-β is PDL with block size b and storing factor β.

Grid. The structure Grid (Konow and Navarro 2013) is a faster but usually larger alternative to WT. It can answer top-k queries quickly if the pattern occurs at least twice in each reported document. If documents with just one occurrence are needed, Grid uses a variant of Sada-C-L to find them. SURF (Gog and Navarro 2015b) is a faster and smaller variant of the same conceptual idea.

We implemented the Brute and PDL variants ourselves, modified the implementation of Grid to use RLCSA, and used the existing implementation of SURF. While WT (Navarro et al, 2014) also supports top-k queries, the existing 32-bit implementation cannot index the large versions of the document collections used in the experiments. As with document listing, we subtracted the time required for finding the lexicographic ranges [ℓ..r] using a CSA from the measured query times. SURF uses a CSA from the SDSL library (Gog et al, 2014), while the rest of the indexes use RLCSA.

11 http://jltsiren.kapsi.fi/rlcsa
12 https://github.com/simongog/surf/tree/single_term
6.3.2 Results

Figure 9 contains the results for single-term top-\(k\) retrieval using the large versions of the real collections. We left Page out from the results, as the number of documents (280) was too low for meaningful top-\(k\) queries. For most of the indexes, the time/space trade-off is given by the RLCSA sample period, while the results for SURF are for the three variants presented in the paper.

The three collections proved to be very different. With Revision, the PDL variants were both fast and space-efficient. When storing factor \(\beta\) was not set, the total query times were dominated by rare patterns, for which PDL had to resort to using Brute-L. This also made block size \(b\) an important time/space trade-off. When the storing factor was set, the index became smaller and slower and the trade-offs became less significant. SURF was larger and faster.
than Brute-D with $k = 10$ but became too slow with $k = 100$. We could not build Grid at all for Revision due to overflowing 32-bit variables.

On Enwiki, the variants of PDL with storing factor $\beta$ set had similar performance to Brute-D. SURF was faster with roughly the same space usage. PDL with no storing factor was much larger than the other solutions. However, it became competitive with $k = 100$, as its performance was almost unaffected by the number of documents requested. We were unable to build Grid for Enwiki.

The third collection, Influenza, was the most surprising of the three. PDL with storing factor $\beta$ set was between Brute-L and Brute-D in both time and space. We could not build PDL without the storing factor, as the document sets were too large for the Re-Pair compressor. The construction of SURF also failed for an unknown reason. Grid was the largest and the fastest index for this collection.

6.4 Document Counting

6.4.1 Indexes

We use two fast document listing algorithms as baseline document counting methods (see Section 6.2.1): Brute-D sorts the query range $DA[\ell..r]$ to count the number of distinct document identifiers, and PDL-RP returns the length of the list of documents obtained. Both indexes use the RLCSA with suffix array sample period set to 32 on non-repetitive datasets, and to 128 on repetitive datasets.

We also consider a number of encodings of Sadakane’s document counting structure (see Section 5). The following ones encode the bitvector $H'$ directly in a number of ways:

- **Sada** uses a plain bitvector representation.
- **Sada-RR** uses a run-length encoded bitvector as supplied in the RLCSA implementation. It uses $\delta$-codes to represent run lengths and packs them into blocks of 32 bytes of encoded data. Each block stores the number of bits and 1s up to its beginning.
- **Sada-RS** uses a run-length encoded bitvector, represented with a sparse bitmap marking the beginnings of the 0-runs and another for the 1-runs. Sparse bitmaps (Okanohara and Sadakane, 2007) store the lower $w$ bits of the position of each 1 in an array, and use gap encoding in a plain bitvector for the high-order bits. Value $w$ is chosen to minimize the size.
- **Sada-RD** uses run-length encoding with $\delta$-codes to represent the lengths. Each block in the bitvector contains the encoding of 128 1-bits, while three sparse bitmaps are used to mark the number of bits, 1-bits, and starting positions of block encodings.
- **Sada-grammar** uses a grammar-compressed bitvector (Navarro and Ordóñez, 2014).

The following encodings use filters in addition to bitvector $H'$:
– Sada-P-G uses Sada for $H'$ and a gap-encoded bitvector for the filter bitvector $F$. The gap-encoded bitvector is also provided in the RLCSA implementation. It differs from the run-length encoded bitvector by only encoding runs of 0-bits.
– Sada-P-RR uses Sada for $H'$ and Sada-RR for $F$.
– Sada-RR-G uses Sada-RR for $H'$ and a gap-encoded bitvector for $F$.
– Sada-RR-RR uses Sada-RR for both $H'$ and $F$.
– Sada-S uses sparse bitmaps for both $H'$ and the sparse filter $F_S$.
– Sada-S-S is Sada-S with an additional sparse bitmap for the 1-filter $F_1$.
– Sada-RS-S uses Sada-RS for $H'$ and a sparse bitmap for $F_1$.
– Sada-RD-S uses Sada-RD for $H'$ and a sparse bitmap for $F_1$.

Finally, ILCP implements the technique described in Section 3.4 using the same encoding as in Sada-RS to represent the bitvectors in the wavelet tree.

Our implementations of the above methods can be found online.\footnote{http://jltsiren.kapsi.fi/rlcsa/ and https://github.com/ahartik/succinct}

### 6.4.2 Results

Due to the use of 32-bit variables in some of the implementations, we could not build all structures for the large real collections. Hence we used the medium versions of Page, Revision, and Enwiki, the large version of Influenza, and the only version of Swissprot for the benchmarks. We started the queries from precomputed lexicographic ranges $[l..r]$ in order to emphasize the differences between the fastest variants. For the same reason, we also left out of the plots the size of the RLCSA and the possible document retrieval structures. Finally, as plain Sada was almost always the fastest method, we scaled the plots to leave out anything much larger than it. The results can be seen in Figure 10.

On Page, the filtered methods Sada-P-RR and Sada-RR-RR are clearly the best choices, being only slightly larger than the baselines and orders of magnitude faster. Plain Sada is much faster than those, but it takes much more space than all the other indexes. Only Sada-grammar compresses the structure better, but it is almost as slow as the baselines.

On Revision, there were many small encodings with similar performance. Among those, Sada-RS-S is the fastest. Sada-S is somewhat larger and faster. As on Page, plain Sada is even faster, but it takes much more space.

The situation changes on the non-repetitive Enwiki. Only Sada-RD-S, Sada-RS-S, and Sada-grammar can compress the bitvector clearly below 1 bpc, and Sada-grammar is much slower than the other two. At around 1 bpc, Sada-S is again the fastest option. Plain Sada requires twice as much space as Sada-S, but it is twice as fast.

Influenza and Swissprot contain, respectively, RNA and protein sequences, making each individual document quite random. Such collections are easy cases for Sadakane’s method, and many encodings compress the bitvector very well. In both cases, Sada-S is the fastest small encoding. On Influenza, the small encodings fit in CPU cache, making them often faster than plain Sada.
Different compression techniques succeed with different collections, for different reasons. This complicates a simple recommendation for a best option. Plain Sada is always fast, while Sada-S is usually smaller without sacrificing too much performance. When more space-efficient solutions are required, the right choice depends on the type of the collection. Our ILCP-based structure, ILCP, also outperforms Sada in space on most collections, but it is always significantly larger and slower than compressed variants of Sada.

6.5 A TF-IDF Index

Finally, we show how our indexes for single-term retrieval can be used for ranked multi-term queries on repetitive text collections. The key idea is to regard our incremental top-k algorithms as abstract representations of the
inverted lists of the individual query terms, sorted by decreasing weight, and then apply any algorithm that traverses those lists sequentially. Since our relevance score will depend on the term frequency and the document frequency of the terms, we will integrate a document counting structure as well.

We use RLCSA as the CSA, PDL–256+F for single-term top-k retrieval, and Sada-S for document counting. We could have integrated the document counts into the PDL structure, but a separate counting structure makes the index more flexible. Additionally, encoding the number of redundant documents in each internal node of the suffix tree (Sada) often takes less space than encoding the total number of documents in each node of the pruned suffix tree (PDL).

Let \( Q = \langle q_1, \ldots, q_m \rangle \) be a query consisting of \( m \) terms \( q_i \in \Sigma^* \). We support ranked queries, which return the \( k \) documents with the highest weights among the documents matching the query. A disjunctive or ranked-OR query matches document \( D \) if at least one of the terms occurs in it, while a conjunctive or ranked-AND query matches \( D \) if all query terms occur in it. Our index supports TF-IDF-like weights

\[
 w(D, Q) = \sum_{i=1}^{m} w(D, q_i) = \sum_{i=1}^{m} f(tf(D, q_i)) \cdot g(df(q_i)),
\]

where \( f \geq 0 \) is an increasing function, \( tf(D, q_i) \) is the term frequency (the number of occurrences) of term \( q_i \) in document \( D \), \( g \geq 0 \) is a decreasing function, and \( df(q_i) \) is the document frequency of term \( q_i \). In this experiment, we use \( f(tf) = tf \) and \( g(df) = \lg(d / \max(df, 1)) \).

The query algorithm uses CSA to find the lexicographic range \([\ell_i..r_i]\) matching each term \( q_i \). We then use PDL to find the sparse suffix tree node \( v_i \) corresponding to range \([\ell_i..r_i]\), and CSA to extract all documents matching term \( q_i \) if no such node exists. We also compute \( df(q_i) \) for all query terms \( q_i \) using Sada. The algorithm then iterates the following loop with \( k' = 2k, 4k, 8k, \ldots \):

1. Extract \( k' \) more documents from the document list of \( v_i \) for each term \( q_i \).
2. If the query is conjunctive, filter out extracted documents that do not match the query terms with completely decompressed document lists.
3. Determine a lower bound for \( w(D, Q) \) for all documents \( D \) extracted so far. If document \( D \) has not been encountered in the document list of \( v_i \), use 0 as a lower bound for \( w(D, q_i) \).
4. Determine an upper bound for \( w(D, Q) \) for all documents \( D \). If document \( D \) has not been encountered in the document list of \( v_i \), use \( tf(D', q_i) \), where \( D' \) is the next unextracted document for term \( q_i \), as an upper bound for \( tf(D, q_i) \).
5. If the query is disjunctive, filter out extracted documents \( D \) with smaller upper bounds for \( w(D, Q) \) than the lower bounds for the current top-\( k \) documents. Stop if the top-\( k \) set cannot change any longer.
6. If the query is conjunctive, stop if the top-\( k \) documents match all query terms and the upper bounds for the remaining documents are lower than the lower bounds for the top-\( k \) documents.
Table 2: Ranked multi-term queries on the Wiki collection. Query type, number of documents requested, and the average number of queries per second with 1, 8, 16, and 32 query threads.

| Query  | k   | 1 thread | 8 threads | 16 threads | 32 threads |
|--------|-----|----------|-----------|------------|------------|
| Ranked-AND | 10  | 152      | 914       | 1699       | 2668       |
|         | 100 | 136      | 862       | 1523       | 2401       |
| Ranked-OR | 10  | 229      | 1529      | 2734       | 4179       |
|         | 100 | 163      | 1089      | 1905       | 2919       |

The algorithm always finds a correct top-$k$ set, but the weights may be incorrect if a disjunctive query stops early.

We tested the performance of the multi-term index on the 1432 MB Wiki collection. RLCSA took 0.73 bpc with sample period 128 (the sample period did not have a significant impact on query performance), PDL–256+F took 3.37 bpc, and Sada-S took 0.13 bpc, for a total of 4.23 bpc (757 MB). Out of the total of 100,000 queries in the query set, there were matches for 31,417 conjunctive queries and 97,774 disjunctive queries.

The results can be seen in Table 2. When using a single query thread, the index can process 136–229 queries per second (around 4–7 milliseconds per query), depending on the query type and the value of $k$. Disjunctive queries are faster than conjunctive queries, while larger values of $k$ do not increase query times significantly. The times are competitive with the state of the art on inverted indexes for natural language texts [Konow et al. 2013]. Since we index strings, our query terms can be phrases, whereas allowing phrases on inverted indexes is a major issue. The speedup from using 32 query threads is around 18x.

7 Conclusions

We have investigated the space/time tradeoffs involved in indexing highly repetitive string collections, with the goal of performing information retrieval tasks on them. Particularly, we considered the problems of document listing, top-$k$ retrieval, and document counting. We have developed new indexes that perform particularly well on those types of collections, and studied how other existing data structures perform in this scenario, and in which cases the indexes are actually better than brute-force approaches. As a result, we offered recommendations on which structures to use depending on the kind of repetitiveness involved and the desired space usage. As a final proof of concept, we have shown how the tools we developed can be assembled to build an efficient index supporting ranked multi-term queries on repetitive string collections.

Acknowledgements This work was supported in part by Academy of Finland grants 268324, 258308, 250345 (CoECGR), and 134287; the Helsinki Doctoral Programme in Computer Science; the Jenny and Antti Wihuri Foundation, Finland; the Wellcome Trust grant 098051, UK; Fondecyt grant 1-140796, Chile; the Millennium Nucleus for Information and
Bibliography

Anick PG, Flynn RA (1992) Versioning a full-text information retrieval system. In: Proc. 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), pp 98–111

Baeza-Yates R, Ribeiro-Neto B (2011) Modern Information Retrieval, 2nd edn. Addison-Wesley

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

Broder A, Eiron N, Fontoura M, Herscovici M, Lempel R, McPherson J, Qi R, Shekita E (2006) Indexing shared content in information retrieval systems. In: Proc. 10th International Conference on Extending Database Technology (EDBT), LNCS 3896, pp 313–330

Böttcher S, Clarke C, Cormack G (2010) Information Retrieval: Implementing and Evaluating Search Engines. MIT Press

Clark D (1996) Compact PAT trees. PhD thesis, University of Waterloo, Canada

Claude F, Munro I (2013) Document listing on versioned documents. In: Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 8214, pp 72–83

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

Claude F, Navarro G (2012) Improved grammar-based compressed indexes. In: Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 7608, pp 180–192

Claude F, Fariña A, Martínez-Prieto M, Navarro G (2010) Compressed $q$-gram indexing for highly repetitive biological sequences. In: Proc. 10th International Conference on Bioinformatics and Bioengineering (BIBE), pp 86–91

Claude F, Fariña A, Martínez-Prieto M, Navarro G (2016) Universal indexes for highly repetitive document collections. Information Systems 61:1–23

Dhaliwal J, Puglisi SJ, Turpin A (2012) Practical efficient string mining. IEEE Transactions on Knowledge and Data Engineering 24(4):735–744

Do HH, Jansson J, Sadakane K, Sung WK (2014) Fast relative Lempel-Ziv self-index for similar sequences. Theoretical Computer Science 532:14–30

Ferrada H, Navarro G (2013) A Lempel-Ziv compressed structure for document listing. In: Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 8214, pp 116–128

Fischer J, Heun V (2011) Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing 40(2):465–492
Gagie T, Gawrychowski P, Kärkkäinen J, Nekrich Y, Puglisi SJ (2012a) A faster grammar-based self-index. In: Proc. 6th Language and Automata Theory and Applications (LATA), LNCS 7183, pp 240–251
Gagie T, Navarro G, Puglisi SJ (2012b) New algorithms on wavelet trees and applications to information retrieval. Theoretical Computer Science 426-427:25–41
Gagie T, Gawrychowski P, Kärkkäinen J, Nekrich Y, Puglisi SJ (2014) LZ77-based self-indexing with faster pattern matching. In: Proc. 11th Latin American Theoretical Informatics Symposium (LATIN), LNCS 8392, pp 731–742
Gog S, Navarro G (2015a) Improved single-term top-k document retrieval. In: Proc. 17th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM Press, pp 24–32
Gog S, Navarro G (2015b) Improved single-term top-k document retrieval. In: Proc. 17th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, pp 24–32, DOI 10.1137/1.9781611973754.3
Gog S, Beller T, Moffat A, Petri M (2014) From theory to practice: Plug and play with succinct data structures. In: Proc. 13th International Symposium on Experimental Algorithms (SEA), Springer, LNCS, vol 8504, pp 326–337, DOI 10.1007/978-3-319-07959-2_28
Grossi R, Gupta A, Vitter J (2003) High-order entropy-compressed text indexes. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp 841–850
He J, Suel T (2012) Optimizing positional index structures for versioned document collections. In: Proc. 35th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), pp 245–254
He J, Yan H, Suel T (2009) Compact full-text indexing of versioned document collections. In: Proc. 18th ACM International Conference on Information and Knowledge Management (CIKM), pp 415–424
He J, Zeng J, Suel T (2010) Improved index compression techniques for versioned document collections. In: Proc. 19th ACM International Conference on Information and Knowledge Management (CIKM), pp 1239–1248
Hernández C, Navarro G (2014) Compressed representations for web and social graphs. Knowledge and Information Systems 40(2):279–313
Hon WK, Patil M, Shah R, Thankachan SV, Vitter JS (2013) Indexes for document retrieval with relevance. In: Space-Efficient Data Structures, Streams, and Algorithms, LNCS 8066, pp 351–362
Konow R, Navarro G (2013) Faster compact top-k document retrieval. In: Proc. DCC, pp 351–360
Konow R, Navarro G, Clarke CLA, López-Ortiz A (2013) Faster and smaller inverted indices with treaps. In: Proc. 36th International Conference on Research and Development in Information Retrieval (SIGIR), pp 193–202
Kreft S, Navarro G (2013) On compressing and indexing repetitive sequences. Theoretical Computer Science 483:115–133
Larsson NJ, Moffat A (2000) Off-line dictionary-based compression. Proceedings of the IEEE Data Compression Conference 88(11):1722–1732
Mäkinen V, Navarro G, Sirén J, Välimäki N (2010) Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology 17(3):281–308

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

Marschall T, et al (2016) Computational pan-genomics: Status, promises and challenges. Tech. rep., Cold Spring Harbor bioRxiv, DOI 10.1101/043430, URL http://biorxiv.org/content/early/2016/03/29/043430

Muthukrishnan S (2002) Efficient algorithms for document retrieval problems. In: Proc. SODA, pp 657–666

Navarro G (2004) Indexing text using the Ziv-Lempel trie. Journal of Discrete Algorithms (JDA) 2(1):87–114

Navarro G (2014) Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. ACM Computing Surveys 46(4):article 52

Navarro G, Mäkinen V (2007) Compressed full-text indexes. ACM Computing Surveys 39(1):art. 2

Navarro G, Nekrich Y (2012) Top-\(k\) document retrieval in optimal time and linear space. In: Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp 1066–1078

Navarro G, Ordóñez A (2014) Grammar compressed sequences with rank/select support. In: Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 8799, pp 31–44

Navarro G, Puglisi SJ, Valenzuela D (2014) General document retrieval in compact space. ACM Journal of Experimental Algorithmics 19(2):article 3

Okanohara D, Sadakane K (2007) Practical entropy-compressed rank/select dictionary. In: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), pp 60–70

Raman R, Raman V, Rao SS (2007) Succinct indexable dictionaries with applications to encoding \(k\)-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4):art. 43

Rochkind M (1975) The source code control system. IEEE Transactions on Software Engineering 1(4):364–370

Sadakane K (2007) Succinct data structures for flexible text retrieval systems. Journal of Discrete Algorithms 5:12–22

Stephens ZD, Lee SY, Faghi F, Campbell RH, Zhai C, Efron MJ, Iyer R, Schatz MC, Sinha S, Robinson GE (2015) Big data: Astronomical or geological? PLoS Biology 13(7):e1002,195

Szpankowski W (1993) A generalized suffix tree and its (un)expected asymptotic behaviors. SIAM Journal on Computing 22(6):1176–1198

Välimäki N, Mäkinen V (2007) Space-efficient algorithms for document retrieval. In: Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 4580, pp 205–215

Weiner P (1973) Linear pattern matching algorithm. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, pp 1–11