Improved Single-Term Top-k Document Retrieval*

Simon Gog† Gonzalo Navarro‡

Abstract

On natural language text collections, finding the k documents most relevant to a query is generally solved with inverted indexes. On general string collections, however, more sophisticated data structures are necessary. Navarro and Nekrich [SODA 2012] showed that a linear-space index can solve such top-k queries in optimal time $O(m + k)$, where m is the query length. Konow and Navarro [DCC 2013] implemented the scheme, managing to solve top-k queries within microseconds with an index using 3.3–4.0 bytes per character (this includes the storage of the collection itself). In this paper we introduce a new implementation using significantly less space, 2.5–3.0 bytes per character (again, including the collection), and retaining similar query times. For short queries, which are the most difficult, our new index actually outperforms the previous one, as well as all the other solutions in the literature. We also show that our index can be built on very large text collections, and that it can handle phrase queries efficiently on natural language text collections. In the latter case, it uses about the same space of the tokenized text (and replaces it), while answering phrase queries an order of magnitude faster than a positional inverted index.

1 Introduction

The landmark problem in Information Retrieval is to find, from a set of *documents*, those that are most relevant to a user-entered *query*. Relevance is generally measured using some function of the occurrences of the query in the documents. An *index*, that is, a data structure built on the documents, is used to solve this problem efficiently. When the documents and queries are formed by so-called "natural language" (i.e., sequences of words from a vocabulary), the *inverted index* is the preferred data structure [2]. The inverted index, however, cannot handle the more general scenario where the documents are arbitrary strings and queries can match any substring. This is the case of bioinformatic sequences, multimedia sequences, source code repositories, and even East Asian languages where word separators are implicit [16].

Handling collections of general strings has proved much more challenging. Suffix trees [25] are indexes using linear space that find all the occ occurrences of a query string $P[1, m]$ in the collection in time $O(m + occ)$. While the k documents most relevant to P can generally be computed from this set of occurrences, it is not efficient to generate all the occurrences of P in order to solve such "top-k" queries. The first efficient linear-space solution for top-k queries on general string collections was given by Hon, Shah and Vitter [13], who achieved $O(m + k \log k)$ time. Later, Navarro and Nekrich [18] improved this time to the optimal $O(m+k)$. While in this latter theoretical work the "linear space" was about $80n$ bytes for a collection of n characters in total, Konow and Navarro [14] engineered a version that used $3.3n$–$4.0n$ bytes[1] and solved queries within microseconds (μs).

Our contributions. We present a new implementation of Navarro and Nekrich's data structure. The main idea of their structure is to associate to each suffix tree node a number of weighted points on consecutive columns of a two-dimensional grid, and then reduce top-k queries to finding the k heaviest points on a 3-sided interval of this grid. Our first novelty is a simple and clever mapping from suffix tree nodes to the range of grid columns associated to their subtrees, which we adapt from a data structure by Sadakane [24] originally designed for document counting (i.e., count the number of documents P appears in). The second main component of our solution is a recent data structure for top-k queries on a grid of points [1], which uses little space and, although does not offer worst-case time guarantees, performs competitively. We also introduce efficient construction algorithms that allow us build our structures on very large text collections.

Our experimental results show that our structure uses $2.5n$–$3.0n$ bytes (which includes the space to store the collection itself), well below the $3.3n$–$4.0n$ of Konow and Navarro [14], while the query time performance is very similar. Indeed, for short queries, where occ is very large and thus top-k queries cannot resort to just finding all the occ occurrences and filtering the top-k documents, our new index is actually faster than the

*Funded in part by Fondecyt Grant 1-140796.

†Karlsruhe Institute of Technology (KIT), Germany.

‡Department of Computer Science, University of Chile, Chile.

[1] See `http://www.dcc.uchile.cl/gnavarro/fixes/dcc13.1.html`.

previous one. It is also orders of magnitude faster than previous heuristics [4], naive solutions [9], and compressed solutions [19], the latter of which does use less space. For example, our index solves any top-10 query in less than 300 μs.

We also consider collections of natural language text, regarded as sequences of words (not characters), so that our index offers a functionality similar to an inverted index. In this case our index takes about the same space of the tokenized collection (i.e., one integer per word, and still including the storage of the collection itself), and solves word and phrase top-k queries more efficiently than an engineered (positional) inverted index.

2 Basic Concepts

Let $\mathcal{D}$ be a collection of N documents (strings) $d_0, \ldots, d_{N-1}$, and let n be the length of their concatenation $\mathcal{C}$. Fig. 1 shows our running example with $N = 3$ documents of total length $n = 13$. The suffix tree [25] of $\mathcal{C}$ is shown on the left of Fig. 2. It is a digital tree built on all the suffixes of $\mathcal{C}$, with unary paths compacted. Internal nodes represent repeated substrings and are circled, whereas leaves represent unique suffixes and are in squares that indicate their starting positions in $\mathcal{C}$. The concatenation of the squared nodes forms the so-called suffix array of $\mathcal{C}$, SA [15]. Below the leaves we show the document id each suffix belongs, which forms the so-called document array D (its cells are easily computed on the fly from SA). The *locus* of a string P is the node p whose string is the shortest one prefixed by P. Each occurrence of P in the collection starts a suffix whose leaf descends from its locus. For example, there are 2 occurrences of $P =$ `AA` (locus v_7), at positions 6 and 5.

The structure of Hon et al. [13] marks also with each document id d all the internal nodes that are lowest common ancestors (LCAs) of two leaves belonging to document d. The figure shows those not leaving from leaves: v_7 is marked with document 1, v_6 with documents 0 and 1, and the root v_0 with the three documents. Upward arrows are drawn for each node v marked with a document d to its nearest ancestor node u that is also marked with d. The arrows are assigned a weight, corresponding to the relevance score of the string of v in document d. Hon et al. prove that, if an internal node p is the locus of P, then there is exactly one upward arrow per distinct document where P appears, going from a subtree of p (or p itself) towards a (strict) ancestor of p. Therefore a top-k query can be solved by finding the k heaviest arrows that cross through node p.

Navarro and Nekrich [18] recast this latter search into a geometric problem. They define a grid where each upward arrow from v to u is assigned a unique column where a point with height equal to the depth of u is placed, with weight equal to that of the arrow. The columns are assigned by preorder of the nodes v. Therefore, the query is reduced to (1) finding the locus p of P, (2) finding the range of columns corresponding to all the arrows leaving from a node descending from the locus p of P, and (3) finding the k heaviest points within the column range, and with row range up to the depth of p. The grid is shown on the right of Fig. 2, with weights corresponding to *term frequencies* (number of times the string appears in the document): the leftmost point refers to the leftmost arrow that goes from v_6 to v_0, corresponding to document d_0 (document ids are not drawn in the grid), with weight 2 (as `A`, the string of v_6, appears twice in d_0) and depth of the arrived node (v_0) equal to zero. Following in preorder of the source nodes of the arrows, we have the second arrow from v_6 to v_0, the arrow from v_7 to v_6, and the arrow from v_{12} to v_0.

Konow and Navarro [14] implemented this scheme with the aim of reducing space. The suffix tree is represented with a compressed suffix array (CSA) [17], which finds the leaf range $[sp, ep]$ of the locus node of P, plus a parentheses representation of the suffix tree topology [22], which finds the locus node p with an LCA operation from the spth and epth leaves. Given p, the topology representation can also compute its depth and subtree size. The numbers of columns induced by each suffix tree node are written in unary and concatenated into a bitvector M (similar to H in Fig. 2, but in preorder, whereas H uses our new ordering), which allows determining the column range in the grid that corresponds to p and its descendants. Konow and Navarro use a grid representation G using $3 \log h$ bits per column, where h is the suffix tree height, which finds the k heaviest points efficiently. In addition, they must store the document identifier and weight of each point (see DOC and w in Fig. 2). Finally, they reduce space by removing from the grid all the arrows with weight 1 (i.e., leaving from leaves; these are already omitted in the figure), and marking them in a second bitvector, L. If there are less than k points in the range with weight more than 1, then the answer set is completed by listing up to k distinct documents appearing under p. This can be done efficiently with $2n + o(n)$ extra bits of space with a range minimum query (RmQ) data structure called RmQC [23, 6].

Related work. Culpepper et al. [4] studied various heuristics to solve top-k queries on top of a CSA and the document array D (typically using $2n$–$4n$ bytes). We compare their best performing variant, GREEDY, in this paper. Culpepper et al. [5] adapted the scheme to large natural language text collections (where each

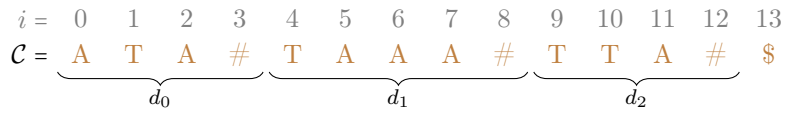

Figure 1: Concatenation $\mathcal{C}$ of our 3-document example collection $\mathcal{D}$.

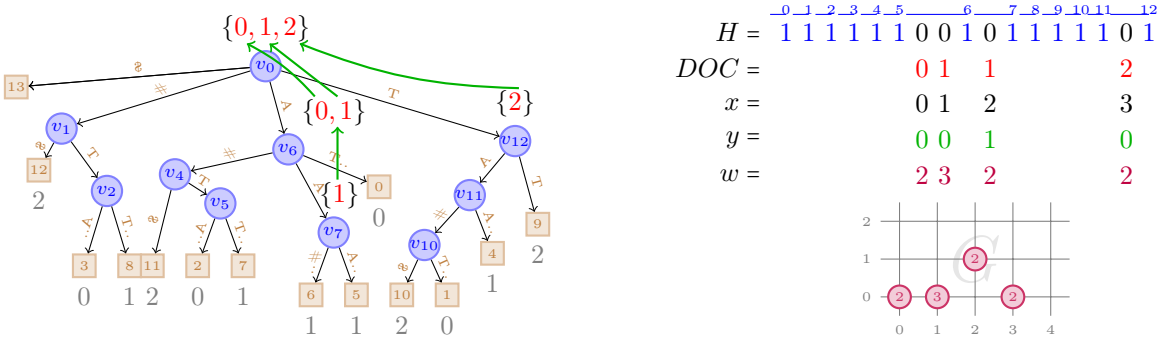

Figure 2: Left: Suffix tree of the example collection in Fig. 1, with the document marks and upward arrows of Hon et al.'s scheme. The index i of inner node v_i corresponds to the index of the rightmost leaf in the subtree of the leftmost child of v_i. Right: Bitvector H concatenates a 1 per new suffix tree node and a 0 per new column induced by an arrow leaving from that node. Each 0 has associated the document id and weight of the corresponding arrow.

word is taken as an atomic symbol), showing that it was competitive with inverted indexes for some queries (see previous work on this line by Patil et al. [20]). The seminal work of Hon et al. [13] also included succinct variants, which were implemented by Navarro and Valenzuela [19] on top of a compressed representation of D. They showed that the extra space induced by Hon et al. was low and significantly reduced the time of earlier heuristics [4]. Their compressed representations used $1.0n$–$2.5n$ bytes, but compression raised query times to the order of milliseconds, much slower than the microseconds achieved in this paper, so their work is not compared here. There is another possibly practical solution by Hon et al. [12], which has not yet been implemented. Finally, we compare with SORT [9], a baseline that also uses the CSA and D. It just obtains the *occ* occurrences of P from D and then extracts the k most relevant documents. In this and previous experimental comparisons, the relevance measure used is the term frequency and the query is a single pattern string; we briefly mention extensions in the conclusions.

3 A New Implementation

In this section we present the main two ideas that lead to a conceptually simpler and, as we will see later, faster and smaller representation than the previous one [14].

Mapping the suffix tree to the grid. We reorganize the grid so that a more space-efficient mapping from a pattern P to a 3-sided range query in the grid is possible. The previous solution [14] used $2t$ bits to represent the topology of the suffix tree (of t nodes, where $n \le t \le 2n$), plus t bits for L, plus between $n/(\sigma-1)$ and n bits for M, where σ is the alphabet size. This gives a total of $3n$ to $7n$ bits for the mapping. Our representation will consist of a single bitvector, H, of length $2n - N$, and the mapping will be simpler.

We first explain how we list the upward arrows leaving the suffix tree nodes in the grid, and then show how we can efficiently map to the grid. We identify each internal node with the position of the rightmost leaf of its leftmost subtree. Fig. 2 already shows this naming for the internal nodes: The root node is named v_0 because the position of its first child (which is also a leaf) is 0, the rightmost leaf of the leftmost child of node v_6 is at position 6, and so on. Note that the names are between 0 and $n-1$ and they are unique (although not all names must exist, e.g., there is no node v_3 in our suffix tree). The bitvector H is generated by first writing n 1s and then inserting a 0 right before the i-th 1 per upward arrow leaving node v_i.

The 0s in H then correspond to the x-domain in the grid, while the y-coordinate (depth of target node), weight and document id of the upward arrow, are stored associated to the corresponding 0 in H (in arrays y, w, and DOC, respectively, see the right of Fig. 2).

Now assume the CSA search yields the leaf interval $[sp, ep]$ for P. These are the positions of the leftmost and rightmost leaves that descend from the locus node p

```
map_interval([sp, ep])
  [sp', ep'] ← [0, −1]
  if sp > 0 then
    sp' ← select(sp − 1, H) − (sp − 1)
  if ep > 0
    ep' ← select(ep − 1, H) − ep
  return [sp', ep']
```

Figure 3: Constant-time mapping from a leaf interval $[sp, ep]$ to the x-domain of the grid G.

of P (although we will not compute p). In the following we will use the key property of our new data layout.

LEMMA 3.1. *A node v_i, if it exists, lies in the subtree of p (including p) if and only if $i \in [sp, ep-1]$.*

Proof. If v_i is in the subtree of p, then its range of descendant leaves is included in that of p, $[sp, ep]$. Since there are no unary nodes, the rightmost child v_r of v_i has at least one descendant leaf, and thus the leaves descending from the leftmost child v_l of v_i are within $[sp, ep-1]$. In particular, the rightmost leaf descending from v_l, i, also belongs to $[sp, ep-1]$. Conversely, if $i \in [sp, ep-1]$ and v_i exists, then v_i is an ancestor of leaf i and so is p, thus they are the same or one descends from the other. However, if p descended from v_i, then i could only be ep (if p was the leftmost child v_l of v_i or belonged to the rightmost path descending from v_l), or it would be outside $[sp, ep]$. ∎

Therefore, all the upward arrows leaving from nodes in the subtree of p are stored contiguously in the grid, and can be obtained by finding the 0s that are between the $(sp-1)$th and the $(ep-1)$th 1 in H (as the 0s are placed before their corresponding 1). Note that no LCA operation on any suffix tree topology is necessary, only operation $select(j, H)$, which finds the position of the jth 1 in H and can be implemented in constant time and $o(n)$ bits of space [3]. The pseudocode is given in Fig. 3.

Note that, since we do not represent the suffix tree topology, we cannot compute the depth of the locus node p. Instead of node depths, we will store the string depths of the nodes (that is, the length of the string they represent) as the y-coordinates. Thus the query can use simply $|P|$ as the y-coordinate limit of the 3-sided query. That is, after applying the mapping of Fig. 3, we find the k heaviest points in the range $[sp', ep'] \times [0, |P|-1]$ of G.

A smaller grid representation. In the previous implementation [14], the grid G is represented with a combination of wavelet tree [11] and range maximum query (RMQ) data structures [6], that ensure query time $O((k + \log h) \log h)$. The price of this guaranteed worst-case time is a heavy representation of the grid, which uses 3 times the space needed to just represent the y vector, and they have to store in addition vector w in plain form.

We use instead a representation that compresses the vectors x, y and w. The K^2-treap [1] is essentially a quad-tree representation of the grid (more precisely, the grid is split into $K \times K$ sub-grids at each iteration, so a quad-tree corresponds to a 2^2-treap). Each node appends K^2 bits to a bitvector T, telling whether each sub-grid is empty or not. The nonempty sub-grids are recursively subdivided, and the final bitvector T represents all the points. The K^2-treap can be navigated using $select(j, T)$ and $rank(i, T)$ (how many 1s are there in $T[0, i]$) operations, which need $o(|T|)$ bits and constant time [3]. Moreover, each node stores explicitly the position and weight of the heaviest point in its sub-grid, and the point is removed from its recursive refinement. This information on each K^2-treap node is associated to the 1-bit of its parent, and stored in the appropriate order in (reshuffled) arrays x, y, and w. Note that values x and y require fewer bits as we advance in the grid subdivision, and weights w can be stored differentially with respect to the maximum weight of the parent grid, so they also generally require fewer bits as we descend.

The search for the top-k points on a K^2-treap starts at its root node. It includes the heaviest point in the result set if the point falls inside the query range. Then, it continues recursively inside all the subgrids that intersect the query range. Those subgrids are not immediately processed, but rather inserted into a global priority queue and processed in order of their maximum weight. The result set always stores the k heaviest points we have seen up to now. As soon as the maximum weight of the next sub-grid to process is not larger than the weight of the kth point in the result set, the process stops.

This process has no good worst-case time complexity. However, the experiments will show that this grid representation uses less space and yields a query time comparable with that of the heavyweight implementation [14].

Time- and space-efficient construction A bottleneck not yet addressed by Konow and Navarro [14] is the efficient construction of the index[2]. Both time-and space-efficiency have to be considered. We will concen-

[2]For example, their liner-time LCA-based index construction method takes 1.5 hours for an 80MB Wikipedia collection, where their peak main memory usage is 12.25 GB.

trate here on the construction of H, the grid G, and its K^2-treap representation. For H we construct an entropy-compressed suffix tree (CST) of $\mathcal{C}$ and a wavelet tree over the document array, WTD, of $n \log N$ bits. We then perform a depth-first-search traversal on the CST and calculate, for each node v_i, the list of its document id marks, by intersecting the document array ranges of v's children. This can be done in time proportional to the intersection result using the intersection on wavelet trees [7]. Since we can calculate the nodes in the order of their names, we can write H and the document ids (DOC) directly to disk. In a second traversal we calculate the upward arrows. For each document d we use a stack to maintain the string depth of the last occurrence of d in the tree. For a node v marked with d we push the string depth of v at the first visit and pop it after all the subtree of v is traversed. Note that this time we can read H and DOC from disk (in streamed mode) and avoid the intersection. In the same traversal we can calculate the weight array w by performing counting queries on WTD. Again, arrays y and w can be streamed to disk.

Finally, the K^2-treap is constructed in-place by a top-down level by level process. Let the input be stored as a sequence of triples (x, y, w) and let 1 be the root level and b be the bottom level. First, we determine the heaviest element by a linear scan, stream its weight out to disk and mark the element as deleted. We then partition the elements of the root level into K^2 ranges, such that all elements in range r ($0 \le r \le K^2$) have the property that $x/K^{b-\ell} \bmod K = r \bmod K$ and $y/K^{b-\ell} \bmod K = r/K$. For each non-empty (resp. empty) range r we add a 1 (resp. 0) to the bitvector T, which is streamed to disk. On the next level $\ell - 1$ we can detect nodes by checking the partitioning condition of the last level, find and mark the maximum weighted entry and apply the partitioning in the node. The time complexity of the process is $O(nK^2 \log_K n)$ and does not use extra space.

Summing up. Our structure consists of five main components: the compressed suffix array (CSA), the CSA-to-grid mapping (H), the K^2-treap over the grid, including coordinates and weights (G), the document ids associated with the grid elements, in the same order of the weights (DOC), and the RmQ structure to retrieve documents occurring once (RmQC).

4 Experiments

For this publication we have implemented the K^2-treap structure and added it to the Succinct Data Structure Library (SDSL) [9]. The K^2-treap implementation is generic (the value K, the bitvector representation of the K^2-ary tree, and the representation of the vector of relative weights, can be parametrized) and complements the description of Brisaboa et al. [1] with the efficient in-place construction described in the previous section. We use a compressed bitvector representation [21] of H, whose slight extra *select* time compared to a plain representation is negligible because only two *select* operations on H are done per query (recall Fig. 3). The new system is dubbed IDX_GN, and we compare it to IDX_KN, the system of Konow and Navarro [14]. As additional reference points we add the GREEDY solution [4] and the SORT framework [9], mentioned in Section 2.

Test environment and data set All experiments were run on a server equipped with an Intel(R) Xeon(R) E5-4640 CPU clocking at 2.40GHz. All experiments use a single core and at most 150GB of the installed 512GB of RAM. All programs were compiled with optimizations using g++ version 4.9.0. We are using test collections from the natural language domain: two Wikipedia dumps of different size, parsed as character and as word sequences [9], and a word parsing of the TREC GOV2 collection [10]. Table 1 summarizes their properties. Our implementation and benchmarks are publicly available[3] as part of the SUccinct Retrieval Framework (SURF), which was introduced in [10]. The experiments can be easily reproduced by running the provided scripts. Detailed information about all index parameters (e.g. the sampling density of the CSA) are provided in the configuration files. Note that the interested reader can also vary the parameters and rerun the benchmarks to study the effects on query time and index size.

Space usage We exemplify the space reduction of IDX_GN compared to IDX_KN in the case of ENWIKI-SML^c(since IDX_KN does not support multi-gigabyte inputs or word alphabets). While the grid mapping takes 46.5 MB ($8.6 + 12.8 + 25.1$ for bitvector B, M, and the tree topology) in IDX_KN, the new solution just takes 6.8 MB for bitvector H (13.0MB in uncompressed form).

The new grid representation as K^2-treap takes 57.7 MB ($21.7 + 24.5 + 11.5$ for weights, y-values, and topology) compared to 77.7 MB (23.3 MB for weights and 55.4 MB for the RmQ-enabled wavelet tree) in IDX_KN. These space savings result in index sizes between 2.5–3.0 times the original collection size for character alphabet collections, see Fig. 4. For word-parsed collections, IDX_GN takes space close to the original input, for example, for the 71.0 GB word parsing[4] of GOV2^w, the index size is 64.6 GB. Recall

[3] `https://github.com/simongog/surf/tree/single_term`

[4] Note that the unparsed GOV2 collection is 426 GB of text. The preprocessing of the Indri search engine (`http://`

Collection	n	N	n/N	σ	$\lvert\mathcal{C}\rvert$ in MB
character alphabet					
ENWIKI-SML^c	68,210,334	4,390	15,538	206	65
ENWIKI-BIG^c	8,945,231,276	3,903,703	2,291	211	8,535
word alphabet					
ENWIKI-SML^w	12,741,343	4,390	2,902	281,577	29
ENWIKI-BIG^w	1,690,724,944	3,903,703	433	8,289,354	4,646
GOV2^w	23,468,782,575	25,205,179	931	39,177,922	72,740

Table 1: Collection statistics: number of characters/words, number of documents, average document length, alphabet size, and total size in MB assuming that the character based collections use one byte per symbol and the word based ones use $\lceil \log \sigma \rceil$ bits per symbol.

that this space includes the CSA component, which can recover any portion of the text collection, and thus the collection does not need to be separately stored.

Retrieval speed The time of a top-k query consists of the time to match the pattern in the CSA, the time to map $[sp, ep]$ to the grid, the time to report the top-k documents using the K^2-treap, and —if less than k documents are found— the time to report frequency-one documents (which are not stored in the grid) using RmQC and, again, the CSA. We examine the cost of the different phases in our first experiment. We have fixed the pattern length ($m = 5$), the index (IDX_GN), and the collection (ENWIKI-BIG^c) since we are first interested in the effects of a varying k. We ran $40,000$ queries, each a pattern of length m extracted from a position chosen uniformly at random from $\mathcal{C}$. We increase k exponentially from 1 to 128 an report the average time per query in Fig. 5. As expected, the pattern matching with the CSA is independent of k and takes about $21\mu s$. The time to retrieve the first document out of the K^2-treap is relatively expensive. It is $71\mu s$ and is dominated by the cost of the priority-queue based search down the treap until a first (heaviest) element within the query range is found. The subsequent documents are cheaper to report. The time spent in the K^2-treap to report 16 documents ($131\mu s$) is about twice the time to report a single document. The average time per document retrieved via the K^2-treap is typically about 3–5 μs for $k \geq 64$. Fig. 5 also depicts the time spent on reporting single-occurrence documents. Essentially, for each such document, a constant number of RmQs and extraction of a suffix array cell from the CSA are performed. The cost of a RmQ is typically below 2 μs [8, Sec. 6.2], while the CSA access accounts for the remaining 100–300μs. The CSA access time is linearly dependent on a space/time tradeoff parameter s, which is set to $s = 32$. Note that top-k queries are meaningful when documents have different weights, and thus large k values that retrieve many documents with frequency 1 are not really interesting.

Fig. 6 shows our second experiment, where we explore the effect of varying the collection. We used $m = 5$ on the character collections and $m = 1$ on the word collections. We fix $k = 10$, a popular value for Web search engines, for which our query time is dominated by the K^2-treap operations. We show that the query time improves significantly if we use an uncompressed bitvector representation for the K^2-treap, while the space increases just by 1%–5%. In the following we will use this index version, denoted IDX_GN*. With this index, the query time is 30–90 μs for any collection.

Fig. 7 shows our last experiment, where we compare the query time of IDX_GN* to the simple solutions GREEDY and SORT, which worsen with larger intervals $[sp, ep]$, as well as IDX_KN. Due to a limitation of the latter implementation, the comparison is only possible on ENWIKI-SML^c. Compared to the simple schemes, IDX_GN* provides an excellent runtime for the important case of short patterns and phrases, for both small and large collections. For ENWIKI-BIG^w, as well as GOV2^w, the query time stays below $300\mu s$ (see Fig. 8), which is at least an order of magnitude faster than engineered positional inverted index implementations, which take a few milliseconds to solve phrase queries. On ENWIKI-SML^c we observe that IDX_GN* outperforms IDX_KN for pattern length ≤ 10, while using considerably less space. The query times for index IDX_GN$^+$, which is created by investing the space saving of IDX_GN* vs IDX_KN into a faster CSA, shows that our new solution outperforms IDX_KN if we use the same amount of memory.

For long queries, the problem is trivial, since there are only a few occurrences of P and the naive method SORT, which just scans the range $[sp, ep]$ of the un-

www.lemurproject.org/indri/) generated the sequence of stems words excluding html tags.

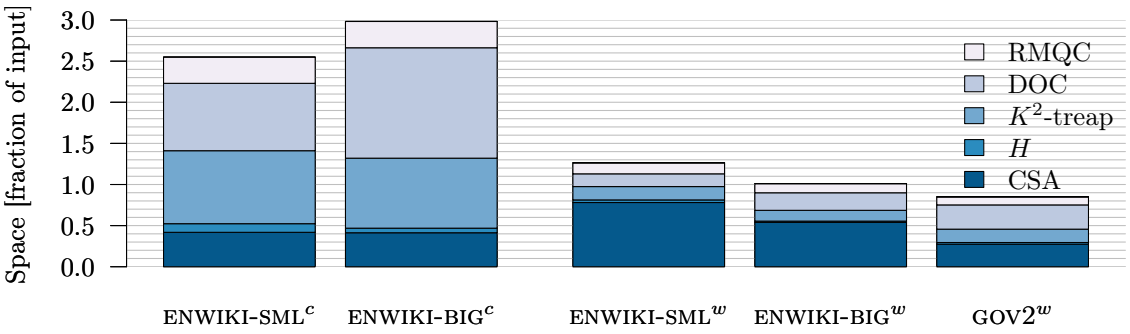

Figure 4: Detailed space breakdown of the new index (IDX_GN) for different collections.

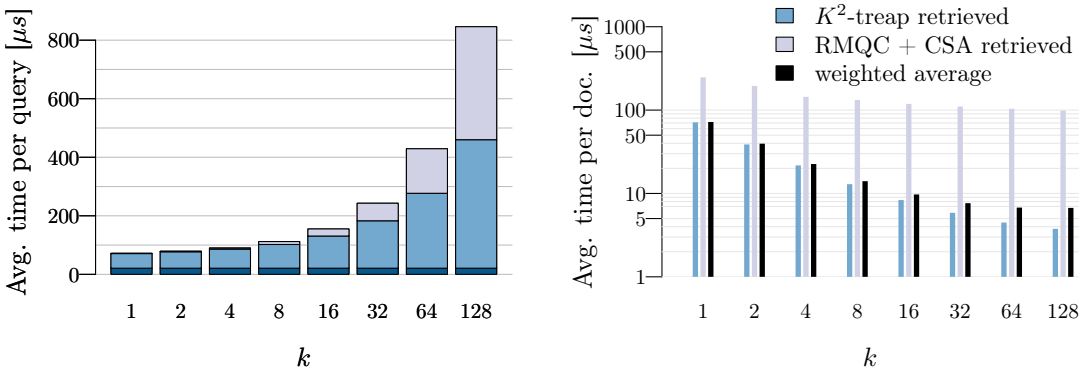

Figure 5: Query times for IDX_GN on ENWIKI-BIG^c, with $m = 5$. Left: Query time depending on k with a detailed breakdown of the three query phases. From the bottom: Matching in the CSA, search in the K^2-treap, and search using RmQC and CSA. Right: Average time per document (mixed), per K^2-treap retrieved document and RmQC+CSA retrieved document. CSA matching time is included in all cases.

compressed D array, is superior. Unlike GREEDY and SORT methods, our solution is robust, that is, it can also efficiently answer queries for short patterns. Note that solutions GREEDY or SORT use about the same space of our index, as both require the CSA plus $n \log N$ bits for the D array. This adds up to 2.0 times the text size on ENWIKI-SML^c and 3.2 on ENWIKI-BIG^c. On the word-based texts, instead, our index is much smaller.

Construction process The construction of IDX_KN for ENWIKI-SML^c takes 85 minutes with a peak memory of 9.3 GB (140 times the collection size!), which compromises its practical applicability. We replaced its linear-time LCA-based calculation of the grid by a more space-efficient $O(n \log N)$ time construction, which only takes 7 minutes to build IDX_GN* with a peak memory of 280 MB. This dramatic improvement made it also possible to build the index for GOV2^w (taking 4 days, memory peak 140 GB).

5 Conclusion

We have proposed a more efficient and conceptually simpler implementation of the top-k document retrieval index of Navarro and Nekrich [18]. Our index significantly improves previous work [14] in space and query time, and it outperforms all other approaches in the literature. We also proposed an efficient construction method, which enables handling large IR collections. We showed on those collections that our index is faster than typical positional inverted outperforms positional inverted indexes on phrase queries.

A relevant open problem is to reduce the space further while retaining competitive query times; current existing approaches using less space [19] are thousand times slower. Another interesting direction is to compete with inverted indexes in weighted Boolean queries, which are much harder than phrase queries for our data structure. Finally, we plan to adapt our index to more complex relevance measures, such as BM25.

References

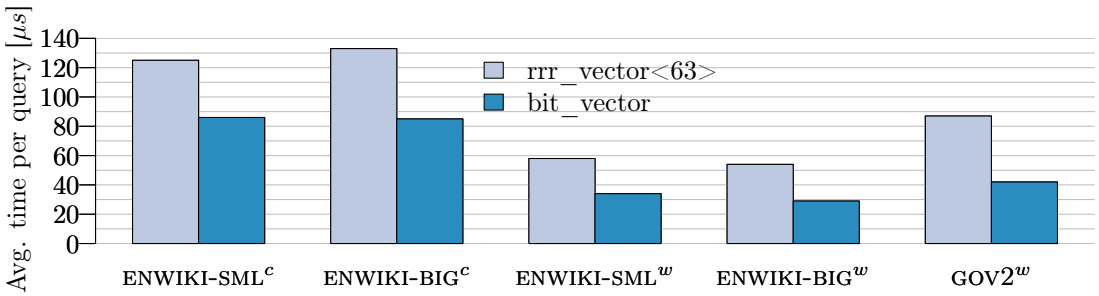

Figure 6: Average query time on different collections, using for the K^2-treap an entropy-compressed bitvector (SDSL class: rrr_vector<63>) and an uncompressed bitvector (bit_vector). We use $m = 5$ for character collections and $m = 1$ for word parsings, and $k = 10$.

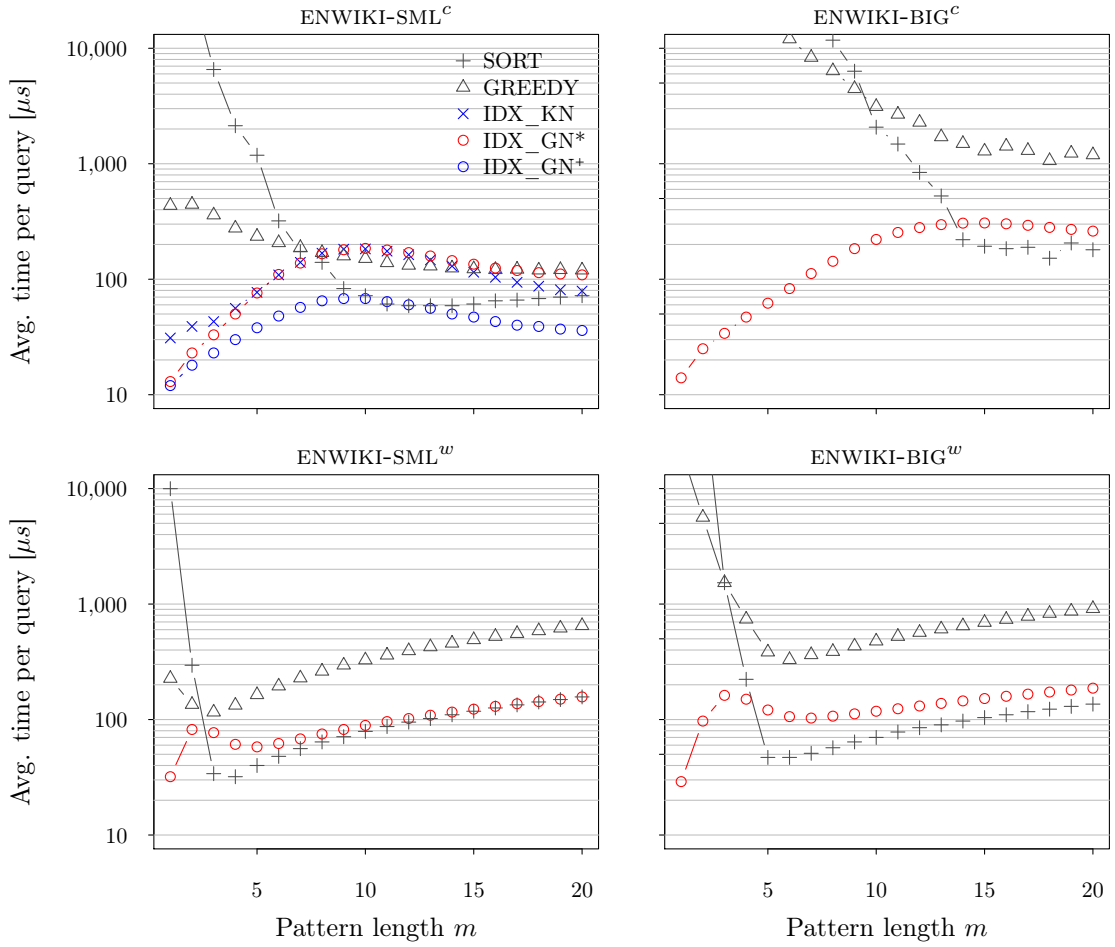

Figure 7: Average top-10 query time depending on collection, index, and pattern length. Compare to [9, Figs. 2–3].

[1] N. Brisaboa, G. de Bernardo, R. Konow, and G. Navarro. k^2-treaps: Range top-k queries in compact space. In *Proc. SEA*, LNCS 8799, pages 215–226, 2014.

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

[3] D. Clark. *Compact Pat Trees*. PhD thesis, Department of Computer Science, University of Waterloo, 1996.

[4] J. S. Culpepper, G. Navarro, S. J. Puglisi, and

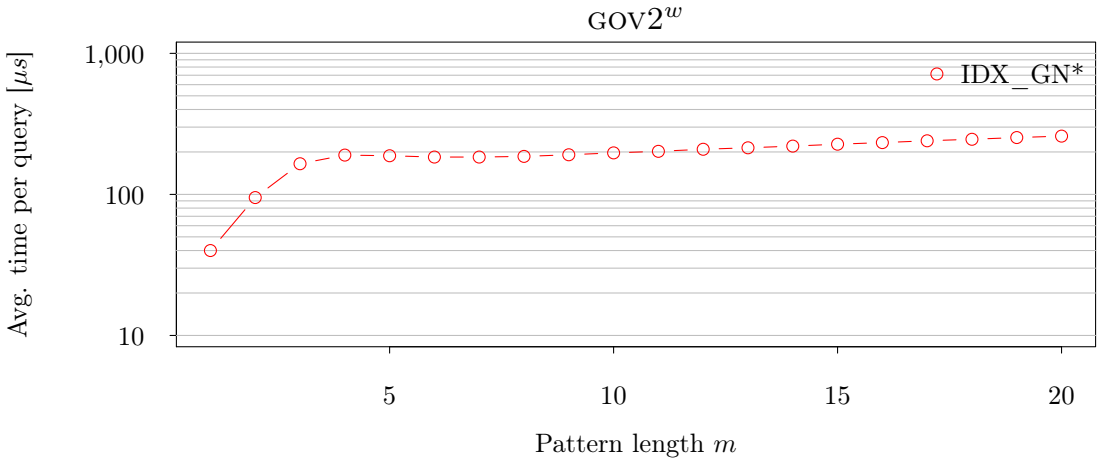

Figure 8: Average top-10 query time on IDX_GN* on GOV2^w with varying pattern length.

A. Turpin. Top-k ranked document search in general text databases. In *Proc. ESA*, pages 194–205, 2010.

[5] J. S. Culpepper, M. Petri, and F. Scholer. Efficient in-memory top-k document retrieval. In *Proc. SIGIR*, pages 225–234, 2012.

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

[7] T. Gagie, G. Navarro, and S.J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sc.*, 426-427:25–41, 2012.

[8] S. Gog. *Compressed Suffix Trees: Design, Construction, and Applications*. PhD thesis, Univ. of Ulm, Germany, 2011.

[9] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337, 2014.

[10] S. Gog and M. Petri. Compact indexes for flexible top-k retrieval. arXiv preprint arXiv:1406.3170.

[11] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.

[12] W.-K. Hon, R. Shah, and S. Thankachan. Towards an optimal space-and-query-time index for top-k document retrieval. In *Proc. CPM*, pages 173–184, 2012.

[13] Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722, 2009.

[14] R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. DCC*, pages 351–360, 2013.

[15] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.

[16] G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comp. Surv.*, 46(4):article 52, 2014.

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

[18] G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. SODA*, pages 1066–1078, 2012.

[19] G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *Proc. SEA*, LNCS 7276, pages 307–319, 2012.

[20] M. Patil, S. V. Thankachan, R. Shah, W.-K. Hon, J. S. Vitter, and S. Chandrasekaran. Inverted indexes for phrases and strings. In *Proc. SIGIR*, pages 555–564, 2011.

[21] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algo.*, 3(4):art. 43, 2007.

[22] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comp. Sys.*, 41(4):589–607, 2007.

[23] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Alg.*, 5(1):12–22, March 2007.

[24] Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Alg.*, 5(1):12–22, 2007.

[25] P. Weiner. Linear pattern matching algorithm. In *Proc. IEEE SWAT*, pages 1–11, 1973.