Document Retrieval Hacks

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland

Bella Zhukova
Department of Computer Science, University of Helsinki, Finland

Abstract

Given a collection of strings, document listing refers to the problem of finding all the strings (or *documents*) where a given query string (or *pattern*) appears. Index data structures that support efficient document listing for string collections have been the focus of intense research in the last decade, with dozens of papers published describing exotic and elegant compressed data structures. The problem is now quite well understood in theory and many of the solutions have been implemented and evaluated experimentally. A particular recent focus has been on highly repetitive document collections, which have become prevalent in many areas (such as version control systems and genomics – to name just two very different sources).

The aim of this paper is to describe simple and efficient document listing algorithms that can be used in combination with more sophisticated techniques, or as baselines against which the performance of new document listing indexes can be measured. Our approaches are based on simple combinations of scanning and hashing, which we show to combine very well with dictionary compression to achieve small space usage. Our experiments show these methods to be often much faster and less space consuming than the best specialized indexes for the problem.

2012 ACM Subject Classification Information systems → Data compression

Keywords and phrases String Processing, Pattern matching, Document listing, Document retrieval, Succinct data structures, Repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.SEA.2021.12

Funding This work was funded in part by the Academy of Finland via grant 319454.

Acknowledgements Our thanks go to Dustin Cobas for prompt help in getting his codebase to compile on our system, and to Massimiliano Rossi for assistance with datasets.

1 Introduction

Given a collection of strings $T_0, T_1, \ldots, T_d$, called *documents* the *document listing* problem is to preprocess the documents and build an index data structure so that later, given a previously unseen string Q (the *pattern* or *query*), we can report all $i \in 0..d$ for which T_i contains Q as a substring.

The document listing problem was introduced, almost 20 years ago now, by Muthukrishnan [14] as a natural variant of the pattern matching problem. At the time, finding *all* the *occ* occurrences of a pattern in a set of texts in time proportional to *occ*, was efficiently solvable using suffix trees and arrays. The algorithmics attractiveness of the document listing problem lay in cases where *docc* – the number of documents containing the pattern – was very much smaller than *occ*, where the "brute force" solution of enumerating over the set of all *occ* occurrences to find the distinct document ids seems wasteful. Muthukrishnan gave a $O(docc + |\mathsf{Q}|)$ time solution, which is optimal.

A convenient way of thinking about Muthukrishnan's approach is in terms of the suffix array [13], SA, for the string T formed by concatenating the T_i strings of the collection into one string $\mathsf{T} = \mathsf{T}_0\$\mathsf{T}_1\$\ldots\mathsf{T}_d\$$ where \$ is a "separator" symbol guaranteed not to be part of any query pattern. The suffix array, $\mathsf{SA}[0, n-1]$, for a string T of length n is an array of integers containing a permutation of $(0 \ldots n-1)$, so that the suffixes of T starting at the

consecutive positions indicated in SA are in lexicographical order: $\mathsf{T}[\mathsf{SA}[i], n] < \mathsf{T}[\mathsf{SA}[i+1], n]$. Because of the lexicographic ordering, all the suffixes starting with a given substring Q of T form an interval $\mathsf{SA}[s, e]$, which can be determined by binary search in $O(|\mathsf{Q}| \log n)$ time. Muthukrishnan's solution to document listing defines an array $\mathsf{DA}[0, n-1]$ in which $\mathsf{DA}[i] = x$ if and only if suffix $\mathsf{T}[\mathsf{SA}[i], n]$ has its starting position inside the area of T corresponding to document T_x. We say that $\mathsf{DA}[i]$ contains the *document id* for the suffix starting at $\mathsf{SA}[i]$.

With DA in hand, document listing for pattern Q then becomes simply a matter of enumerating the distinct elements in the interval $\mathsf{DA}[s, e]$. Muthukrishnan shows this can be done in optimal $\mathrm{O}(docc)$ time via a range minimum query data structure after finding interval $[s, e]$ in $\mathrm{O}(m)$ time using the suffix tree of T. The solution requires $\mathrm{O}(n \log n)$ bits of space, and subsequent work by several authors [2, 4, 5, 7, 8, 16, 22] has aimed to reduce space in pursuit of practical solutions (we refer the reader to [15] for a survey of results prior to 2014, and [4] for more recent results). Almost all solutions make use of DA in some form.

A particular recent focus [2, 4, 5, 7, 16] has been on document retrieval indexes for *highly repetitive document collections*, which have become prevalent in many areas, such as version control systems and genomics – to name just two very different sources (see [17, 18] for recent surveys of results on highly repetitive data in general).

Our contribution. The main results of this paper are twofold:

1. We describe and implement a handful of algorithms for document listing that work by gathering the distinct document ids in $\mathsf{DA}[s, e]$ while scanning that interval. These algorithms are so simple as to be almost trivial, but nonetheless seem to have escaped scrutiny to date. We show experimentally that these algorithms are very fast, and represent a new baseline against which the speed of more complex document listing indexes should be gauged.
2. We show that the DA for a highly-repetitive collection can be effectively compressed via relative Lempel-Ziv (RLZ) parsing, a compression method that is known to support fast extraction of arbitrary intervals from its underlying sequence. We show that RLZ-compressed document arrays combine well with the aforementioned scan-based document listing algorithms, leading to the smallest (or near smallest) indexes we know of for that problem, which are often an order of magnitude or faster than the best competing methods.

Roadmap. In the next section we describe our new scan-based method for document listing and compare it experimentally to another, previously described, brute-force method for the problem. Then, in Section 3 we describe how DA is amenable to RLZ compression and combine this representation of DA with variants of our scan-based document listing algorithm. We also show that for certain types of repetitive document collection, run-length encoding can be an effective way to compress DA. Section 4 compares our new document listing indexes to state-of-the-art methods, showing them to be significantly faster and often less space consuming. We then conclude with some possible directions for future work.

2 Refined Brutes

We begin with a "brute-force" document listing method that scans, copies and sorts $\mathsf{DA}[s, e]$. It is the only published method we know of that explicitly inspects every element of $\mathsf{DA}[s, e]$, and is used in several papers [4, 5, 7] where it is reported to be faster than specialized document listing methods when $e - s + 1 = occ$ is small. We refer to this method as sort[1].

[1] The same method is referred to as Packed-sort in [4, 5] and SORT in [7].

The approach taken is to allocate a buffer of *occ* integers, copy the contents of $\mathsf{DA}[s, e]$ into it, sort the buffer (bringing duplicate document ids together) and then scan the buffer removing duplicates. The final contents of the buffer are the distinct elements in $\mathsf{DA}[s, e]$.

While sort certainly gets the job done, it also looks and sounds suspiciously like a straw man: all that movement of data. The obvious alternative to sorting, of course, is something akin to the counting part of counting sort: scan $\mathsf{DA}[s, e]$ and use an array B of d elements (initially all 0) to simply record which document ids are present in the interval – indeed, d bits will do. Whenever $\mathsf{B}[\mathsf{DA}[i]] = 0$ we add document $\mathsf{DA}[i]$ to the result set (again implemented as a vector) and set $B[\mathsf{DA}[i]]$ to 1. If desirable, we can reuse B between queries by scanning the result set at the end of the scan of $\mathsf{DA}[s, e]$ and resetting all the set bits to 0. We call this document listing method bv (for bit vector).

A possible concern with bv is the d extra bits it uses. The minimum extra space (without trying too hard) for document listing is the size of the result set itself $docc \log d$ bits. We note, however, that d bits can easily be less than the $\mathrm{O}(occ \log n)$ bits that sort uses when it copies $\mathsf{DA}[s, e]$ to its buffer for sorting. In any case, we can reduce the working space in at least two ways.

For cases when $\mathrm{O}(occ \log n)$ extra space is preferable, hashing is a simple option. To this end, we implemented a simple linear probing hash table that indicates on inserting $\mathsf{DA}[i]$ whether that element is already present in the table. If not, it is inserted into the table and added to the result set. The hash table allocates space for $2\lceil \log occ \rceil$ elements, ensuring a low load factor (maximum 0.5) and therefore fast insert time. We call this variant hash and implemented a second version of it, uset, that uses a C++ `std::unordered_set` as a sanity check to our hand-rolled linear-probing hash table. Working space usage can be further reduced to $\mathrm{O}(docc \log n)$ by using a search tree to accumulate distinct elements instead of a hash table. We implemented such a variant, which we denote tree, variant using a C++ `std::set`, which is backed by a red-black tree.

We measured the performance of the above scan-based document listing algorithms (including the previously described sort-based method) on three document collections. These data sets are described in Section 4 and Table 1. We tried all variants on two trivial encodings of DA. The first, Plain stores DA as an array of 32-bit unsigned integers. The second, Packed, packs elements of DA into $\lceil \log d \rceil$ bits each, with each element still being accessible in $\mathrm{O}(1)$ time, albeit with a higher constant of proportionality than element access with the Plain encoding.

Results for query patterns with 4-mers of high frequencies are shown in Figure 1. The bitvector-based scanning approach (bv) is a clear winner on all data sets. It is on average an order of magnitude faster than the sort-based baseline (sort) on the Page data set, and two order of magnitude faster on Revision and Influenza. The linear probing hash-based method (hash) is also significantly faster than sort on all data sets (1.32-2.27×). Somewhat unsurprisingly, Plain document arrays always led to faster queries than did Packed document arrays for all methods. We did not include Packed-tree and Packed-uset to save space in the plots because they are dominated by other methods. Figure 1 also shows that the differences between the two variants Plain-bv and Packed-bv are bigger than the differences for other variants. Taking into account that the number of DA access must be the same in all scanning algorithms, we do not yet have a good explanation for it, but it could be due to the overhead of the function call required with Packed-bv that makes the loop more difficult to unroll for the compiler.

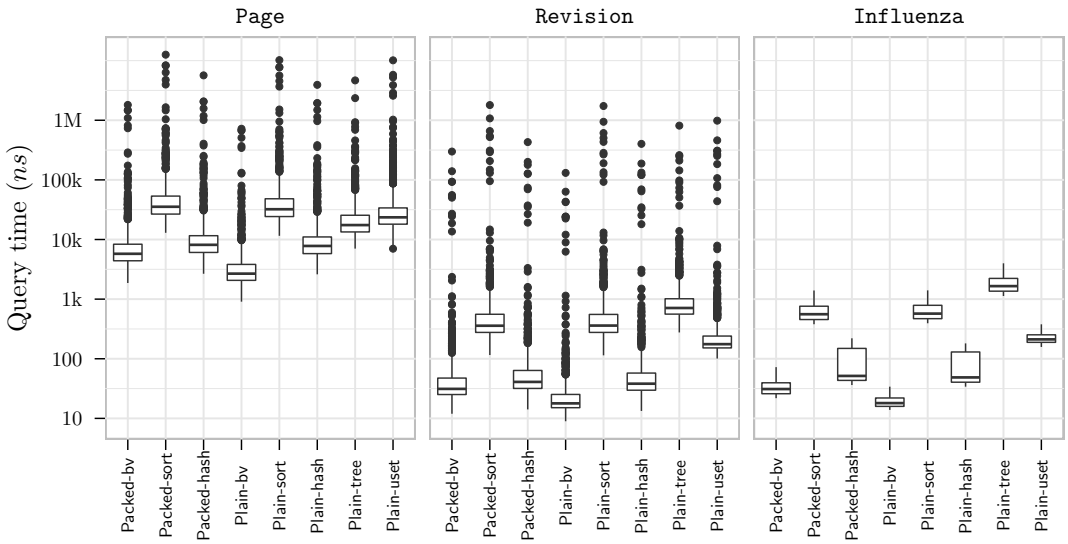

Figure 1 Boxplots showing time for Packed-sort versus our variants. These results are obtained on query patterns with k-mers of length 4 with high frequencies (see Section 4 and Table 1 for details of data sets and patterns). Note that the vertical axis is logarithmic.

3 Petite Brutes

On highly repetitive collections, the size of the compressed index used to find the interval of DA containing the document occurrences for the query pattern is dwarfed by the space used by DA. Even when DA is stored bit-packed in $n\lceil \log d \rceil$ bits, it is still almost 70 times bigger than the interval finding component on all our data sets. Reducing the space used by DA is therefore important.

As several authors have now observed [4,19], repeated substrings in T give rise to repeated intervals in DA. To see this, consider two lexicographically adjacent suffixes of $T[\mathsf{SA}[i], n]$ and $T[\mathsf{SA}[i+1], n]$. If SA[i] and SA[i+1] are preceded by an identical symbol $c \neq \$$ (i.e. $\mathsf{BWT}[i] = \mathsf{BWT}[i+1] = c \neq \$$) then the lexicographical ordering of suffixes in SA dictates that suffixes $T[\mathsf{SA}[i]-1, n]$ and $T[\mathsf{SA}[i+1]-1, n]$ will be adjacent too, and so $\mathsf{DA}[i, i+1]$ will be repeated. More generally, a run of x suffixes $\mathsf{SA}[i, i+x]$ having identical preceding symbols implies the sequence $\mathsf{DA}[i, i+x]$ is repeated in DA.

In [4], Cobas and Navarro employ grammar compression [1] to capture such repetitions and so reduce the size of DA. They preprocess the resulting grammar so that the compressed representation of DA supports the extraction of arbitrary intervals, and use this to implement sort-based document listing: the relevant interval is extracted and copied to a buffer, which is then sorted so that distinct elements can be reported.

In this section we explore compression of DA using relative Lempel-Ziv (RLZ) dictionary compression, as an alternative to grammar compression of DA. Our rational for RLZ is twofold. Firstly, it is known to support fast random-access decompression of arbitrary substrings (in our context, intervals of DA), and, secondly, it has been recently shown to be effective at compressing suffix arrays. Given the tight relationship between SA and DA, it is reasonable to expect RLZ to compress DA well too.

3.1 RLZ-Compressed Document Array

RLZ parsing [9, 10] is a variant of the classic LZ77 parsing [24], in which a sequence X is compressed relative to a second sequence R (the reference) by encoding X as a sequence of z substrings, or *phrases*, that occur in R. In our context $\mathsf{X} = \mathsf{DA}$.

Intuitively, if the substrings of sequence R are "similar" to those in X, then the parsing will produce a small number of phrases. There are a number of ways to determine a good R for a given X. Random sampling substrings from X has been shown to give good results in practice [9] and also in theory [6]. Several authors have advocated more judicious selection of substrings [11] and reference pruning methods to eliminate sparsely used parts of the reference [23]. We return to the problem of reference selection for DA below.

Data Structure. We now describe how, given an reference R, the resulting RLZ parsing of DA is encoded to facilitate fast random access to arbitrary intervals of DA. The approach is essentially the way in which random access is supported in RLZ-compressed text [10], but we include the description here for completeness.

The RLZ parsing of DA is stored in two arrays, S and P, both of length z. S contains the starting position in DA of each phrase in ascending order. Elements of S are kept in a predecessor data structure. P contains either literal DA values or positions in R as output by the parsing algorithm (the second components of each pair). The type of the ith (literal or repeat) is determined from the phrase length, which is in turn computed from the phrase starting positions: $\ell_i = S[i+1] - S[i]$. If $\ell_i = 1$ then $P[i]$ should be interpreted as containing the value of a literal phrase, and otherwise $P[i]$ is the position in R at which the ℓ_i symbols constituting the ith phrase begin.

Scanning (i.e. decoding) an arbitrary interval $\mathsf{DA}[s, e]$ is performed as follows. An output buffer B of size $e - s + 1$ will contain the decoded elements. At a high level, the phrases covering $\mathsf{DA}[s, e]$ are decoded and copied to B (some parts of the first and last phrase may not be) until B is full, at which point we are done. To this end, begin by finding the index in S of the predecessor of s. Let x denote this index, and so $S[x] \leq s$. If $P[x]$ is a literal phrase, copy its value to the output buffer. Otherwise, ($P[x]$ is non-literal) find the position where the interval in this phrase begins, which is $s - S[x]$ elements from the start of the phrase. The length of the phrase is $\ell = S[x+1] - S[x]$. So we need to decode from this phase $\min(l - (s - S[x]), e - s + 1)$ elements. If $S[x] = s$, to decode phrase x we access $R[P[x]]$, copy $R[P[x]]$ to the output buffer, continuing then to copy $(R[P[x] + 1])$ to B, and so on until either the whole phrase has been decoded, or the output buffer is full. And if $S[x] < s$, the copying starts from $(R[P[x]] + (s - S[x]))$. After decoding phrase x, if the output buffer is not full, then phrase $x + 1$ is decoded, and so on, until all $e - s + 1$ values have been decoded.

The time to decode the desired interval from the RLZ-compressed DA is $\mathrm{O}(e - s + \log\log n)$, where $\log\log n$ is the time need for the initial predecessor query.

Reference Selection. In earlier work [20, 21] we explored compression of SA using a combination of differential encoding and RLZ compression, with [20] using random sampling of substrings, and [21] obtaining superior compression performance via a more sophisticated algorithm (adapted from [11]), in which substrings are selected for inclusion in the reference according to the abundance of smaller substrings of length k (k-mers) contained therein.

We adopt a similar approach to derive a good reference for DA, which we now describe. DA is logically divided into n/s substrings of equal length s called *segments* (the last segment may have length $< s$). The frequency of each distinct k-mer in DA is then computed. Each

segment is assigned an initial score according to the k-mers it contains. In particular, let $f(x)$, for k-mer x be the frequency of x in DA. Let $x \in X_i$ denote that k-mer x has at least one occurrence in segment X_i. Then the initial score for segment X_i, $i \in [0..n/s+1]$ is the ℓ_p norm of the vector of its constituent k-mers, calculated as:

$$\text{score}(X_i) = (\Sigma_{x \in X_i} f(x)^p)^{1/p}.$$

The highest scoring X_i is then selected for inclusion in the reference, and frequencies $f(.)$ are then reduced for every k-mer $x \in X_i$, in particular $f(x)$ is reduced by the frequency of k-mer of x in X_i. This process of segment selection and subsequent score adjustment is repeated until the sum of the lengths of the selected segments has reached the target reference length (an input parameter).

Apart from the target reference length, there are three parameters to this process: s, the segment size; k, the k-mer length; and p, which affects the way in which k-mer frequencies affect segment scores (we set $p = 0.5$ in all experiments in this paper). Figure 2 shows the effect of parameters s and k on the index size for the Influenza dataset for a fixed target reference size. The figure shows that the choice of k and s can significantly influence the size of the index and should be chosen with care. Having said this, the range illustrated in the figure across all tested (k, s) settings is from just under 130MB to just over 270MB for the 336,798,466-entry long DA sequence. This corresponds to a range of 3.2 to 6.7 bits per symbol, which is significantly less than the $\log d = \log(227,356) = 18$ bits per symbol required to store DA uncompressed. Our point here is that while finding the minimum index size may be difficult, finding a "good" one seems not to be.

We also found that not every position in our constructed references was actually covered by any phrase, and removing the symbols at those positions led to around a small reduction in reference length: 1.5% for Page, 9.35% for Revision, and 2.64% for Influenza.

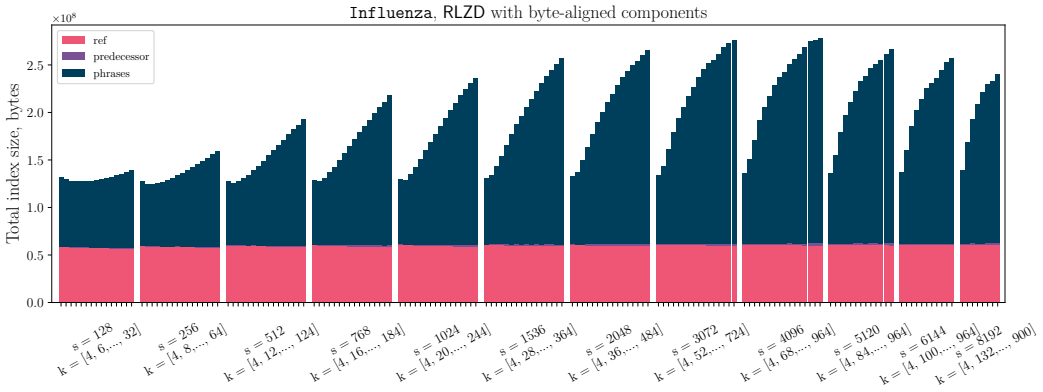

Figure 2 Effect of segment size s and k-mer length on overall index size (reference and phrases), in bytes. Target reference size requested for every case is the same, and is equal to 61.24MB. Actual reference size (ref) in the index varies a little due to removal of symbols that were not referred to during the RLZ compression (see text). Predecessor size (pred) grows with the number of phrases, though it is difficult to see on this plot.

3.2 Run-Length Compressed Document Array

To our initial surprise, we found that DA for the Page collection ($n = 1,036,000,000$) consisted of just $17,224,529$ runs of equal document ids. We believe this phenomenon can be explained as follows. Recall that in Page all versions of a given Wikipedia entry are treated

as a single document, sharing the same document id. For a given entry with, say, document id x, substrings corresponding to terminology (or, e.g., dates) specific to that entry are likely to be shared across versions, and suffixes prefixed with such patterns will be concentrated together in SA, with a corresponding concentration of id x in DA.

With this in mind, we implemented a run-length encoded version of DA that supports random access to intervals via a predecessor data structure holding the starting positions of runs in DA. We call this data structure RLED, and include it in experiments only for the Page collection (Revision and Influenza both had average run length around 1 in their DAs – far too low for RLED to acheive any compression).

3.3 Experiments Results

We implemented the RLZ- and RLE-based encodings of DA and combined them with the scan-based "brute-force" methods for document listing described in Section 2. Results of experiments on our three test collections are shown in Figure 3. Plain-bv, the fastest uncompressed method from Section 2 is included as a reference point. On the Page dataset, the methods using RLE are clearly fastest, with RLED-bv being the fastest overall. However the RLZ-based methods also show good performance, with RLZD-bv being only marginally slower than the uncompressed Plain-bv baseline (0.07 vs. 0.06 nanoseconds per query). On the Revision and Influenza datasets, where the RLE method does not apply, we observe a similar pattern in the RLZ-based methods as was observed in Section 2 for the uncompressed methods: RLZD-bv is fastest, followed by RLZD-hash.

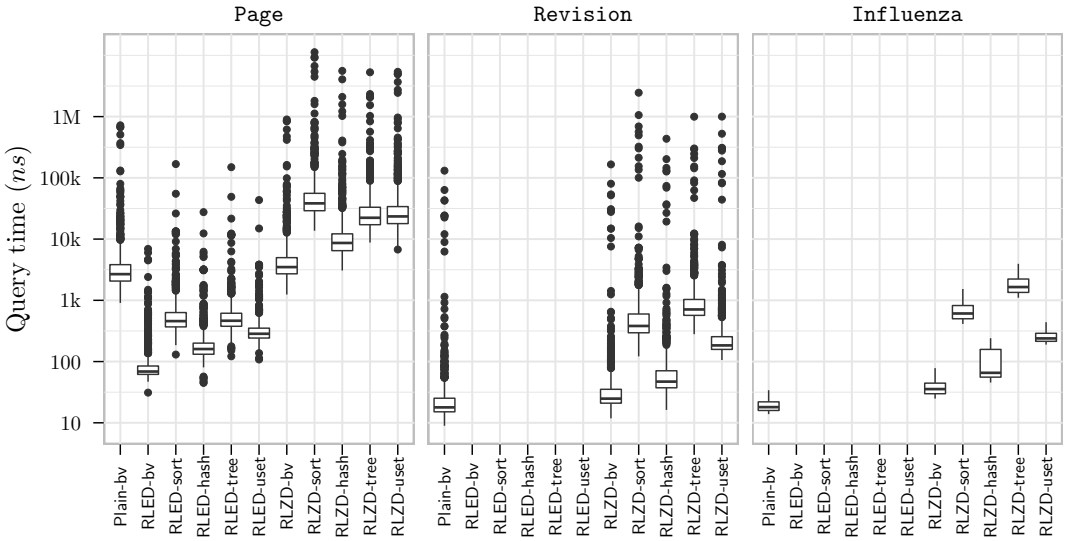

Figure 3 Boxplots showing query times for our RLZ- and RLE-compressed DA variants. We include the fastest among uncompressed methods, Plain-bv, as a reference point. Note that the vertical axis is logarithmic. These results were obtained on patterns with 4-mers of high frequency (see Section 4 and Table 1 for details of data sets and patterns). The RLZD index was constructed with the following parameters: for Page $k = 312$, $s = 2048$, reference size 7.95MB, for Revision $k = 8$, $s = 1024$, 36.38MB reference, and for Influenza $k = 12$, $s = 256$, 61.24MB reference. Reference sizes were computed experimentally. For Revision and Influenza there is no data for RLED because on these datasets it does not achieve any compression (see text, Section 3.2).

Table 1 Statistics for document collections: *Collection* name; *Size* in megabytes; *Docs*, number of documents; *Doc size*, average document length; *k-mer*, length of k-mers in the patterns; *Frequency*, frequency of k-mers chosen for patterns; number of *Patterns*; *Occs*, average number of occurrences; *Doc occ*, average number of document occurrences; *Occs/doc*, average ratio of occurrences to document occurrences.

Collection	*Size*, MB (n)	*Docs* (D)	*Doc size* (n/D)	*k-mer*	*Frequency*	*Patterns*	*Occs* (occ)	*Doc occs* ($docc$)	*Occs/doc* ($\frac{occ}{docc}$)
Page	1 037	280	38 883 145	4	high	1 000	318 136.00	153.81	2 068.32
					mid		11 994.35	34.23	350.43
					low		500.50	4.80	104.16
				8	high		87 081.25	55.31	1 574.51
					mid		7 148.71	19.90	359.25
					low		500.50	6.54	76.56
Revision	1 035	65 565	16 552	4	high		317 512.93	31 850.79	9.97
					mid		12 017.48	5 937.12	2.02
					low		500.50	389.86	1.28
				8	high		86 895.30	11 627.18	7.47
					mid		7 147.44	3 814.97	1.87
					low		500.50	390.65	1.28
Influenza	321	227 356	1 480	4	high	141	1 819 407.80	224 449.40	8.11
					mid		562 380.65	161 793.77	3.48
					low	142	75.18	70.52	1.07
				8	high		41 721.84	37 822.00	1.10
					mid	1 000	9 784.96	9 665.94	1.01
					low		500.50	498.79	1.00

4 Performance Comparison

In this section we report on the practical performance of our document listing approaches to other state-of-the art solutions.

4.1 Experimental Setup

Test Machine. All our experiments[2] were conducted on a 2.10 GHz Intel Xeon E7-4830 v3 CPU equipped with 30 MiB L3 cache and 1.5 TiB of main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS was Linux (Ubuntu 18.04.5 LTS) running kernel 5.4.0-58-generic. Programs were compiled using `g++` version 7.5.0. All given runtimes were recorded with the C++11 `high_resolution_clock` time measurement facility.

Datasets. Table 1 summarizes the collections and patterns used. Page and Revision are repetitive collections generated from a Finnish-language Wikipedia archive with full version history: 280 pages with a total of 65,565 revisions. In Page, all the revisions of a page form a single document. In the case of Revision, each page revision becomes a separate document. Influenza is composed of 227,356 sequences of the H. influenzae virus genomes[3].

Query Patterns. To form the query patterns for each dataset (see statistics in Table 1) we first computed the set of all distinct substrings of printable characters of lengths 4 and 8, and their number of occurrences. Each such set of these substrings were then sorted based

[2] Code is available at `https://www2.helsinki.fi/en/researchgroups/algorithmic-bioinformatics/compressed-data-structures`.
[3] All data sets are available at `https://jltsiren.kapsi.fi/rlcsa`.

on their occurrences, and divided into three equal subsets based on their frequencies: high, middle, and low. In the subset if there were less than a 1000 patterns, then all the patterns in this subset form a corresponding file with queries, and if there were more than 1000, we choose a 1000 out of them that represent the subset: for high – 1000 most frequent, for mid – 1000 that are exactly at the middle of the list with the sorted frequencies, and for the low – 1000 least frequent. Patterns with high *occ* represent hard cases for our scan-based methods.

Indexes Measured. We used the recent study of Cobas and Navarro [4] as a guide for selecting the best-performing document listing methods to include for each data set. In particular, we measured the following indexes.

- Brute Force. Apart from sort, the method from Section 2, which is referred to as Packed-sort in [4], we also include Brute-C a variant from [4] that uses a grammar-compressed DA, augmented with the length of the expansion of each nonterminal. If the resulting grammar tree has height h the interval DA$[s, e]$ can be extracted in time $O(h + e - s)$.
- Grammar-Compressed Document Array (GCDA), the proposal of Cobas and Navarro [4], which precomputes answers to document listing queries and stores them grammar compressed to reduce space. We set parameters b and β as described in [4].
- Sadakane (Sada). Sada-D is the index of Sadakane [22] in which the query time was sped up by explicitly storing DA, as first suggested in [5].
- Interleaved Longest Common Prefix (ILCP). ILCP-D is a variant of the ILCP index of Gagie et al. [5] that uses DA. ILCP-C uses, instead, Cobas and Navarro's [4] grammar-compressed DA, which can access any cell of DA in $O(h)$ time.

All indexes tested (including our own) use the same RLCSA implementation [12] to find the relevant interval $[s, e]$ of DA for each query pattern. This RLCSA uses $(r \log \sigma + 2r \log(n/r))(1 + o(1))$ bits of space, where r is the number of runs in the BWT of T, and finds the interval in $O(m \log r)$ time. RLCSA for Page and Revision required 0.13 bits per symbol, and 0.26 bits per symbol for Influenza.

4.2 Results

Figure 4 shows the tradeoff between time and space for the indexes that were shown in [4] to be among fastest and smallest, and the ones described in this paper: Packed-bv, which was the second fastest among uncompressed versions (Figure 1) after Plain-bv (but uses much less space); and the two fastest variants for RLZD and RLED: -bv and -hash (Figure 3). RLZD on these datasets has shown the best compression on Page (0.341 for RLZD and 0.412 for Brute-C, which is the closest) and second best after Brute-C on Revision (0.812 for RLZD and 0.595 for Brute-C) and Influenza (2.958 for RLZD and 2.541 for Brute-C). These space results for RLZD were received with the following parameters to generate the reference: for Page k = 312, s = 2048, ref = 1,987,456 integers requested, for Revision k = 8, s = 1024, ref = 9,096,048, and for Influenza k = 12, s = 256, ref = 15,310,000. Ref values were computed experimentally. In terms of time RLZD-bv yields only to Packed-bv (and in half of the cases outperforms it), but the time difference is marginal and Packed-bv requires 6-20× more space.

RLED dominates on Page (where there are few and large documents), especially on high frequent patterns, where the closest competitor GCDA is 6 times slower. RLZD-bv in this case is 41 times slower than RLED-bv, but requires 3 times less space; compared to GCDA, RLZD-bv is almost 5 times slower, but needs almost half the space (GCDA takes 0.584 bits per symbol, RLZD 0.341). On the moderately frequent pattern set the time difference between RLZD-bv and GCDA becomes negligible, and RLZD-bv is starting to win on longer patterns.

On low frequent queries RLZD-bv becomes 3 times faster than GCDA, only 1.6 times slower than RLED-bv and is indeed only marginally slower than Packed-bv.

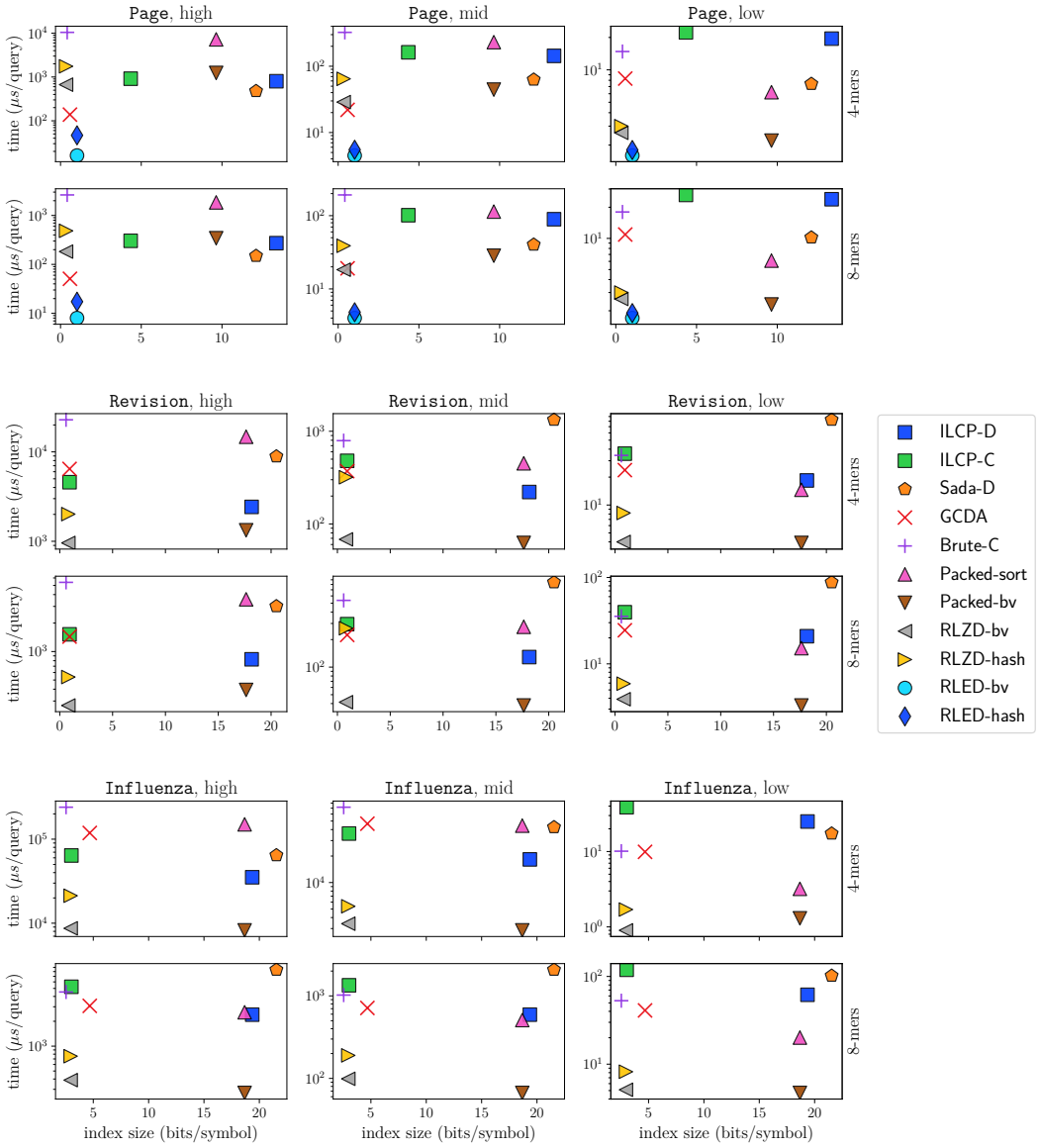

Figure 4 Document listing indexes on real repetitive collections. The x axis shows the total size of the index in bits per symbol. The y axis shows the average time per query in microseconds. Note that the vertical axis is logarithmic.

In the case of Revision, where are more and smaller documents, RLZD-bv is about 1.4-1.5× faster than Packed-bv on frequent patterns, showing almost the same time results on middle and low frequent queries. ILCP-D is 2.5-3× slower on high frequent patterns (taking more than 22 times more space, requiring 18.155 bit per symbol), slowing down to 4.5-5× on low frequent. GCDA is 5.3-6.7× slower than RLZD-bv here. Brute-C with the index size of 1.36× smaller than RLZD, takes $8.5 - 23.8\times$ more time compared to RLZD-bv. Here RLZD-hash is 9.7-14× slower than RLZD-bv variant.

Influenza, with many small documents, is the worst case for many indexes. Here the smallest index, Brute-C, takes 1.16× less space than the nearest competitor, RLZD, and from 10.3 (on 8-mers with low frequency) upto 27.6 (on 4-mers with high frequency) more time than RLZD-bv. Packed-bv is 1.06 upto 1.42 times faster (271 microseconds for Packed-bv versus 388 for RLZD-bv on 8-mers with high frequency) and requires 6.3× more space. The other three nearest competitors: ILCP-D is 4-28× slower and as big as Packed-bv, GCDA is 8-13.6× slower, requiring 1.6 more space, and ILCP-C, which is third best space-wise, requires 1.14 more space (0.923 bits per symbol) and 4.7-10.2 more time.

In most cases the best previous indexes (from [4]) are either much slower, much larger, or both, compared to the ones we describe.

5 Conclusions and Future Work

We have shown that very simple algorithms based on scanning intervals of the document array lead to very fast document listing times on three highly repetitive data sets of versioned documents and genome collections. We have further shown these approaches to work well with new compressed representations of the document array based on relative Lempel-Ziv parsing and run-length encoding. We speculate that there are many ways to further engineer and improve the approaches we have described, however, our experiments here strongly indicate that they already exhibit significant performance improvements over existing methods. Our indexes are among the smallest document listing approaches known (at least on the data sets tested) and are an order of magnitude or faster.

There are numerous avenues for future work. Firstly, the results of this paper should lead directly to greatly improved performance for the metagenomics applications of document listing investigated in [3]. Extending our results to more complex versions of the document listing problem, such as document listing with frequencies and top-k document retrieval should also be possible. It may also be fruitful to apply RLZ compression to improve other document listing methods, as was done with grammar compression by Cobas and Navarro [4].

Finally, it would seem important to determine the interval size at which specialized document listing methods begin to overhaul scanning-based methods. Such an investigation would necessarily involve indexing large test collections, which bring their own different challenges for the more intricate indexes, such as index construction. Our results here show that scanning-methods still have the edge for interval sizes into the millions.

References

1 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.

2 F. Claude and I. Munro. Document listing on versioned documents. In *Proc. SPIRE*, LNCS 8214, pages 72–83, 2013.

3 D. Cobas, V. Mäkinen, and M. Rossi. Tailoring r-index for document listing towards metagenomics applications. In *Proc. SPIRE*, LNCS 12303, pages 291–306. Springer, 2020.

4 D. Cobas and G. Navarro. Fast, small, and simple document listing on repetitive text collections. In *Proc. SPIRE*, LNCS 11811, pages 482–498, 2019.

5 T. Gagie, A. Hartikainen, K. Karhu, J. Kärkkäinen, G. Navarro, S. J. Puglisi, and J. Sirén. Document retrieval on repetitive collections. *Information Retrieval*, 20:253–291, 2017.

6 T. Gagie, S. J. Puglisi, and D. Valenzuela. Analyzing relative Lempel-Ziv reference construction. In *Proc. SPIRE*, LNCS 9954, pages 160–165, 2016.

7 S. Gog, R. Konow, and G. Navarro. Practical compact indexes for top-k document retrieval. *ACM Journal of Experimental Algorithmics*, 22(1):article 1.2, 2017.

8 W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722. IEEE, 2009.

9 C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proceedings of the VLDB Endowment*, 5(3):265–273, 2011.

10 S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. SPIRE*, LNCS 6393, pages 201–206, 2010.

11 K. Liao, M. Petri, A. Moffat, and A. Wirth. Effective construction of relative lempel-ziv dictionaries. In *Proc. WWW*, pages 807–816. ACM, 2016.

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

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

14 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.

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

16 G. Navarro. Document listing on repetitive collections with guaranteed performance. *Theoretical Computer Science*, 777:58–72, 2019.

17 G. Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 2020. To appear.

18 G. Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

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

20 S. J. Puglisi and B. Zhukova. Relative Lempel-Ziv compression of suffix arrays. In *Proc. SPIRE*, LNCS 12303, pages 89–96. Springer, 2020.

21 S. J. Puglisi and B. Zhukova. Smaller RLZ-compressed suffix arrays. In *Proc. Data Compression Conference*, pages 213–222. IEEE Computer Society, 2021.

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

23 J. Tong, A. Wirth, and J. Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proc. SIGIR*, pages 283–292. ACM, 2014.

24 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.