Inf Retrieval J (2017) 20:253–291
DOI 10.1007/s10791-017-9297-7

INFORMATION RETRIEVAL EFFICIENCY

Document retrieval on repetitive string collections

Travis Gagie[1] · Aleksi Hartikainen[2] · Kalle Karhu[3] · Juha Kärkkäinen[4] · Gonzalo Navarro[5] · Simon J. Puglisi[4] · Jouni Sirén[6]

Received: 28 May 2016 / Accepted: 28 February 2017 / Published online: 1 April 2017

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. 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. Finally, we show how the tools we

Preliminary partial versions of this paper appeared in Proc. CPM 2013, Proc. ESA 2014, and Proc. DCC 2015. Part of this work was done while the first author was at the University of Helsinki and the third author was at Aalto University, Finland.

✉ Jouni Sirén
jouni.siren@iki.fi

Travis Gagie
travis.gagie@gmail.com

Aleksi Hartikainen
ahartik@gmail.com

Kalle Karhu
kalle.karhu@iki.fi

Juha Kärkkäinen
tpkarkka@cs.helsinki.fi

Gonzalo Navarro
gnavarro@dcc.uchile.cl

Simon J. Puglisi
puglisi@cs.helsinki.fi

developed can be combined to solve ranked conjunctive and disjunctive multi-term queries under the simple tf-idf model of relevance. We thoroughly evaluate the resulting techniques in various real-life repetitiveness scenarios, and recommend the best choices for each case.

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), an immensely successful technology that can by now be considered mature. The inverted index has well-known limitations, however: the text must be easy to parse into *terms* or *words*, and queries must be sets of words or sequences of words (*phrases*). Those limitations are acceptable in most cases when *natural language* text collections are indexed, and they enable the use of an extremely simple index organization that is efficient and scalable, and that has been the key to the success of Web-scale information retrieval.

Those limitations, on the other hand, hamper the use of the inverted index in other kinds of string collections where partitioning the text into words and limiting queries to word sequences is inconvenient, difficult, or meaningless: 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 about alternatives to the inverted index 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, exceeding Moore's Law rate by a wide margin (Stephens et al. 2015). This growth brings new scientific opportunities but it also creates new computational problems.

[1] CeBiB — Center of Biotechnology and Bioengineering, School of Computer Science and Telecommunications, Diego Portales University, Santiago, Chile

[2] Google Inc, Mountain View, CA, USA

[3] Research and Technology, Planmeca Oy, Helsinki, Finland

[4] Department of Computer Science, Helsinki Institute of Information Technology, University of Helsinki, Helsinki, Finland

[5] Department of Computer Science, CeBiB — Center of Biotechnology and Bioengineering, University of Chile, Santiago, Chile

[6] Wellcome Trust Sanger Institute, Cambridge, UK

[1] www.wikipedia.org.

[2] From the Internet Archive, www.archive.org/web/web.php.

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 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 (Rochkind 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, to compute tf-idf, a simple and popular relevance model. 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, 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 various proposals to reduce 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; Belazzougui et al. 2015) support only *pattern matching*, that is, they count or list the occ occurrences of a pattern P in the whole collection. Of course one can retrieve the occ occurrences and then answer any of our three document retrieval queries, but the time will be $\Omega(occ)$. Instead, there are optimal-time indexes for string collections that solve document listing in time $\mathcal{O}(|P| + \mathsf{df})$ (Muthukrishnan 2002), top-k retrieval in time $\mathcal{O}(|P| + k)$ (Navarro and Nekrich 2012), and

[3] Such as p7zip, http://p7zip.sourceforge.net.

document counting in time $\mathcal{O}(|P|)$ (Sadakane 2007). The first two solutions, however, use a lot of space even for classical, non-repetitive collections. While 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. It then 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. Third, we show that a solution for document counting (Sadakane 2007) that uses just two bits per symbol (bps) 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. Finally, we show how the different components of our solutions can be assembled to offer tf-idf ranked conjunctive and disjunctive multi-term queries on repetitive string collections.

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 of up to 1 GB we perform document listing and top-k retrieval in 10–100 microseconds per result and using 1–2 bits per symbol. For counting, we use as little as 0.1 bits per symbol and answer queries in less than a microsecond. Multi-term top-k queries can be solved with a throughput of 100–200 queries per second, which we show to be similar to that of a state-of-the-art inverted index. Of course, we do not aim to compete with inverted indexes in the scenarios where they can be applied (mainly, in natural language text collections), but to offer similar functionality in the case of generic string collections, where inverted indexes cannot be used.

This article collects our earlier results appearing in *CPM 2013* (Gagie et al. 2013), *ESA 2014* (Navarro et al. 2014a), and *DCC 2015* (Gagie et al. 2015), where we focused on exploiting repetitiveness in different ways to handle different document retrieval problems. Here we present them in a unified form, considering the application of two new techniques (ILCP and PDL) and an existing one (Sadakane 2007) to the three problems (document listing, top-k retrieval, and document counting), and showing how they interact (e.g., the need to use fast document counting to choose the best document listing method). In this article we also consider a more complex document retrieval problem we had not addressed before: top-k retrieval of multi-word queries. We present an algorithm that uses our (single-term) top-k retrieval and document counting structures to solve ranked multi-term conjunctive and disjunctive queries under the tf-idf relevance model.

The article is organized as follows (see Table 1). In Sect. 2 we introduce the concepts needed to follow the presentation. In Sect. 3 we introduce the Interleaved LCP (ILCP) structure and show how it can be used for document listing and, with a different representation, for document counting. In Sect. 4 we introduce our second structure,

Table 1 The techniques we study and the document retrieval problems we solve with them

Problem	ILCP	PDL	Sadakane
Listing	Section 3.3	Section 4.1	
Top-k		Section 4.2	
Counting	Section 3.4		Section 5

Precomputed Document Lists (PDL), and describe how it can be used for document listing and, with some reordering of the lists, for top-k retrieval. Section 5 then returns to the problem of document counting, not to propose a new data structure but to study a known one (Sadakane 2007), which is found to be compressible in a repetitiveness scenario (and, curiously, on totally random texts as well). Section 6 shows how our developments can be combined to build a document retrieval index that handles multi-term queries. Section 7 empirically studies the performance of our solutions on the three document retrieval problems, also comparing them with the state of the art for generic string collections, repetitive or not, and giving recommendations on which structure to use in each case. Finally, Sect. 8 concludes and gives some future work directions.

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 (Manber and Myers 1993). Assume that we have a collection of d strings, each terminated with a special symbol "\$" (which we consider to be lexicographically smaller than any other symbol), 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, $\mathsf{SA}[1..n]$, which is an array of pointers to all the suffixes sorted in increasing lexicographic order, that is, $T[\mathsf{SA}[i]..n] < T[\mathsf{SA}[i+1]..n]$ for all $1 \leq i < n$. To find all the 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 $\mathcal{O}(m + occ)$. On a suffix array, we obtain the range $\mathsf{SA}[\ell..r]$ 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 $\mathcal{O}(m \lg n + occ)$.

We will make use of compressed suffix arrays (Navarro and Mäkinen 2007), which we will call generically $\mathsf{CSA}s$. Their size in bits is denoted $|\mathsf{CSA}|$, their time to find ℓ and r is denoted $\mathsf{search}(m)$, and their time to access any cell $\mathsf{SA}[i]$ is denoted $\mathsf{lookup}(n)$. 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 $\mathsf{rank}_c(S, i)$, which counts the number of occurrences of symbol c in $S[1..i]$, and $\mathsf{select}_c(S, j)$, which gives the position of the jth occurrence of symbol c in S. For

bitvectors, one can compute both functions in $\mathcal{O}(1)$ time using $o(n)$ bits on top of S (Clark 1996). If S contains m 1s, we can also represent it using $m \lg \frac{n}{m} + \mathcal{O}(m)$ bits, so that rank takes $\mathcal{O}\left(\lg \frac{n}{m}\right)$ time and select takes $\mathcal{O}(1)$ (Okanohara and Sadakane 2007).[4]

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. 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 rank and select capabilities to the bitvectors associated with the nodes, the wavelet tree can compute any $S[i] = c$, $\mathsf{rank}_c(S, i)$, or $\mathsf{select}_c(S, j)$) 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 $n \lg \sigma + o(nh)$, where h is the wavelet tree height, independent of the way the alphabet is partitioned (Grossi et al. 2003).

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* $\mathsf{DA}[1..n]$ of T, in which each cell $\mathsf{DA}[i]$ stores the identifier of the document containing $T[\mathsf{SA}[i]]$; an array $C[1..n]$, in which each cell $C[i]$ stores the largest value $h < i$ such that $\mathsf{DA}[h] = \mathsf{DA}[i]$, or 0 if there is no such value h; and a data structure supporting range-minimum queries (RMQs) over C, $\mathsf{rmq}_C(i, j) = \arg\min_{i \le k \le j} C[k]$. These data structures take a total of $\mathcal{O}(n \lg n)$ bits. Given a pattern $P[1..m]$, the suffix tree is used to find the interval $\mathsf{SA}[\ell..r]$ 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 $\mathsf{DA}[i]$. Thus a recursive algorithm finding all those positions i starts with $k = \mathsf{rmq}_C(\ell, r)$. If $C[k] \ge \ell$ it stops. Otherwise it reports document $\mathsf{DA}[k]$ and continues recursively with the ranges $C[\ell..k-1]$ and $C[K+1..r]$ (the condition $C[k] \ge \ell$ always uses the original ℓ value). In total, the algorithm uses $\mathcal{O}(m + \mathsf{df})$ time, where df is the number of documents returned.

Sadakane (2007) proposed a space-efficient version of this algorithm, using just $|\mathsf{CSA}| + \mathcal{O}(n)$ bits. 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 $\mathsf{DA}[i] = \mathsf{rank}_1(B, \mathsf{SA}[i])$ 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 comparisons $C[k] \ge \ell$ are replaced by marking the documents already reported in a bitvector $V[1..d]$ (initially all 0s), so that $V[\mathsf{DA}[i]] = 1$ iff document $\mathsf{DA}[i]$ has already been reported. If $V[\mathsf{DA}[i]] = 1$ the recursion stops, otherwise it sets $V[\mathsf{DA}[i]]$, reports $\mathsf{DA}[i]$, 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 $C[K+1..r]$ (Navarro 2014). The total time is then $\mathcal{O}(\mathsf{search}(m) + \mathsf{df} \cdot \mathsf{lookup}(n))$.

[4] This is achieved by using a constant-time $\mathsf{rank/select}$ solution (Clark 1996) to represent their internal bitvector H.

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 the classical document listing technique of 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 $\mathsf{LCP}_S[1..|S|]$ of a string S is defined such that $\mathsf{LCP}_S[1] = 0$ and, for $2 \le i \le |S|$, $\mathsf{LCP}_S[i]$ is the length of the longest common prefix of the lexicographically $(i-1)$th and ith suffixes of S, that is, of $S[\mathsf{SA}_S[i-1]..|S|]$ and $S[\mathsf{SA}_S[i]..|S|]$, where 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 $T[1..n] = S_1 \cdot S_2 \cdots S_d$ be the concatenation of documents S_j, DA the document array of T, and LCP_{S_j} the longest-common-prefix array of string S_j. Then the *interleaved LCP array of T* is defined, for all $1 \le i \le n$, as

$$\mathsf{ILCP}[i] = \mathsf{LCP}_{S_{\mathsf{DA}[i]}}\left[\mathsf{rank}_{\mathsf{DA}[i]}(\mathsf{DA}, i)\right].$$

That is, if the suffix $\mathsf{SA}[i]$ belongs to document S_j (i.e., $\mathsf{DA}[i] = j$), and this is the rth suffix of SA that belongs to S_j (i.e., $r = \mathsf{rank}_j(\mathsf{DA}, i)$), then $\mathsf{ILCP}[i] = \mathsf{LCP}_{S_j}[r]$. Therefore the order of the individual LCP arrays is preserved in ILCP.

Example Consider the documents $S_1 = \texttt{"TATA\$"}$, $S_2 = \texttt{"LATA\$"}$, and $S_3 = \texttt{"AAAA\$"}$. Their concatenation is $T = \texttt{"TATA\$LATA\$AAAA\$"}$, its suffix array is $\mathsf{SA} = \langle 15, 10, 5, 14, 9, 4, 13, 12, 11, 7, 2, 6, 8, 3, 1\rangle$ and its document array is $\mathsf{DA} = \langle 3, \mathbf{2}, 1, 3, \mathbf{2}, 1, 3, 3, 3, \mathbf{2}, 1, \mathbf{2}, \mathbf{2}, 1, 1\rangle$. The LCP arrays of the documents are $\mathsf{LCP}_{S_1} = \langle 0, 0, 1, 0, 2\rangle$, $\mathsf{LCP}_{S_2} = \langle \mathbf{0}, \mathbf{0}, \mathbf{1}, \mathbf{0}, \mathbf{0}\rangle$, and $\mathsf{LCP}_{S_3} = \langle 0, 0, 1, 2, 3\rangle$. Therefore, $\mathsf{ILCP} = \langle 0, \mathbf{0}, 0, 0, \mathbf{0}, 0, 1, 2, 3, \mathbf{1}, 1, \mathbf{0}, \mathbf{0}, 0, 2\rangle$ interleaves the LCP arrays in the order given by DA (notice the fonts above).

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

Lemma 1 *Let $T[1..n] = S_1 \cdot S_2 \cdots S_d$ be the concatenation of documents S_j, SA its suffix array and DA its document array. Let $\mathsf{SA}[\ell..r]$ 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 $\mathsf{DA}[\ell..r]$ are in the same positions as the values strictly less than m in $\mathsf{ILCP}[\ell..r]$.*

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

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

Now let f_j be the leftmost occurrence of j in $\mathsf{DA}[\ell..r]$. This means that $\mathsf{SA}[f_j]$ is the lexicographically first suffix of S_j that starts with P. By the definition of ℓ_j, it holds that $\ell_j = \mathsf{rank}_j(\mathsf{DA}, f_j)$. Thus, by definition of ILCP, it holds that $\mathsf{ILCP}[f_j] = \mathsf{LCP}_{S_j}[\mathsf{rank}_j(\mathsf{DA}, f_j)] = \mathsf{LCP}_{S_j}[\ell_j] < m$, whereas all the other $\mathsf{ILCP}[k]$ values, for $\ell \le k \le r$, where $\mathsf{DA}[k] = j$, must be $\ge m$. □

Example In the example above, if we search for $P[1..2] = \text{"TA"}$, the resulting range is $\mathsf{SA}[13..15] = \langle 8, 3, 1 \rangle$. The corresponding range $\mathsf{DA}[13..15] = \langle 2, 1, 1 \rangle$ indicates that the occurrence at $\mathsf{SA}[13]$ is in S_2 and those in $\mathsf{SA}[14..15]$ are in S_1. According to the lemma, it is sufficient to report the documents $\mathsf{DA}[13] = 2$ and $\mathsf{DA}[14] = 1$, as those are the positions in $\mathsf{ILCP}[13..15] = \langle 0, 0, 2 \rangle$ 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 (Sect. 2.3): instead of recursing until we have listed all the positions k such that $C[k] < \ell$, we recurse until we list all the positions k such that $\mathsf{ILCP}[k] < m$. 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 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),[5] 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.[6]), the ILCP array of T is formed by $\rho \le r + \mathcal{O}(s \lg(r + s))$ runs of equal values.*

Proof Before applying the edit operations, we have $T = S_1 \cdots S_d$ and $S_j = S\$$ for all j. At this point, ILCP is formed by at most $r + 1$ runs of equal values, since the d equal suffixes $S_j[\mathsf{SA}_{S_j}[i]..r + 1]$ must be contiguous in the suffix array SA of T, in the area $\mathsf{SA}[(i-1)d + 1..id]$. Since the values $l = \mathsf{LCP}_{S_j}[i]$ are also equal, and ILCP values are the LCP_{S_j} values listed in the order of SA, it follows that $\mathsf{ILCP}[(i-1)d + 1..id] = l$ forms a

[5] This model states that the statistical dependence of a symbol from previous ones tends to zero as the distance towards them tends to infinity. The A2 model includes, in particular, the Bernoulli model (where each symbol is generated independently of the context), stationary Markov chains (where the probability of each symbol depends on the previous one), and kth order models (where each symbol depends on the k previous ones, for a fixed k).

[6] This is a very strong kind of convergence. A sequence X_n tends to a value β 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, $\lim_{n\to\infty} \sup_{N>n} \Pr(|X_N/\beta - 1| > \epsilon) = 0$.

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 $\mathcal{O}(\lg(r+s))$ (Szpankowski 1993), the suffix will possibly be moved in SA only for $h = \mathcal{O}(\lg(r+s))$. Thus, a.s., only $\mathcal{O}(\lg(r+s))$ suffixes are moved in SA, and possibly the corresponding runs in ILCP are broken. Hence $\rho \leq r + \mathcal{O}(s \lg(r+s))$ a.s. □

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 ρ.

3.3 Document listing

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

With this representation, it holds that $\mathsf{ILCP}[i] = \mathsf{VILCP}[\mathsf{rank}_1(L, i)]$. We can map from any position i to its run $i' = \mathsf{rank}_1(L, i)$ in time $\mathcal{O}(\lg(n/\rho))$, and from any run i' to its starting position in ILCP, $i = \mathsf{select}(L, i')$, in constant time.

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

This is sufficient to emulate the document listing algorithm of Sadakane (2007) (Sect. 2.3) 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 in the same way as L, so that it requires $d \lg(n/d) + \mathcal{O}(d)$ bits and lets us compute any value $\mathsf{DA}[i] = \mathsf{rank}_1(B, \mathsf{SA}[i])$ in time $\mathcal{O}(\mathsf{lookup}(n))$. Finally, we build the compact RMQ data structure (Fischer and Heun 2011) on VILCP, requiring $2\rho + o(\rho)$ bits. We note that this RMQ structure does not need access to VILCP to answer queries.

Assume that we have already found the range $\mathsf{SA}[\ell..r]$ in $\mathcal{O}(\mathsf{search}(m))$ time. We compute $\ell' = \mathsf{rank}_1(L, \ell)$ and $r' = \mathsf{rank}_1(L, r)$, which are the endpoints of the interval $\mathsf{VILCP}[\ell'..r']$ containing the values in the runs in $\mathsf{ILCP}[\ell..r]$. Now we run Sadakane's algorithm on $\mathsf{VILCP}[\ell'..r']$. Each time we find a minimum at $\mathsf{VILCP}[i']$, we remap it to the run $\mathsf{ILCP}[i..j]$, where $i = \max(\ell, \mathsf{select}(L, i'))$ and $j = \min(r, \mathsf{select}(L, i'+1) - 1)$. For each $i \leq k \leq j$, we compute $\mathsf{DA}[k]$ using B and RLCSA as explained, mark it in $V[\mathsf{DA}[k]] \leftarrow 1$, and report it. If, however, it already holds that $V[\mathsf{DA}[k]] = 1$, 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 $\mathsf{VILCP}[i']$ found.

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

```
function listDocuments(ℓ, r)
    (ℓ′, r′) ← (rank_1(L, ℓ), rank_1(L, r))
    return list(ℓ′, r′)

function list(ℓ′, r′)
    if ℓ′ > r′: return ∅
    i′ ← rmq_VILCP(ℓ′, r′)
    i ← max(ℓ, select(L, i′))
    j ← min(r, select(L, i′ + 1) − 1)
    res ← ∅
    for k ← i . . . j:
        g ← rank_1(B, SA[k])
        if V[g] = 1: return res
        V[g] ← 1
        res ← res ∪ {g}
    return res ∪ list(ℓ′, i′ − 1) ∪ list(i′ + 1, r′)
```

Fig. 1 Pseudocode for document listing using the ILCP array. Function listDocuments(ℓ, r) lists the documents from interval $\mathsf{SA}[\ell..r]$; list(ℓ', r') returns the distinct documents mentioned in the runs ℓ' to r' that also belong to $\mathsf{DA}[\ell..r]$. 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 *res* are known to be disjoint

Proof Let $j = \mathsf{DA}[k]$ be the leftmost occurrence of document j in $\mathsf{DA}[\ell..r]$. By Lemma 1, among all the positions where $\mathsf{DA}[k'] = j$ in $\mathsf{DA}[\ell..r]$, k is the only one where $\mathsf{ILCP}[k] < m$. 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 $\mathsf{DA}[k'] = j$, since it has a larger ILCP value and is to the right of k. Therefore, when $V[\mathsf{DA}[k]] = 0$, that is, the first time we find a $\mathsf{DA}[k] = j$, it must hold that $\mathsf{ILCP}[k] < m$, 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 $V[\mathsf{DA}[k']] = 1$, the document has already been reported. Thus this is not its leftmost occurrence and then $\mathsf{ILCP}[k'] \geq m$ 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 ρ is small on repetitive collections (Lemma 2):

Theorem 1 *Let $T = S_1 \cdot S_2 \cdots S_d$ 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* $\mathsf{search}(m)$ *and accessing* $\mathsf{SA}[i]$ *in time* $\mathsf{lookup}(n)$. *Let ρ be the number of runs in the* ILCP *array of T. We can store T in $|\mathsf{CSA}| + \rho \lg(n/\rho) + \mathcal{O}(\rho) + d \lg(n/d) + \mathcal{O}(d) = |\mathsf{CSA}| + \mathcal{O}((\rho + d) \lg n)$ bits such that document listing takes $\mathcal{O}(\mathsf{search}(m) + \mathsf{df} \cdot (\mathsf{lookup}(n) + \lg n))$ 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 (Sect. 2.2), 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 ith leftmost leaf is at depth $1 + 2\lfloor \lg i \rfloor = \mathcal{O}(\lg i)$. The number of wavelet tree nodes up to depth d is $\sum_{i=1}^{(d+1)/2} 2^i = 2(2^{(d+1)/2} - 1)$. The number of nodes up to the depth of the mth leftmost leaf is maximized when m is of the form $m = 2^{d-1}$, reaching $2(2^d - 1) = 4m - 2 = \mathcal{O}(m)$. See Fig. 2.

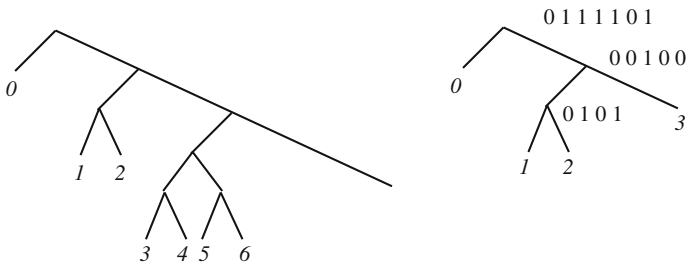

Fig. 2 *On the left*, the schematic view of our skewed wavelet tree; *on the right*, the case of our running example where it represents $\mathsf{VILCP} = \langle 0, 1, 2, 3, 1, 0, 2\rangle$

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

The number of documents where P appears, df, is the number of times a value smaller than m occurs in $\mathsf{ILCP}[\ell..r]$. 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 $[\mathsf{rank}_0(W, \ell - 1) + 1..\mathsf{rank}_0(W, r)]$ and to the right child with the interval $[\mathsf{rank}_1(W, \ell - 1) + 1..\mathsf{rank}_1(W, r)]$, stopping the recursion on empty intervals. This method arrives at all the wavelet tree leaves corresponding to the distinct values in $\mathsf{ILCP}[\ell..r]$. Moreover, if 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 $\mathsf{ILCP}[\ell..r]$.

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 $\mathcal{O}(m)$ wavelet tree nodes, in time $\mathcal{O}(m)$.

A complication is that VILCP is the array of run length heads, so when we start at $\mathsf{VILCP}[\ell'..r']$ and arrive at each leaf l with interval $[\ell'_l..r'_l]$, we only know that $\mathsf{VILCP}[\ell'..r']$ contains from the ℓ'_lth to the r'_lth occurrences of value l in $\mathsf{VILCP}[\ell'..r']$. We store a reordering of the run lengths so that the runs 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 $\mathsf{select}(L', \cdot)$, to count the total length spanned by the ℓ'_lth to r'_lth 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 $\mathsf{ILCP}[\ell..r]$. Figure 3 gives the pseudocode.

Theorem 2 *Let $T = S_1 \cdot S_2 \cdots S_d$ be the concatenation of d documents S_j, and CSA a compressed suffix array on T that searches for any pattern $P[1..m]$ in time $\mathsf{search}(m)$. 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 S_j. Then we can store T in $|\mathsf{CSA}| + \rho(\lg \lambda + 2 \lg(n/\rho) + \mathcal{O}(1)) = |\mathsf{CSA}| + \mathcal{O}(\rho \lg n)$ bits such that the number of documents where a pattern $P[1..m]$ occurs can be computed in time $\mathcal{O}(m + \mathsf{search}(m))$.*

```
function countDocuments(ℓ, r)
    (ℓ′, r′) ← (rank_1(L, ℓ), rank_1(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 v is a leaf:
        l ← l − 1
        if ℓ′ > r′: return 0
        return select(L′, r′ + 1) − select(L′, ℓ′)
    (ℓ_1, r_1) ← (rank_1(v.W, ℓ′ − 1) + 1, rank_1(v.W, r′))
    return count(v.left, ℓ′ − ℓ_1 + 1, r′ − r_1) + count(v.right, ℓ_1, r_1)
```

Fig. 3 Document counting with the ILCP array. Function countDocuments(ℓ, r) counts the distinct documents from interval $\mathsf{SA}[\ell..r]$; 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 $\mathsf{DA}[\ell..r]$. 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

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 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 a block size b and a constant $\beta \geq 1$, we build a sampled suffix tree that allows us to answer document listing queries efficiently. For any suffix tree node v, it holds that:

1. node v is sampled and thus set D_v is directly stored; or
2. $|\mathsf{SA}_v| < b$, and thus documents can be listed in time $\mathcal{O}(b \cdot \mathsf{lookup}(n))$ by using a CSA and the bitvectors B and V of Sect. 2.3; 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 sampled suffix tree.

The purpose of rule 2 is to ensure that suffix array intervals solved by brute force are not longer than b. The purpose of rule 3 is to ensure that, if we have to rebuild an answer by merging a list of answers precomputed at descendant sampled suffix tree nodes, then the

merging costs no more than β per result. That is, we can discard answers of nodes that are close to being the union of the answers of their descendant nodes, since we do not waste too much work in performing the unions of those descendants. Instead, if the answers of the descendants have many documents in common, then it is worth storing the answer at the node too; otherwise merging will require much work because the same document will be found many times (more than β on average).

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 SA_{v_i} of the selected nodes cover the entire suffix array. Given node v and its parent w, we select v if $|\mathsf{SA}_v| \leq b$ and $|\mathsf{SA}_w| > b$, and store D_v with the node. These nodes v become the leaves of the sampled suffix tree, and we assume that they are numbered from left to right. We then assume that all the ancestors of those leaves belong to the sampled suffix tree, and proceed upward in the suffix tree removing some of them. 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 sampled suffix tree, and store D_v there.

When the document collection is repetitive, the document array $\mathsf{DA}[1..n]$ is also repetitive. This property has been used in the past to compress DA using grammars (Navarro et al. 2014b). 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 $\mathsf{DA}[\ell..r]$ that corresponds to node v.

Let $v_1, \ldots, v_L$ be the leaf nodes and $v_{L+1}, \ldots, v_{L+I}$ the internal nodes of the sampled suffix tree. We use grammar-based compression to replace frequent subsets in sets $D_{v_1}, \ldots, D_{v_{L+I}}$ with grammar rules expanding to those subsets. Given a set Z and a grammar rule $X \to Y$, 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 $X \to Y$, 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 $\{v_1, \ldots, v_{L+I}\}$ and $\{1, \ldots, d\}$, with an edge from v_i to j if $j \in D_{v_i}$. Let $X \to Y$ be a grammar rule, and let V be the set of nodes v_i such that rule $X \to Y$ can be applied to set D_{v_i}. As $Y \subseteq D_{v_i}$ for all $v \in V$, the induced subgraph with vertex sets V and Y is a complete bipartite graph or a *biclique*. Many Web graph compression algorithms are based on finding bicliques or other dense subgraphs (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 $D_{v_1}, \ldots, D_{v_{L+I}}$ 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 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 $A[\mathsf{select}(B_A, i)..\mathsf{select}(B_A, i+1) - 1]$. The bitvector takes $|A|(1 + o(1))$ bits of space and answers select queries in $\mathcal{O}(1)$ time. 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 sampled suffix tree. A bitvector $\mathrm{B}_L[1..n]$ marks the first cell of interval SA_{v_i} for all leaf nodes v_i, allowing us to convert interval $\mathsf{SA}[\ell..r]$ into a range of nodes $[ln..rn] = [\mathsf{rank}_1(B_L, \ell)..\mathsf{rank}_1(B_L, r+1) - 1]$. Using the format of Okanohara and Sadakane (2007) for B_L, the bitvector takes $L \lg(n/L) + \mathcal{O}(L)$ bits, and answers rank queries in $\mathcal{O}(\lg(n/L))$ time and select queries in constant time. A second bitvector $B_F[1..L + I]$, using $(L + I)(1 + o(1))$ bits and supporting rank queries in constant time, 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 $j = L + F[\mathsf{rank}_1(B_F, i)]$. 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 $\mathcal{O}((r+1-\ell)\,\mathsf{lookup}(n))$ time, set(i) takes $\mathcal{O}(|D_{v_i}|)$ time, and parent(i) takes $\mathcal{O}(1)$ time. Function decompress(ℓ, r) produces set res in time $\mathcal{O}(|res| \cdot \beta h)$, where h is the height of the sampled suffix tree: finding each set may take $\mathcal{O}(h)$ time, and we may encounter the same document $\mathcal{O}(\beta)$ times. Hence the total time for listDocuments(ℓ, r) is $\mathcal{O}(\mathsf{df} \cdot \beta h + \lg n)$ for unions of precomputed answers, and $\mathcal{O}(b \cdot \mathsf{lookup}(n))$ otherwise. If the text follows the A2 model of Szpankowski (1993), then $h = \mathcal{O}(\lg n)$ and the total time is on average $\mathcal{O}(\mathsf{df} \cdot \beta \lg n + b \cdot \mathsf{lookup}(n))$.

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 β. In a bad case like $T = a^{\ell-1}\$b^{\ell-1}\$c^{\ell-1}\$\ldots$, the suffix tree is formed by d long paths and the sampled suffix tree contains at least $d(n/d - b) = \Theta(n)$ nodes (assuming $bd = o(n)$), so the total space is $\mathcal{O}(n \lg n)$ bits as in a classical suffix tree. In a good case, such as a balanced suffix tree (which also arises on texts following the A2 model), the sampled suffix tree has $\mathcal{O}(n/b)$ nodes. Although each such node v may store a list D_v with b entries, many of those entries are similar when the collection is repetitive, and thus their compression is effective.

```
function listDocuments(ℓ, r)
    (res, ln) ← (∅, rank_1(B_L, ℓ))
    if select(B_L, ln) < ℓ:
        r' ← min(select(B_L, ln + 1) − 1, r)
        (res, ln) ← (list(ℓ, r'), ln + 1)
        if r' = r: return res
    rn ← rank_1(B_L, r + 1) − 1
    if select(B_L, rn + 1) ≤ r:
        ℓ' ← select(B_L, rn + 1)
        res ← res ∪ list(ℓ', r)
    return res ∪ decompress(ln, rn)

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

function parent(i)
    par ← F[rank_1(B_F, i)]
    return (par + L, N[par])

function set(i)
    res ← ∅
    ℓ ← select(B_A, i)
    r ← select(B_A, i + 1) − 1
    for j ← ℓ to r:
        if A[j] ≤ d: res ← res ∪ {A[j]}
        else: res ← res ∪ rule(A[j] − d)
    return res

function rule(i)
    ℓ ← select(B_G, i)
    r ← select(B_G, i + 1) − 1
    return G[ℓ..r]

function list(ℓ, r)
    res ← ∅
    for i ← ℓ to r:
        res ← res ∪ {rank_1(B, SA[i])}
    return res
```

Fig. 4 Document listing using precomputed answers. Function listDocuments(ℓ, r) lists the documents from interval $\mathsf{SA}[\ell..r]$; decompress(ℓ, r) decompresses the sets stored in nodes $v_\ell, \ldots, v_r$; 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 ith grammar rule; and list(ℓ, r) lists the documents from interval $\mathsf{SA}[\ell..r]$ by using CSA and bitvector B

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 in decreasing order of their "frequencies", that is, the number of times the string represented by v appears in the documents. Ties are 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 need, instead, 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 $\mathcal{O}(\sqrt{s})$ distinct frequencies, and the frequencies can be encoded in $\mathcal{O}(\sqrt{s}\lg s)$ bits.

There are two basic approaches to using the PDL structure for top-k document retrieval. First, 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, and it works well with repetitive collections thanks to the grammar-compression of the lists. Note that this enables *incremental top-k queries*, where value k is not given beforehand, but we extract documents with successively lower scores and can stop at any time. Note also that, in this version, it is not necessary to store the frequencies.

Alternatively, we can build the PDL structure as in Sect. 4.1, with some parameter β, to achieve better space usage. Answering queries will then be slower as we have to decompress multiple 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 $2n + o(n)$ 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 unrelated 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 (i.e., the suffix tree node arrived at when searching for 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 D_v be again the set of distinct document identifiers in the corresponding range $\mathsf{DA}[\ell..r]$, and let $\mathsf{count}(v) = |D_v|$ be the size of that set. If node v has children u and w, we define the number of *redundant* suffixes as $h(v) = |D_u \cap D_w|$. This allows us to compute df recursively: $\mathsf{count}(v) = \mathsf{count}(u) + \mathsf{count}(w) - h(v)$. By using the leaf nodes descending from v, [ℓ..r], as base cases, we can solve the recurrence:

$$\mathsf{count}(v) = \mathsf{count}(\ell, r) = (r + 1 - \ell) - \sum_u h(u),$$

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

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

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 ℓ and r:

$$\mathsf{count}(\ell, r) = 2(r - \ell) - (\mathsf{select}_1(H', r) - \mathsf{select}_1(H', \ell)) + 1.$$

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 $\mathsf{count}()$ 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 symbol a, the subtrees of the binary suffix tree corresponding to patterns x and ax have an 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 $\mathsf{count}(v) > 1$. Given a range $[\ell..r-1]$ of nodes in the binary suffix tree, the corresponding subtree of the pruned tree is $[\mathsf{rank}_1(F,\ell)..\mathsf{rank}_1(F,r-1)]$. 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 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 instead subtract the number of 1s in $F_1[\ell..r-1]$ from the result of the query. 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 r, over an alphabet of size σ. We call string S *unique*, if it occurs at most once in every document. The subtree of the binary 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 a given document is at most $r^2/(2\sigma^{2k})$, 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 symbols. By the union bound, the probability the string is non-unique is at most $dr^2/(2\sigma^{2k})$. Let $N(i)$ be the number of non-unique strings of length $k_i = \lg_\sigma(r\sqrt{d}) + i$. As there are σ^{k_i} strings of length k_i, the expected value of $N(i)$ is at most $r\sqrt{d}/(2\sigma^i)$. The expected size of the smallest cover of unique strings is therefore at most

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

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 (dr). See Fig. 5 for an experimental confirmation of this analysis.

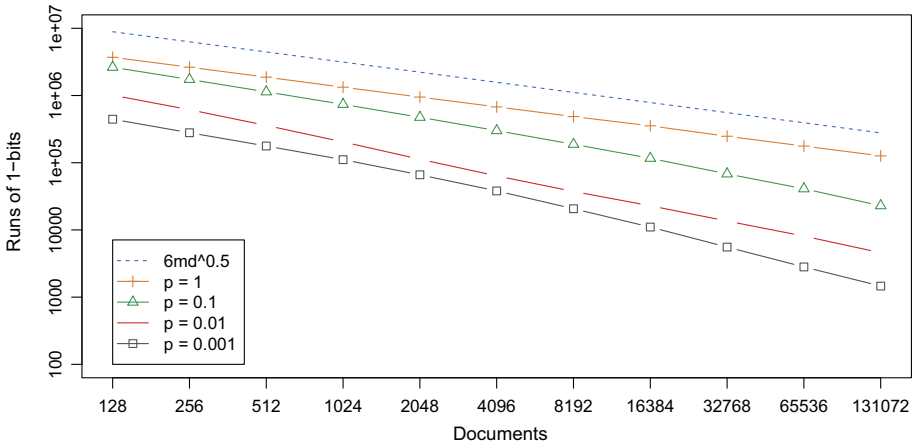

Fig. 5 The number of runs of 1-bits in Sadakane's bitvector H' on synthetic collections of DNA sequences ($\sigma = 4$). Each collection has been generated by taking a random sequence of length $m = 2^7$–2^{17}, duplicating it $d = 2^{17}$–2^7 times (making the total size of the collection 2^{24}), and mutating the sequences with random point mutations at probability $p = 0.001$–1. The mutations preserve zero-order empirical entropy by replacing the mutated symbol with a randomly chosen symbol according to the distribution in the original sequence. The *dashed line* represents the expected case upper bound for $p = 1$

6 A multi-term index

The queries we defined in the Introduction are *single-term*, that is, the query pattern P is a single string. In this section 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 algorithm of Sect. 4.2 as an *abstract representation* 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 (Sects. 3.4 or 5).

Let $Q = \langle q_1, \ldots, q_m \rangle$ be a query consisting of m patterns q_i. We support ranked queries, which return the k documents with the highest scores among the documents matching the query. A *disjunctive* or *ranked-OR* query matches document D if at least one of the patterns occurs in it, while a *conjunctive* or *ranked-AND* query matches D if all query patterns occur in it. Our index supports both conjunctive and disjunctive queries with tf-idf-like scores

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

where $f \geq 0$ is an increasing function, $\mathsf{tf}(D, q_i)$ is the *term frequency* (the number of occurrences) of pattern q_i in document D, $g \geq 0$ is a decreasing function, and $\mathsf{df}(q_i)$ is the *document frequency* of pattern q_i. For example, the standard tf-idf scoring scheme corresponds to using $f(\mathsf{tf}) = \mathsf{tf}$ and $g(\mathsf{df}) = \lg(d/\max(\mathsf{df}, 1))$.

From Sect. 4.2, we use the incremental variant, which stores the full answers for all the suffix tree nodes above leaves. The query algorithm uses CSA to find the lexicographic range $[\ell_i..r_i]$ matching each pattern q_i. We then use PDL to find the sparse suffix tree node

v_i corresponding to range $[\ell_i..r_i]$ and fetch its list D_{v_i}, which is stored in decreasing term frequency order. If v_i is not in the sparse suffix tree, we use instead the CSA to build D_{v_i} by brute force from $\mathsf{SA}[\ell_i..r_i]$. We also compute $\mathsf{df}(q_i) = \mathsf{count}(v_i)$ for all query patterns q_i with our document counting structure. 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 pattern q_i.
2. If the query is conjunctive, filter out extracted documents that do not match the query patterns 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 $\mathsf{tf}(D', q_i)$, where D' is the next unextracted document for pattern q_i, as an upper bound for $\mathsf{tf}(D, q_i)$.
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 further.
6. If the query is conjunctive, stop if the top-k documents match all query patterns and the upper bounds for the remaining documents are lower than the lower bounds for the top-k documents.

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

7 Experiments and discussion

7.1 Experimental setup

7.1.1 Document collections

We performed extensive experiments with both real and synthetic collections.[7] Most of our document collections were relatively small, around 100 MB in size, as some of the implementations (Navarro et al. 2014b) 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 more important than collection size, as a 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 fair number of real and synthetic collections with different characteristics for our experiments. We describe them next, and summarize their statistics in Table 2.

A note on collection size The index structures evaluated in this paper should be understood as promising algorithmic ideas. In most implementations, the construction algorithms do not scale up for collections larger than a couple of gigabytes. This is often

[7] See http://jltsiren.kapsi.fi/rlcsa for the datasets and full results.

Table 2 Statistics for document collections (small, medium, and large variants)

Collection	Size (n) (MB)	CSA size (RLCSA) (MB)	Documents (d)	Avg. doc size (n / d)	Patterns	Occurrences ($\overline{occ}$)	Document occs ($\overline{\mathsf{df}}$)	Occs per doc ($\overline{occ/\mathsf{df}}$)
Page	110	2.58	60	1,919,382	7658	781	3	242.75
	641	9.00	190	3,534,921	14,286	2601	6	444.79
	1037	17.45	280	3,883,145	20,536	2889	7	429.04
Revision	110	2.59	8834	13,005	7658	776	371	2.09
	640	9.04	31,208	21,490	14,284	2592	1065	2.43
	1035	17.55	65,565	16,552	20,536	2876	1188	2.42
Enwiki	113	49.44	7000	16,932	18,935	1904	505	3.77
	639	309.31	44,000	15,236	19,628	10,316	2856	3.61
	1034	482.16	90,000	12,050	19,805	17,092	4976	3.44
Influenza	137	5.52	100,000	1436	1000	24,975	18,547	1.35
	321	10.53	227,356	1480	1000	59,997	44,012	1.36
Swissprot	54	25.19	143,244	398	10,000	160	121	1.33
Wiki	1432	42.90	103,190	14,540				
DNA	95		100,000		889–1000			
Concat	95		10–1000		7538–15,272			
Version	95		10,000		7537–15,271			

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 (**DNA**, **Concat**, and **Version**), most of the statistics vary greatly

intentional. In this line of research, being able to easily evaluate variations of the fundamental idea is more important than the speed or memory usage of construction. As a result, many of the construction algorithms build an explicit suffix tree for the collection and store various kinds of additional information in the nodes. Better construction algorithms can be designed once the most promising ideas have been identified. See "Appendix 2" for further discussion on index construction.

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 8834, 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 7000, 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. 2014b). 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 & Chili corpus.[8] **DNA** is similar to **Influenza**. Each collection has $d = 1$, 10, 100, or 1000 base documents, 100,000/d variants of each base document, and mutation rate $p = 0.001$, 0.003, 0.01, 0.03, or 0.1. We take a prefix of length 1000 from the Pizza & Chili DNA file and generate the base documents by mutating the prefix at probability $10p$ under the same model as in Fig. 5. 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, or 1000 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, 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**.

7.1.2 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 ≥ 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 1000 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[9] consisting of 411,394 terms in 100,000 queries.

[8] http://pizzachili.dcc.uchile.cl.

[9] http://trec.nist.gov/data/terabyte06.html.

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, filtered out duplicates, and chose the 1000 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**.

7.1.3 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.

We executed the query benchmarks in the following way:

1. Load the RLCSA with the desired sample period for the current collection into memory.
2. Load the query patterns corresponding to the collection into memory and execute find queries in the RLCSA. Store the resulting lexicographic ranges $[\ell..r]$ in vector V.
3. Load the index to be benchmarked into memory.
4. Iterate through vector V once using a single thread and execute the desired query for each range $[\ell..r]$. Measure the total wall clock time for executing the queries.

We divided the measured time by the number of patterns, and listed the average time per query in milliseconds or microseconds and the size of the index structure in bits per symbol. There were certain exceptions:

- **LZ** and **Grammar** do not use a CSA. With them, we iterated through the vector of patterns as in step 4, once the index and the patterns had been loaded into memory. The average time required to get the range $[\ell..r]$ in CSA-based indexes (4−6 μs, depending on the collection) was negligible compared to the average query times of **LZ** (at least 170 μs) and **Grammar** (at least 760 μs).
- We used the existing benchmark code with **SURF**. The code first loads the index into memory and then iterates through the pattern file by reading one line at a time. To reduce the overhead from reading the patterns, we cached them by using `cat> /dev/null`. Because **SURF** queries were based on the pattern instead of the corresponding range $[\ell..r]$, we executed find queries first and subtracted the time used for them from the subsequent top-k queries.
- In our tf-idf index, we parallelized step 4 using the OpenMP `parallel for` construct.
- We used the existing benchmark code with Terrier. We cached the queries as with **SURF**, set `trec.querying.outputformat` to `NullOutputFormat`, and set the logging level to `off`.

7.2 Document listing

We compare our new proposals from Sects. 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.

7.2.1 Indexes

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

Sadakane (**Sada**) This family of algorithms is based on the improvements of Sadakane (2007) to the algorithm of Muthukrishnan (2002). **Sada-L** is the original algorithm, while **Sada-D** uses an explicit document array DA instead of retrieving the document identifiers with locate.

ILCP (**ILCP**) This is our proposal in Sect. 3.3. The algorithms are the same as those of Sadakane (2007), but they run on the run-length encoded ILCP array. As for **Sada**, **ILCP-L** obtains the document identifiers using locate on the CSA, whereas **ILCP-D** stores array DA explicitly.

Wavelet tree (**WT**) This index stores the document array in a wavelet tree (Sect. 2.2) to efficiently find the distinct elements in $\mathsf{DA}[\ell..r]$ (Välimäki and Mäkinen 2007). The best known implementation of this idea (Navarro et al. 2014b) 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. 2014b), 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 (**PDL**) This is our proposal in Sect. 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[10] and stores the dictionary in plain form. We use block size $b = 256$ and storing factor $\beta = 16$, which have proved to be good general-purpose parameter values.

Grammar-based (**Grammar**) This index (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**) This index (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.

[10] http://www.dcc.uchile.cl/gnavarro/software.

We implemented **Brute**, **Sada**, ILCP, and the **PDL** variants ourselves[11] 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 **ILCP-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 average time required for obtaining the ranges was 4–6 µs per pattern, depending on the collection, which is negligible compared to the average time used by **Grammar** (at least 760 µs) and **LZ** (at least 170 µs).

7.2.2 Results

Real collections Figures 6 and 7 contain the results for document listing with small and large real collections, respectively. 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 $occ/\mathsf{df} < 4$ on the average. The additional effort made by **Sada-L** and **ILCP-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. **ILCP-L** uses less space than **Sada-L** when the collection is repetitive and contains many similar documents (i.e., on **Revision** and **Influenza**); otherwise **Sada-L** is slightly smaller.

The two **PDL** alternatives usually achieve similar performance, but in some cases **PDL-BC** uses much less space. **PDL-BC**, in turn, can use significantly more space than **Brute-L**, **Sada-L**, and **ILCP-L**, but is always orders of magnitude faster. The document sets of versioned collections such as **Page** and **Revision** are very compressible, making the collections very suitable for **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 **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, **Sada-D** and **ILCP-D** can be significantly faster than **PDL-BC**, but they also use much more space. From the non-CSA-based indexes, **Grammar** reaches the Pareto-optimal curve on **Revision** and **Influenza**, while being too slow or too large on the other collections. We did not build **Grammar** for the large version of **Page**, as it would have taken several months.

In general, we can recommend **PDL-BC** as a medium-space alternative for document listing. When less space is available, we can use **ILCP-L**, which offers robust time and space guarantees. If the documents are small, we can even use **Brute-L**. Further, we can

[11] http://jltsiren.kapsi.fi/rlcsa.

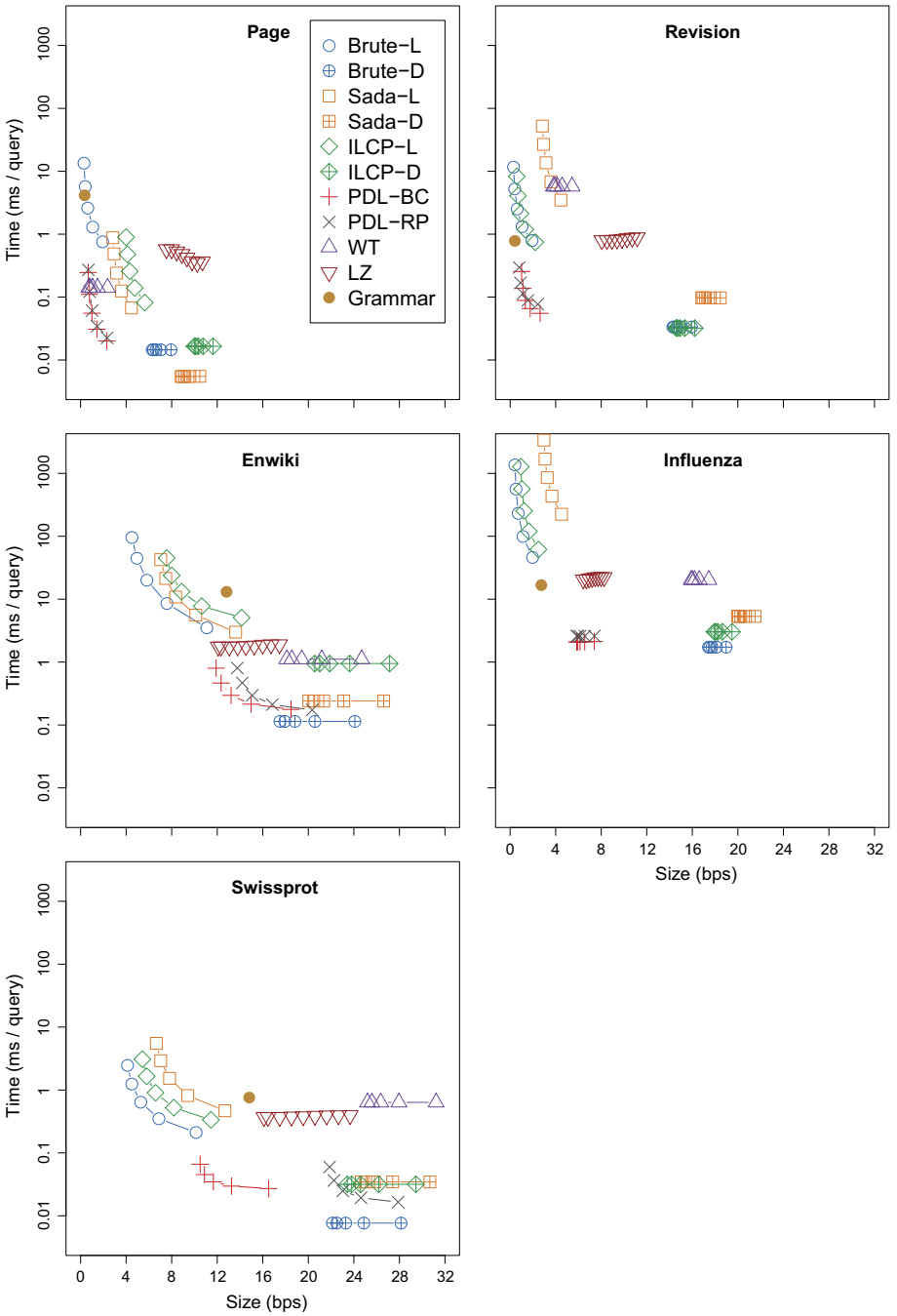

Fig. 6 Document listing on small real collections. The total size of the index in bits per symbol (x) and the average time per query in milliseconds (y)

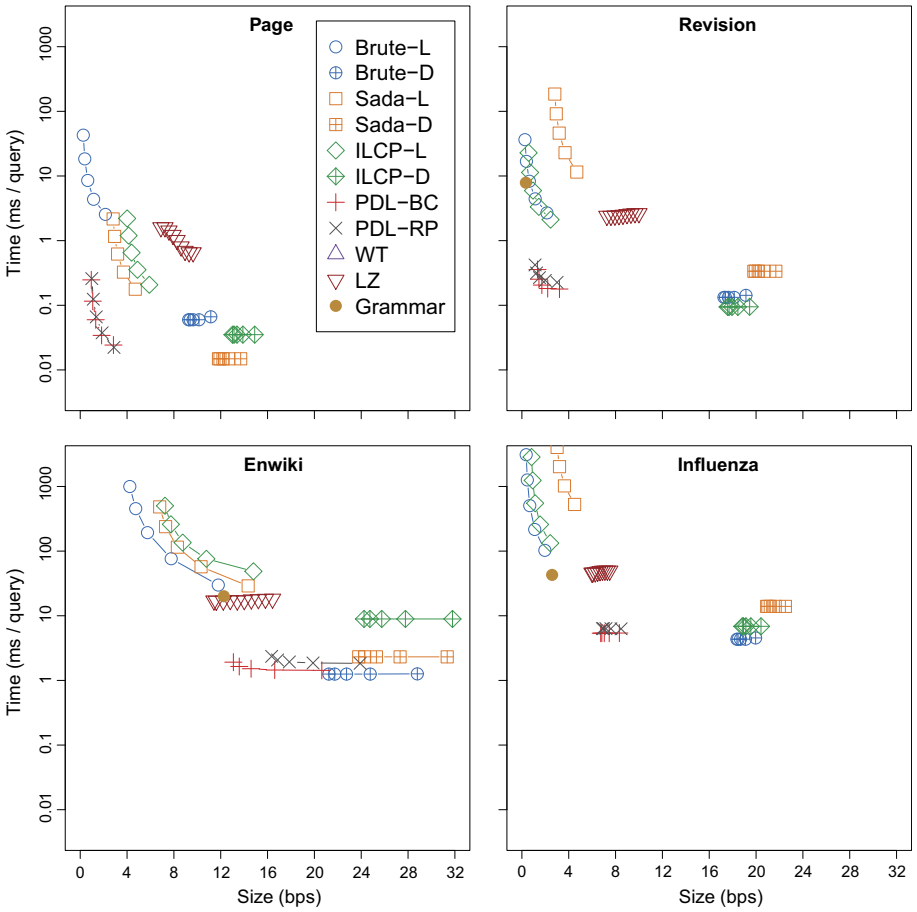

Fig. 7 Document listing on large real collections. The total size of the index in bits per symbol (x) and the average time per query in milliseconds (y)

use fast document counting to compare df with $occ = r - \ell + 1$, and choose between **ILCP-L** and **Brute-L** according to the results.

Synthetic collections Figures 8 and 9 show 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 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 **Sada-L** and **ILCP-L**) were left out. When **PDL-BC** and **PDL-RP** had a very similar performance, only one of them was chosen for the plot.

On **DNA**, **Grammar** was a good solution for small mutation rates, while **LZ** was good with larger mutation rates. With more space available, **PDL-BC** became the fastest algorithm. **Brute-D** and **ILCP-D** were often slightly faster than **PDL**, when there was enough space available to store the document array. On **Concat** and **Version**, **PDL** was

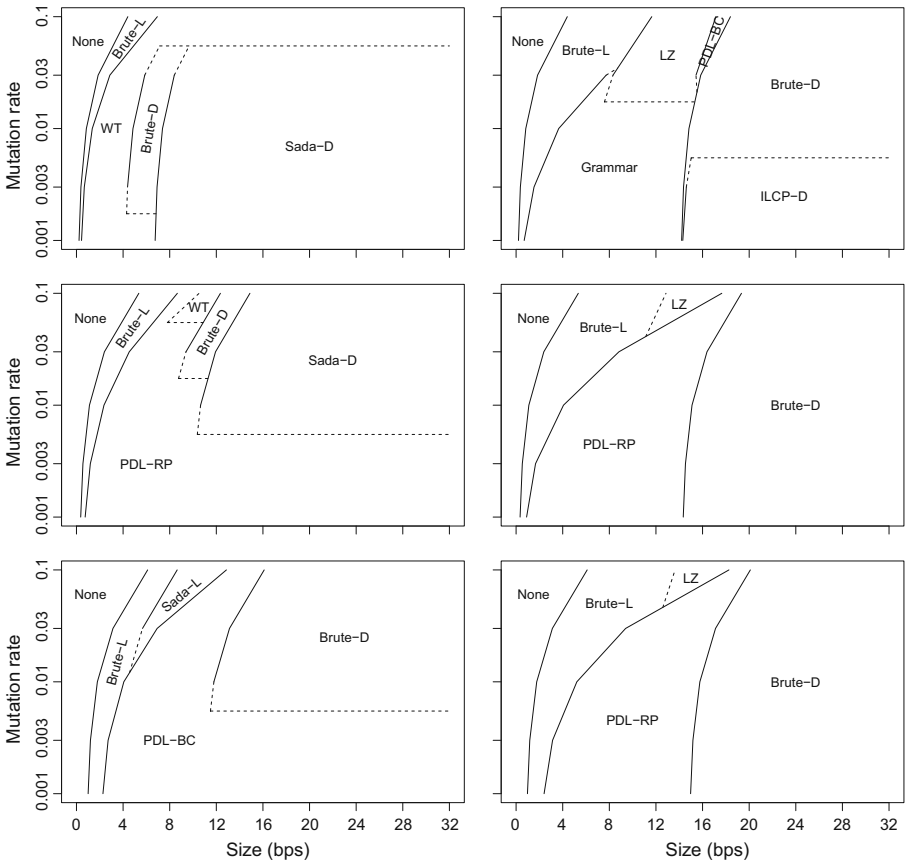

Fig. 8 Document listing on synthetic collections. The fastest solution for a given size in bits per symbol and a mutation rate. From *top* to *bottom*: 10, 100, and 1000 base documents with **Concat** (*left*) and **Version** (*right*). **None** denotes that no solution can achieve that size

usually a good mid-range solution, with **PDL-RP** being usually smaller than **PDL-BC**. The exceptions were the collections with 10 base documents, where the number of variants (1000) was clearly larger than the block size (256). With no other structure in the collection, **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 DA were usually the fastest choices.

7.3 Top-*k* retrieval

7.3.1 Indexes

We compare the following top-*k* retrieval algorithms. Many of them share names with the corresponding document listing structures described in Sect. 7.2.1.

Brute force (**Brute**) These algorithms correspond to the document listing algorithms **Brute-D** and **Brute-L**. To perform top-*k* retrieval, we not only collect the distinct

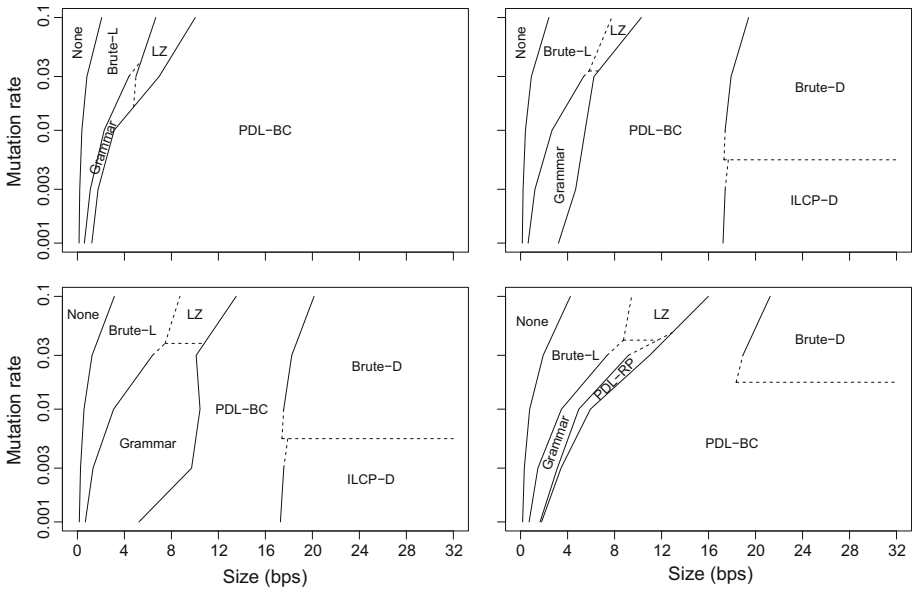

Fig. 9 Document listing on synthetic collections. The fastest solution for a given size in bits per symbol and a mutation rate. **DNA** with 1 (*top left*), 10 (*top right*), 100 (*bottom left*), and 1000 (*bottom right*) base documents. **None** denotes that no solution can achieve that size

document identifiers after sorting $\mathsf{DA}[\ell..r]$, we also record the number of times each one appears. The k identifiers appearing most frequently are then reported.

Precomputed document lists (**PDL**) We use the variant of **PDL-RP** modified for top-k retrieval, as described in Sect. 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.

Large and fast (**SURF**) This index (Gog and Navarro 2015b) is based on a conceptual idea by Navarro and Nekrich (2012), and improves upon a previous implementation (Konow and Navarro 2013). 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, **SURF** uses a variant of **Sada-L** to find them.

We implemented the **Brute** and **PDL** variants ourselves[12] and used the existing implementation of **SURF**.[13] While **WT** (Navarro et al. 2014b) also supports top-k queries, the 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 $[\ell..r]$ 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.

7.3.2 Results

Figure 10 contains the results for top-k retrieval using the large versions of the real collections. We left **Page** out of the results, as the number of documents (280) was too low for

[12] http://jltsiren.kapsi.fi/rlcsa.

[13] http://github.com/simongog/surf/tree/single_term.

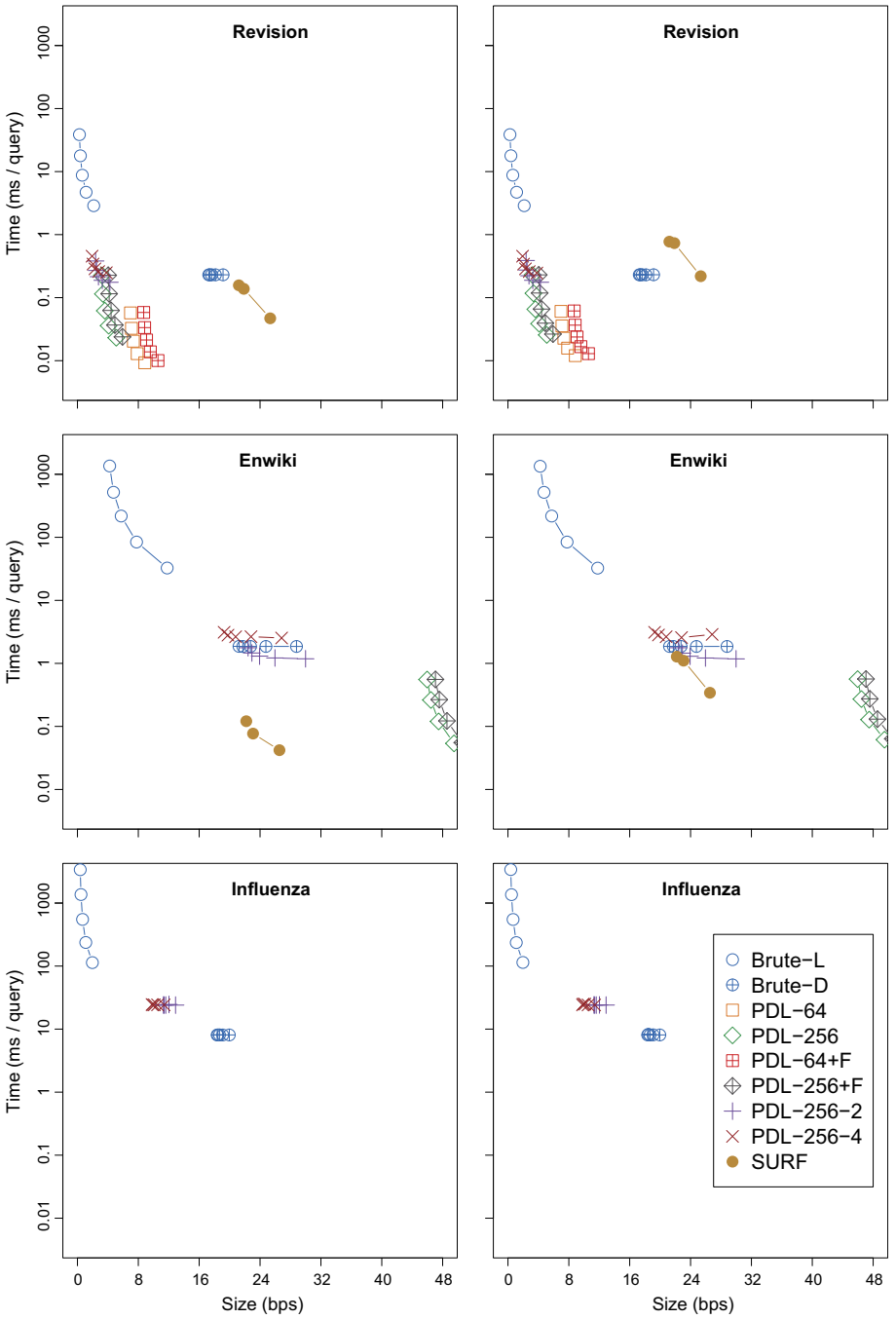

Fig. 10 Single-term top-k retrieval on real collections with $k = 10$ (*left*) and $k = 100$ (*right*). The total size of the index in bits per symbol (x) and the average time per query in milliseconds (y)

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 β 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 slow with $k = 100$.

On Enwiki, the variants of **PDL** with storing factor β set had a performance similar 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, its time performance became competitive for $k = 100$, as it was almost unaffected by the number of documents requested.

The third collection, **Influenza**, was the most surprising of the three. **PDL** with storing factor β 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 with this dataset.

7.4 Document counting

7.4.1 Indexes

We use two fast document listing algorithms as baseline document counting methods (see Sect. 7.2.1): **Brute-D** sorts the query range $\mathsf{DA}[\ell..r]$ 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 Sect. 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 δ-codes to represent run lengths and packs them into blocks of 32 bytes of encoded data. Each block stores how many bits and 1s are there before it.
- **Sada-RS** uses a run-length encoded bitvector, represented with a sparse bitmap (Okanohara and Sadakane 2007) marking the beginnings of the 0-runs and another for the 1-runs.
- **Sada-RD** uses run-length encoding with δ-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-Gr** 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 Sect. 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.[14]

7.4.2 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 $[\ell..r]$ 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 it was almost always the fastest method, we scaled the plots to leave out anything much larger than plain **Sada**. The results can be seen in Fig. 11. Table 5 in "Appendix 1" lists the results in further detail.

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-Gr** 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-Gr** can compress the bitvector clearly below 1 bit per symbol, and **Sada-Gr** is much slower than the other two. At around 1 bit per symbol, **Sada-S** is again the fastest option. Plain **Sada** requires twice as much space as **Sada-S**, but is also 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** was 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, which 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**.

7.5 The multi-term tf-idf index

We implement our multi-term index as follows. We use RLCSA as the CSA, **PDL-256+F** for single-term top-k retrieval, and **Sada-S** for document counting. We could have

[14] http://jltsiren.kapsi.fi/rlcsa and http://github.com/ahartik/succinct.

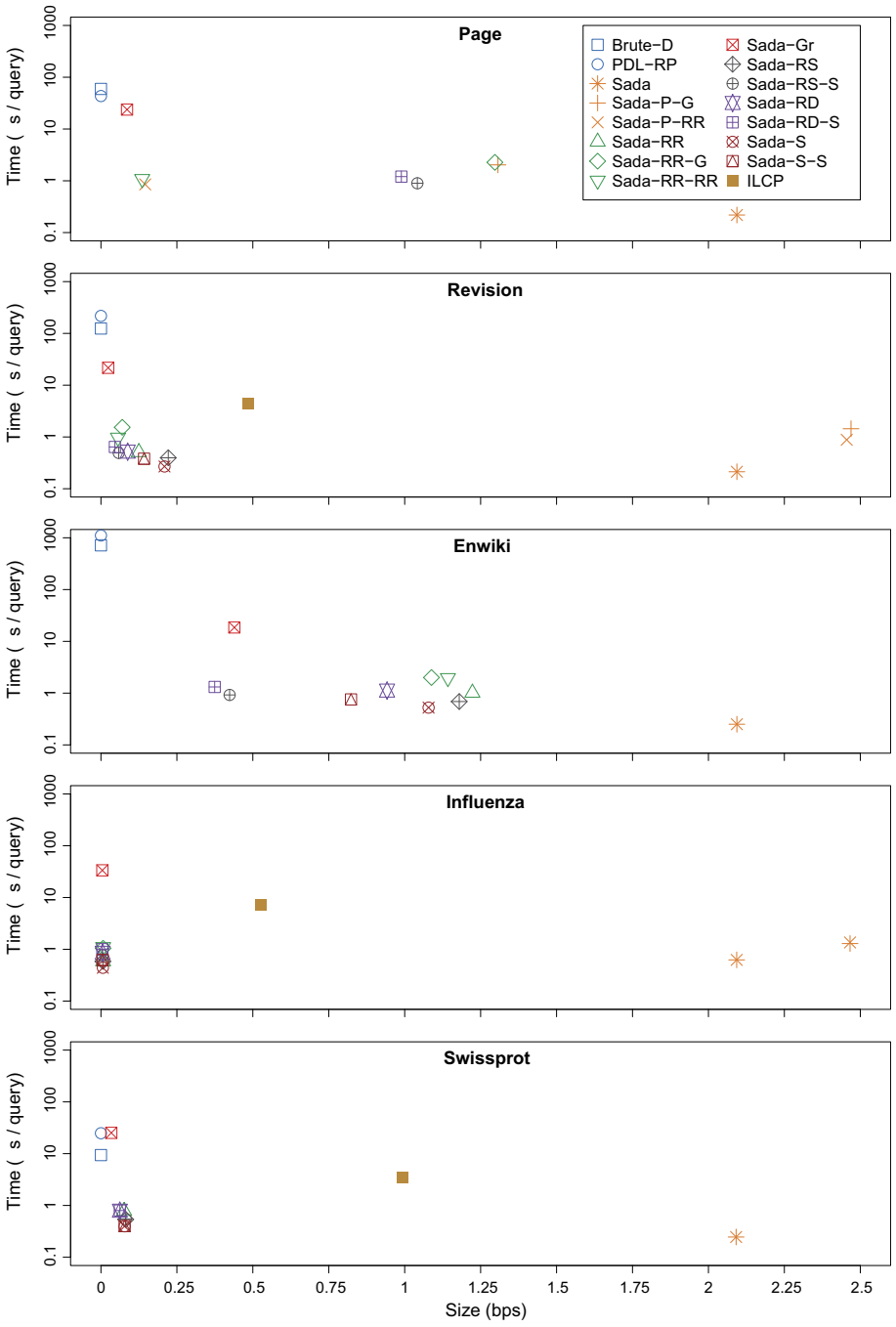

Fig. 11 Document counting on different datasets. The size of the counting structure in bits per symbol (x) and the average query time in microseconds (y). The baseline document listing methods are presented as having size 0, as they take advantage of the existing functionalities in the index

Table 3 Ranked multi-term queries on the **Wiki** collection

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

Query type, number of documents requested, and the average number of queries per second with 1, 8, 16, and 32 query threads

Table 4 Our index (**PDL**) and an inverted index (**Terrier**) on the **Wiki** collection

Index	Vocabulary	Posting lists	Collection	Size (MB)	Queries/s	
PDL	39.2M substrings	8840M documents	1500M symbols	757	229 ($k = 10$)	163 ($k = 100$)
Terrier	0.134M tokens	42.3M documents	133M tokens	90.1	231 ($k = 10$)	228 ($k = 100$)

The size of the vocabulary, the posting lists, and the collection in millions of elements, the size of the index in megabytes, and the number of Ranked-OR queries per second with $k = 10$ or 100 using a single thread

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 sampled suffix tree (**PDL**). We use the basic tf-idf scoring scheme.

We tested the resulting performance on the 1432 MB **Wiki** collection. RLCSA took 0.73 bps with sample period 128 (the sample period did not have a significant impact on query performance), **PDL-256+F** took 3.37 bps, and **Sada-S** took 0.13 bps, for a total of 4.23 bps (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 3. When using a single query thread, the index can process 136–229 queries per second (around 4–7 ms 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. Note that our ranked disjunctive query algorithm preempts the processing of the lists of the patterns, whereas in the conjunctive ones we are forced to expand the full document lists for all the patterns; this is why the former are faster. The speedup from using 32 threads is around 18x.

Since our multi-term index offers a functionality similar to basic inverted index queries, it seems sensible to compare it to an inverted index designed for natural language texts. For this purpose, we indexed the **Wiki** collection using **Terrier** (Macdonald et al. 2012) version 4.1 with the default settings. See Table 4 for a comparison between the two indexes.

Note that the similarity in the functionality is only superficial: our index can find *any text substring*, whereas the inverted index can only look for indexed *words and phrases*. Thus our index has an index point per symbol, whereas **Terrier** has an index point per word (in addition, inverted indexes usually discard words deemed uninteresting, like stopwords). Note that **PDL** also chooses frequent strings and builds their lists of documents, but since it has many more index points, its posting lists are 200 times longer than

those of **Terrier**, and the number of lists is 300 times larger. Thanks to the compression of its lists, however, **PDL** uses only 8 times more space than **Terrier**. On the other hand, both indexes have similar query performance. When logging and output was set to minimum, **Terrier** could process 231 top-10 queries and 228 top-100 queries per second under the tf-idf scoring model using a single query thread.

8 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 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.

We do not aim to outperform inverted indexes on natural language text collecions, where they are unbeatable, but rather to offer similar capabilities on generic string collections, where inverted indexes cannot be applied. Our developments are at the level of algorithmic ideas and prototypes. In order to have our most promising structures scale up to real-world information systems, where inverted indexes are now the norm, various research problems must be faced:

1. Our construction algorithms scale up to a few gigabytes. This limits the collection sizes we can handle, even if they are repetitive and thus the final structures are much smaller. For example, our PDL structure first builds the classical suffix tree and then samples it. Using construction space proportional to that of the final structures in the case of repetitive scenarios, or building efficiently using the disk, is an important research problem.
2. When the datasets are sufficiently large, even the compressed structures will have to operate on disk. Inverted indexes are extremely disk-friendly, which makes them perform well on huge text collections. We have not yet studied this aspect of our structures, although PDL seems well-suited to this case: it traverses one or a few contiguous lists (which should be decompressed in main memory) or a contiguous area of the suffix array.
3. Our data structures are static, that is, they must be rebuilt from scratch when documents are inserted in the collection or deleted from it. Inverted indexes tolerate updates much better, though they are not fully dynamic either. Instead, since in many scenarios updates are not so frequent, popular solutions combine a large part of the collection that is indexed and a small recent part that is traversed sequentially. It is likely that our structures will also perform well under such a scheme, as long as we manage to rebuild the index periodically within controlled space and time.
4. We showed that our structures can handle multi-term queries under the simple tf-idf scoring scheme. While this can be acceptable in some applications for generic string collections, information retrieval on natural language texts uses, nowadays, much more sophisticated formulas. Inverted indexes have been adapted to successfully

support those formulas that are used for a first filtration step, such as BM25. Studying how to extend our indexes to handle these is another interesting research problem.

5. One point where our indexes could outperform inverted indexes is in phrase queries, where inverted indexes must perform costly list intersections. Our suffix-array based indexes, instead, need not do anything special. For a fair comparison, we should regard the text as a sequence of tokens (i.e., the terms that are indexed by the inverted index) and build our indexes on them. The resulting structure would then only answer term and phrase queries, just like an inverted index, but would be must faster at phrases.

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 Coordination in Networks (ICM/FIC P10-024F), Chile; Basal Funds FB0001, Conicyt, Chile; and European Unions Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie Grant Agreement No. 690941. Finally, we thank the reviewers for their useful comments, which helped improve the presentation, and Meg Gagie for correcting our grammar.

Appendix 1: Detailed results

Table 5 shows the precise numerical results displayed in Fig. 11, to allow for a finer-grained comparison.

Table 5 Document counting on different datasets

	Page	Revision	Enwiki	Influenza	Swissprot
Brute-D	59.419 µs	**124.286** µs	**714.481** µs	**4557.310** µs	**9.392** µs
	0.000 b	**0.000** b	**0.000** b	**0.000** b	**0.000** b
PDL-RP	**43.356** µs	217.804 µs	1107.470 µs	6221.610 µs	24.848 µs
	0.000 b	0.000 b	0.000 b	0.000 b	0.000 b
Sada	**0.218** µs	**0.213** µs	**0.250** µs	0.624 µs	**0.246** µs
	2.094 b	**2.094** b	**2.094** b	2.093 b	**2.091** b
Sada-P-G	2.030 µs	1.442 µs	1.608 µs	1.291 µs	–
	1.307 b	2.469 b	2.694 b	2.466 b	–
Sada-P-RR	**0.852** µs	0.882 µs	1.572 µs	1.356 µs	–
	0.146 b	2.455 b	2.748 b	2.466 b	–
Sada-RR	1.105 µs	0.506 µs	1.013 µs	0.581 µs	**0.779** µs
	5.885 b	0.125 b	1.223 b	0.007 b	**0.076** b
Sada-RR-G	2.268 µs	1.535 µs	2.001 µs	1.046 µs	–
	1.297 b	0.070 b	1.088 b	0.007 b	–
Sada-RR-RR	**1.088** µs	0.974 µs	1.960 µs	1.108 µs	–
	0.136 b	0.056 b	1.142 b	0.007 b	–
Sada-Gr	23.750 µs	**21.643** µs	18.542 µs	33.502 µs	**25.236** µs
	0.086 b	**0.024** b	0.439 b	0.005 b	**0.034** b

Table 5 continued

	Page	Revision	Enwiki	Influenza	Swissprot
Sada-RS	0.742 µs	0.396 µs	0.688 µs	0.584 µs	0.538 µs
	5.991 b	0.222 b	1.180 b	0.006 b	0.082 b
Sada-RS-S	0.897 µs	**0.492** µs	**0.923** µs	**0.767** µs	0.545 µs
	1.042 b	**0.059** b	**0.424** b	**0.005** b	0.082 b
Sada-RD	1.019 µs	0.521 µs	1.119 µs	0.856 µs	**0.792** µs
	3.717 b	0.088 b	0.942 b	0.006 b	**0.062** b
Sada-RD-S	1.205 µs	**0.641** µs	**1.316** µs	1.005 µs	0.799 µs
	0.989 b	**0.046** b	**0.374** b	0.005 b	0.062 b
Sada-S	0.604 µs	**0.269** µs	**0.525** µs	**0.439** µs	**0.396** µs
	5.729 b	**0.209** b	**1.079** b	**0.006** b	**0.078** b
Sada-S-S	0.735 µs	**0.380** µs	**0.755** µs	0.624 µs	0.399 µs
	3.432 b	**0.142** b	**0.823** b	0.006 b	0.078 b
ILCP	4.399 µs	4.482 µs	6.033 µs	7.252 µs	3.414 µs
	18.454 b	0.484 b	4.575 b	0.525 b	0.992 b

The average query time in microseconds and the size of the counting structure in bits per symbol. Results on the Pareto frontier have been highlighted. The baseline document listing methods **Brute-D** and **PDL-RP** are presented as having size 0, as they take advantage of the existing functionalities in the index. We did not build **Sada-P-G**, **Sada-P-RR**, **Sada-RR-G**, and **Sada-RR-RR** for **Swissprot**, because the filter was empty and the remaining structure was equivalent to **Sada** or **Sada-RR**

Appendix 2: Index construction

Our construction algorithms prioritize flexibility over performance. For example, the construction of the tf-idf index (Sect. 6) proceeds as follows:

1. Build RLCSA for the collection.
2. Extract the LCP array and the document array from the RLCSA, traverse the suffix tree by using the LCP array, and build **PDL** with uncompressed document sets.
3. Compress the document sets using a Re-Pair compressor.
4. Build the **Sada-S** structure using a similar algorithm as for **PDL** construction.

See Table 6 for the time and space requirements of building the index for the **Wiki** collection.

Scaling the index up for larger collections requires faster and more space-efficient construction algorithms for its components. There are some obvious improvements:

Table 6 Building the tf-idf index for the **Wiki** collection

	RLCSA		**PDL**	Re-Pair	
Sada-S	Total				
Time (min)	10.5	39.2	123	74.7	248
Memory (GB)	19.6	111	202	92.8	202

Construction time in minutes and peak memory usage in gigabytes for RLCSA construction, **PDL** construction, compressing the document sets using Re-Pair, **Sada-S** construction, and the entire construction

- RLCSA construction can be done in less memory by building the index in multiple parts and merging the partial indexes (Sirén 2009). With 100 parts, the indexing of a repetitive collection proceeds at about 1 MB/s using 2–3 bits per symbol (Sirén 2012). Newer suffix array construction algorithms achieve even better time/space trade-offs (Kärkkäinen et al. 2015).
- We can use a compressed suffix tree for **PDL** construction. The SDSL library (Gog et al. 2014) provides fast scalable implementations that require around 2 bytes per symbol.
- We can write the uncompressed document sets to disk as soon as the traversal returns to the parent node.
- We can build the H array for **Sada-S** by keeping track of the lowest common ancestor of the previous occurrence of each document identifier and the current node. If node v is the lowest common ancestor of consecutive occurrences of a document identifier, we increment the corresponding cell of the H array. Storing the array requires about a byte per symbol.

The main bottleneck in the construction is Re-Pair compression. Our compressor requires 24 bytes of memory for each integer in the document sets, and the number of integers (8.9 billion) is several times larger than the number of symbols in the collection (1.5 billion). It might be possible to improve compression performance by using a specialized compressor. If interval $\mathsf{DA}[\ell..r]$ corresponds to suffix tree node u and the collection is repetitive, it is likely that the interval $\mathsf{DA}[\ell'..r']$ corresponding to the node reached by taking the suffix link from u is very similar to $\mathsf{DA}[\ell..r]$.

References

Anick, P. G., & Flynn, R. A. (1992). Versioning a full-text information retrieval system. In *Proceedings of the 15th annual international ACM conference on research and development in information retrieval (SIGIR)* (pp. 98–111).

Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern information retrieval* (2nd ed.). Reading: Addison-Wesley.

Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., & Raffinot, M. (2015). Composite repetition-aware data structures. In *Proceedings of the 26th annual symposium on combinatorial pattern matching (CPM)* (pp. 26–39).

Broder, A., Eiron, N., Fontoura, M., Herscovici, M., Lempel, R., McPherson, J., et al. (2006). Indexing shared content in information retrieval systems. In *Proceedings of the 10th international conference on extending database technology (EDBT)*, LNCS 3896 (pp. 313–330).

Büttcher, S., Clarke, C., & Cormack, G. (2010). *Information retrieval: Implementing and evaluating search engines*. Cambridge: MIT Press.

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

Claude, F., Fariña, A., Martínez-Prieto, M., & Navarro, G. (2010). Compressed q-gram indexing for highly repetitive biological sequences. In *Proceedings of the 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.

Claude, F., & Munro, I. (2013). Document listing on versioned documents. In *Proceedings of the 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 *Proceedings of the 19th international symposium on string processing and information retrieval (SPIRE)*, LNCS 7608 (pp. 180–192).

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

Do, H. H., Jansson, J., Sadakane, K., & Sung, W. K. (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 *Proceedings of the 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, S. J., (2012a) A faster grammar-based self-index. In *Proceedings of the 6th international conference on language and automata theory and applications (LATA)*, LNCS 7183 (pp. 240–251).

Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., & Puglisi, S. J. (2014). LZ77-based self-indexing with faster pattern matching. In *Proceedings of the 11th Latin American theoretical informatics symposium (LATIN)*, LNCS 8392 (pp. 731–742).

Gagie, T., Hartikainen, A., Kärkkäinen, J., Navarro, G., Puglisi, S. J., & Sirén, J. (2015). Document counting in compressed space. In *Proceedings of the 25th data compression conference (DCC)* (pp. 103–112).

Gagie, T., Karhu, K., Navarro, G., Puglisi, S. J., & Sirén, J. (2013). Document listing on repetitive collections. In *Proceedings of the 24th annual symposium on combinatorial pattern matching (CPM)*, LNCS 7922 (pp. 107–119).

Gagie, T., Navarro, G., & Puglisi, S. J. (2012b). New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, *426–427*, 25–41.

Gog, S., Beller, T., Moffat, A., & Petri, M. (2014). From theory to practice: Plug and play with succinct data structures. In *Proceedings of the 13th international symposium on experimental algorithms (SEA)*, LNCS 8504 (pp. 326–337).

Gog, S., & Navarro, G. (2015a). Improved single-term top-*k* document retrieval. In *Proceedings of the 17th workshop on algorithm engineering and experiments (ALENEX)* (pp. 24–32).

Gog, S., & Navarro, G. (2015b). Improved single-term top-*k* document retrieval. In *Proceedings of the 17th Workshop on Algorithm Engineering and Experiments (ALENEX)* (pp. 24–32).

Grossi, R., Gupta, A., & Vitter, J. (2003). High-order entropy-compressed text indexes. In *Proceedings of the 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 *Proceedings of the 35th international ACM 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 *Proceedings of the 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 *Proceedings of the 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, W. K., Patil, M., Shah, R., Thankachan, S. V., & Vitter, J. S. (2013). Indexes for document retrieval with relevance. *Space-Efficient Data Structures, Streams, and Algorithms, LNCS*, *8066*, 351–362.

Kärkkäinen, J., Kempa, D., & Puglisi, S. J. (2015). Parallel external memory suffix sorting. In *Proceedings of the 26th annual symposium on combinatorial pattern matching (CPM)*, LNCS 9133 (pp. 329–342).

Konow, R., & Navarro, G. (2013). Faster compact top-k document retrieval. In *Proceedings of the 23rd data compression conference (DCC)* (pp. 351–360).

Kreft, S., & Navarro, G. (2013). On compressing and indexing repetitive sequences. *Theoretical Computer Science*, *483*, 115–133.

Larsson, N. J., & Moffat, A. (2000). Off-line dictionary-based compression. *Proceedings of the IEEE*, *88*(11), 1722–1732.

Macdonald, C., McCreadie, R., Santos, R., & Ounis, I. (2012). From puppy to maturity: Experiences in developing Terrier. In *Proceedings of the SIGIR 2012 workshop in open source information retrieval* (pp. 60–63).

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. http://biorxiv.org/content/early/2016/03/29/043430.

Muthukrishnan, S. (2002). Efficient algorithms for document retrieval problems. In *Proceedings of the 13th annual ACM–SIAM symposium on discrete algorithms (SODA)* (pp. 657–666).

Navarro, G. (2004). Indexing text using the Ziv–Lempel trie. *Journal of Discrete Algorithms*, *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), article 2.

Navarro, G., & Nekrich, Y. (2012). Top-*k* document retrieval in optimal time and linear space. In *Proceedings of the 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 *Proceedings of the 21st international symposium on string processing and information retrieval (SPIRE)*, LNCS 8799 (pp. 31–44).

Navarro, G., Puglisi, S. J., & Sirén, J. (2014a). Document retrieval on repetitive collections. In *Proceedings of the 22nd annual european symposium on algorithms (ESA B)*, LNCS 8737 (pp. 725–736).

Navarro, G., Puglisi, S. J., & Valenzuela, D. (2014b). 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 *Proceedings of the 9th workshop on algorithm engineering and experiments (ALENEX)* (pp. 60–70).

Raman, R., Raman, V., & Rao, S.S. (2007). Succinct indexable dictionaries with applications to encoding *k*-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms, 3*(4), article 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.

Sirén, J. (2009). Compressed suffix arrays for massive data. In *Proceedings of the 16th symposium on string processing and information retrieval (SPIRE)*, LNCS 5721 (pp. 63–74).

Sirén, J. (2012). *Compressed full-text indexes for highly repetitive collections*. PhD thesis, University of Helsinki.

Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., et al. (2015). Big data: Astronomical or genomical? *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 *Proceedings of the 18th annual symposium on combinatorial pattern matching (CPM)*, LNCS 4580 (pp. 205–215).

Weiner, P. (1973). Linear pattern matching algorithm. In *Proceedings of the 14th annual IEEE symposium on switching and automata theory* (pp. 1–11).