Efficient Compressed Indexing for Approximate Top-k String Retrieval [⋆]

Héctor Ferrada and Gonzalo Navarro

Department of Computer Science, University of Chile.
{hferrada,gnavarro}@dcc.uchile.cl

Abstract. Given a collection of strings (called documents), the *top-k document retrieval* problem is that of, given a string pattern p, finding the k documents where p appears most often. This is a basic task in most information retrieval scenarios. The best current implementations require 20–30 bits per character (bpc) and k to $4k$ microseconds per query, or 12–24 bpc and 1–10 milliseconds per query. We introduce a Lempel-Ziv compressed data structure that occupies 5–10 bpc to answer queries in around k microseconds. The drawback is that the answer is approximate, but we show that its quality improves asymptotically with the size of the collection, being over 85% already for patterns of length 4–6 on rather small collections, and improving for larger ones.

1 Introduction

Finding the k documents most relevant to a search query is the most basic information retrieval problem. Originally defined on natural language text collections, its generalization to collections of arbitrary strings turns out to be a problem arising naturally in areas like bioinformatics, multimedia databases, software repositories, and so on [11]. For example, one might want to find the genes where a certain motif appears most often (as they may deserve further biological analysis), modules where a function is called most often (to spot cohesion issues in software design), songs containing most occurrences of a certain sequence (to hint plagiarism), and so on. On East Asian languages like Chinese and Korean, classical solutions for Western natural languages are not applicable, and they are usually handled as generic string collections as well.

Our collection will contain D *documents*, which are strings $d_1, \ldots, d_D$, over an alphabet $[1, \sigma]$, of total length $n = \sum |d_i|$. We preprocess them to build an *index*. Later, given a pattern string $p[1, m]$ and a threshold k, we must list the k documents where p appears most often. In natural language searching the measure of relevance can be more sophisticated than just the number of occurrences of p, but frequency is still a key component. Usually even more complex measures are used in a second step, where the top-k documents are further filtered to obtain the final result [14].

⋆ Partially funded by Fondecyt grant 1-140796, Chile.

Hon et al. [4] proposed a first index for this problem, but its space usage is superlinear, $O(n \log n)$ words; their implementation also uses too much space. Later, Hon et al. [6] presented a structure using linear space, that is, $O(n)$ words or $O(n \log n)$ *bits*. They solved queries in $O(m + k \log k)$ time. Navarro and Nekrich [12] reduced the time to the optimal $O(m + k)$. Konow and Navarro [7] implemented this index, obtaining an index that uses 20–30 bits per character (bpc)[1] and answers top-k queries in k to $4k$ microseconds (μsec). Their time complexity is $O(m + (k + \log\log n)\log\log n)$ with high probability, on statistically typical texts [15]. Shah et al. [5] proposed another index that is not yet implemented, but it is likely to perform similarly, and has a time complexity of $O(m + (\log\log n)^6 + k(\log\sigma \log\log n)^{1+\epsilon})$ for any constant $\epsilon > 0$. Navarro and Valenzuela [13] aimed at using less space, reaching 12–24 bpc depending on the compressibility of the collections, but retrieval times are an order of magnitude higher, 1 to 10 milliseconds (the time complexity is upper-bounded by $O(m + k \log^{4+\epsilon} n)$). There are several other theoretical proposals [11] that promise to use much less space than current implementations, but that are most likely to be even slower in practice (as already hinted in current studies [13]).

In this paper we introduce an index that uses much less space and time than current alternatives. It is based on Lempel-Ziv compression, precisely LZ78 [16], which compresses texts by building a dictionary of frequent strings (called phrases) and then parsing the text as a sequence of n' phrases. It holds $n' \le n/\lg_\sigma n$ for any text, and moreover $n' \lg n = nH_k + O(n(k\log\sigma + \log\log n)/\log_\sigma n)$ for any k, where H_k is the k-th order empirical entropy of the text [8]. This is $n' \lg n = nH_k + o(n\log\sigma)$ for any $k = o(\log_\sigma n)$. Our structure builds on previous LZ78-compressed indexes called LZ-indexes, developed for finding all the occurrences of p [10, 2] and for listing all the documents where p occurs [3]. Like these indexes, our structure uses, in practice, $(2+\epsilon)n'\lg n + O(n'\log\sigma) = (2+\epsilon)nH_k + o(n\log\sigma)$ bits, and it solves top-k queries in time $O(m + k\log D)$. In practice, the space is around 5–10 bpc to achieve a query time around k μsec. This time/space tradeoff is well below that of previous implementations.

In exchange, our top-k answer is approximate, as we consider only the occurrences of p within phrases. If the text is generated by a stationary source, the occurrences of any pattern p appear regularly, every d positions on average (e.g., $d = \sigma^m$ if the symbols are generated uniformly and independently). On the other hand, since $n' \le n/\lg_\sigma n$, only $(n/d)m(n'/n) \le (n/d)m/\lg_\sigma n$ of those occurrences hit a phrase boundary on average. This means that that a fraction of $1 - m/\lg_\sigma n$ of the occurrences are within phrases (the fraction improves to $1 - mH_k/\lg n$ on compressible texts). Thus, we are considering asymptotically all of the occurrences of p when building the approximate top-k answers for short enough patterns, $m = o(\log_\sigma n)$. Note that, if $m \ge \lg_\sigma n$, then it occurs $O(1)$ times on average in the collection, and then a plain listing of all the documents where it appears [3] is an appropriate tool to find its top-k documents.

We show that, already on moderate collections of $n =$ 25–130 MB, the quality of the answer (measured as the number of occurrences of p on the k retrieved

[1] The space results they report [7] are somewhat underestimated, as we show here.

documents as a percentage of the number of occurrences on the actual top-k documents) is always over 85% for short patterns (m = 4–6), improving as the collection size grows and as the collection becomes more compressible with LZ78.

2 The LZ-Index

Assume we concatenate the documents $d_1 \cdots d_D$ (each terminated with a special symbol \$) into a text $T[1, n]$ over alphabet $[1, \sigma]$. The LZ78 compression algorithm cuts the text into n' distinct *phrases*, each of which is equal to the longest possible previous phrase plus the following new symbol. Each phrase is then replaced by the *id* of the previous corresponding phrase and the extra symbol. The number of bits output by the compressor is $|\mathsf{LZ78}| = n'(\lg n' + \lg \sigma)$, which converges to the statistical entropy of T [8], and it always holds $n' \leq n/\lg_\sigma n$. The LZ-index [10] is a text index built on the LZ78 parsing of the text, and it supports locating the occurrences of a pattern $p[1, m]$ in T. The index is formed by the following components (among others not relevant for this paper).

1. ***LZTrie***: a trie composed of all the phrases produced by the LZ78 parsing. Note that the set of phrases is prefix-closed (the prefix of a phrase is also a phrase), so LZTrie has n' nodes. It stores the phrase identifier of each node.
2. ***RevTrie***: a trie storing the reversed phrases. It is not prefix-closed, so there are *empty* nodes not associated to phrases. It contains originally t_{rev} nodes. We contract unary paths of empty nodes to a single edge, after which the trie has $n_{rev} = n' + n_e \leq 2n'$ nodes, where n_e empty nodes remain after contracting. The phrase numbers of the n' nonempty nodes are stored.
3. ***Node***: an array mapping from phrase numbers to their preorder in LZTrie.

The modern version [2] of the LZ-index uses $(2+\epsilon)|\mathsf{LZ78}|(1+o(1))$ bits, for any $\epsilon>0$. The occurrences of pattern p are divided into type 1 (those completely inside a phrase), and types 2 and 3 (those spanning two or more phrases, respectively). Those are found separately at search time. In this paper we are only interested in finding occurrences of type 1. For those, we search for p^r (the reversed pattern) in RevTrie, arriving at node v^r. Each node u^r descending from v^r (including v^r) corresponds to an occurrence of type 1 where p appears at the end of the phrase. The other occurrences of type 1 are the nodes u' that descend from u in LZTrie, where u corresponds to u^r. Thus, for each node u^r that is nonempty, we read the phrase id f_u of u^r, compute $u = Node(f_u)$, and report all the phrase ids in the subtree of u. It takes $O(m + occ)$ time to report the occ type-1 occurrences.

3 An LZ-Index for Approximate Top-k Retrieval

Our top-k retrieval LZ-Index is a modification of the original LZ-Index. This tree will be stored as in previous work [3]:

***LZTrie*:** We store only the topology and the documents where the phrases lie, using in total $n' \lg D + O(n')$ bits.

- **P_{lz}:** The LZTrie topology represented with parentheses in a preorder traversal, and made navigable in $O(1)$ time using $2n' + o(n')$ bits (`FF` [1]).
- **D_{lz}:** An array storing, for each node v of LZTrie in preorder, the document identifier (using $\lceil \lg D \rceil$ bits) where its corresponding phrase lies.

***Revtrie*:** We store the structures needed to carry out searches directly on it, without the help of the LZTrie, using in total $t_{rev} \lg \sigma + O(t_{rev})$ bits. In theory t_{rev} can be as large as n but in practice it is much closer to $n_{rev} \leq 2n'$.

- **P_{rev}:** The tree topology using parentheses and made constant-time navigable, using $2t_{rev} + o(t_{rev})$ bits (`FF` [1]).
- **E_{rev}:** A bitmap marking empty nodes, in preorder, using t_{rev} bits.
- **U_{rev}:** A bitmap marking empty unary nodes (i.e., contracted), from those that are marked empty in E_{rev}, using $t_{rev} - n'$ bits.
- **L_{rev}:** A sequence of the n_{rev} letters that label the non-contracted edges leading to the nodes, in preorder. Used to find the child nodes at searching.
- **M_{rev}:** A sequence of the $t_{rev} - n_{rev}$ letters that label the contracted edges leading to the nodes, in preorder. Used to check that the characters in the contracted edge match the search pattern.

All the bitmaps are stored with sublinear extra data structures that solve *rank*/*select* operations in constant time [9]. This allows, for example, finding the jth 0 or 1 in the bitmap in constant time, or count the number of 0s or 1s in any bitmap interval.

***Node*:** This is recast as a mapping from nonempty RevTrie nodes to their LZTrie preorder numbers, using $n' \lg n' + O(n')$ bits.

***Top*:** To solve top-k document retrieval for any $k \leq k^*$, where k^* is a parameter defined at construction time, we will store the top-k^* answers, in decreasing frequency order, for some RevTrie nodes. We use a parameter g to define the RevTrie nodes that will store their top-k^* answer: These will be the (empty or nonempty) nodes representing a string with at least gk^* occurrences of type 1 in T. Empty unary nodes will not store their answer set, as this is the same of its child. The marking will be node for all the k^* values in $[1..D]$ that are a power of 2. Nodes v will store their top-k^* answers for the maximum k^* value for which they are marked. This is implemented with the following additional structures:

- **B_{top}:** A bitmap of size n_{rev} marking which RevTrie nodes have top-k^* answers precomputed, in preorder.
- **K_{top}:** The sequences of k^* most frequent documents where each node marked in B_{top} appears, concatenated in the same order of B_{top}. The identifiers are stored using $\lceil \lg D \rceil$ bits, in decreasing frequency order.
- **LK_{top}:** A bitmap marking the starting positions of the sequences in K_{top}.
- **A_{top}:** Since there may be less than k^* distinct documents where the marked node appears, this bitmap indicates whether a node marked in B_{top} already lists all of the possible documents.

The larger g, the fewer RevTrie nodes store their top-k^* documents: While in theory we might store up to $n_{rev}\, k^* \lg D$ bits, in practice this is much closer to $(n_{rev}/(gk^*))\, k^* \lg D = (n_{rev} \lg D)/g$, which added over the $\lg D$ values for k^* gives $(n_{rev} \lg^2 D)/g$ bits. The other bitmaps use $O(n_{rev})$ bits.

The overall space is, in practice, upper bounded by $n'(\lg n' + \lg D + 2\lg\sigma + 2(\lg^2 D)/g + O(1))$ bits. Thus a value like $g = \Theta(\lg D)$ obtains space similar to the original pattern-matching LZ-index, $(2+\epsilon)n' \lg n + O(n' \log\sigma)$ bits [2].

Querying. At query time, we find the RevTrie node corresponding to p^r, move to its highest descendant not marked in U_{rev}, v^r, and check if it is marked in B_{top}. If marked and either (1) A_{top} indicates it stores all the possible documents, or (2) LK_{top} indicates that it stores $k' \geq k$ top documents, then we return the first k documents stored for v^r in K_{top} and finish. Otherwise, we need to solve the answer by brute force, by traversing all its type-1 occurrences. By construction, this takes place only if v^r has $k' < k$ answers stored (including the case $k' = 0$). If k^* is the power of 2 next to k, then v^r does not store its top-k^* answers, thus by construction it has less than gk^* occurrences of type 1. Therefore the brute-force process must traverse $O(gk)$ occurrences.

In order to solve v^r by brute force, we use P_{rev} to compute its preorder i_v and its subtree size s_v. Thus all the subtree of v^r has the preorder interval $[i_v, i_v + s_v - 1]$. Then we use *rank* on E_{rev} to map it to the interval $[i_1, i_2]$ of nonempty preorder values. For each i in this interval, we compute $i_u = Node(i)$, which is the preorder of the corresponding node in LZTrie, and then use P_{lz} to obtain the corresponding node u in LZTrie. Then we compute the size s_u of u and obtain the interval $[i_u, i_u + s_u - 1]$ of all the descendants of u in LZTrie. We process all the document identifiers in $D_{lz}[i_u, i_u + s_u - 1]$, for all the nodes u in LZTrie that correspond to all the RevTrie descendants u^r of v^r, accumulating their frequencies and then choosing the k highest ones. The whole process takes $O(m + gk + k\log k) = O(m + k\log D)$ time.

4 Experimental Results

We use various document collections, following previous work [13, 7] and exploring different aspects of statistical compressibility, size, number of documents, and repetitiveness: `ClueWiki` (English, few large documents), `DNA` (synthetic, mildly repetitive with 5% mutations among documents), `KGS` (Go game records), `Wiki` (more and shorter documents), `Proteins` (many more documents, almost incompressible), and `TodoCL` (a snapshot of the Chilean Web, with real queries, used to measure quality). Table 1 shows their main characteristics (column "compress" shows how the LZ78-based Unix Compress program compresses them).

Our machine is an Intel Xeon with 8 processors of 2.4GHz and 12MB cache, with 96GB RAM. It runs Linux 2.6.32-46-server, and we use gcc with full optimization and no multithreading. We chose 40,000 patterns of lengths $m = 3$ and $m = 8$ extracted randomly from the collection.

4.1 Time and Space

Table 1 shows the size of our structure with $g = 512$, where it uses almost its minimum possible space, and with $g = 128$, where it achieves around k μsec to

Collection	n (MB)	D	n/n'	compress (bpc)	$g = 512$ (bpc)	$g = 128$ (bpc)
`ClueWiki`	131	3,334	17.24	3.63	4.50	6.31
`DNA`	95	10,000	11.50	2.68	4.86	5.30
`KGS`	25	18,838	14.97	1.85	5.13	6.23
`Wiki`	80	40,000	9.58	3.34	6.73	7.43
`Proteins`	56	143,244	6.43	4.61	9.58	10.10
`TodoCL.200`	200	48,186	9.86	3.91	7.32	6.65

Table 1. Main measures related to the space usage of our index. We refer here to the first 200MB of `TodoCL`.

solve queries, as we will see. The minimum space ranges from 1.2 to 2.8 times the space of Unix Compress. For this value of g our analysis predicts a factor around 2. On the other hand, our index uses around 5–10 bpc with $g = 128$.

Fig. 1 gives the breakdown of the space obtained by our index on those collections, for increasing values of g. The components are LZTrie (the tree topology and the document identifiers, which dominate), RevTrie (the tree topology and the letters), array Node, and Top (the storage of the best documents for some precomputed nodes). We show the breakdown as cumulative space curves. As g increases, the Top component is reduced and the structure becomes slower.

Now we compare our structure with previous work. We consider search patterns of lengths $m = 3$ in Fig. 2 and $m = 8$ in Fig. 3, and measure the cost to compute the top-10 and top-100 documents, for $g = 512, 256, 128, \ldots$. We denote DCC'13 the existing fast and large structure [7] and denote SEA'12 a choice of relevant space/time tradeoffs of the existing small and slow structure [13]. In most texts, our structure uses 5–7 bpc to solve top-k queries in around k μsec. The exception is `Proteins`, where it uses around 10 bpc due to its incompressibility. Except on `Proteins`, where it uses over 20 bpc, structure SEA'12 can use similar or less space than ours, but at the cost of being 4–5 orders of magnitude slower. Even if using much more space, SEA'12 is at least 10 times slower than ours. Structure DCC'13, on the other hand, is 4–50 times slower than ours, and uses 2.5–7 times our space.

4.2 Quality

The drawback is that our structure delivers approximate top-k answers. We present in Fig. 4 two measures of the quality of the answer. On the left we show traditional recall, measured in the following way: for each value $k' \in [1, k]$, we measure how many of the (correct) top-k' documents are reported within the (approximate) top-k results. For example, the point at $k' = 1$ indicates how many times the most relevant document is contained in our top-k answer. The point at $k' = k$ indicates how many of the correct top-k documents are actually returned. This measure is interesting in applications where the top-k answer is postprocessed with a more sophisticated relevance function in order to deliver a final answer of $k' \ll k$ results. The figure shows that, in this scenario, the k'

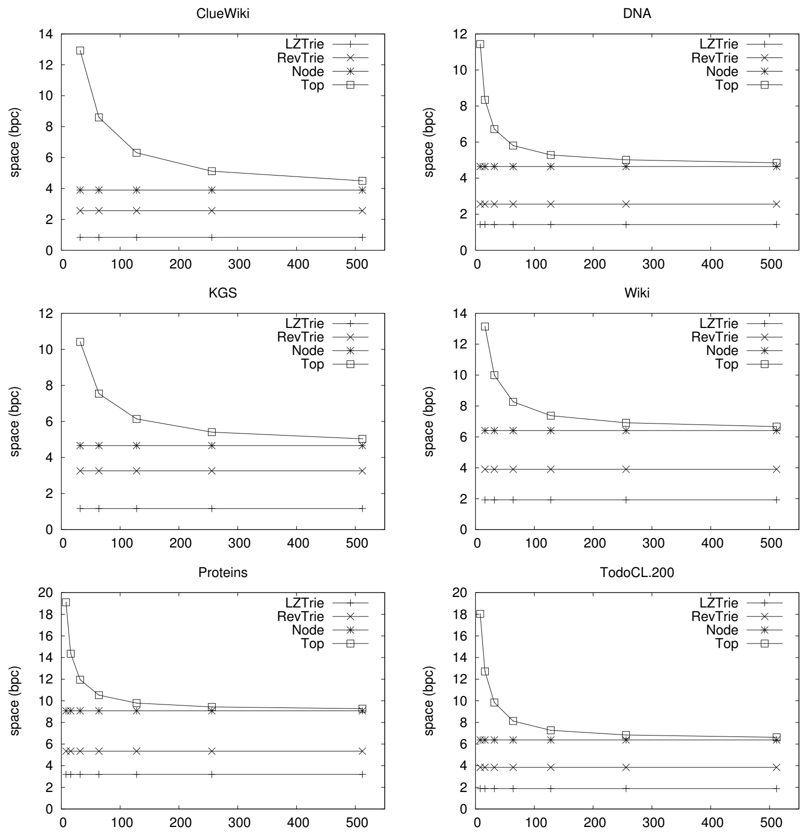

Fig. 1. Space breakdown of our structures for different k and g values (g is the x-axis).

most relevant candidates are frequently in the approximate top-k answer set, for small k'. However, when k' becomes closer to k, the recall degrades, more or less depending on the collection, and faster for $m = 8$ than for $m = 3$. On the other hand, there are no significant differences between the results for $k = 10$ and for $k = 100$. Results are particularly bad for `DNA`, `KGS`, and `Proteins`.

If our index fails to return a top-k document but returns another one with the same frequency, we take it as a hit, as both are equally good. In this sense, recall is a too strict measure of relevance: if the system returns a document with only slightly fewer occurrences than the correct one, it counts as zero. As the frequency is only a rough measure of relevance, a much more precise measure of quality is the sum of the frequencies of the documents in the approximate top-k answer as a fraction of the sum in the correct top-k answer.

Fig. 4 (right) shows the results under this measure of quality. All collections perform very well for $k = 3$, reaching 90%–100% of quality even for $k' = k$. For $k = 8$, collections `ClueWiki` and `KGS` still achieve a reasonable quality over 80%, `DNA` over 60%, `Wiki` over 50%, and `Proteins` only 10%. These differences

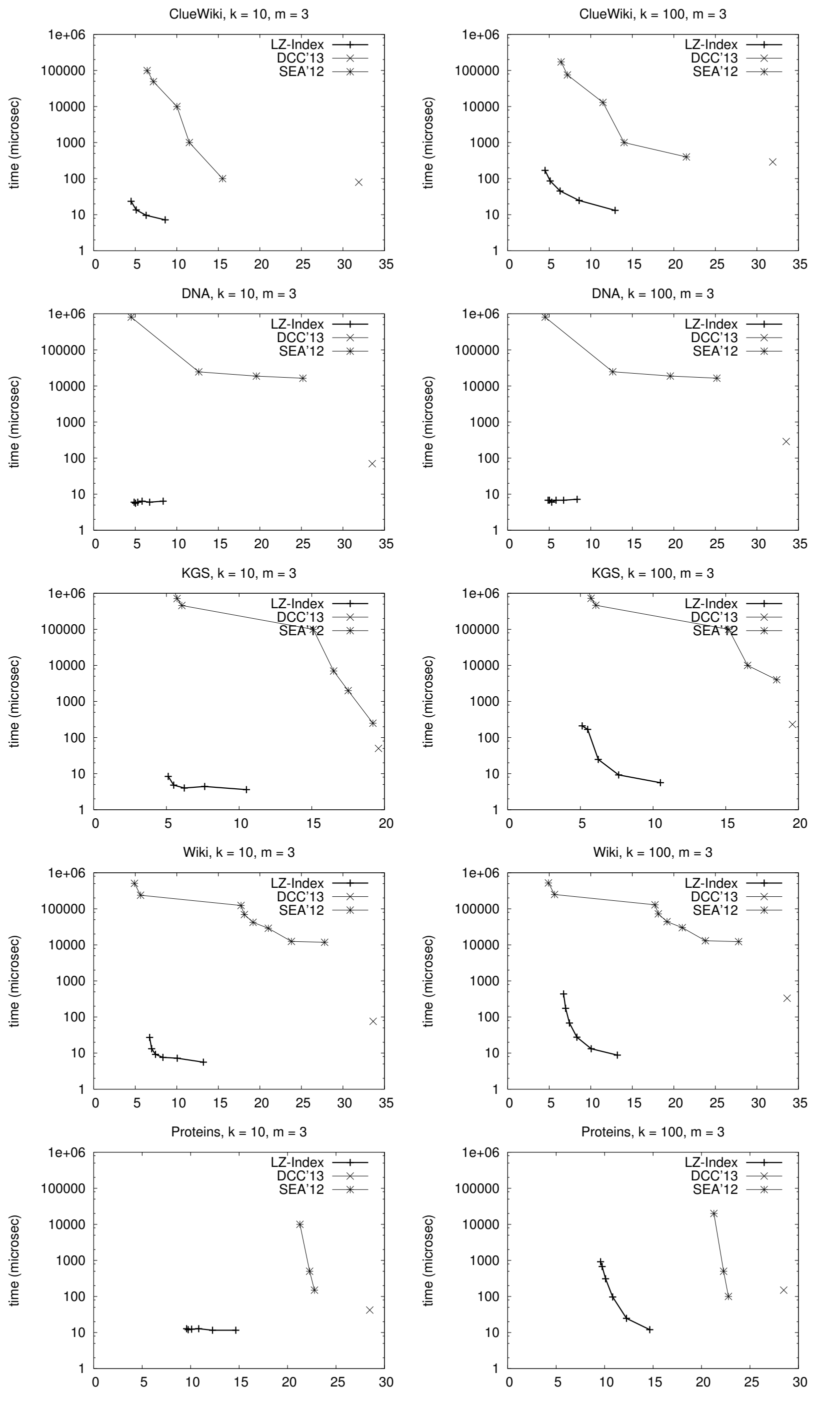

Fig. 2. Space/time comparison for pattern length $m = 3$. Space (bpc) is the x-axis.

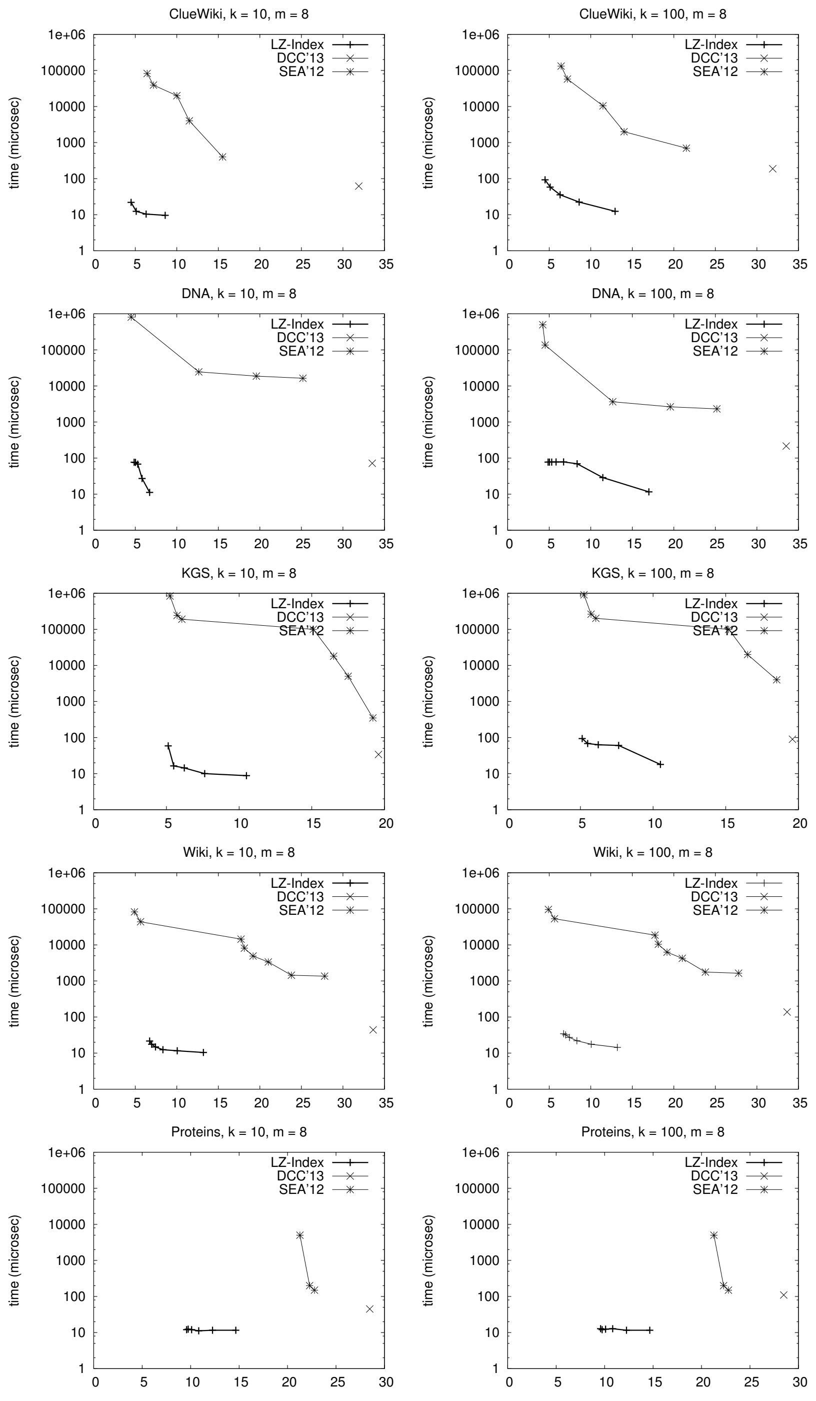

Fig. 3. Space/time comparison for pattern length $m = 8$. Space (bpc) is the x-axis.

in quality can be predicted with Table 1: the less compressible the collection, the smaller n/n', and the worse quality obtained for a given pattern length m.

On the other hand, the fact that better quality is obtained for shorter patterns coincides with our probabilistic analysis. Fig. 5 illustrates this effect more closely, for increasing pattern lengths. It can be seen that, for the moderate collection sizes of 25–130 MB we considered, we obtain quality well above 85% for $m =$ 4–6, depending on the text type and its n/n' value. Fig. 6 shows the case of real query words (of length > 3 to exclude most stopwords, average length 7.2) and 2-word phrases (average length 8.0), on increasing prefixes of `TodoCL` converted to lowercase (as case is generally ignored in natural language queries). As predicted, the quality improves with n, from 33%–46% on 200 MB ($n/n' = 10.1$) up to 59%–72% on 1.6 GB ($n/n' = 12.5$).

5 Conclusions

We have introduced a top-k retrieval index for general string collections, based on Lempel-Ziv compression. The index is orders of magnitude faster, and uses much less space, than previous work. In exchange, it delivers approximate top-k answers, which is acceptable in most applications. We analytically show that the answers tend to exactness asymptotically, when the collection is large enough compared to the pattern length. Our experiments also show that the quality of the answer is good enough for short patterns already on our moderate-size text collections. The larger the text collection, or the more compressible it is with LZ78, the longer the patterns that can be searched with high quality. In this sense, the index is a very promising alternative to handle the large text collections one aims at in real life.

We obtain good-quality results for real word queries on a moderately large text collection. Our next step is to use our index to find top-k candidate documents for the individual words of multiword queries and then postprocessing the result into weighted conjunctive or disjunctive queries [14].

In natural language, retrieving approximate top-k answers to improve efficiency is a common practice. This avenue has not been explored much for general string collections. Our work shows that this idea is promising, as large space and time reductions are possible while still returning answers of good quality.

References

1. D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. ALENEX*, pages 84–97, 2010.
2. D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1):54–101, 2012.
3. H. Ferrada and G. Navarro. A Lempel-Ziv compressed structure for document listing. In *Proc. SPIRE*, LNCS 8214, pages 116–128, 2013.
4. W.-K. Hon, M. Patil, R. Shah, and S.-B. Wu. Efficient index for retrieving top-k most frequent documents. *J. Discr. Alg.*, 8(4):402–417, 2010.

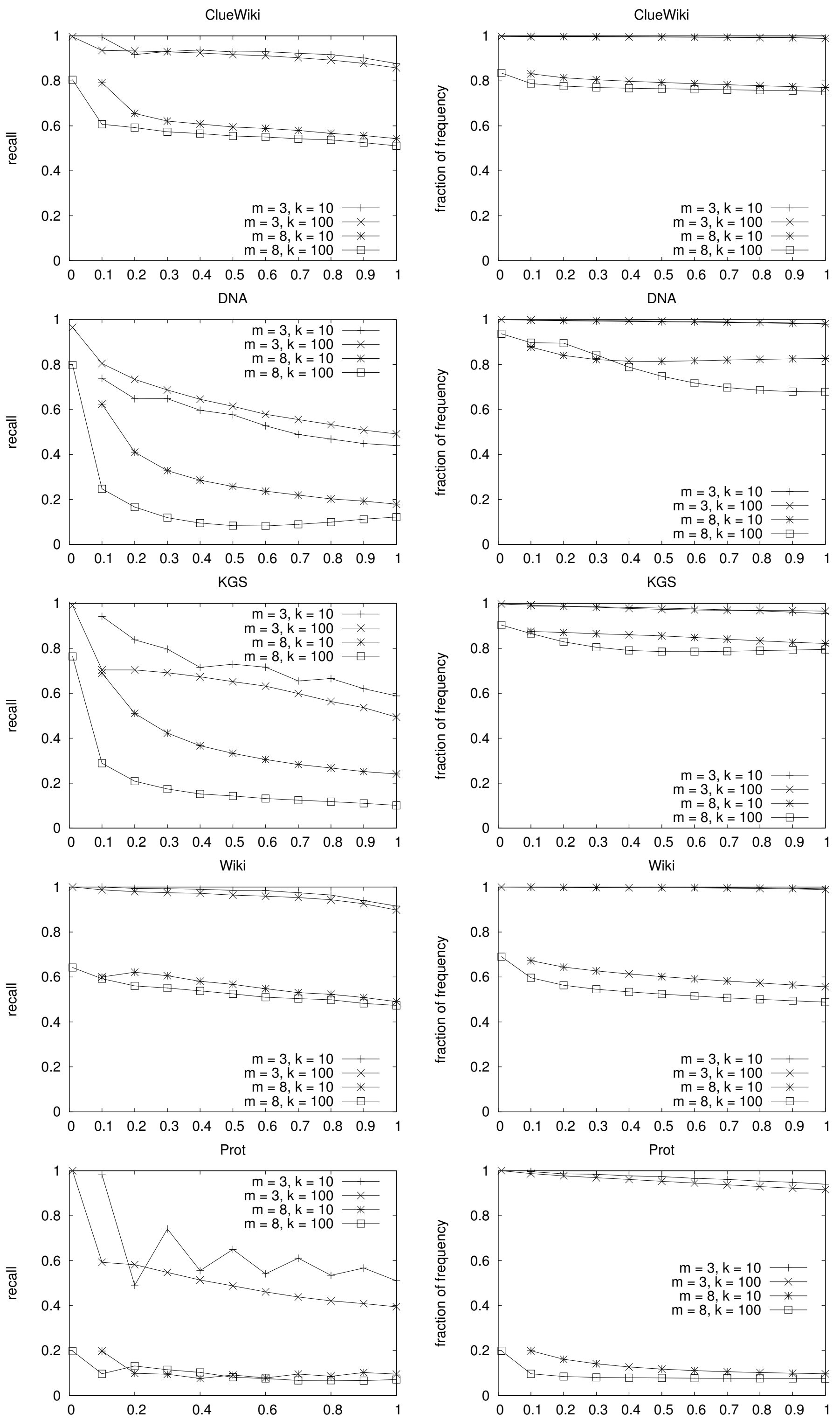

Fig. 4. Recall (left) and quality (right) of our approximate top-k solution, as a function of the fraction of the answer (x-axis).

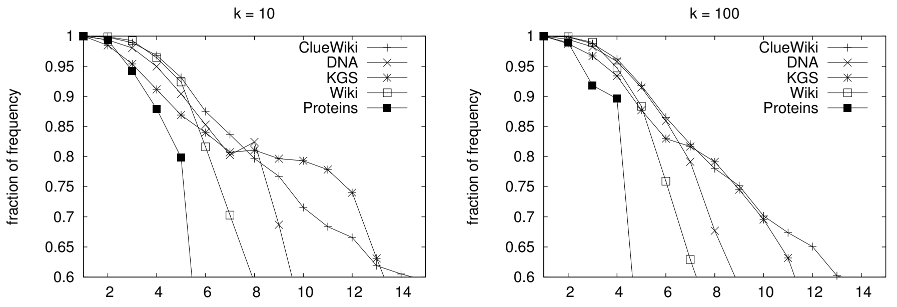

Fig. 5. Quality of our approximate top-k solution, as a function of the pattern length, for top-10 (left) and top-100 (right).

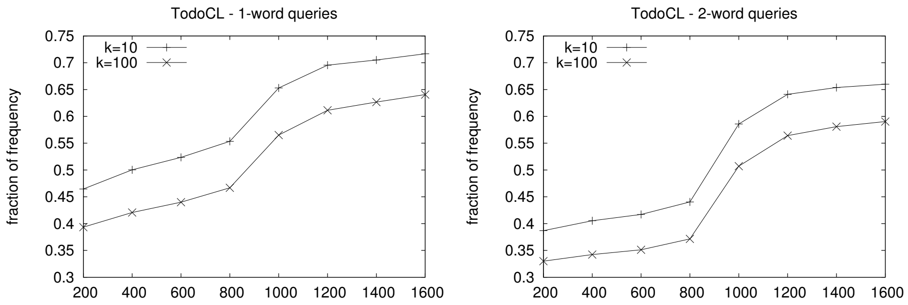

Fig. 6. Quality of our approximate top-k solution, as a function of the prefix size of `TodoCL` in MB, for words (left) and phrases of 2 words (right).

5. W.-K. Hon, R. Shah, and S. V. Thankachan. Towards an optimal space-and-query-time index for top-k document retrieval. In *Proc. CPM*, pages 173–184, 2012.
6. W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722, 2009.
7. R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. DCC*, pages 351–360, 2013.
8. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. Comp.*, 29(3):893–911, 1999.
9. I. Munro. Tables. In *Proc. FSTTCS*, pages 37–42, 1996.
10. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discr. Alg.*, 2(1):87–114, 2004.
11. G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comp. Surv.*, 46(4):article 52, 2014.
12. G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. SODA*, pages 1066–1077, 2012.
13. G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *Proc. SEA*, LNCS 7276, pages 307–319, 2012.
14. C. Clarke S. Büttcher and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, 2010.
15. W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Comp.*, 22(6):1176–1198, 1993.

16. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.*, 24(5):530–536, 1978.