

# Document Counting in Compressed Space\*

Travis Gagie<sup>1</sup>, Aleksi Hartikainen<sup>1</sup>, Juha Kärkkäinen<sup>1</sup>,  
Gonzalo Navarro<sup>2</sup>, Simon J. Puglisi<sup>1</sup> and Jouni Sirén<sup>2</sup>

<sup>1</sup> Helsinki Institute of Information Technology (HIIT),  
Department of Computer Science, University of Helsinki, Finland

<sup>2</sup> Center of Biotechnology and Bioengineering (CeBiB),  
Department of Computer Science, University of Chile, Chile

## Abstract

We address the problem of counting the number of strings in a collection where a given pattern appears, which has applications in information retrieval and data mining. Existing solutions are in a theoretical stage. In this paper we implement these solutions and explore compressed variants, aiming to reduce data structure size. Our main result is to uncover some unexpected compressibility properties of the fastest known data structure for the problem. By taking advantage of these properties, we can reduce the size of the structure by a factor of 5-400, depending on the dataset.

## 1 Introduction

In the classic pattern matching problem, we are given a text string  $T[1, n]$  and a pattern string  $P[1, m]$ , and must count or report all the positions in  $T$  at which  $P$  occurs. Document retrieval problems are natural variants of this classic problem in which  $T$  is composed of  $d$  smaller strings, or *documents*. The three main document retrieval problems considered to date are: *document counting*, where the task is to compute the number of documents containing  $P$ ; *document listing*, where we must return a list of all the documents that contain  $P$ ; and *top- $k$  listing*, returning the  $k$  documents most relevant to  $P$ , given some relevance measure (for example, the  $k$  documents that contain  $P$  most often). From an algorithmic point of view, these problems are interesting because the number of occurrences of  $P$  in  $T$ , denoted  $occ$ , may be very much larger than  $docc$ , the number of distinct documents in which the pattern occurs, and so tailored solutions may outperform those based on brute-force application of classical pattern matching.

In recent years, document retrieval problems have been the subject of intense research in both the string algorithms and information retrieval communities (see recent surveys [8, 11]). The vast majority of this work has been on the latter two problems (listing and top- $k$ ). Indeed, there have been only two results on document counting [15, 5], and no investigation into their practicality has ever been undertaken.

---

\*Funded in part by Fondecyt Project 1-140796; Basal Funds FB0001, Conicyt, Chile; the Jenny and Antti Wihuri Foundation, Finland; and by Academy of Finland grants 268324 and 258308.

However, competitive listing and top- $k$  solutions require fast algorithms for counting. In recent work [13] it was shown that the best choice of listing and top- $k$  algorithm in practice strongly depends on the *docc/occ* ratio, and thus the ability to compute *docc* quickly may allow the efficient selection of an appropriate listing/top- $k$  algorithm at query time. Secondly, from an information retrieval point of view, *docc* (known in that community as *document frequency*, or *df*) is a necessary component of most ranking formulas [16, 2], and so fast computation of it is desirable. Document counting is also important for data mining applications on strings (or *string mining*, see, e.g., [3]), where the value *docc/d* for a given pattern is its *support* in the collection.

**Results.** Our main result is to show that Sadakane’s data structure inherits the repetitiveness present in the underlying data, which can be exploited to reduce its space occupancy, obtaining a compressed data structure. Surprisingly, the structure also becomes repetitive with random and near-random data, such as DNA sequences. We show how to take advantage of this redundancy in a number of different ways, leading to different space-time trade-offs. The best of these compressed representations are 5–400 times smaller than the original, depending on the dataset, while being only marginally slower, and sometimes faster, at answering counting queries.

## 2 Background

Let  $T[1, n]$  be a concatenation of a collection of  $d$  documents. We assume that each document ends with a special character  $\$$  that is lexicographically smaller than any other character of the alphabet. The *suffix array (SA)* of the collection is an array  $\text{SA}[1, n]$  of pointers to the suffixes of  $T$  in lexicographic order. The *document array (DA)*  $\text{DA}[1, n]$  is a related array, where  $\text{DA}[i]$  is the identifier of the document where suffix  $T[\text{SA}[i], n]$  begins. The *suffix tree (ST)* is a versatile text index based on building a trie for the suffixes of the text, and compacting unary paths into single edges. If we list the leaves of the suffix tree in lexicographic order, we get the suffix array.

Many succinct and compressed data structures are based on *bitvectors*. A bitvector is a binary sequence  $B[1, n]$ , with additional data structures to support **rank** and **select**. Operation  $\text{rank}_1(B, i)$  counts the number of 1-bits in the prefix  $B[1, i]$ , while  $\text{select}_1(B, i)$  finds the 1-bit of rank  $i$ . These operations can also be defined for 0-bits, as well as on general sequences. Several different encodings are commonly used for the binary sequence. *Plain* bitvectors store the sequence as-is, while *entropy-compressed* bitvectors reduce its size to close to the zero-order entropy. *Gap encoding* stores the distances between successive minority bits, while *run-length encoding* stores the lengths of successive runs of 1-bits and 0-bits. *Grammar-compressed* bitvectors use a context-free grammar to encode the sequence.

The *compressed suffix array (CSA)* [7] and the *FM-index (FMI)* [4] are space-efficient text indexes based on the *Burrows-Wheeler transform (BWT)* [1] — a permutation of the text originally developed for data compression. The Burrows-Wheeler transform  $\text{BWT}[1, n]$  is easily obtained from the text and the suffix array:  $\text{BWT}[i] = T[\text{SA}[i] - 1]$ , with  $T[0] = \$$ . As the CSA and the FMI are very similar data structures, we collectively name them compressed suffix arrays (CSAs) in this paper.

We consider text indexes supporting four kinds of queries: 1) `find`( $P$ ) returns the range  $[sp, ep]$ , where the suffixes in  $\text{SA}[sp, ep]$  start with pattern  $P$ ; 2) `locate`( $P$ ) returns  $\text{SA}[sp, ep]$ ; 3) `count`( $P$ ) returns the number of documents containing pattern  $P$ ; 4) `list`( $P$ ) returns the identifiers of the distinct documents containing pattern  $P$ . For queries 2–4, we also consider variants where the parameter is the suffix array range  $[sp, ep]$  or the suffix tree node  $v$  corresponding to pattern  $P$ . CSAs support the first two queries; `find` is relatively fast, while `locate` can be much slower. The main time/space trade-off in a CSA, the *suffix array sample period*, affects the performance of `locate` queries. Larger sample periods result in slower and smaller indexes.

### 3 Prior Methods for Document Counting

In this section we review the two prior methods for document counting, one by Sadakane [15] and another by Gagie et al. [5].

**Sadakane’s method.** Sadakane [15] showed how to solve `count` in constant time adding 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, with nodes  $u$  and  $w$  as its children, we determine the number of redundant suffixes  $h(v) = |\text{list}(u) \cap \text{list}(w)|$ . This allows us to compute `count` recursively:  $\text{count}(v) = \text{count}(u) + \text{count}(w) - h(v)$ . By using the leaf nodes descending from  $v$ ,  $[sp, ep]$ , as base cases, we can solve the recurrence:

$$\text{count}(v) = \text{count}(sp, ep) = (ep + 1 - sp) - \sum_u h(u),$$

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

We form 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

$$\text{count}(sp, ep) = (ep + 1 - sp) - \sum_{i=sp}^{ep-1} H[i].$$

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  $H[i]$  0-bits. We can now compute the sum by counting the number of 0-bits between the 1-bits of ranks  $sp$  and  $ep$ :

$$\text{count}(sp, ep) = 2(ep - sp) - (\text{select}_1(H', ep) - \text{select}_1(H', sp)) + 1.$$

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

**Counting with the interleaved LCP array.** Muthukrishnan [10] defined, for efficiently computing `list`( $P$ ), an array  $C[1, n]$  so that  $C[i] = j$  if  $j$  is the rightmost position preceding  $i$  such that  $\text{DA}[i] = \text{DA}[j]$ . He uses the property that the first occurrence  $\text{DA}[i]$  of each document in  $\text{DA}[sp, ep]$  is the only one for which  $C[i] < sp$ .

This property makes  $C$  useful for counting, as we only have to determine the number of values below  $sp$  in  $C[sp, ep]$ . This can be done in  $O(\log n)$  time using a wavelet tree [7] on  $C$ , of size  $n \log n + o(n \log n)$  bits. Gagie et al. [6] used a more sophisticated representation, achieving  $n \log d + o(n \log d) + O(n)$  bits of space and query time  $O(\log(ep - sp + 1))$  to compute  $\text{count}(sp, ep)$ .

Both time and space are not competitive with Sadakane’s method. However, a more recent approach [5] could be space-competitive, especially on repetitive document collections. Let  $\text{lcp}(S, T)$  be the length of the *longest common prefix* of sequences  $S$  and  $T$ . The *LCP array* of  $T[1, n]$  is an array  $\text{LCP}[1, n]$ , where  $\text{LCP}[i] = \text{lcp}(T[\text{SA}[i - 1], n], T[\text{SA}[i], n])$ . We get the *interleaved LCP array*  $\text{ILCP}[1, n]$  by building separate LCP arrays for each of the documents, and interleaving them according to the document array. As  $\text{ILCP}[i] < |P|$  iff position  $i$  contains the first occurrence of  $\text{DA}[i]$  in  $[sp, ep]$ , we can solve  $\text{count}$  by counting the number of values less than  $|P|$  in  $\text{ILCP}[sp, ep]$ . This is efficiently done with a wavelet tree of  $\text{ILCP}$ . The advantage of using  $\text{ILCP}$  is that, if the documents are similar to each other, then  $\text{ILCP}$  will have many runs of about  $d$  equal values (i.e., the same suffix coming from all the  $d$  documents), and thus it can be run-length compressed. The wavelet tree is built only on the run heads, and  $\text{count}(sp, ep)$  is computed from the run heads and the run lengths.

## 4 Compressed Document Counting

As described above, Sadakane’s structures requires  $2n + o(n)$  bits, irrespective of the underlying data. However, even this can be a considerable overhead on highly compressible collections, taking significantly more space than the CSA (on top of which Sadakane’s structure operates).

Fortunately, as we now show, the bitvector  $H'$  used in Sadakane’s method is highly compressible. There are five main ways of compressing the bitvector, with different combinations of them working better with different datasets.

1. Let  $V_v$  be the set of nodes of the binary suffix tree corresponding to node  $v$  of the original suffix tree. As we only need to compute  $\text{count}(v)$  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 1-bits in  $H'$ .
3. When the documents in the collection have a versioned structure, we can also use *grammar compression*. To see this, consider a substring  $x$  that occurs in many documents, but at most once in each document. If each occurrence of substring  $x$  is preceded by character  $a$ , the subtrees of the binary suffix tree corresponding to

patterns  $x$  and  $ax$  have identical structure, and  $\text{DA}[\text{find}(x)] = \text{DA}[\text{find}(ax)]$ . Hence the subtrees are encoded identically in bitvector  $H'$ .

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

5. We can also use filters based on array  $H$  instead of  $\text{count}$ . If  $H[i] = 0$  for the most cells, we can use a *sparse filter*  $F_S[1, n-1]$ , where  $F_S[i] = 1$  iff  $H[i] > 0$ , and build bitvector  $H'$  only for those nodes. We can also encode positions with  $H[i] = 1$  separately with an *1-filter*  $F_1[1, n-1]$ , where  $F_1[i] = 1$  iff  $H[i] = 1$ . With an 1-filter, we do not write 0-bits in  $H'$  for nodes with  $H[i] = 1$ , but subtract the number of 1-bits in  $F_1[sp, ep-1]$  from the result of the query instead. It is also possible to use a sparse filter and an 1-filter simultaneously. In that case, we set  $F_S[i] = 1$  iff  $H[i] > 1$ .

We analyze the number of runs of 1-bits in bitvector  $H'$  in the expected case. Assume that our document collection consists of  $d$  random documents, each of length  $m$ , over an alphabet of size  $\sigma$ . We call string  $S$  *unique*, if it occurs at most once in every document. The subtree of the sparse suffix tree corresponding to a unique string is encoded as a run of 1-bits in bitvector  $H'$ . Therefore any set of unique strings that covers all leaves of the tree will define an upper bound for the number of runs.

Consider a random string of length  $k$ . The probability that the string is non-unique is at most  $dm^2/(2\sigma^{2k})$ . Let  $N(i)$  be the number of non-unique strings of length  $k_i = \log_\sigma(m\sqrt{d}) + i$ . As there are  $\sigma^{k_i}$  strings of length  $k_i$ , the expected value of  $N(i)$  is at most  $m\sqrt{d}/(2\sigma^i)$ . The expected size of the smallest cover of unique strings is therefore at most

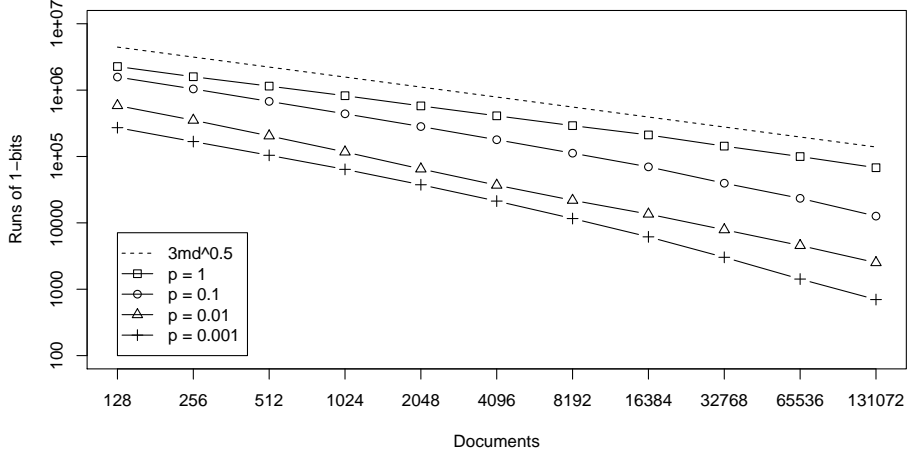
$$(\sigma^{k_0} - N(0)) + \sum_{i=1}^{\infty} (\sigma N(i-1) - N(i)) = m\sqrt{d} + (\sigma - 1) \sum_{i=0}^{\infty} N(i) \leq \left(\frac{\sigma}{2} + 1\right) m\sqrt{d},$$

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

## 5 Experiments

### 5.1 Implementation

We use two fast document listing algorithms as baseline document counting methods. **Brute-D** uses an explicit document array, sorting a copy of the query range  $\text{DA}[sp, ep]$  to count the number of distinct document identifiers. **PDL-RP** [13] is a variant of precomputed document listing, using grammar compression to space-efficiently store



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

the answers for list queries for a carefully selected subset of suffix tree nodes. As the basic text index, both algorithms use RLCSA [9], a practical implementation of the compressed suffix array intended for repetitive datasets. The suffix array sample period was set to 32 on non-repetitive datasets, and to 128 on repetitive datasets.

We used both RLCSA components and newly implemented bitvectors for several variants of Sadakane’s method. First, we have a set of basic (i.e., not applying filtering) versions of this method, depending on how bitvector  $H'$  is encoded:

**Sada** uses a plain bitvector representation.

**Sada-RR** uses a run-length encoded bitvector as supplied in the RLCSA implementation. It uses  $\delta$ -codes to represent run lengths and packs them into blocks of 32 bytes of encoded data. Each block stores the number of bits and 1-bits up to its beginning.

**Sada-RS** uses a run-length encoded bitvector, represented with a sparse bitvector (like that of **Sada-S**) marking the beginnings of the 0-runs and another for the 1-runs.

**Sada-RD** uses run-length encoding with  $\delta$ -codes to represent the lengths. The bitvector is cut into blocks of 128 1-bits, and three sparse bitvectors (as in **Sada-S**) are used to mark the number of bits, 1-bits, and starting positions of block encodings.

**Sada-grammar** uses grammar-compressed bitvectors [12].

There are also various versions that include filtering, and differ on how the bitvector  $F$  is represented (we only study the most promising combinations):

**Sada-P-G** uses **Sada** for  $H'$  and a gap-encoded bitvector for  $F$ . This gap-encoding is provided in the RLCSA implementation, which is similar to that of run-length encoding but only runs of 0-bits are considered.

**Sada-P-RR** uses **Sada** for  $H'$  and a run-length encoded bitvector (as **Sada-RR**) for  $F$ .

**Table 1:** Statistics for document collections. Collection size in megabytes, RLCSA size without suffix array samples in megabytes and in bits per character, number of documents, average document length, number of patterns, average number of occurrences and document occurrences, and the ratio of occurrences to document occurrences.

Collection	Size		RLCSA		$d$	$n/d$	Patterns	$\overline{occ}$	$\overline{docc}$	$occ/docc$
	MB	MB	bpc							
Page	641	9.00	0.11	190	3534921	14286	2601	6	444.79	
Revision	640	9.04	0.11	31208	21490	14284	2592	1065	2.43	
Enwiki	639	309.31	3.87	44000	15236	19628	10316	2856	3.61	
Influenza	321	10.53	0.26	227356	1480	1000	59997	44012	1.36	
Swissprot	54	25.19	3.71	143244	398	10000	160	121	1.33	

**Sada-RR-G** uses Sada-RR for  $H'$  and a gap-encoded bitvector for  $F$ .

**Sada-RR-RR** uses Sada-RR for  $H'$  and the same encoding for  $F$ .

**Sada-S** uses sparse bitmaps for both  $H'$  and the sparse filter  $F_S$ . Sparse bitmaps store the lower  $w$  bits of the position of each 1-bit in an array, and use gap encoding in a plain bitvector for the high-order bits. Value  $w$  is selected to minimize the size [14].

**Sada-S-S** is Sada-S with an additional sparse bitmap for  $F_1$

**Sada-RS-S** uses Sada-RS for  $H'$  and a sparse bitmap (as in Sada-S) for  $F_1$ .

**Sada-RD-S** uses Sada-RD for  $H'$  and a sparse bitmap (as in Sada-S) for  $F_1$ .

Finally, ILCP implements the technique described in Section 3, using the same encoding as in Sada-RS to represent the bitvectors of the wavelet tree.

The implementations were written in C++ and compiled on g++ version 4.8.1.<sup>1</sup> Our test environment was a machine with two 2.40 GHz quad-core Xeon E5620 processors (12 MB cache each) 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.

## 5.2 Experimental data

We compared the performance of the document counting methods on five real datasets. Three of the datasets consist of natural language texts in XML format, while two contain biological sequences. See Table 1 for some basic statistics on the datasets.

**Page** is a repetitive collection of 190 pages with a total of 31208 revisions from a Finnish language Wikipedia archive with full version history. The revisions of each page are concatenated to form a single document. For patterns, we chose all Finnish words of length  $\geq 5$  that occur in the collection.

**Revision** is the same as **Page**, except that each revision is a separate document.

<sup>1</sup>The implementations are available at <http://jltsiren.kapsi.fi/rlcsa> and <https://github.com/ahartik/succinct>.

**Enwiki** is a nonrepetitive collection of 44000 pages from a snapshot of the English language Wikipedia. As patterns, we used search terms from an MSN query log with stop words filtered out. We generated 20000 patterns according to term frequencies, and selected those that occur in the collection.

**Influenza** is a repetitive collection of the genomes of 227356 influenza viruses. For patterns, we extracted 100000 random substrings of length 7, filtered out duplicates, and kept the 1000 patterns with the largest *occ/docc* ratios.

**Swissprot** is a nonrepetitive collection of 143244 protein sequences. Patterns are 10000 substrings of length 5 with the largest *occ/docc* ratio out of a sample of 200000.

### 5.3 Results

Results are shown in Figure 2. As plain **Sada** was almost always the fastest method, we scaled plots to leave out anything much larger than it. The baseline document listing methods have size 0, as they exploit functionality already present in the index.

On **Page**, the filtered methods **Sada-P-RR** and **Sada-RR-RR** were clearly the best choices. Plain **Sada** was much faster, but took much more space than the rest of the index. Only **Sada-grammar**, which was quite slow, compressed the structure much. On **Revision**, there were many small encodings with similar performance. Among the very small encodings, **Sada-RS-S** was the fastest. **Sada-S** was somewhat larger and faster. Like with **Page**, plain **Sada** was even faster, but took much more space.

The situation changed on the non-repetitive **Enwiki**. Only **Sada-RD-S**, **Sada-RS-S**, and **Sada-grammar** could compress the bitvector well below 1 bpc, and **Sada-grammar** was much slower than the other two. At around 1 bpc, **Sada-S** was again the fastest option. Plain **Sada** required twice as much space as **Sada-S**, but was twice as fast.

**Influenza** and **Swissprot** contain, respectively, DNA and protein sequences, so each individual sequence is quite random. Such collections are easy cases for Sadakane’s method and many encodings compressed the bitvector very well. On both datasets, **Sada-S** was the fastest small encoding. On **Influenza** the small encodings easily fit in CPU cache, and so were often faster than plain **Sada**.

It is interesting that different compression techniques succeed in different collections. **Sada-grammar** was the only consistently small variant.

The ILCP-based structure, **ILCP**, was always significantly larger and slower than compressed variants of **Sada**.

## 6 Conclusions

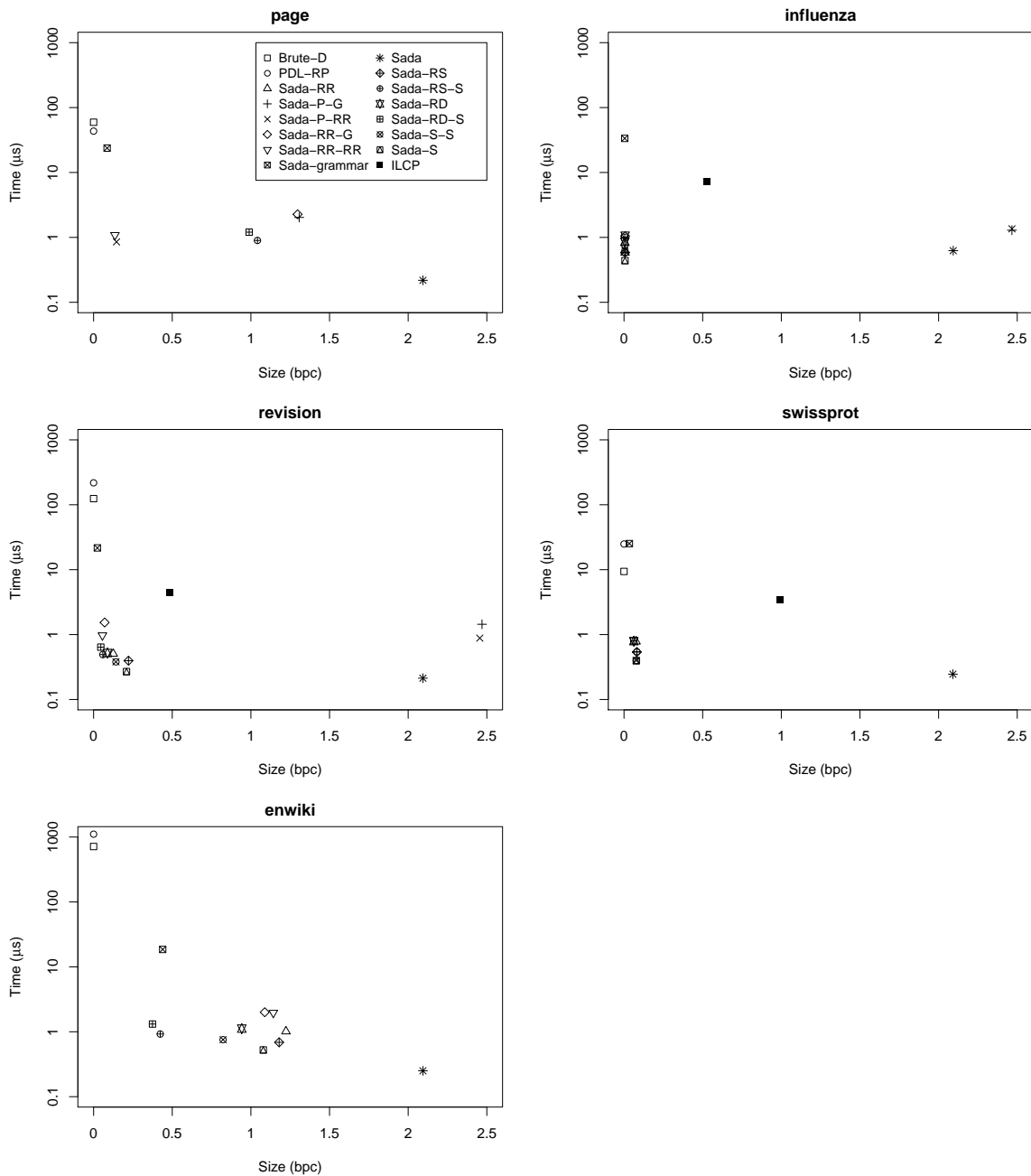
We investigated the time/space trade-offs in document counting data structures, implementing both known solutions and new methods. While Sadakane’s method was the fastest choice, we found that it can be compressed significantly below the original  $2n + o(n)$  bits, for a document collection of total size  $n$ . We achieved 5-fold compression on the natural language **Enwiki** dataset. When the dataset was repetitive



or contained random sequences, but not both, the best compressed encodings were around 20 times smaller than Sadakane’s original structure. With both repetitive data and random sequences in the *Influenza* collection, we achieved up to 400-fold compression. In all cases, the query times were around 1 microsecond or less.

## References

- [1] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- [2] S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.
- [3] J. Dhaliwal, S. J. Puglisi, and A. Turpin. Practical efficient string mining. *IEEE Transactions on Knowledge and Data Engineering*, 24(4):735–744, 2012.
- [4] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
- [5] T. Gagie, K. Karhu, G. Navarro, S. J. Puglisi, and J. Sirén. Document listing on repetitive collections. In *Proc. CPM*, LNCS 7922, pages 107–119, 2013.
- [6] T. Gagie, J. Kärkkäinen, G. Navarro, and S.J. Puglisi. Colored range queries and document retrieval. *Theoretical Computer Science*, 483:36–50, 2013.
- [7] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. ACM-SIAM SODA*, pages 841–850, 2003.
- [8] W.-K. Hon, M. Patil, R. Shah, S. V. Thankachan, and J. S. Vitter. Indexes for document retrieval with relevance. LNCS 8066, pages 351–362, 2013.
- [9] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comp. Bio.*, 17(3):281–308, 2010.
- [10] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. ACM-SIAM SODA*, pages 657–666, 2002.
- [11] G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014.
- [12] G. Navarro and A. Ordóñez. Grammar compressed sequences with rank/select support. In *Proc. SPIRE*, LNCS 8799, pages 31–44, 2014.
- [13] G. Navarro, S. J. Puglisi, and J. Sirén. Document retrieval on repetitive collections. In *Proc. ESA*, LNCS 8737, pages 725–736, 2014.
- [14] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ACM-SIAM ALENEX*, 2007.
- [15] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5:12–22, 2007.
- [16] J. Zobel and A. Moffat. Exploring the similarity space. *SIGIR Forum*, 32(1):18–34, 1998.



**Figure 2:** Document counting on different datasets. We show the average time in microseconds required by a count query, as a function of the size of the document counting structure in bits per character.