Lempel-Ziv Compressed Structures for Document Retrieval[☆]

Héctor Ferrada

Universidad Austral de Chile, Chile

Gonzalo Navarro

CeBiB — Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile

Abstract

Document retrieval structures index a collection of string documents, to quickly retrieve those that satisfy a given condition for a pattern string p. For document listing, we must list all the documents where p appears. For top-k retrieval, we must list the k most relevant documents for p, for some relevance criterion (such as the frequency of p in the documents). There exist optimal-time and linear-space solutions for both problems, but they use too much space in practice. Most current research uses compressed suffix arrays, but fast indices still use 17–21 bpc (bits per character), whereas small indices take several milliseconds to return each answer.

This work presents the first document retrieval structures based on Lempel-Ziv compression. We build on the LZ78 parsing of the collection, which yields structures using 7–10 bpc that dominate an important area in the current space/time tradeoff map. In addition, the structures allow for much more efficient partial or approximate answers, which are acceptable in many applications: the structure for document listing outputs the first 75%–80% of the answers at a rate of one answer per microsecond, and the top-k retrieval

[☆]Early parts of this work appeared in *Proc. SPIRE 2013* [9] and *Proc. SPIRE 2014* [10]. Supported by Fondecyt Grant 1-170048, Chile; Basal Funds FB0001, Conicyt, Chile; and a CONICYT-Chile Doctoral Scolarship.

Email addresses: hferrada@inf.uach.cl (Héctor Ferrada), gnavarro@dcc.uchile.cl (Gonzalo Navarro)

structure returns a result of 90% quality at the same rate and using just 4–6 bpc. Current indices using such a little space are orders of magnitude slower, whereas indices achieving those speeds are 2–4 times larger.

Key words: Document retrieval, document listing, top-k queries, string databases, compressed data structures.

---

1. Introduction

Web searching is the best-known example of a document retrieval problem. A data structure (called an index) is built on a set of documents so that, later, given a query, some documents that best fit the query are returned. This is only an example of a wider set of similar problems, which arise when managing software repositories, genome and protein collections, music sequences, time series, log files, file systems, and so on. In its more general form, documents are simply sequences of symbols (that is, strings) and queries are also strings (called patterns). While different applications require different types of searches, the basic search for simple string patterns underlies most document retrieval solutions.

There are several kinds of document retrieval problems. The most basic one, *document listing*, aims to retrieve all the documents where the pattern appears. For example, one might want to retrieve all the genes containing some biological marker, all the MIDI files containing a short theme, or all the source files calling a function. A more sophisticated problem is *top-k retrieval*, where a given *score function* defines how relevant is a document to a pattern, and one retrieves the k documents most relevant to the given pattern. For example, one might want to retrieve the genomes where a given DNA motif appears most frequently, the security log files where a suspicious sequence of accesses to the system occurs most often, the days where a word was tweeted most frequently, and so on. While the mere frequency of the pattern in the document is a typical score function, more sophisticated ones are common in Web searching and information retrieval on natural language.

While inverted indices are by far the most popular structures for document retrieval problems on natural language text collections [3], document retrieval on more general sequence collections requires other data structures. Suffix trees [47] and suffix arrays [27] are appropriate data structures to index general sequences, and optimal-time indices for document listing [31] and top-k retrieval [21, 36] have been built on them. These take $O(1)$ time

per retrieved document and require linear space. However, this is in practice many times the size of the collection itself, which renders those indices hard to apply in real scenarios.

Much research has been done on reducing the space of document retrieval indices [34]. In practice, the fastest ones [8, 24, 17] still require 17–21 bpc (bits per character), that is, 2–3 times the collection size if symbols are represented as bytes. More succinct representations [43, 21, 38, 4] may use as little as 7 bpc, but at the cost of slowing down query times by several orders of magnitude (they take at best $O(\lg^{1+\epsilon} n)$ time per retrieved document, where n is the collection size measured in symbols and $\epsilon > 0$ is a small constant).[1]

All the current reduced-space solutions build on compressed suffix arrays [35], which offer suffix array functionality within the space of the compressed text collection. This functionality includes counting and locating the positions where the pattern occurs in the text collection, and the compression is statistical [28]. An alternative to compressed suffix arrays are the LZ-indices [12, 32, 33, 41, 2], which build on the Lempel-Ziv 1978 (LZ78) compression method [48]. These are faster than compressed suffix arrays to locate the pattern occurrences, yet they cannot count them without locating them all.

In this article we show that LZ-indices offer a novel and attractive approach to document retrieval on general sequences. LZ78 parses the text into n' so-called phrases, which are at most $n/\lg_\sigma n$ (where σ is the alphabet size) and in practice 1/20 – 1/6 of n. We manage to use the fast and large document retrieval structures on the sequence of n' phrases, not of n symbols, which reduces their size by an order of magnitude, while still taking advantage of their speed.

We obtain indices that are competitive with the best succinct solutions. Our indices use $(3\text{–}5)nH_l + O(n)$ bits, where H_l is the l-th order entropy of the collection, for any $l = o(\lg_\sigma n)$. This is in practice 7–10 bpc in most texts. In a (very pessimistic) worst-case each retrieved answer may take up to $O(m \lg^2 n)$ time for document listing (where m is the pattern length), and $O(\lg^3 n)$ for top-k retrieval, but this is as low as $O(1)$ for the majority of the answers. In practice, each answer is returned in 10–100 microseconds (μs) on our machine[2], which places these indices in a dominating position on a

[1] We use lg instead of log to indicate that the logarithm is in base 2, where it matters.

[2] An Intel Xeon with 8 processors of 2.4GHz and 12MB cache running Linux, see Section 4.3.

large portion of the space/time tradeoff map.

Furthermore, both indices allow for much faster and/or smaller variants that yield partial or approximate answers. The document listing index retrieves in $O(1)$ time (1 μs in practice) the first 75%–80% of the answers. This is useful in scenarios where only some documents must be shown, or they are shown progressively to the user. The top-k retrieval index may return approximate answers within 1–5 μs per answer, while using just $2nH_l + O(n)$ bits (4–6 bpc). This is tolerable in many scenarios where score functions are already an approximation. For example, in Information Retrieval it is customary to take the top-k documents for a large k and then use a more sophisticated scoring function to choose the best among those k [6, 46]. We prove that the index becomes more accurate asymptotically (as the collection size grows) and show experimentally that, in most collections, the retrieved documents add up to 90% of the occurrences of the actual top-k documents for pattern lengths ≤ 8, even for small collections.

Existing (exact) indices using the space of our approximate solutions are several orders of magnitude slower, and those reaching their speeds are 2–4 times larger. This opens the door to a deeper study of these possibilities in existing indices as well.

This article wraps up the results in our previous conference publications [9, 10], including better implementations and more extensive experimental results. The implementations of both document listing and top-k retrieval now offer space/time tradeoffs based on tuning parameters; we also include tips and advices for practical implementations. The experiments include new data sets, new baselines to compare with, and new tests and measurements. Section 2 gives the theoretical background and Section 3 describes the original LZ78-based pattern-matching index. Section 4 discusses the current results, putting our contribution in context at the end. Section 5 describes our solution for document listing, and Section 6 details our proposal for approximate top-k document retrieval. We conclude in Section 7 with final remarks and future work directions.

2. Preliminaries

2.1. Document Retrieval

We are given a collection ∇ of D documents $d_1, d_2, ..., d_D$ of total length $n = \sum_{i=1}^{D} |d_i|$ characters. We preprocess ∇ to build an index, such that later

we can efficiently support queries on it for pattern strings $p_{1..m}$. We define the following Document Retrieval (DR) problems:

Document Listing (DL): List all the distinct documents of ∇ that contain p as a substring.

Top-k Retrieval (top-k). Given also a number k with the query, list k documents of ∇ that contain p. These must be the ones maximizing a given *score function* $w(p, d)$ that measures the relevance of document d for pattern p. A simple and popular score function is the *term frequency* $tf(p, d)$, which is the number of times p occurs as a substring of d.

2.2. Bitvectors

The *bitvector* is a fundamental data structure in compressed text indexing. It represents a sequence $B_{1..n}$ of n bits, supporting access to any position as well as two important queries on prefixes of B, called *rank* and *select*:

- $access(B, i)$ returns the bit at position i, for any $1 \leq i \leq n$.
- $rank_b(B, i)$ returns the number of occurrences of bit $b \in \{0, 1\}$ until and including position i, for any $1 \leq i \leq n$.
- $select_b(B, i)$ returns the position of the ith bit $b \in \{0, 1\}$, for any $1 \leq i \leq rank_b(B, n)$.

There are solutions using $o(n)$ bits on top of B that answer these queries in $O(1)$ time [22, 7]. It is also possible to retain those times while compressing B [40] to $nH_0(B)+o(n)$ bits, where $H_0(B)$ is the *zero-order empirical entropy* [28] of B. If B has m 1s, then $H_0(B) = (m/n)\lg(n/m)+((n-m)/n)\lg(n/(n-m)) = (m/n)\lg(n/m)+O(m/n)$, using $\lg(n/(n-m)) = \lg(1+m/(n-m)) \leq \frac{1}{\ln 2}\, m/(n-m)$.

2.3. Trees

A general ordinal tree allows any number of children per node, and distinguishes their order. Such a tree with n nodes can be represented using just $2n+o(n)$ bits so that a large number of tree navigation and query operations can be carried out in constant time [22, 30, 5, 39].

The most recent representation [39] has been shown to be the most efficient one supporting full navigation functionality [1]. It represents the tree

using $2n$ parentheses (seen as bits). In particular, if we deploy the parentheses in DFUDS format (Depth-First Unary Degree Sequence [5]), then all the children of a node have contiguous indices in the representation, which is useful to implement tries (see next).

2.4. Tries

The *trie*, or *digital tree* [15, 23], is a structure that organizes a set of strings as a tree, so that the search for a string $p_{1..m}$ can be performed in $O(m)$ time. Each trie node represents a different prefix of the string set, and each edge is labeled by an alphabet symbol. The root node represents the empty string, and each other node represents the concatenation of the edge labels in the path from the root to the node. Tries are built in time proportional to the total length of the strings stored. To search for p, we traverse the trie downwards from the root, following the symbols of p and selecting the matching edge labels. With a similar method it is also possible to find the strings prefixed by p or, if p is not in the set, the longest prefix of p that coincides with the prefix of some string in the set.

If a trie has n nodes, we can store its topology using $2n + o(n)$ bits using the DFUDS representation (Section 2.3). This representation can be extended to assign increasing labels from $[1..\sigma]$ to the children of a node, so that one can navigate towards the desired child in $O(1)$ time [5]. This scheme requires $n \lg \sigma + 2n + o(n)$ bits and searches for $p_{1..m}$ in time $O(m)$.

2.5. Suffix Trees

The *suffix tree (ST)* [47] of a string $T_{1..n}$ is essentially a compacted trie of all the suffixes of T, that is, of the n strings $T_{i..n}$. Compacted means that, in the suffix tree, paths formed by unary nodes (i.e., with a single child) are replaced by a single edge labeled with the concatenation of the edge labels in the path. The children of a node are ordered lexicographically from left to right. Each leaf represents a suffix $T_{i..n}$ and stores the position i. Internal nodes represent substrings of T that appear more than once.

Since every occurrence of $p_{1..m}$ in T is a prefix of some suffix, the suffix tree can be used for *pattern matching*, that is, finding the occurrences of p in T. We just traverse the tree as for trie search. If at some point we have no edge to follow, then p does not occur in T. If we use all the symbols of p, then the current node is called the *locus* of p in the suffix tree. Suffix tree edges may have several symbols, and thus p may be consumed in the middle of an edge. In this case the locus is the child node of the edge. Every leaf

descending from the locus and storing a position i represents an occurrence of p at $T_{i..i+m-1}$. If we store the number of leaves descending from each suffix tree node, we can count the number of occurrences of p.

If we can find each child in constant time, then the suffix tree counts the number of occurrences of p in time $O(m)$, and locates the *occ* occurrences in T in time $O(m + occ)$. The suffix tree uses $O(n \lg n)$ bits of space and can be built in $O(n)$ time. In practice, it uses 10–20 times the size of T.

A *Generalized Suffix Tree (GST)* on ∇ is a suffix tree built on the concatenation of all the documents in ∇, $T_{1..n} = d_1 d_2 ... d_D$. Each document d_i is assumed to be terminated with a special symbol $\$_i$ that does not appear elsewhere. We can perform pattern searches on the GST just as on a suffix tree, finding all the occurrences of p in ∇. This is not, however, an efficient way to perform DL, because we extract all the individual occurrences of p in each document we report.

2.6. Suffix Arrays

The *suffix array (SA)* [27] is a structure that reduces the size of the suffix tree to about 4 times the text size (but still $O(n \lg n)$) in exchange for supporting fewer functionalities. It still supports pattern matching, however. The SA is an array of n integers with the positions i associated with the suffix tree leaves, in left-to-right order. It can also be seen as the set of (indices of) suffixes $T_{1..n}$ in lexicographic order. Since all the suffixes starting with p are contiguous in the SA, it can be binary searched for p to delimit the segment of all the suffixes starting with p, in time $O(m \lg n)$.

Like the GST, when the text is a concatenation of the documents of a collection ∇, we speak of the *generalized suffix array (GSA)* of ∇.

Figure 1 shows an example of suffix trees and arrays.

2.7. Wavelet Trees

The *wavelet tree (WT)* [18] for a symbol sequence $S_{1..n}$ over an alphabet Σ of size σ is a balanced binary tree, which can compute $access(S, i)$, $rank_c(S, i)$ and $select_c(S, i)$, for any $c \in \Sigma$, in time $O(\lg \sigma)$. These are the natural extensions of the bitvector operations to general alphabets.

The wavelet tree uses $n \lg \sigma + o(n \lg \sigma)$ bits, and can be built in $O(n \lg \sigma)$ time. Each node handles a subset of Σ: the root handles the whole Σ and each leaf a singleton $\{c\}$. The alphabet is split into two contiguous halves[3]

[3]If the alphabet size is odd, one half will have one more element than the other.

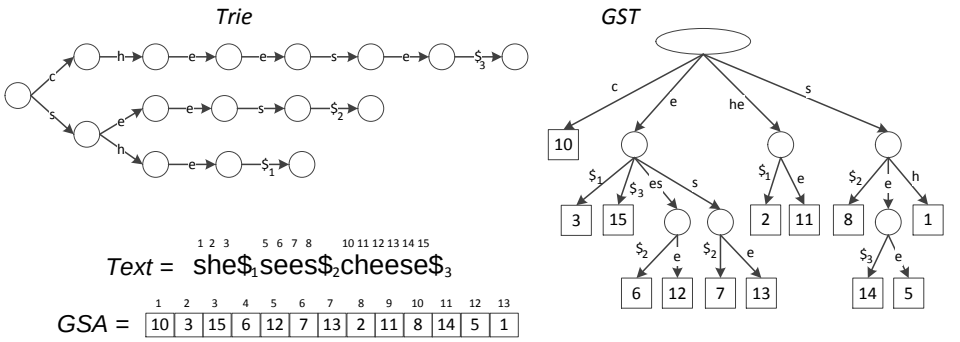

Figure 1: A trie on the words `she`$\$_1$, `sees`$\$_2$ and `cheese`$\$_3$, and also the GST and GSA for the text `she`$\$_1$`sees`$\$_2$`cheese`$\$_3$. For simplicity, in GST and GSA the suffixes starting at symbols $\$_i$ have been omitted. The edges leading to leaves only show their first symbol.

at each node. Each node v represents the subsequence S_v of S formed by the characters it handles. Instead of the subsequence S_v, the node stores a bitvector B_v, so that $B_v[i] = 0$ iff the symbol $S_v[i]$ is handled by the left child, and $B_v[i] = 1$ otherwise. The bitvectors support *rank* and *select* operations, and this is sufficient for the wavelet tree to offer the given functionality.

To compute $access(S, i)$, we move down in the tree from the root towards the leaf that handles $S[i]$, and then report the symbol of the leaf. At each node v, if $B_v[i] = 0$, we track the symbol towards the left child, and otherwise towards the right child. In the first case we recompute the position with $i = rank_0(B_v, i)$, and in the second with $i = rank_1(B_v, i)$. Operations $rank_c(S, i)$ and $select_c(S, i)$ are handled similarly.

Wavelet trees can also represent an $n \times n$ grid with n points, one per column: $(1, y_1), (2, y_2), \ldots, (n, y_n)$, by regarding the points as a sequence $S_{1..n} = y_1 y_2 \ldots y_n$ on the alphabet $[1..n]$. The wavelet tree takes $n \lg n + o(n \lg n)$ bits to represent the points, and can retrieve the t points in any rectangle $[x_m, x_M] \times [y_m, y_M]$ in time $O((t + 1) \lg n)$, as follows. We start at the root bitvector B_v with the range $[x_m, x_M]$. Then we go to the left child with the new range $[rank_0(B_v, x_m - 1) + 1, rank_0(B_v, x_M)]$, *and* to the right child with the range $[rank_1(B_v, x_m - 1) + 1, rank_1(B_v, x_M)]$. We stop the recursion at any node v where either the range is empty or there is no intersection between the sub-alphabet of $[1..n]$ handled by v and the range

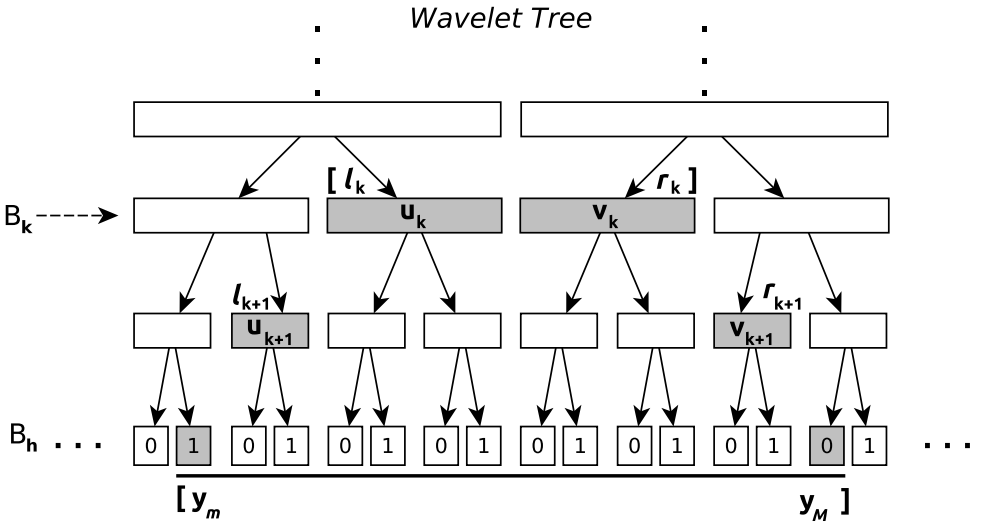

Figure 2: A wavelet tree where we shadowed the $O(\lg n)$ nodes that cover the leaves range $[y_m, y_M]$. The range is covered with at most two maximal nodes per level. It is sufficient to map an original range $[x_m, x_M]$ from the root to those $O(\lg n)$ nodes to find all the points in $[x_m, x_M] \times [y_m, y_M]$. Then those points can be reported one by one, or their total amount can be counted in time $O(\lg n)$.

$[y_m, y_M]$. When we reach a leaf, we report its corresponding y value (we can report the x value as well, by going upwards as for *select*). Since the range $[y_m, y_M]$ is covered by $O(\lg n)$ wavelet tree nodes, it is possible to count the number t of points in a rectangle in $O(\lg n)$ time, by adding up $x_M - x_m + 1$ on those nodes that cover $[y_m, y_M]$. Figure 2 exemplifies.

2.8. Compressed Suffix Arrays

A *compressed suffix array (CSA)* is a succinct representation of a suffix array [18, 19, 42, 12, 13, 35]. We denote $|CSA|$ its space, which is always $O(n \lg \sigma)$ bits for a text $T_{1..n}$ over an alphabet of size σ. Depending on the index, the space can be close to the zero-order entropy of T, $nH_0(T)+o(n \lg \sigma)$ bits, and even less, the lth order entropy [28] for any $l < \alpha \lg_\sigma n$ and constant $0 < \alpha < 1$, $nH_l(T) + o(n \lg \sigma)$ bits.

A CSA can retrieve any suffix array entry $SA[i]$ in time $\mathsf{lookup}(n)$, which is typically of the form $O(\lg^{1+\epsilon} n)$ for some constant $\epsilon > 0$. Within the same

time it also recovers any cell of the inverse permutation, $SA^{-1}[j]$, which is the position of the suffix $T_{j..n}$ in SA. It can also determine the range $SA[l, r]$ of the suffixes starting with $p_{1..m}$ in time $O(\mathsf{search}(m))$, which is typically $O(m \lg n)$ or $O(m \lg \sigma)$. Therefore, the CSA locates all the *occ* occurrences of p in time $O(\mathsf{search}(m) + occ\, \mathsf{lookup}(n))$.

Finally, a CSA can retrieve any text substring $T_{x..y}$, and thus it acts as a replacement of the text.

2.9. Range Minimum Queries

Given an array $A[1, n]$, a *range minimum query (RMQ)* is of the form $RMQ_A(i, j) = \text{argmin}_{i \le k \le j} A[k]$, that is, the position of the minimum in $A[i..j]$. The optimal-size solution [14] requires just $2n + o(n)$ bits and computes any $RMQ_A(i, j)$ in constant time, without accessing the original array A at query time.

2.10. LZ78 Compression

The LZ78 compression algorithm [48] parses the text $T_{1..n}$ to be compressed into a sequence of *phrases*. Each phrase is formed by appending a new character to the longest possible previous phrase, and is represented with the index of the phrase used and the new character appended. The result is a collection of n' phrases, where $n' \le n/\lg_\sigma n$, and thus the output of the compressor has at most $n'(\lg n' + \lg \sigma) \le n \lg \sigma + o(n \lg \sigma)$ bits if $\lg \sigma = o(\lg n)$. On compressible texts, however, the space decreases. Actually, the number of bits output by the LZ78 compressor can be bounded as $|LZ78| = n'(\lg n + \lg \sigma) \le n\, H_l(T) + o(n \lg \sigma)$ for any $l = o(\lg_\sigma n)$ [25].

Figure 3 shows an example of LZ78 parsing. The output of the LZ78 compressor are the pairs $(0, a)(0, b)(1, a)(2, b)(1, b)(0, c)(6, a)(4, \$_1)(2, b)(3, a)(4, c)$ $(5, a)(6, \$_2)(9, a)(4, b)(12, b)(12, \$_3)$. Phrase number 0 corresponds to the empty string, and otherwise phrase number i refers to the ith phrase formed during the parsing. The figure also shows a trie with all the phrases, where the node numbers are the phrase indices. Note that the set is *prefix-closed*, that is, the prefix of a phrase is also a phrase, and thus every trie node corresponds to a distinct phrase.

This trie is used for efficient parsing in $O(n)$ time. It is built as we parse: we traverse the trie with the text to be parsed, and every time we fall off the trie we add a new child with the symbol that was not found among the children, thus creating the new phrase. Then we return to the root and resume the parsing.

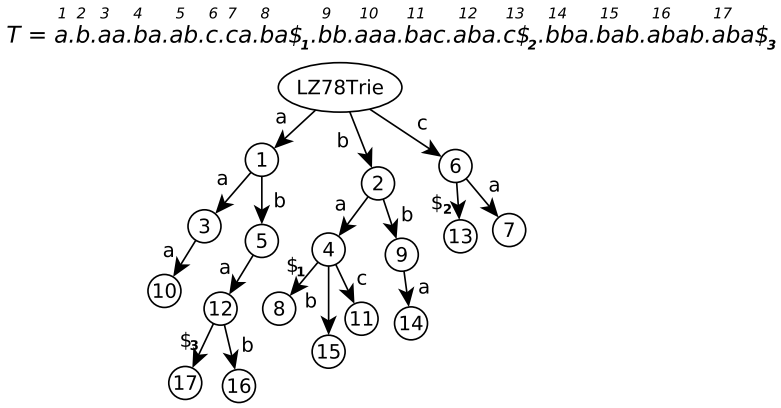

Figure 3: The resulting phrases when applying the LZ78 parsing of a collection with 3 texts, $T = abaabaabccaba\$_1\, bbaaabacabac\$_2\, bbababababababa\$_3$. The result is a dictionary of $n' = 17$ phrases, which are enumerated and separated by points in the figure. We also show how the phrases are organized in a trie.

3. The LZ-Index

In this section we describe the basic components of the *LZ-Index* [32], which also offers pattern matching capabilities, and on which we build our document retrieval solutions.

The LZ-Index builds on the LZ78 parsing of the text $T_{1..n}$ to index. Its first two components are two tries, which store the set of phrases obtained for T using LZ78 (called *LZTrie*, the same shown in Figure 3), and the trie of the reversed phrases (called *RevTrie*), that is, the phrases read backwards. Note that LZTrie can be used to find the phrases that start with p, and RevTrie to find those that end with p (by looking for the reverse of p). Note that the set of reversed phrases is not prefix-closed, therefore RevTrie may contain nodes that do not correspond to any phrase.

These two tries are represented in compact form (Section 2.4), so that they support the efficient navigation described in Section 2.3. Apart from basic navigation toward children and parents, we can find in constant time the preorder index (or just preorder, for short) of a node v, the node with a given preorder, and the range of preorders for the subtree rooted at v. We also store an array that associates the phrase number with each node. The space per trie is $n' \lg n' + O(n' \lg \sigma)$ bits [32, 2].

During the search process, it is necessary to travel from a node in RevTrie

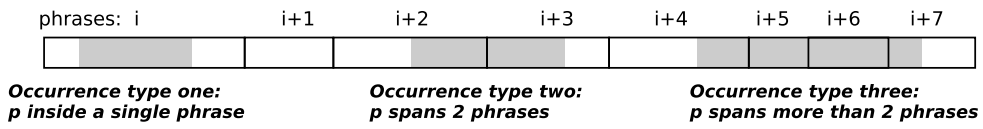

Figure 4: The three types of occurrences according to how many phrases they span.

to the node in LZTrie that represents the same phrase. For this task, the index includes an array called *Node*, which does the mapping between phrase identifiers and preorders in LZTrie. This array uses other $n' \lg n'$ bits.

The last basic member of the LZ-Index, *Range*, is a data structure used to find the occurrences that begin inside a phrase and end in the next one. This is a two-dimensional $n' \times n'$ grid where we store n' points. If the $(k+1)$th text phrase is represented by the node with preorder i in LZTrie and the kth phrase is represented by the node with preorder j in RevTrie (counting only nodes that represent phrases), then a point at row i and column j is placed in the grid. Note that with the LZTrie preorder (i.e., the row) we obtain the phrase identifier of a point. The grid is implemented with a wavelet tree (Section 2.7) using $n' \lg n'(1 + o(1))$ bits, so that all the t points in a rectangular query range are retrieved in time $O((t+1) \lg n')$.

With these components, the occurrences of a pattern $p_{1..m}$ in $T_{1..n}$ are found as follows, according to the three possible ways p can occur across the phrases of T (see Figure 4).

1. Find the occurrences completely contained in a single phrase (occ_{t1} occurrences of type 1): Search for p^r (the reversed pattern) in RevTrie, arriving at node v^r. Every node u^r in the subtree of v^r corresponds to an occurrence of p at the end of a phrase. Any other phrase formed from that of u^r also contains p, and those form all the occurrences of type 1. Thus, any occurrence of type 1 is at an LZTrie node that descends from u, where u is the LZTrie node that corresponds to u^r. Therefore, for each node u^r, we travel from RevTrie to LZTrie using Node, and report every phrase in the corresponding subtree of LZTrie. The search time for p^r in RevTrie is $O(m)$, and then each occurrence of type 1 is reported

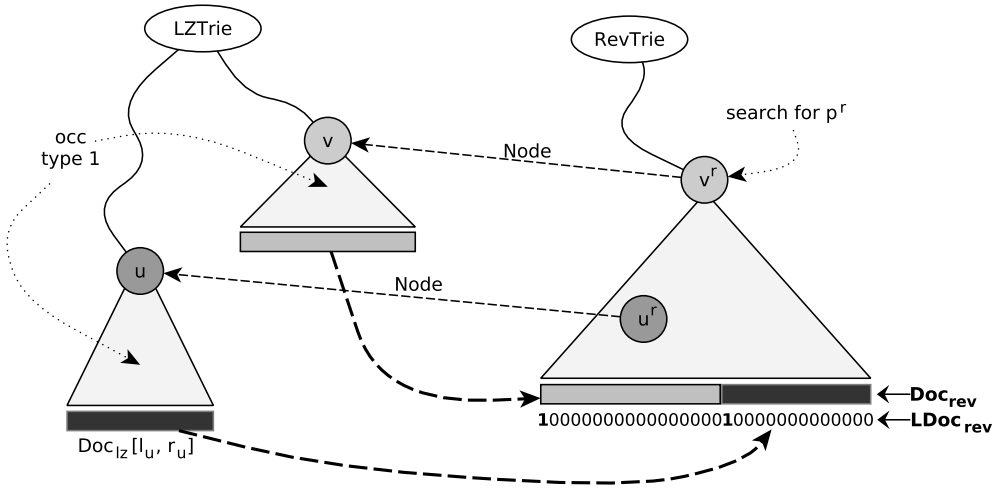

Figure 5: The structures to report occurrences of type 1. The arrays *Doc* and *LDoc* are used for document listing.

in $O(1)$ time, for a total time of $O(m + occ_{t1})$. See Figure 5 (disregard the arrays of the bottom for now).

2. Find the occurrences that span two consecutive phrases (occ_{t2} occurrences of type 2). The pattern is split in every possible way into $p = p_{start} \cdot p_{end}$. For each such split, we search for p^r_{start} in RevTrie (finding locus v^r) and for p_{end} in LZTrie (finding locus u). Both searches take $O(m)$ time (for each division of p), and obtain the preorder ranges $[l_v, r_v]$ and $[l_u, r_u]$ of occurrences for all prefixes and suffixes. Now we query Range for $[l_v, r_v] \times [l_u, r_u]$, retrieving all the phrase numbers k that end with p_{start} such that $k + 1$ starts with p_{end}. Since this is done for every split, the cost is $O(m^2 + m \lg n + occ_{t2} \lg n)$. See Figure 6.
3. Find the occurrences that span more than two consecutive phrases (occ_{t3} occurrences of type 3). Since p must contain a whole phrase in each such occurrence, and every phrase is distinct in the LZ78 parsing, there are only $O(m^2)$ possible occurrences of this type. These are found with a more laborious process [32] that takes time $O(m^3)$.

The total space of the LZ-Index is $4n' \lg n' + O(n' \lg \sigma)$ bits, which is at most $4nH_l + o(n \lg \sigma)$ for any $l = o(\lg_\sigma n)$. The time for locating occ occurrences is $O(m^3 + m \lg n + occ \lg n)$. Later improvements on this structure [2] reduce the time to $O(m^2 + m \lg n + occ \lg n)$ by handling in a better way

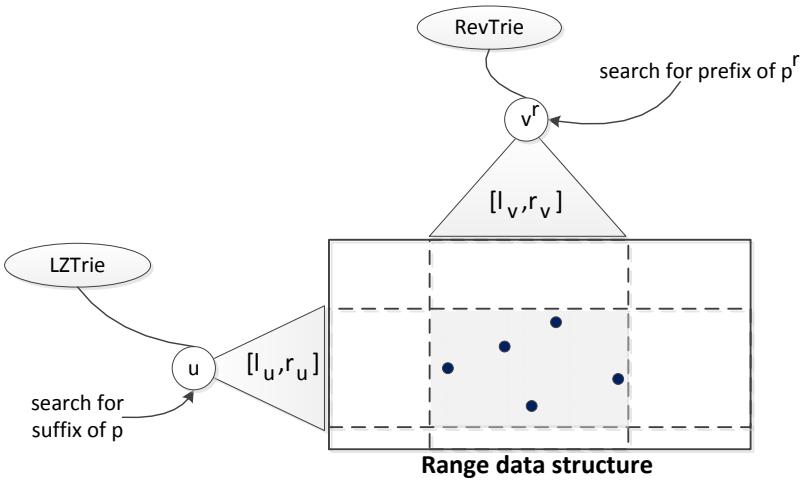

Figure 6: The scheme to report the occurrences of type 2.

the occurrences of type 3. A practical advantage of the LZ-Index compared to CSAs is that it is faster when many occurrences must be reported [33]. Note that, in this case, the pattern p is usually short and then most of the occurrences are of type 1, which are reported in $O(1)$ time. We exploit this property in our DR indices.

4. Related Work and our Contribution in Context

In this section we describe the current solutions for document listing and top-k retrieval [34], and then put our contribution in context. We will reuse some of the ideas in our new indices; these are described in more detail.

4.1. Document Listing

We begin by introducing the method of Muthukrisman [31], which is the first optimal-time solution for document listing in linear space. He builds his structure on the GST (Section 2.5) of $T_{1..n} = d_1 d_2 \ldots d_D$. He also introduces a useful array in the DR field, called *document array* $E[1..n]$: $E[i]$ stores the identifier j of the document d_j where the suffix $SA[i]$ begins. A third structure is the array $C[1..n]$, where $C[j] = i$ iff i is the largest index with

$i < j$ and $E[i] = E[j]$. If i does not exist, then $C[j] = 0$. To complete the framework, he builds an RMQ structure (Section 2.9) on the array C.

To answer a DL query for a pattern $p_{1..m}$, the first task is to search for p in the GST, reaching the node $v = locus(p)$. The leaves of v then form the lexicographic range of the suffixes $SA[l..r]$ that start with p. Thus the answer to the query are the *distinct* values in $E[l..r]$. To find them in optimal time, the key property is that there is exactly one value $C[i] < l$ for each distinct document $E[i]$. Thus a recursive process determines in each step $i = RMQ_C(l, r)$ and, if $C[i] < l$, it reports document $E[i]$ and continues recursively with the ranges $[l, i-1]$ and $[i+1, r]$. Each recursive branch stops when $C[i] \geq l$. Then the ndoc distinct answers are found in optimal time $O(m + \mathsf{ndoc})$.

The disadvantage of Muthukrishnan's solution is that it uses too much space. Even if it is $O(n \lg n)$ bits (i.e., linear), the constant factor of the space is high. Sadakane [43] introduced a succinct variant of this proposal, reducing the space to $|CSA| + O(n)$ bits, replacing the GST by a CSA (Section 2.8), simulating array E with a bitvector and the CSA, and getting rid of array C. Note that C is not accessed in modern RMQ structures (Section 2.9), but we still need to ask whether $C[i] < l$. Sadakane finds an equivalent condition: he replaces the test by marking the documents already reported in a bitvector $V_{1..D}$ and stopping when a marked document is found again. The query time is now $O(\mathsf{search}(m) + \mathsf{ndoc} \cdot \mathsf{lookup}(n))$.

The time can be improved by building a wavelet tree (Section 2.7) representation of array E [45]. Then array C can be simulated with $C[i] = select_{E[i]}(E, rank_{E[i]}(E, i-1))$, and the original algorithm of Muthukrishnan can be applied. Using a compressed suffix array, this requires $|CSA| + n \lg D(1 + o(1))$ bits and reports the ndoc documents in time $O(\mathsf{search}(m) + \mathsf{ndoc} \cdot \lg D)$. Later [16] it was shown that the wavelet tree can report the distinct values in $E[l..r]$ with a simpler and faster procedure, leading to DL time $O(\mathsf{search}(m) + \mathsf{ndoc} \cdot \lg(D/\mathsf{ndoc}))$.

While the wavelet-tree based solutions are faster than Sadakane's, their space includes a significant component of $n \lg D$ bits, which is not far from $n \lg n$. Navarro et al. [38] achieved nearly 50% reduction of the wavelet tree space in practice, at the price of nearly doubling the query time.

Hon et al. [20] went in the other direction, reducing Sadakane's space to just $|CSA| + D \lg(n/D) + o(n)$ bits, and answering DL queries in time $\mathsf{search}(m) + \mathsf{ndoc} \lg^{1+\epsilon} n \, \mathsf{lookup}(n)$ time, for any constant $\epsilon > 0$. This is obtained by grouping consecutive entries of C and running the algorithm on

the minima of the groups, then scanning the selected groups by brute force.

Our contribution. In Section 5 we describe an extension of the LZ-Index that uses $4n' \lg n' + n' \lg D + 2n' \lg \sigma + o(n' \lg n') + 3n + o(n) \leq 5nH_l(T) + 3n + o(n \lg \sigma)$ bits. It solves DL queries in time $O(m^2 \lg n + \mathsf{ndoc}\, m \lg^2 n)$. As the complexities hint, this is significantly smaller and slower than the solutions that build a wavelet tree on E [45, 16], and generally faster and larger than Sadakane's initial solution [43] and the compressed wavelet trees [38].

An interesting aspect in our solution is that most of the documents (ndoc_1) are obtained from occurrences of type 1 (Section 3), and those are obtained in optimal time, $O(m + \mathsf{ndoc}_1)$, at a speed comparable to the fast wavelet-tree based solutions. We prove that most of the answers are of this type, $\mathsf{ndoc}_1 = \mathsf{ndoc} - o(\mathsf{ndoc})$, thus those taking more time are asymptotically vanishing, $o(\mathsf{ndoc})$. A variant of our index that finds only those ndoc_1 answers is close to the smallest solutions in space, and to the fastest solutions in time. In various applications, finding a good fraction of the answers, or finding them progressively, is of interest, and in this case our index excels.

4.2. Top-k Retrieval

It took longer to obtain a time-optimal and linear-space solution for the more sophisticated problem. Hon et al. [20] obtained $O(m + k \lg k)$ time and $O(n \lg n)$ bits, and their framework was later exploited by Navarro and Nekrich [36] to reduce the time to the optimal $O(m + k)$. Both solutions, however, use too much space if implemented directly.

Hon et al. [20] also gave the first succinct solution for term-frequency scores: a structure that uses $2|CSA| + o(n) + D \lg \frac{n}{D} + O(D)$ bits and answers in time $O(\mathsf{search}(m) + k \lg^{3+\epsilon} n \cdot \mathsf{lookup}(n))$, again for any constant $\epsilon > 0$. The method takes a fixed κ value and chooses every gth left-to-right leaf in the GST, for $g = \kappa \lg^{2+\epsilon} n$. Then it samples all the lowest common ancestors of successive chosen leaves. The sampled GST nodes are $O(n/g)$, and are represented in a reduced tree τ_κ. For each such node, the top-κ answer is stored explicitly, with the score of each document.

Now, given the locus node $v = locus(p)$, it can be shown that there are only $O(g)$ leaves between its subtree and the subtree covered by the highest marked node u below v. Then those $O(g)$ leaves are traversed in order to "correct" the precomputed top-κ answer stored at u. The second $|CSA|$-bit space is used to compute the frequencies of p in the documents found along this traversal, to see if they make it to the top-κ list.

The final structure stores a tree τ_κ for every $\kappa = 2^t$, $0 \leq t \leq \lg D$, therefore $O(\lg D)$ trees at different resolutions are stored. For the queries one uses the tree τ_κ built for $\kappa = \lceil \lg k \rceil$.

Navarro et al. [38] implemented this index, replacing the $|CSA|$ bits used to compute frequencies by the (compressed) wavelet tree of E, which in practice works better. It is faster to obtain the document identifiers, and also enables smarter strategies to obtain all the distinct documents in the range of $O(g)$ leaves that must be collected. They also showed that it was better to store a unique tree τ that merges the nodes of all the τ_κ, so that each node stores the top-κ answers for the largest κ where this node belongs to τ_κ. Their structure reaches as little as 7 bpc, depending on the compressibility of the collection, and retrieval times are 1–10 milliseconds (ms).

An alternative proposal [4] replaces E by monotone minimum perfect hash functions, which use $O(n \lg\lg\lg D)$ bits instead of the $n \lg D$ of the wavelet tree. The top-k retrieval time becomes $O(\mathsf{search}(m)+k \lg k \lg^{1+\varepsilon} n\, \mathsf{lookup}(n))$. In practice it is significantly slower than the previous implementation [38], and it uses less space only when the collection is incompressible. There are several other theoretical proposals [34] that promise to use much less space than current implementations, but they will most likely be even slower.

Faster indices, yet using more space, are built as space-reduced versions of the optimal-time index [36]. Konow and Navarro [24] gave an implementation that uses over 28 bpc on our datasets[4] and answers top-k queries in time $O(m + (k + \lg\lg n) \lg\lg n)$ with high probability. In practice, this is k–$4k$ μs. A more recent version [17] reduces the space to 17–21 bpc in our datasets, and maintains the same query time performance.

Our contribution. In Section 6 we describe an extension of the LZ-Index that uses $2n' \lg n' + n' \lg D + 2n' \lg \sigma + o(n' \lg n') + O(n) \leq 3nH_l(T) + O(n) + o(n \lg \sigma)$ bits. It solves top-k queries using term frequencies as the relevance measure, in time $O(m^2 \lg n + k \lg^3 n)$, where m is the pattern length, under a wide set of statistical text models with finite memory. This turns out to dominate an important area of the space/time tradeoff map, between the extremes of the smallest and slowest indices [38] and the fastest and largest ones [24, 17].

Moreover, within just $n' \lg n' + n' \lg D + 2n' \lg \sigma + o(n' \lg n') + O(n) \leq 2nH_l(T) + O(n) + o(n \lg \sigma)$ bits, the index gives an approximate top-k answer in time $O(m + k \lg^2 n)$, k–$5k$ μs in practice. The index is as small as the

[4]The space results they report [24] were somewhat underestimated [17].

smallest indices [38] but 3–6 orders of magnitude faster, and as fast as the fastest ones [24, 17] but 2–4 times smaller.

The approximation is obtained by considering only the occurrences of type 1. As expected from the results of listing, where as said we lose only $o(\mathsf{ndoc})$ occurrences by disregarding the others, we prove that the result becomes asymptotically more accurate as the text collection grows. We also show experimentally that the accumulated term frequencies of the chosen documents becomes in most cases 90% of that of the correct top-k documents for $m \leq 8$, even for relatively small collections. That is, we preserve 90% of the "quality", in a sense.

4.3. Experiments

We run our experiments on several text collections that were already considered in previous work [38, 24], as well as other larger ones.

- `ClueWiki`: A sample of ClueWeb09. These are Web pages from the English Wikipedia (`boston.lti.cs.cmu.edu/Data/clueweb09/`).
- `Wiki`: A collection of more and shorter documents than `ClueWiki`.
- `KGS`: A collection of sgf-formatted Go game records from year 2009 (`www.u-go.net/gamerecords`).
- `Proteins`: A collection of sequences of human and mouse proteins (`www.ebi.ac.uk/swissprot`).
- `DNA`: A synthetic collection, slightly repetitive with 5% mutations among documents.
- `Influenza`: A repetitive collection of the genomes of influenza viruses. We take the first 70MB.
- `TodoCL`: A collection formed by snapshots of the Chilean Web. This includes real queries, which we use to measure quality. We take the first 100MB for most experiments, and up to 2.05GB for experiments on collection growth.
- `TREC`: The TREC Corpus FT91 to 94 (`http://trec.nist.gov`). We take the first 3.5GB and use it for experiments on collection growth.

Collection	n (MB)	D	n/n'	compress (bpc)
`ClueWiki`	131	3,334	17.24	2.78
`Wiki`	80	40,000	9.58	3.34
`KGS`	25	18,838	14.97	1.85
`Proteins`	56	143,244	6.38	4.61
`DNA`	95	10,000	11.50	2.68
`Influenza`	70	49,588	21.18	1.89
`TodoCL`	100	22,850	9.02	3.82
`TREC`	3500	846,869	19.42	3.74

Table 1: Main characteristics of the text collections.

Table 1 summarizes the main characteristics of these collections: size n, number of documents D, average LZ78 phrase length n/n' (the larger, the more compressible for our index), and bpc obtained by the LZ78-based Unix `compress` program (another measure of LZ78 compressibility).

The machine used for all experiments is an Intel Xeon with 8 processors of 2.4GHz and 12MB cache, with 96GB RAM. It runs on Linux 2.6.32-46-server, and we use gcc with full optimization.

5. An LZ-based Index for Document Listing

We adapt the LZ-Index described in Section 3 to carry out document listing (DL). The resulting index is called LZ-DLIndex. We solve DL by considering the same 3 types of occurrences of pattern matching. The key idea is, instead of collecting each individual occurrence of p, to simulate Muthukrishnan's DL algorithm (Section 4.1) on ranges of occurrences, even when the information is more fragmented than on a suffix array. We first give a broad description of the ideas and then enter into details.

For the occurrences of type 1, the pattern matching algorithm finds the locus v^r of p^r in RevTrie and traverses its whole subtree. It maps each node u^r in the subtree of v^r to u in LZTrie, and then traverses the whole subtree of u. Now we want to report all the distinct documents found across this process. We will virtually[5] *expand* each node u^r in RevTrie with the subtree of u, recording the document where each node belongs. The result will be an array analogous to $E_{1..n}$, where we can use Muthukrishnan's DL algorithm on the

[5]In the sense that we will not actually represent it, as explained next.

range covered by v^r. We do not store the array E itself, but we simulate access to any position indirectly via the LZTrie, which will store the document where each phrase belongs. We will store an RMQ structure on the (virtual) array C corresponding to E. This structure uses $2n + o(n)$ bits (Section 2.9) and allows us to find each new document in $O(1)$ time without accessing C. The other accesses to C will be replaced with Sadakane's workaround (Section 4.1).

For the occurrences of type 2, we find the $O(\lg n')$ nodes that cover the y-interval $[l_u, r_u]$ and project the x-interval $[l_v, r_v]$ to each such node (Section 2.7). Each point to report belongs to some document, and we want again to report all the distinct documents. We can do it by brute force (reporting the document of every individual point, avoiding repetitions), or apply Muthukrishnan's algorithm on each of the $O(\lg n')$ ranges $[x_m, x_M]$ in the wavelet tree nodes that cover the y-coordinate interval. Access to the corresponding arrays E is done via mapping the points to the LZTrie. In addition, we store in each wavelet tree node the RMQ structures on the corresponding C arrays. Therefore, to each bitvector B_v we add an RMQ structure using $2|B_v| + o(|B_v|)$ bits, reusing a technique for RMQs on two-dimensional point sets [37]. Again, other accesses to C are avoided in the same way as Sadakane.

Finally, occurrences of type 3 are $O(m^2)$ in total and are dealt with one by one. We will show that the whole process takes time $O(m^2 \lg n + \mathsf{ndoc}\, m \lg^2 n)$. However, the ndoc_1 documents found with occurrences of type 1 are listed in time $O(m + \mathsf{ndoc}_1)$. Now we describe the techniques in further detail.

5.1. Structure

Tries. We store the topologies of both tries plus the labels of RevTrie as in previous work [32, 2]. These require $2n' \lg \sigma + O(n')$ bits and support constant-time traversals. Note that LZTrie has n' nodes, and thus its topology is represented with $2n' + o(n')$ bits (Section 2.4). However, RevTrie may have up to n nodes, because not every node corresponds to a phrase. From those nodes, some are *unary*, that is, have just one child, and some are *empty*, that is, do not represent any phrase. Since RevTrie has at most n' leaves and exactly n' nonempty nodes, it has at most $2n'$ non-unary nodes. Thus we can represent only the (at most) $3n'$ nodes that are non-unary or nonempty, and collapse the remaining unary paths. Only the symbols that are not in those paths are stored. This leads to a representation that uses $2n' \lg \sigma + O(n')$ bits. The symbols from unary paths are extracted via the

connection with the LZTrie [32, 2]. We also mark in a bitvector $Q_{1..3n'}$ which nodes (in preorder) are nonempty, so that we can compute the preorder of a node v among the nonempty nodes as $rank_1(Q, preorder(v))$.

Documents. Instead of storing the phrase identifiers for the n' nodes of LZTrie, we store the identifiers of the document where they occur. We also store the RMQ structure associated with the virtual array $E_{1..n}$ we have described. In total we store $n' \lg D + 3n + o(n)$ bits, in the following structures (see the arrays on the bottom of Figure 5).

Doc_{lz}**:** The array of n' document identifiers of the LZTrie phrases in preorder, stored explicitly in $n' \lg D$ bits. This is equivalent to the document array of Muthukrishnan (Section 4.1), but restricted to phrases.

Doc_{rev}**:** A sequence of n document identifiers built as follows. We traverse RevTrie in preorder, and for each nonempty node v^r, let v be the corresponding LZTrie node. Let $Doc_{lz}[l_v, r_v]$ be the range of all the descendants of v (included). We then append $Doc_{lz}[l_v, r_v]$ to Doc_{rev}. The total length of Doc_{rev} is n because n is the internal path length (sum of all node depths) in LZTrie, and each LZTrie node is appended to Doc_{rev} once per ancestor it has in LZTrie.

We do not store Doc_{rev}, but only the $2n$-bit RMQ structure on its corresponding C array (Section 4.1); recall that the RMQ structure does not need to access C (Section 2.9). This will be sufficient to run Muthukrishnan's DL algorithm (Section 4.1) on top of Doc_{rev}. When it needs to access Doc_{rev}, we will take the corresponding cell from Doc_{lz}.

$LDoc_{rev}$**:** A bitvector of n bits that marks the Doc_{rev} positions where the intervals $Doc_{lz}[l_v, r_v]$ start. Since it has only n' bits set, it is represented in compressed form (Section 2.2), so it can use less than n bits.

Node. A mapping from RevTrie to LZTrie. If the node v^r in RevTrie with nonempty preorder i corresponds to the node v in LZTrie with preorder j, then $Node[i] = j$. Array *Node* uses $n' \lg n'$ bits.

Range. An enhanced binary wavelet tree. Each wavelet tree node implicitly represents a sequence of points (i.e., pairs of phrases $(k, k+1)$). Now consider the array of their corresponding documents (we are not interested in pairs of phrases that span two documents, as no matches occur there). In addition

to the bitvector B_v of node v, we store the RMQ structure corresponding to the C array of its (virtual) array of documents (Section 4.1). The total space of Range is then $3n' \lg n' + o(n' \lg n')$ bits.

Space. Overall, the LZ-DLIndex requires $4n' \lg n' + n' \lg D + 2n' \lg \sigma + o(n' \lg n') + 3n + o(n) \leq 5nH_l(T) + 3n + o(n \lg \sigma)$ bits. This is close to the original LZ-Index size [32].

5.2. Queries

We solve DL incrementally, considering the three types of occurrences.

5.2.1. Occurrences of type 1

We search for p^r in RevTrie, arriving at node v^r. Let $[i_v, j_v]$ be the range of preorders of nonempty nodes descending from v^r. We find the interval $I = Doc_{rev}[s_v, e_v]$ of all the documents that contain occurrences of type 1, where $s_v = select_1(LDoc_{rev}, i_v)$ and $e_v = select_1(LDoc_{rev}, i_v + 1) - 1$. Next, we report all the distinct documents in I with Muthukrishnan's algorithm using RMQs. For each new position pos of a document $Doc_{rev}[pos]$ reported by an RMQ, we need to report the document identifier. We determine the nonempty preorder $j = rank_1(LDoc_{rev}, pos)$ of the RevTrie node holding that position, and then the preorder of this node in LZTrie, $i = Node[j]$. The difference $d = pos - select_1(LDoc_{rev}, j)$ provides the offset of this position within the leaf interval of the LZTrie node with preorder i. Thus, the document is $Doc_{lz}[i + d]$. The overall time of this step is thus $O(m + \mathsf{ndoc}_1)$.

5.2.2. Occurrences of type 2

We proceed as in the original LZIndex for reporting occurrences from Range, but now we use the RMQ structures in the wavelet tree of Range to report documents. We consider all the $m - 1$ partitions $p = p_{start} \cdot p_{end}$ and search for these prefixes and suffixes in the tries. Each such partition then becomes a range search for $[l_v, r_v] \times [l_u, r_u]$ in Range, and is decomposed into $O(\lg n')$ intervals $[x_m, x_M]$ in different wavelet tree nodes v. Each point in those intervals represents a position in a document. The distinct documents in each interval $[x_m, x_M]$ are obtained using Muthukrishnan's algorithm on the RMQs built for the node. To obtain the document identifier for each reported position $pos \in [x_m, x_M]$, we track the position down in the wavelet tree until reaching the leaf, which indicates the row of Range. Since the rows of Range correspond to LZTrie preorders, we simply access Doc_{lz} at the leaf index.

Although unlikely, in the worst case we can output the same document in each of the $O(\lg n')$ intervals for each of the $m-1$ partitions, and each requires $O(\lg n')$ time for tracking the point down to the leaves. This gives $O(m^2)$ time for the RevTrie searches plus a (very pessimistic) worst-case bound of $O(\mathsf{ndoc}_2\, m \lg^2 n)$ time for the ndoc_2 occurrences of type 2.

5.2.3. Occurrences of type 3

Occurrences of type 3 are extracted one by one as in previous work [2], using Doc_{lz} to give document numbers for the matches found. The only difference is that this algorithm needs to find the LZTrie node of phrase $k+1$ given the RevTrie node of phrase k. This is easy because their Node structure maps from phrase numbers to LZTrie nodes, but ours maps RevTrie preorders to LZTrie preorders.

Instead, we use Range to find the phrase $k+1$ given a phrase number k in RevTrie: If the phrase k corresponds to preorder i_v in RevTrie, then tracking down the i_vth position from the root to the leaf of Range ends up precisely in the preorder of phrase $k+1$. In terms of the wavelet tree functionality, the answer is simply $access(i_v)$. This increases the complexity to find these occurrences to $O(m^2 \lg n)$.

5.2.4. Time

The total query time is $O(m^2 \lg n + \mathsf{ndoc}\, m \lg^2 n)$, where we remind that this is a very pessimistic upper bound. We also note that the occurrences of type 1 are reported very early, in time $O(m + \mathsf{ndoc}_1)$. If the text is generated by an ergodic 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 $n' \leq n/\lg_\sigma n$, only $O((n/d)m/\lg_\sigma n)$ of those occurrences hit a phrase boundary on average. This means that that a fraction of $1 - O(m/\lg_\sigma n)$ of the occurrences are of type 1, and also $\mathsf{ndoc}_2 = O(\mathsf{ndoc}\, m/\lg_\sigma n) = o(\mathsf{ndoc})$ if $m = o(\lg_\sigma n)$. Thus we report almost all of the occurrences in $O(1)$ time each. If we just lose those $o(\mathsf{ndoc})$ occurrences not of type 1, our time is the optimal $O(m + \mathsf{ndoc})$.

We show in the experiments that, indeed, our index is particularly competitive to show the first occurrences (those of type 1), which are the most for short patterns.

5.3. Implementation

To obtain a practical implementation of the scheme, we make some changes that, although they do not preserve the theoretical space and time guaran-

tees, perform much better in practice. These refer largely to the implementation of the tries.

The mechanism to avoid storing symbols of unary paths in RevTrie and instead extract them from LZTrie is slow in practice. Instead, we store them in RevTrie. Moreover, we perform all the searches in RevTrie, and do not represent LZTrie at all. RevTrie then has t_{rev} nodes, which can be as large as n, but in practice it is much less.

We represent RevTrie in DFUDS form (Section 2.3), using $2t_{rev} + o(t_{rev})$ bits, plus a bitvector that marks the nonempty nodes in DFUDS order, so as to compute the nonempty preorders that are used in searches. We also store a string with the $2t_{rev}$ symbols that label the edges, in the same order they are stored in DFUDS. This allows (1) performing binary searches on the labels toward the children of a node, to efficiently find the one to follow, (2) having in consecutive positions the symbols that label unary paths, so as to compare them efficiently with p. The constant-time method to find the label given in DFUDS [5] is theoretical, and is better replaced with searches on this string.

RevTrie is used directly to find the occurrences of type 1, and also to search for p^r_{start} when looking for occurrences of type 2. To find p_{end}, since we cannot search in LZTrie, we look for p^r_{end} in RevTrie. If it does not exist, or it leads to an empty node, then p_{end} is not a phrase and there are no phrases starting with p_{end} (as LZTrie is prefix-closed). If instead we reach a node u^r, with nonempty preorder t, then $i = Node[t]$ is the LZTrie preorder of the corresponding node u, which represents p_{end}. It is also the left end $l_u = i$ of the preorder interval of the descendants of u. To find the right end, r_u, we compute the size ℓ of its interval in Doc_{lz} using $LDoc_{rev}$: $\ell = select_1(LDoc_{rev}, t+1) - select_1(LDoc_{rev}, t)$, and then $r_u = l_u + \ell - 1$. Now we have the row interval to search Range.

Finally, we can reduce the space of the RMQs in Range by storing them only for the highest levels of the wavelet tree. The lowest ones have shorter bitvectors, and then traversing them sequentially is not much different from applying Sadakane's algorithm to find the different documents (moreover, as they are closer to the leaves, obtaining their document identifiers is cheaper). This gives a space/time tradeoff.

5.4. Experimental Results

A 64-bit implementation of our index, *LZ-DLIndex*, is left public[6]. We also use an RMQ implementation of our own[7] [11], which requires around $2.2n$ bits. The bitvector implementations are obtained from the SDSL library[8].

We also implement the classical DL solution of Sadakane [43], which we also leave public[9]. As the CSA, we use the FM-Index implemented in the SDSL library, and use different suffix array samplings to obtain space/time tradeoffs.

5.4.1. Space study

Table 2 gives the space obtained by our LZ-DLIndex structure on the collections described in Table 1. The total bpc for each main component is shown in bold, and between parentheses its percentage of the total size of the structure. `Influenza`, `ClueWiki` and `KGS` are the most compressible ones, reaching 6.3–8.2 bpc, whereas `DNA`, `Wiki`, `TodoCL` and `Proteins` are the least compressible ones. All are, as roughly expected from the space analysis, 3.7–5.2×|LZ78|, where $|\mathsf{LZ78}| = n'(\lceil \lg n' \rceil + \lceil \lg \sigma \rceil)/n$. We show how |LZ78| relates to n/n', and how it roughly coincides with the output size of `compress`, a classical LZW Unix compressor (shown in Table 1).

The Doc component dominates the space, with 37%–53% of the total index size. It includes the document identifiers with their boundary values and the RMQ_C data structure on RevTrie. For Range we used the lowest and smallest version of the index, where the wavelet tree of Range does not include any RMQ structure (this corresponds to the highest point of the LZ-DLIndex in Figures 7 and 8). Range and Node use 15%–25% of the index size each. The distribution varies a bit on the less compressible collections, where the fraction of Node and Range increases, reaching 25% each. Note that component Range can be omitted if we only want to list the occurrences of type 1, in which case the index size is reduced by 15%–25%.

Table 3 shows the number of documents listed by the queries, averaging over 3,000 patterns randomly extracted from the collections. Many of the listed documents are obtained as type-1 occurrences (70%–96% for $m = 6$, and 50%–92% for $m = 10$ if we exclude `DNA`). This shows that we could

[6] At `https://github.com/hferrada/LZ-DLIndex.git`.

[7] Available at `https://github.com/hferrada/rmq.git`.

[8] From `https://github.com/simongog/sdsl-lite.git`.

[9] At `https://github.com/hferrada/Sada-DLIndex.git`.

Collection	**RevTrie**	**Doc**	**Node**	**Range**	**Total (/|LZ78|)**	|LZ78| (n/n')
ClueWiki	**1.69(23%)**	**3.30(45%)**	**1.33(18%)**	**1.04(14%)**	**7.39** (4.31×)	1.71 (17.24)
	0.18(topology)	*0.70(Doc_{lz})*				
	1.39(labels)	*2.34(RMQ_C)*				
	0.12(empty)	*0.26($LDoc_{rev}$)*				
Wiki	**1.93(18%)**	**4.38(40%)**	**2.51(23%)**	**2.07(19%)**	**10.89** (3.68×)	2.96 (9.58)
	0.18(topology)	*1.67(Doc_{lz})*				
	1.39(labels)	*2.34(RMQ_C)*				
	0.12(empty)	*0.37($LDoc_{rev}$)*				
KGS	**2.03(25%)**	**3.65(44%)**	**1.40(17%)**	**1.13(14%)**	**8.21** (4.56×)	1.80 (14.97)
	0.23(topology)	*1.00(Doc_{lz})*				
	1.61(labels)	*2.36(RMQ_C)*				
	0.19(empty)	*0.29($LDoc_{rev}$)*				
DNA	**1.10(12%)**	**3.87(42%)**	**2.09(23%)**	**2.08(23%)**	**9.14** (3.89×)	2.35 (11.50)
	0.24(topology)	*1.22(Doc_{lz})*				
	0.80(labels)	*2.32(RMQ_C)*				
	0.06(empty)	*0.33($LDoc_{rev}$)*				
Proteins	**2.06(11%)**	**5.63(37%)**	**3.76(25%)**	**3.76(25%)**	**15.21** (4.33×)	3.51 (6.38)
	0.44(topology)	*2.82(Doc_{lz})*				
	1.51(labels)	*2.34(RMQ_C)*				
	0.11(empty)	*0.47($LDoc_{rev}$)*				
Influenza	**0.95(15%)**	**3.36(53%)**	**1.04(17%)**	**0.95(15%)**	**6.30** (5.21×)	1.21 (21.18)
	0.14(topology)	*0.75(Doc_{lz})*				
	0.75(labels)	*2.34(RMQ_C)*				
	0.06(empty)	*0.27($LDoc_{rev}$)*				
TodoCL	**2.05(18%)**	**4.40(39%)**	**2.66(23%)**	**2.21(20%)**	**11.32** (3.24×)	3.40 (9.02)
	0.35(topology)	*1.66(Doc_{lz})*				
	1.51(labels)	*2.35(RMQ_C)*				
	0.19(empty)	*0.39($LDoc_{rev}$)*				

Table 2: Space breakdown of the main components in our LZ-DLIndex structure, with values in bpc. For RevTrie and Doc columns, the space is the sum of the components detailed below them (bpc values in italics). The Range column does not include the RMQ structures to speed up the index. The percentages refer to the total size of the index. The column (/|LZ78|) indicates the ratio of the total size over |LZ78|, and the last column, in turn, gives also (n/n').

obtain a significant part of the result using just the fastest listing and without representing Range.

5.4.2. Space/time tradeoffs

Figures 7 and 8 compare our LZ-DLIndex structures in three modes: (i) the full mode where it returns all the documents for a DL-query (called LZ-Index in the plots); (ii) a mode where it also can return all the documents but we take the time needed to return only those that were found for occurrences of type 1, and use the minimum space for Range (called "up to type 1"); and (iii) a mode where it can only return the documents found by occurrences of type 1 as it does not store Range at all (called "only type 1"). For the full

	$m = 6$				$m = 10$				
Collection	Type 1	Type 2	Type 3	ndoc	Type 1	Type 2	Type 3	ndoc	D
ClueWiki	**1860.51**	**69.37**	**0.24**	**1930.12**	**1437.01**	**119.79**	**2.05**	**1558.85**	**3,334**
	96.4%	*3.6%*	*0.0%*	*57.89%*	*92.2%*	*7.7%*	*0.1%*	*46.76%*	
Wiki	**921.66**	**290.04**	**0.16**	**1211.86**	**135.79**	**76.50**	**0.97**	**213.26**	**40,000**
	76.1%	*24.0%*	*0.0%*	*3%*	*63.7%*	*35.9%*	*0.5%*	*0.5%*	
KGS	**4702.26**	**1691.87**	**1.40**	**6395.53**	**2012.27**	**739.57**	**4.66**	**2756.49**	**18,839**
	73.5%	*26.5%*	*0.0%*	*33.9%*	*73.0%*	*26.8%*	*0.2%*	*14.63%*	
DNA	**7527.03**	**1630.21**	**0.01**	**9157.25**	**32.72**	**98.37**	**0.14**	**131.22**	**10,001**
	82.2%	*17.8%*	*0.0%*	*91.56%*	*24.9%*	*75.0%*	*0.1%*	*1.31%*	
Proteins	**52.01**	**21.53**	**0.07**	**73.61**	**25.57**	**16.59**	**3.50**	**45.66**	**143,244**
	70.7%	*29.2%*	*0.0%*	*0.05%*	*56.0%*	*36.3%*	*7.7%*	*0.03%*	
Influenza	**16901.13**	**3302.72**	**0.09**	**20203.94**	**995.46**	**452.63**	**1.18**	**1449.27**	**49,588**
	83.7%	*16.3%*	*0.0%*	*57.89%*	*68.7%*	*31.2%*	*0.1%*	*2.92%*	
TodoCL	**467.09**	**173.88**	**0.16**	**641.13**	**35.85**	**35.34**	**0.93**	**72.12**	**22,850**
	72.9%	*27.1%*	*0.0%*	*2.81%*	*49.7%*	*49.0%*	*1.3%*	*0.03%*	

Table 3: Number of occurrences of each type, for pattern lengths $m = 6$ and $m = 10$. Under each number, we give the percentages of the documents output. For the three types of occurrences these refer to ndoc, and for column ndoc this refers to D.

mode, we obtain a space/time tradeoff by representing RMQs only for the highest levels of Range, as explained.

We also compare Sadakane's DL structure [43], showing seven points that use suffix array sampling steps of 4, 8, 16, 32, 64, 128 and 256. We also compare some variants of the proposal that stores a wavelet tree of the document array [38]: (i) the variant using document arrays as plain wavelet trees [45] (WT Plain), (ii) a representation with grammar-compressed wavelet trees (WT RePair), and (iii) an intermediate one called WT Alpha[10]. In order to compute the occurrences interval $SA[l, r]$ in this index we incorporate a CSA with no sampling in order to minimize space (the sampling is not needed here). We use the same FM-Index used for Sadakane.

It can be seen that adding RMQs to Range, while theoretically appealing, increases the space without giving a significant speedup in practice. Our LZ-DLIndex is between, on one extreme, Sadakane and WT RePair, which use less space but may be orders of magnitude slower, and on the other extreme, WT Plain, which is orders of magnitude faster but uses much more space. In some collections, like ClueWiki, Wiki, DNA, and TodoCL, WT Re-

[10]We ran the 32-bit code given by the authors [38], which can build the variants (i) and (ii) for any data collection. The "alpha" structure could be built only on the four data collections used in their publication, which include ClueWiki, KGS, and Proteins.

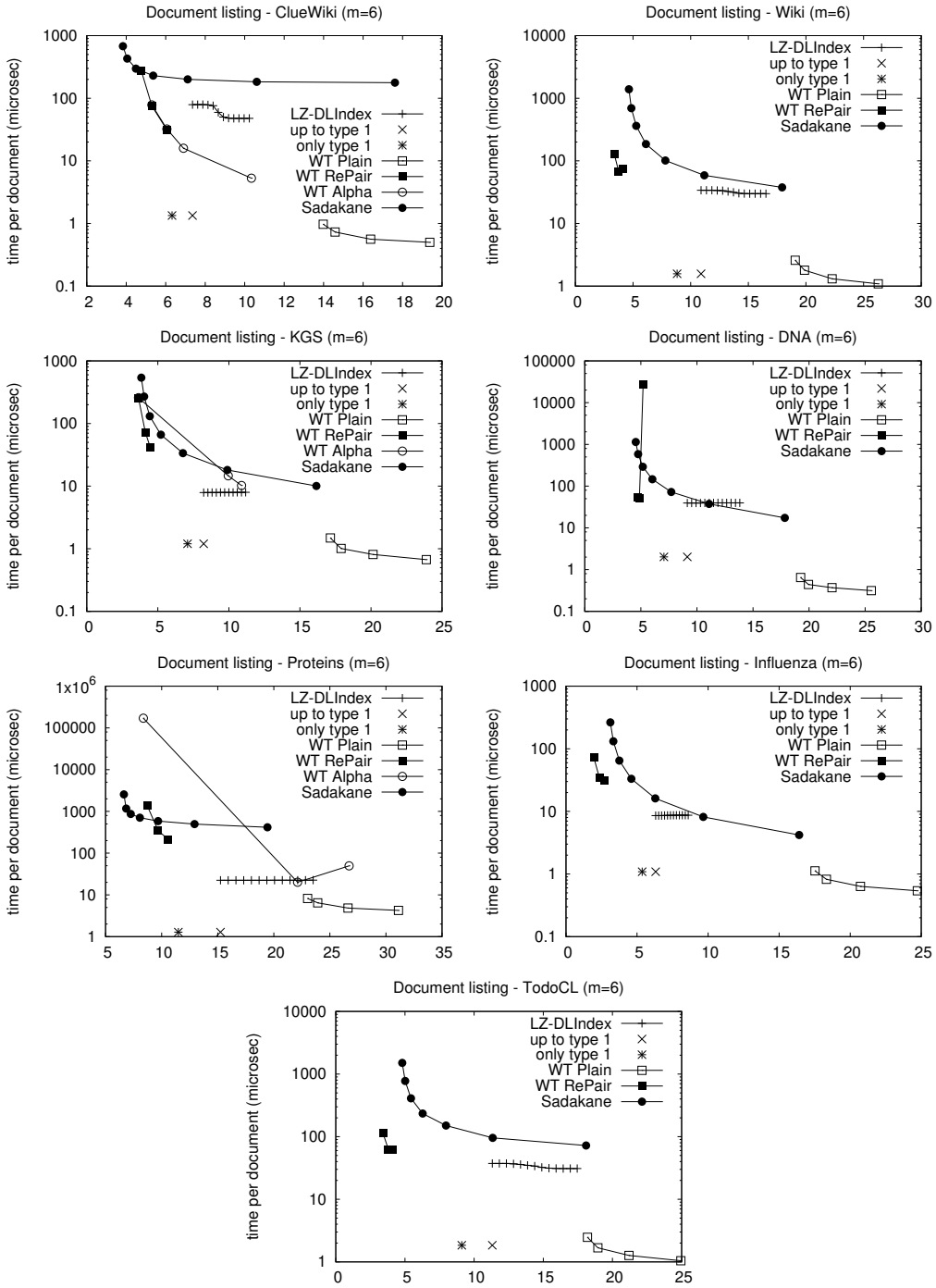

Figure 7: Space/time document listing comparison for pattern length $m = 6$. The x-axis is the space in bpc.

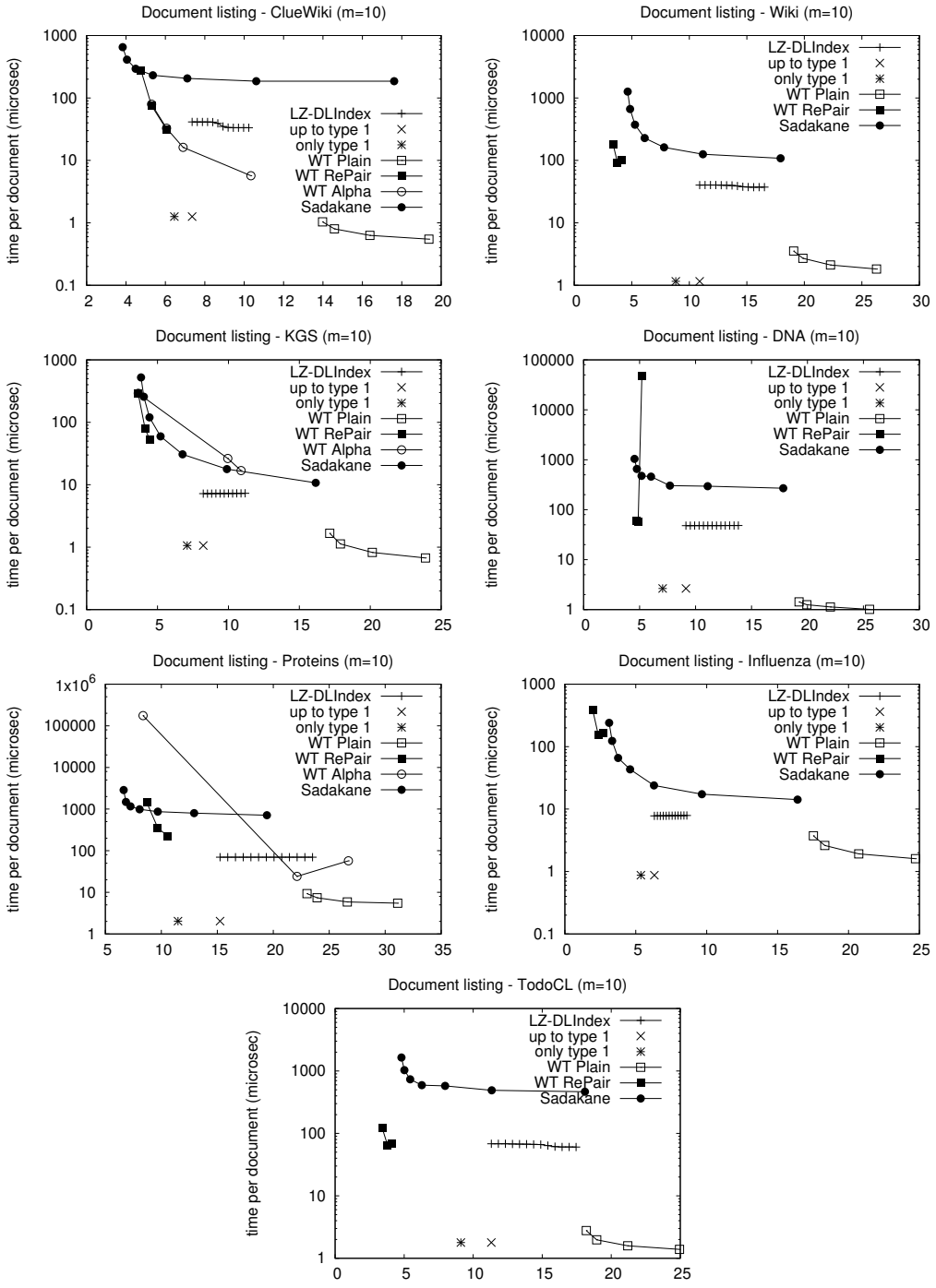

Figure 8: Space/time document listing comparison for pattern length $m = 10$. The x-axis is the space in bpc.

Pair outperforms the LZ-DLIndex in both time and space, whereas in `KGS`, `Proteins`, and `Influenza`, the LZ-DLIndex is much faster. The LZ-DLIndex is comparable to WT Alpha in various cases, but it is much easier to tune.

The partial variants of the LZ-DLIndex reach much better time, similar and even faster than those of WT Plain. They return about one result per microsecond. Therefore, in scenarios where we return the occurrences progressively, for example to be displayed in an interface, the "up to type 1" structure is very efficient, as it retrieves the first occurrences very fast.

The variant that can only return the occurrences of type 1 is also significantly smaller. Next we study the fraction of the total set that is found with this type of occurrences.

5.4.3. Quality

Now we measure the quality of our small and fast approximation of the LZ-DLIndex. As explained, it returns the documents where p is contained in at least one full phrase. Our analysis showed that, as n grows, the fraction of these documents should asymptotically approach the total answer set.

Figure 9 explores the behavior on a large collection, `TodoCL`, with a real query log. We tested one-word and two-word queries. As expected, the ratio of documents returned grows with n and decreases with the query length. When we reach 2.2GB, the approximation returns about 80% of all the answers. Note that this was already around 75% on just 200MB.

6. An LZ-based Index for Top-k Retrieval

Our structure for solving top-k retrieval, called LZ-TopkIndex, is inspired by the succinct approach of Hon et al. [20, 38], which stores top-κ answers for some chosen suffix tree nodes. Our idea is to solve the queries by brute force in principle, that is, obtaining all the occurrences and selecting the k most frequent documents. However, if this implies processing more than $g \cdot k$ occurrences, and the pattern has occurrences of type 1, then the top-k answers will be precomputed in the corresponding RevTrie node. Here g is a space/time tradeoff parameter.

Parameter g then determines which RevTrie nodes will store a top-κ answer, and for which κ. The (empty or nonempty) RevTrie nodes representing a string with at least g occurrences are then marked in a bitvector. Yet, we never mark empty unary nodes because their set of occurrences is the same as for their child. Each marked node stores a number κ of documents where

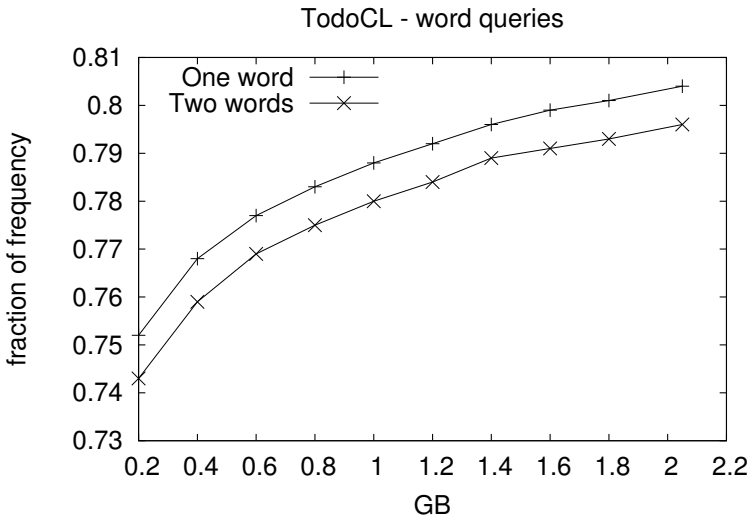

Figure 9: Fraction of the real answers of our LZ-DLIndex found as type 1, for real queries, as a function of the prefix size of `TodoCL`, for words and two-word phrases.

it appears most often: if the node has o occurrences, then it stores $\kappa = \lfloor o/g \rfloor$ precomputed answers. This guarantees that, if $k > \kappa$ and we need to find all the o occurrences by brute force, it is because p has less than gk occurrences, and thus the effort to collect them individually is no more than $O(g \lg n')$ per result returned (dominated by the occurrences of type 2).

Inspired by the fact that it is so fast to retrieve the occurrences of type 1 and these form a large portion of the documents where p appears, we also design an approximate variant of the index that only considers occurrences of type 1, which we call LZ-TopkApp. Now the brute-force approach only counts the occurrences of type 1, and the κ values are calculated according to those numbers. Still, the top-κ precomputed lists store the correct top-κ answers. The space and time of LZ-TopkApp are then close to those of the LZ-DLIndex variant "only type 1"; in particular the cost per occurrence is just $O(g)$. According to the analysis in Section 5.2.4, LZ-TopkApp returns the top-k answers considering a fraction of the occurrences of p that tends to 1 as n increases, thus we expect the quality of the answer to increase with n. We now give the details of the structures.

6.1. Structure

The structure of LZ-TopkApp includes the LZTrie, RevTrie, and Node components of the LZ-DLIndex. We also include Doc_{lz}, but not Doc_{rev} nor

$LDoc_{rev}$. In addition, LZ-TopkIndex includes Range, to find the occurrences of type 2, but not the RMQ structures the LZ-DLIndex associates with it. In exchange, we include a new structure, *Top*, where the precomputed top-κ answers are maintained. Apart from the bitvector B_{top} that tells which nodes are marked, we store the top-κ answers for each marked node in an array K_{top}, and mark the beginning of each answer set in a bitvector LK_{top}. The detail is as follows:

B_{top}: A bitvector marking which RevTrie nodes have top-κ answers precomputed, in preorder.

K_{top}: The sequences of κ most frequent documents where each node marked in B_{top} appears, concatenated in the same order of B_{top}. The identifiers are stored using $\lg D$ bits, in decreasing frequency order.

LK_{top}: A bitvector marking the starting positions of the sequences in K_{top}.

A_{top}: Since there may be less than κ distinct documents where the marked node appears, this bitvector indicates whether a node marked in B_{top} already lists all of the possible documents.

Space. The larger g, the fewer RevTrie nodes store their top-κ documents. Consider a RevTrie node. If it has o occurrences, then it stores $\kappa \leq o/g$ precomputed answers (including zero, being not marked, if $o < g$). Adding over all the RevTrie nodes representing strings of the same length, no more than n/g precomputed results are stored, since the occurrences must be disjoint and can only add up to n. Therefore, if h is the maximum length of a phrase (or, equivalent, the height of LZTrie), we can have n/g results per length, adding up to $h(n/g)\lg D$ bits in total. We then choose $g = h\lg D$ to ensure this space is $O(n)$ bits.

Therefore, the space of LZ-TopkIndex is $2n'\lg n' + n'\lg D + 2n'\lg\sigma + o(n'\lg n') + O(n) \leq 3nH_l(T) + o(n\lg\sigma) + O(n)$ bits, and that of LZ-TopkApp is $n'\lg n' + n'\lg D + 2n'\lg\sigma + O(n) \leq 2nH_l(T) + o(n\lg\sigma) + O(n)$ bits.

6.2. Queries

We search for the locus v^r of p in RevTrie as before. Then we check in B_{top} whether the node is marked. If it is, *rank*/*select* queries on LK_{top} yield the range of top-κ document identifiers stored for v^r. If $k \leq \kappa$, or A_{top}

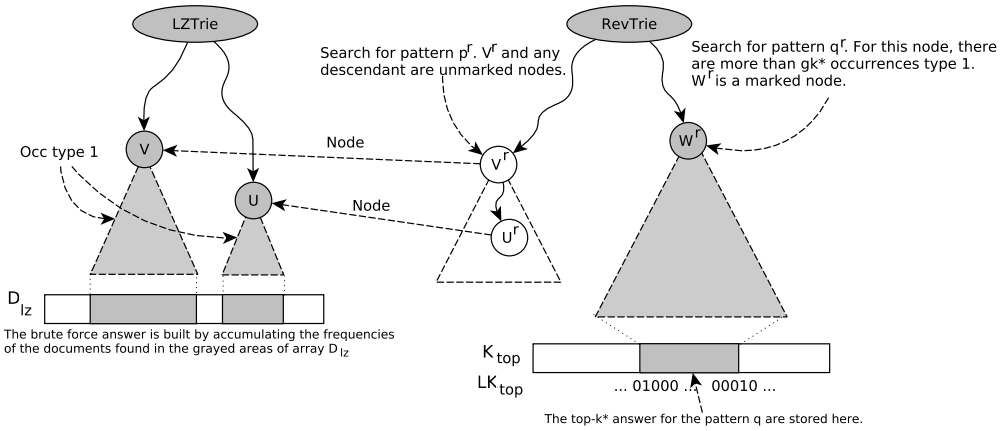

Figure 10: The main data structures of our approximate top-k index. The search for the pattern q^r reaches node w^r in RevTrie, which is marked in B_{top}. The marks in LK_{top} also indicate that there are $\kappa \geq k$ document identifiers stored. Therefore, the answer is retrieved from K_{top} using the marks in LK_{top}. The search for p^r, instead, reaches node v^r in RevTrie. Since this node is not marked, the answer is computed by accumulating frequencies from the document array of phases, D_{lz}. We use $k*$ for κ in the drawing.

indicates that v^r stores all the documents where it appears, we return the top-min(k, κ) documents stored for v^r and finish.

Otherwise, the stored answers are not sufficient (or do not exist) and we have to proceed by brute force. Then, just as for the basic index, we collect all the occurrences of type 1 by mapping every descendant u^r of v^r to node u in LZTrie using Node, and traversing the range of Doc_{lz} covered by u. Along this process, we accumulate the frequencies of the documents found in an initializable array [29, Sec. III.8.1], and at the end collect the k documents with the highest frequencies. In case of the LZ-TopkIndex, we also collect the occurrences of type 2 and 3, to ensure that the answer is completely correct.

Figure 10 illustrates the main components of our index and how we retrieve top-k answers in both cases: when the locus of the pattern contains the answer precomputed, or when the output is computed by brute force.

Time. The LZ-TopkIndex guarantees to spend $O(g \lg n')$ time per occurrence returned when p has occurrences of type 1. Otherwise, there is no guarantee. However, let us follow the analysis of Section 5.2.4. On texts generated by ergodic sources, the probability that p, appearing o times in T, has no

occurrences of type 1, is $(1 - \Theta(m/\lg_\sigma n))^o$. Taking the worst value $m = 2$ and multiplying by the cost $O(o \lg n')$ to find all such occurrences, this is upper bounded by $e^{-\Theta(o/\lg_\sigma n)} o \lg n$, which is maximized for $o = \Theta(\lg_\sigma n)$. Thus we absorb this case, on average, by adding $O(\lg^2 n)$ time. Considering the time for searching the tries and handling the occurrences of type 3, we obtain $O(m \lg^2 n + kg \lg n)$ time. The LZ-TopkApp structure, instead, reports nothing when p has no occurrences of type 1, and otherwise spends $O(g)$ time per occurrence returned. Thus its total time is always $O(m + kg)$.

If we assume that the text is generated by a memoryless source, then the LZTrie can be thought of as the trie induced by n' infinite and statistically independent strings. Under a wide set of probabilistic models, the height of such a trie is $h = O(\lg n')$ [44]. The result still holds if T is generated by a finite-memory source, where each symbol depends on $O(1)$ previous symbols. Therefore, we have that $g = h \lg D = O(\lg^2 n)$. Our previous calculations then yield time $O(m \lg^2 n + k \lg^3 n)$ for LZ-TopkIndex and $O(m + k \lg^2 n)$ for LZ-TopkApp.

6.3. Experimental Results

We leave public a 64-bit implementation of our indexes LZ-TopkIndex[11] and LZ-TopkApp[12]. We compare our index with previous work [24, 38] in terms of query time and space usage. We use the same document collections described in Table 1.

6.3.1. Space study

Figure 11 gives the space breakdown for DL-TopkApp, for various values of g. We group the data structures into four components: (1) LZTrie contains the tree topology and the document identifiers Doc_{lz}; (2) RevTrie considers the tree topology, the symbols of the edges, and the other bitvectors to perform pattern searches; (3) Node is the array mapping RevTrie to LZrie; and (4) Top counts the storage of the best documents for marked nodes and the bitvectors to extract them. Only the size of Top varies with g. It can be seen that reasonable values of g, depending on the collection, start at 32–256. The impact of g is slightly smaller on DL-TopkIndex.

[11] At `https://github.com/hferrada/LZ-TopK.git`.

[12] At `https://github.com/hferrada/LZ-AppTopK.git`.

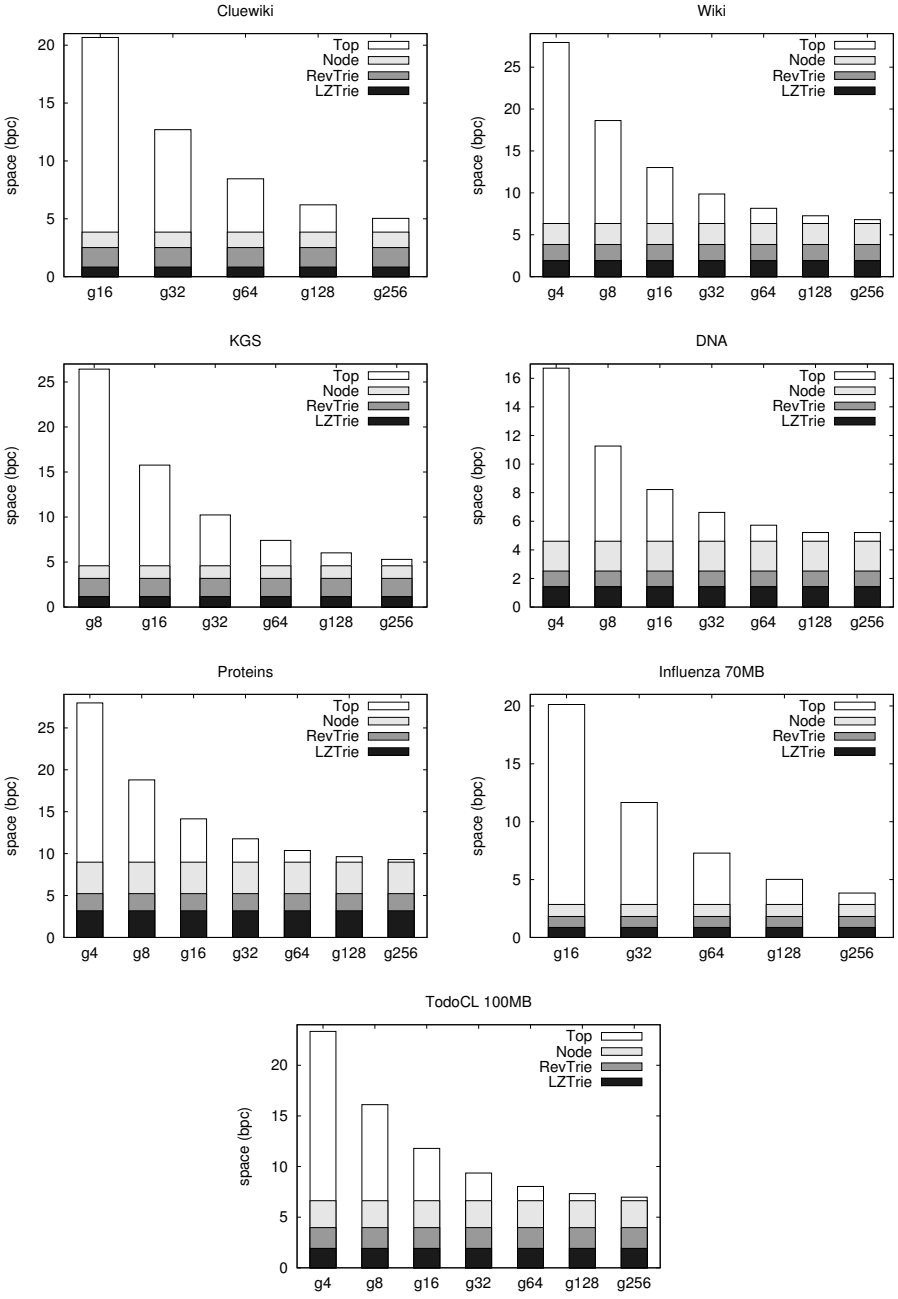

Figure 11: Space breakdown of our LZ-TopkApp structures for different g values (g is the x-axis).

6.3.2. Space/time tradeoffs

We compare our top-k indexes with the best previous solutions. We denote IDX-KN [24] and IDX-GN [17] the implementations of a fast and large structure [36]. We also include a choice of relevant space/time tradeoffs from the small and slow structure based on Hon et al.'s sampling [20] combined with wavelet trees [38], which we call HON-WT.

We consider search patterns of lengths $m = 6$ and $m = 10$ in Figures 12 and 13. We take strings from random positions in the collection, checking that they appear in a least k documents. We test $k = 10$ and $k = 100$. For our indexes we try the values $g = 256, 128, 64 \ldots$ until the size of component Top exceeds 24 bpc. For LZ-TopkApp we also include the case $g = +\infty$ (i.e., not precomputing any answer) to see if storing answers is worth the space.

Since n/n' is the average node depth in LZTrie, we set $g = (n/n') \lg D$ as a natural value. According to Table 1, this yields values in the range 110–330 for g. In most texts LZ-TopkApp uses 4–7 bpc with those values of g (except `Proteins`, where it uses 10 bpc), and solves top-k queries in around k–$5k$ μs. LZ-TopkIndex uses 5–8 bpc (12 bpc on `Proteins`) and solves queries in $10k$–$100k$ μs. Using a smaller g improves performance significantly in some cases, while increasing the space still within competitive bounds. Instead, not using top-κ answers at all significantly increases the times.

Our LZ-TopkIndex provides an interesting space/time tradeoff. Only HON-WT is able to use less space in some cases, generally at the price of being an order of magnitude slower. There is no other competing structure until we reach the space of IDX-GN, which is up to 2 orders of magnitude faster but almost always uses 17–21 bpc. IDX-KN is always slower and larger than IDX-GN.

The structure HON-WT can also use similar or less space than LZ-TopkApp, but at the cost of being 3–6 orders of magnitude slower. Even if using much more space, HON-WT is at least 2 orders of magnitude slower than LZ-TopkApp. On the side of the large and fast structures, IDX-GN obtains time similar to LZ-TopkApp, but it uses 2–4 times more space.

6.3.3. Quality

Our LZ-TopkApp index offers an excellent space/time tradeoff. However, it does not always ensure that the answer is completely accurate. In order to estimate how good the approximation is, we computed two measures of quality for the top-k approximation. The first one is the traditional *recall*, measured in the following way: for each value $k' \in [1, k]$, we measure how

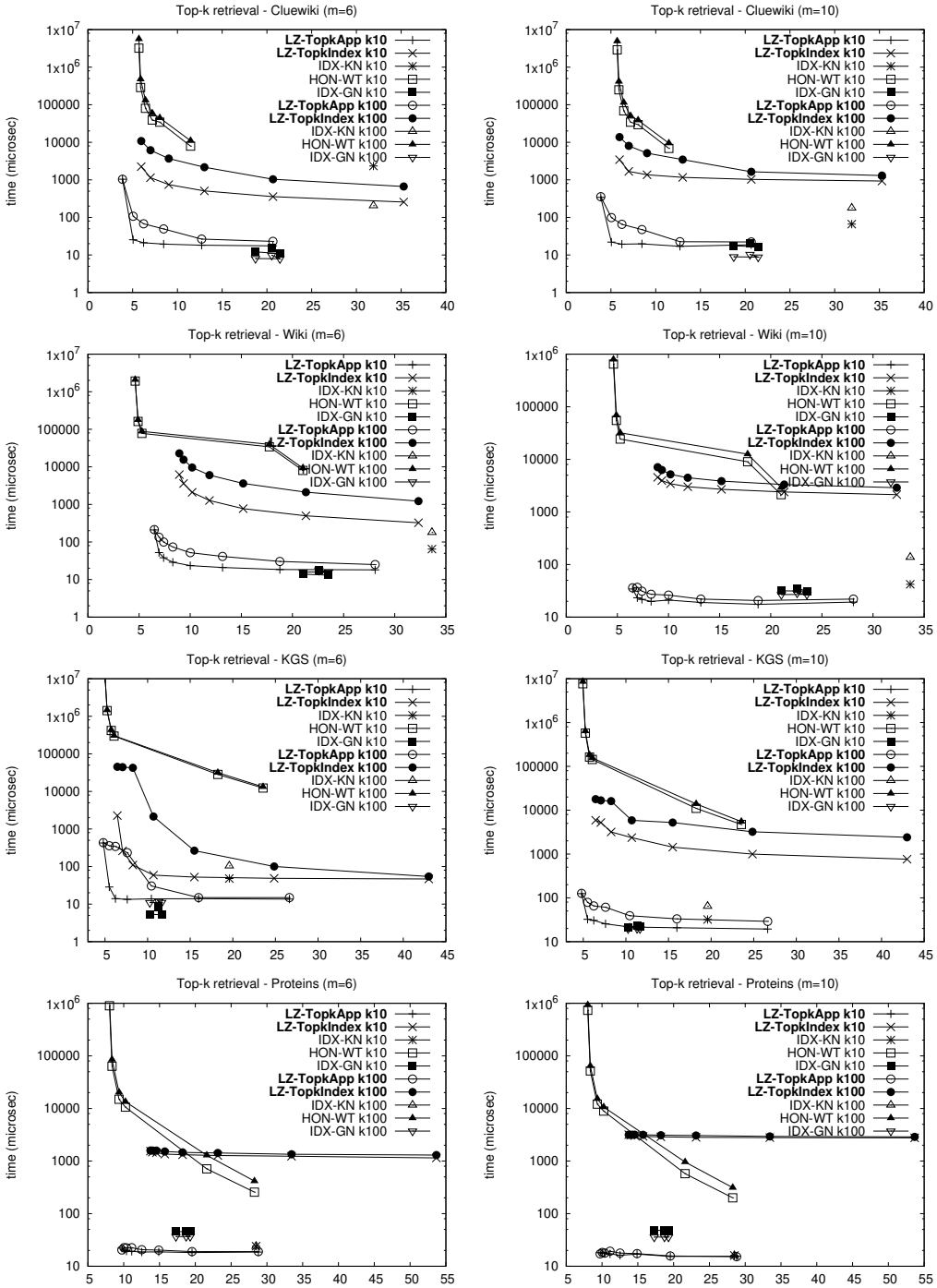

Figure 12: Space/time top-k comparison for pattern length $m = 6$ (left) and $m = 10$ (right). Space (bpc) is the x-axis.

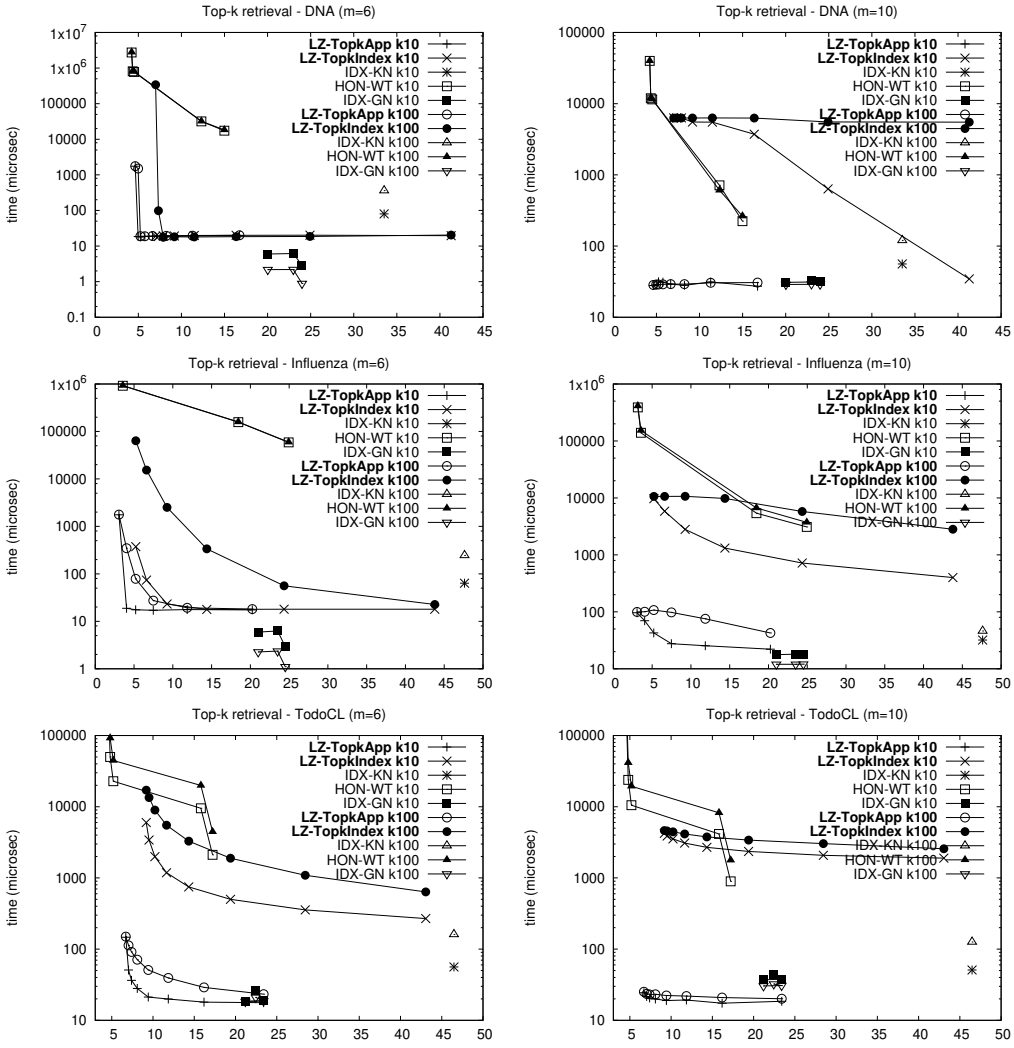

Figure 13: Space/time top-k comparison for pattern length $m = 6$ (left) and $m = 10$ (right). Space (bpc) is the x-axis.

many of the (correct) top-k' documents are reported within the (approximate) top-k results. This is shown in Figure 14. In this experiment we have selected the largest g value for each collection, which ensures that the total size of the index is around 12–16 bpc.

The point at $k' = 1$ (i.e., 0.1 in the x-axis for $k = 10$ and 0.01 for $k = 100$) indicates how many times the most relevant document is contained in the top-k approximate answer. The point at $k' = k$ (i.e., 1.0) gives traditional recall: how many of the correct top-k documents are actually returned.

This indicator is useful for applications where the top-k answer is post-processed with a more sophisticated relevance function in order to deliver a final answer of $k' \ll k$ results. For example, except for $m = 10$ on `Proteins` (where few occurrences of type 1 are found), we obtain a recall of 70%–100% if we use this top-k approximation to later extract the best 30% of the results (0.3 in the plots).

In most collections the recall is 60%–100% even for $k' = k$ (except on `Proteins` and `DNA`, which do not compress well). There are no large differences between $k = 10$ and $k = 100$. When there are, the quality is much better for $k = 100$.

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 too strict of a 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 fairer 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. This is the second indicator we compute. We omit the figures because the improvement is not that large compared to recall: now we obtain 70%–100% of quality for $k' = k$ (except for `Proteins` and `DNA`, which do not improve much), and 80%–100% for $k' = 30\%$ of k (except for `Proteins`).

On the other hand, the fact that better quality is obtained for shorter patterns coincides with our probabilistic analysis. Figure 15 illustrates this effect more closely, for increasing pattern lengths (using our second measure of quality from now on). For the moderate collection sizes of 25–130 MB we considered, we obtain quality well above 80% for $m = 2$–8 in top-10 (`Proteins`, again, is the exception). In most of the collections, the quality is over 90% for $m \leq 10$. For top-100, we obtain quality well above 80% for $m \leq 14$ (except for `DNA`, where the results are good only up to $m \leq 12$).

Our analysis also predicts that the quality improves as n grows. In the

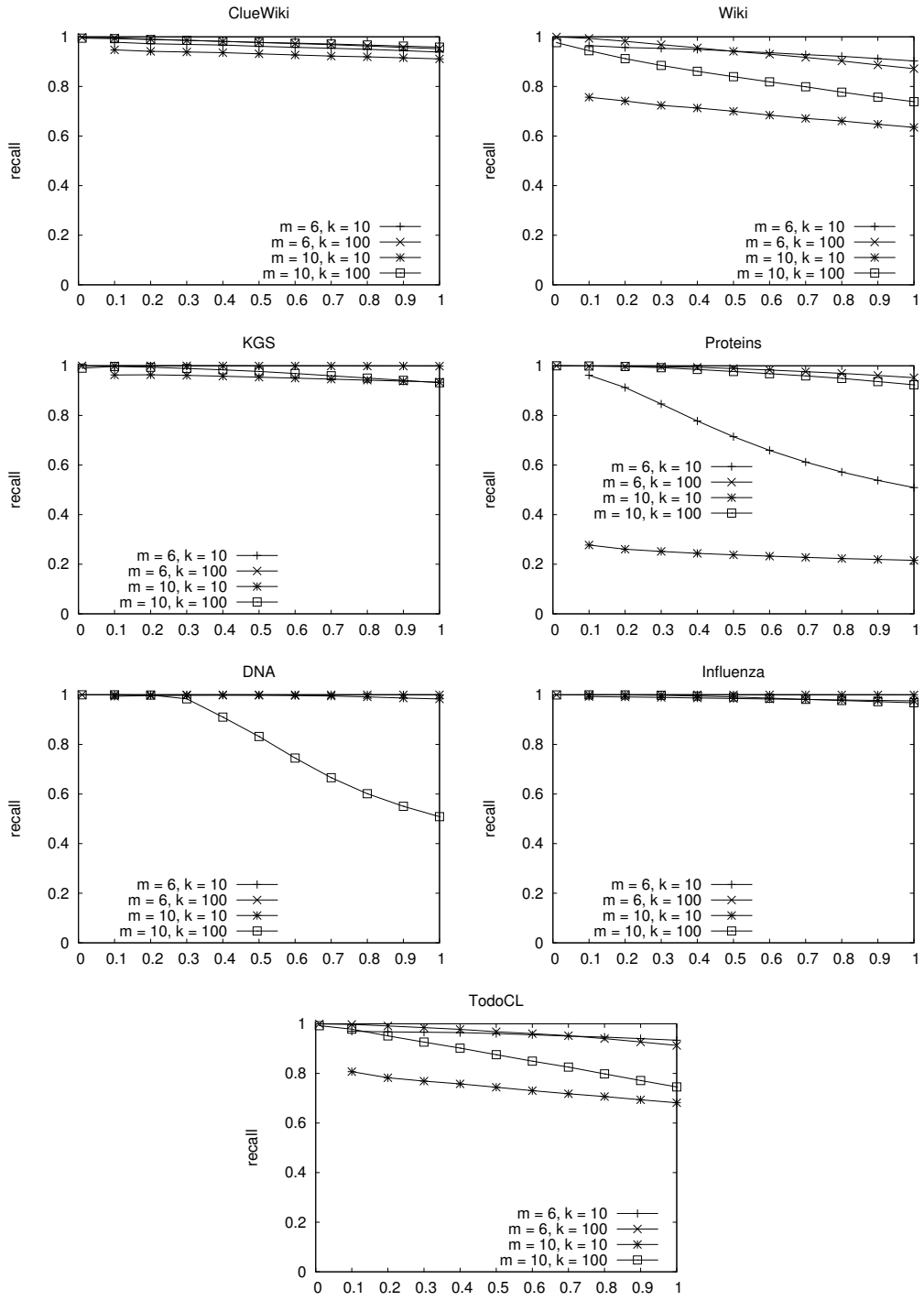

Figure 14: Recall of our approximate top-k solution, as a function of the fraction of the answer (x-axis).

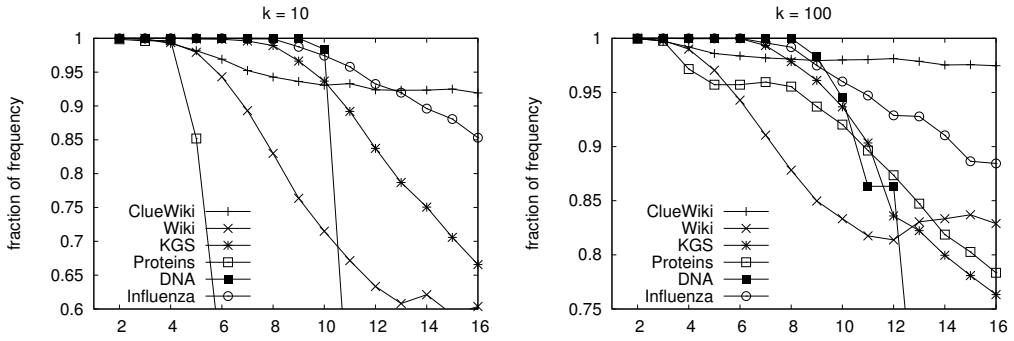

Figure 15: Quality of our approximate top-k solution, as a function of the pattern length, for top-10 (left) and top-100 (right). Each pattern appears at least in k documents.

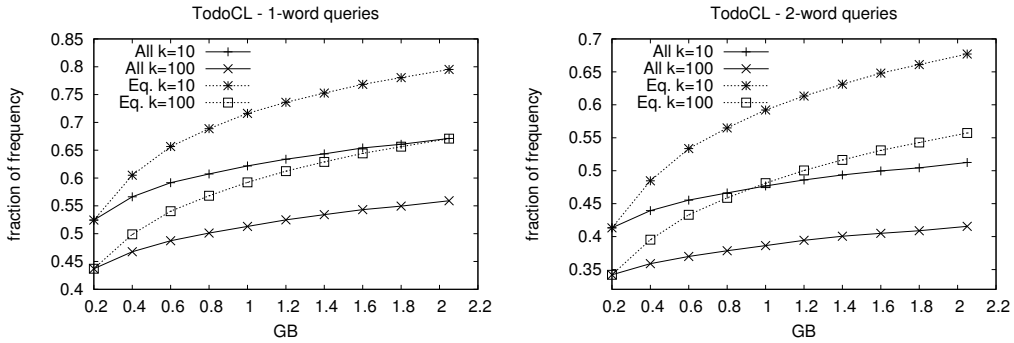

Figure 16: Fraction of the real answer found by LZ-TopkApp for real queries, as a function of the prefix size of `TodoCL` for words (left) and phrases of two words (right). Solid lines include new sets of queries for each prefix (labels "All k"). Dashed lines consider always the same set of queries from the first 200MB of the collection (labels "Eq. k").

next experiment we build the structure for increasing prefixes of `TodoCL`. Figure 16 (solid lines) shows the quality obtained for real query words (of length > 3 to exclude most stopwords), with average length 7.2, and 2-word phrases, with average length 8.0. We convert `TodoCL` to lowercase (as the distinction is generally not made in natural language queries). As predicted, the quality improves with n, from 44%–52% on 200MB ($n/n' = 10.1$) up to 56%–67% on 2.05GB ($n/n' = 12.7$) for words; and for 2-word phrases from 34%–42% on 200MB up to 42%–52% on 2.05GB.

The percentages are much lower than before, because many queries may appear just a few times in the collection. In those cases, a brute-force pattern matching is a better approach. Our LZ-TopkApp index performs better when the words appear many times, and thus a top-k query is more relevant.

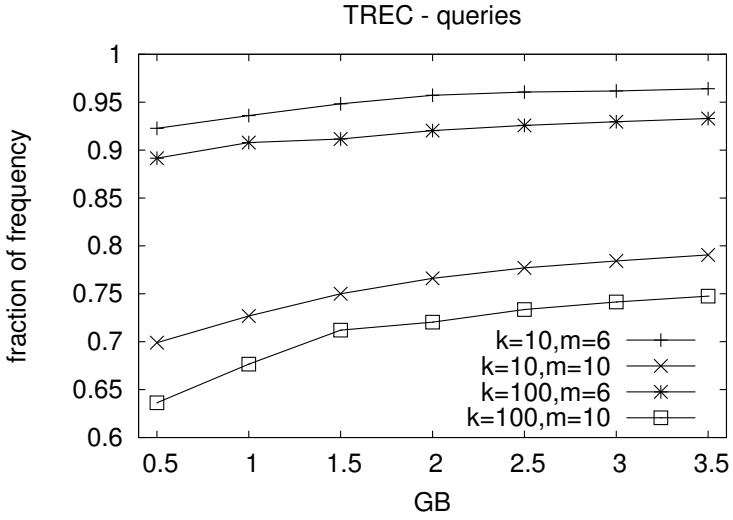

Figure 17: Fraction of the real answer found by LZ-TopkApp as a function of the prefix size of TREC, for arbitrary patterns of lengths 6 and 10, in top-10 and top-100 queries.

Figure 16 (dashed lines) repeats the experiment, but now we only use patterns that appear in the first 200MB, to query the structure for all the prefixes. The results are much better because the queries appear more often.

In the last experiment, we measure the improvement with n without the problem of real queries that may appear infrequently, and with another large text collection, TREC. We extract patterns of lengths $m = 6$ and $m = 10$ from random text positions, and that appear at least in k documents, for $k = 10$ and $k = 100$. The resulting quality is shown in Figure 17. Once again, our index gives an answer of high quality on large text collections.

7. Conclusions

All the current indices for document retrieval on general sequence collections build on suffix trees and arrays. They either take a considerable amount of space (17–21 bpc) or are considerably slow (milliseconds per query). In this article we take a completely different approach to the problem, and build instead on the LZ-Index family [32]. This is a pattern-matching index based on the LZ78 compression of the text $T_{1..n}$, which cuts it into $n' \leq n/\lg_\sigma n$ phrases (σ is the alphabet size). We borrow some of the structures of the existing indices, but manage to apply them on the n' phrases instead of the

n text positions. Therefore, we can use fast structures that use much space on a reduced set of positions, thereby obtaining low space and time.

Our indices extend the LZ-Index to carry out two typical document retrieval tasks: document listing (LZ-DLIndex) and top-k retrieval (LZ-TopkIndex). Both structures achieve competitive space/time tradeoffs compared to existing solutions, dominating a significant part of the space/time tradeoff map. In addition, they can be transformed into indices yielding approximate solutions in strikingly low space and time. The LZ-DLIndex typically uses 7 bpc and outputs most of the documents (75%–80%) very fast, each in about 1 microsecond. An approximate variant of our top-k index (LZ-TopkApp) typically uses 4–7 bpc and returns each result in about 1–5 microseconds. Its results are approximate, but the quality of the result is over 90% in typical cases. In the many applications where a partial or approximate answer is acceptable, these LZ-Index extensions perform as fast as the fastest previous solutions while using 2–4 times less space, and use as little space as the smallest previous solutions while being orders of magnitude faster. We have left our implementations publicly accessible from `https://github.com/hferrada`.

We believe this new approach to low-space indices for document retrieval is promising, and that several more results can be obtained, in particular exploiting other Lempel-Ziv pattern matching indices [41, 26]. Adapting classical indices to retrieve approximate solutions can also prove to be a productive avenue to improve current solutions by orders of magnitude while still providing useful functionality.

References

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 12th Workshop on Algorithm Engineering and Experiments (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] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Pearson Education, 2nd edition, 2011.

[4] D. Belazzougui, G. Navarro, and D. Valenzuela. Improved compressed

indexes for full-text document retrieval. *Journal of Discrete Algorithms*, 18:3–13, 2013.

[5] D. Benoit, E. Demaine, J. I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[6] S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[7] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1998.

[8] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. 18th Annual European Symposium on Algorithms (ESA), part II*, LNCS 6347, pages 194–205, 2010.

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

[10] H. Ferrada and G. Navarro. Efficient compressed indexing for approximate top-k string retrieval. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8799, pages 18–30, 2014.

[11] H. Ferrada and G. Navarro. Improved range minimum queries. *Journal of Discrete Algorithms*, 43:72–80, 2017.

[12] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[13] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[14] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[15] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[16] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

[17] S. Gog and G. Navarro. Improved single-term top-k document retrieval. In *Proc. 17th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 24–32, 2015.

[18] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[19] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[20] W. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. *Proc. IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 713–722, 2009.

[21] W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. Space-efficient frameworks for top-k string retrieval. *Journal of the ACM*, 61(2):article 9, 2014.

[22] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[23] D. E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[24] R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. 23rd Data Compression Conference (DCC)*, pages 351–360, 2013.

[25] S. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 2000.

[26] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[27] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[28] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[29] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.

[30] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2002.

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

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

[33] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics*, 13:article 2, 2009.

[34] G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014. 47 pages.

[35] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

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

[37] G. Navarro, Y. Nekrich, and L. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

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

[39] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[40] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, 2007.

[41] L. M. S. Russo and A. L. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval*, 11(4):359–388, 2008.

[42] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[43] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[44] W. Szpankowski. On the height of digital trees and related problems. *Algorithmica*, 6(2):256–277, 1991.

[45] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 205–215, 2007.

[46] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 105–114, 2011.

[47] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[48] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.