Top-k Document Retrieval in Compact Space and Near-Optimal Time [⋆]

Gonzalo Navarro[1] and Sharma V. Thankachan[2]

[1] Dept. of Computer Science, University of Chile, Chile, gnavarro@dcc.uchile.cl
[2] Dept. of Computer Science, Louisiana State University, USA, thanks@csc.lsu.edu

Abstract. Let $\mathcal{D}=\{d_1, d_2, ...d_D\}$ be a given set of D string documents of total length n. Our task is to index $\mathcal{D}$ such that the k most relevant documents for an online query pattern P of length p can be retrieved efficiently. There exist linear space data structures of $O(n)$ words for answering such queries in optimal $O(p+k)$ time. In this paper, we describe a compact index of size $|\mathsf{CSA}| + n \lg D + o(n \lg D)$ bits with near optimal time, $O(p+k \lg^* n)$, for the basic relevance metric *term-frequency*, where $|\mathsf{CSA}|$ is the size (in bits) of a compressed full-text index of $\mathcal{D}$, and $\lg^* n$ is the iterated logarithm of n.

1 Introduction and Related Work

Top-k document retrieval is the problem of preprocessing a text collection so that, given a search pattern $P[1..p]$ and a threshold k, we retrieve the k documents most "relevant" to P, for some definition of relevance. This is the basic problem of search engines and forms the core of the Information Retrieval (IR) field [5]. In this paper we focus on the popular *term frequency* as the relevance measure, that is, the number of times P appears in a document.

The inverted index successfully solves top-k queries in various IR scenarios. However, they apply to text collections that can be segmented into "words", so that only whole words can be queried. This excludes many East Asian languages such as Chinese and Korean, where automatic segmenting is an open problem, and is troublesome even in highly synthetic languages such as German and Finnish. A simple solution for those cases is to treat the text as an uninterpreted sequence of symbols and look for any substring in those sequences. This string model is also appealing in other applications like bioinformatics, software repositories, multimedia databases, and so on. Supporting document retrieval queries on those general string collections has proved much more challenging.

Sufix trees [28] and arrays [15] are useful tools to search string collections. These structures solve the *pattern matching problem*, that is, count or list all the occ individual occurrences of P in the collection. Obtaining the k most relevant documents from that set requires time $\Omega(\mathsf{occ})$, usually much higher than k. Only recently [13, 9, 12, 18] was the top-k problem solved satisfactorily, finally reaching the optimal time $O(p+k)$. Those solutions, like suffix trees, have the drawback of

[⋆] Funded in part by Fondecyt Grant 1-110066.

requiring $O(n \lg n)$ bits of space on a collection of length n, whereas the collection itself would require no more than $n \lg \sigma$ bits, where σ is the alphabet size. This renders these indexes impractical on moderate and large text collections.

Compressed Suffix Arrays (CSAs) satisfactorily solve the pattern matching problem within the size of the compressed text collection, under some entropy model [17]. They can in addition retrieve any substring of any document, and hence replace the collection with a compressed version that in addition supports queries. We call their space $|\mathsf{CSA}| \leq n \lg \sigma(1+o(1))$, which can be thought of as the minimum space in which the text collection can be represented.

A similar result for top-k queries has been sought. Various solutions use $2|\mathsf{CSA}|+o(n)$ bits [24, 12, 7, 3], culminating with the fastest solution so far in this family, $O(p+k \lg k \lg^{1+\epsilon} n)$ time by Hon et al. [11]. Recently, asymptotically optimal space $|\mathsf{CSA}|+o(n)$ bits was obtained as well [26], being $O(p+k \lg^2 k \lg^{1+\epsilon} n)$ the best time achieved so far [20].

In all those solutions there is a significant time factor per element returned, of at least $\lg k \lg^{1+\epsilon} n$. It seems unlikely that this factor can disappear in this type of solutions. Experimental comparisons [6, 19] show that these schemes are impractically slow compared to those that use $n \lg D + o(n \lg D)$ bits to store a so-called *document array* [16, 27]. We call *compact* the solutions that use $|\mathsf{CSA}| + n \lg D + o(n \lg D)$ bits. The best practical results to date [6, 21, 3, 14] are nearly compact. Their space requirement, 1–3 times the collection size (and including it), while not optimal, is affordable in many practical situations.

It is therefore relevant to ask which is the best time performance that can be achieved within compact space. The time-optimal result of Navarro and Nekrich [18], $O(p + k)$ time, requires $O(n(\lg D + \lg \sigma))$ bits. While of the same order of compact solutions, the constants are still way too large in practice. There have been some attempts to achieve truly compact solutions. Hon et al. [10] obtained $O(p + (\lg\lg n)^6 + k(\lg \sigma \lg\lg n)^{1+\epsilon})$ time, for any constant $\epsilon > 0$, using compact space. Alternatively, they obtain time $O(p + (\lg\lg n)^4 + k \lg\lg n)$ using $|\mathsf{CSA}| + 2n \lg D + o(n \lg D)$ bits. Konow and Navarro [14] achieved time $O(p + (\lg\lg n)^2 + k \lg\lg n)$ using $|\mathsf{CSA}| + n(\lg D + 4 \lg\lg n)(1+o(1))$ bits, but the result holds only on typical texts, not in the worst case.

In this paper we show that it is possible to get very close to optimal time within compact space. We prove the following result, where we remark that the top-k results are not returned in sorted order of relevance.

Theorem 1 *There exists a compact index of* $|\mathsf{CSA}| + n \lg D + o(n \lg D)$ *bits and near-optimal* $O(p + k \lg^* n)$ *query time time, for the (unsorted) top-k frequent document retrieval problem, where* $\lg^* n$ *is the iterated logarithm of* n.

In Section 5 we show that, with slightly extra space, we can achieve even $O(p + k \lg^* k)$ time.

2 The Data Structure

Three main components of our structure are a generalized suffix tree (GST), the document array, and some precomputed answer lists. These are described next.

Generalized Suffix Tree (GST): Let $T = d_1 d_2 d_3 ... d_D$ be the text (of length n) obtained by concatenating all the documents in $\mathcal{D}$. The last character of each document is \$, a special symbol that does not appear anywhere else in T. Each substring $T[i..n]$, with $i \in [1..n]$, is called a *suffix* of T. The *suffix tree* for T (or, equivalently, the *generalized suffix tree (GST)* of $\mathcal{D}$) is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in the suffix tree is labeled by a string, and $path(x)$ of a node x (node x refers to the node with preorder rank x) is the concatenation of edge labels along the path from the *root* of GST to node x. Let ℓ_i for $i \in [1..n]$ represent the (preorder rank of) the ith leftmost leaf in GST. Then $path(\ell_i)$ represents the ith lexicographically smallest suffix of T. A node x is called the *locus* of a pattern P, if it is the node closest to the *root* with $path(x)$ prefixed by P.

The *suffix array* $SA[1..n]$ is an array of length n, where $SA[i]$ is the starting position (in T) of the ith lexicographically smallest suffix of T. An important property of SA is that the starting positions of all the suffixes with the same prefix are always stored in a contiguous region of SA. Based on this property, we define the *suffix range* of P in SA to be the maximal range $[sp, ep]$ such that for all $i \in [sp, ep]$, $SA[i]$ is the starting point of a suffix of T prefixed by P.

A compressed representation of suffix array is called a Compressed Suffix Array (CSA). We will use a recent CSA [1], which obtains high-order entropy compression and can compute the suffix range $[sp, ep]$ of any given pattern $P[1..p]$ in $O(p)$ time. We also maintain the tree topology of GST in (at most) $4n + o(n)$ bits [25], with constant-time support of the operations $parent(x)$ (the parent of node x), $lca(x, y)$ (the lowest common ancestor of nodes x and y), $left\text{-}leaf(x)/right\text{-}leaf(x)$ (the leftmost/rightmost leaf in the subtree rooted at node x), and $leaf(i)$ (the ith leftmost leaf), and mapping from nodes to their preorder ranks and back. The total space of this component is $|\mathsf{CSA}| + O(n)$ bits.

Document Array (DA): Define a bit-vector $B[1..n]$, such that $B[i] = 1$ iff $T[i] = \$$. Then suffix $T[i, n]$ belongs to document d_r if $r = 1 + rank_B(i)$, where $rank_B(i)$ is the number of 1s in $B[1, i]$. The document array $DA[1..n]$ is defined as $DA[j] = r$ if the suffix $SA[j]$ belongs to document d_r. Moreover, we say that the corresponding leaf node ℓ_j is *marked* with document d_r. Now,

– $rank_{DA}(r, i)$ returns the number of occurrences of r in $DA[1, i]$;
– $select_{DA}(r, j)$ returns i where $DA[i] = r$ and $rank_{DA}(r, i) = j$; and
– $access_{DA}(i)$ returns $DA[i]$;

Then we have use the following representation for DA [2].

Lemma 1 *The document array DA can be stored in* $n \lg D + o(n \lg D)$ *bits and support queries* $rank_{DA}$, $select_{DA}$ *and* $access_{DA}$ *in times* $O(\lg \lg n)$, $O(f(n, D))$ *and* $O(1)$ *respectively, where* $f(n, D) = \omega(1)$ *is any non-constant function.*

The so-called *partial rank* query can be added to this repertoire [3].

Lemma 2 *Operation* $rank_{DA}(DA[i], i)$ *can be supported in constant time by storing* $O(n \lg \lg D) = o(n \lg D)$ *additional bits on top of the DA.*

Thus the total space of this component is $n \lg D + o(n \lg D)$ bits.

Precomputed Answer Lists: We start with the following definitions:

- $L(x)$ is the set of leaves in the subtree of node x in GST.
- $L(x \backslash y) = L(x) \setminus L(y)$, the leaves in the subtree of x, but not in that of y.
- $score(r, x)$ is the number of leaves in $L(x)$ marked with document d_r (i.e., $|\{\ell_i \in L(x), DA[i] = r\}|$).

We use the following scheme to identify a subset S_g of *marked nodes* in GST [12, 21]: Let g be a parameter called *grouping factor*, then mark every gth leftmost leaf in GST, and then mark the lowest common ancestor (LCA) of every consecutive pair of marked leaves. Then, we have the following lemma [12, 21].

Lemma 3 *The above marking scheme ensures the following properties:*

1. *The number of marked nodes is $|S_g| = \Theta(n/g)$.*
2. *If it exists, the closest marked descendant node y of any unmarked node x is unique, and $|L(x \backslash y)| < 2g$.*
3. *If there exists no marked node in the subtree of x, then $|L(x)| < 2g$.*

Let $F(x, k)$ represent the list (or set) of top-k documents d_r, along with $score(r, x)$, corresponding to a pattern with locus node x in GST. Clearly we cannot afford to maintain $F(x, k)$ for all possible x's and k's. Rather, we will maintain the lists $F(x, z)$ only for marked nodes x's (for various g values) and for k's that are powers of 2. Then $F(x, k)$ for any x and k will be efficiently computed using that sampled data. The next section describes how we store and retrieve the sampled lists.

3 Storing and Retrieving the Lists $F(x, z)$

The following is a key result in our scheme.

Lemma 4 *Let $g_h = z(\lg^{(h)} n)^2$ for any $1 \leq h < \lg^* n$, where $\lg^{(1)} n = \lg n$, $\lg^{(h)} n = \lg(\lg^{(h-1)} n)$, and $\lg^{(\lg^* n)} n \leq 1$. Then $F(x, z)$ for all $x \in S_{g_h}$ can be encoded in $s_h = s_{h-1} + O(n/\lg^{(h)} n)$ bits, and $F(x, z)$ for any given $x \in S_{g_h}$ can be decoded in time $t_h = t_{h-1} + O(z)$, where $s_1 = O(n/\lg n)$ and $t_1 = O(z)$.*

Proof. We use induction. Consider the base case $h = 1$. For every $x \in S_{g_1}$, we maintain the list $F(x, z)$ explicitly (using $O(\lg n)$ bits per element), along with a pointer to the location where it is stored, in $s_1 = O(|S_{g_1}| z \lg n) = O(n/\lg n)$ bits. Thus the list $F(x, z)$, for any $x \in S_{g_1}$, can be decoded in time $t_1 = O(z)$.

Now consider the grouping factor is g_h for $h \geq 2$. As we cannot afford to use $\Theta(\lg n)$ bits per element, we introduce encoding schemes that reduce it to $O(\lg^{(h)} n)$ bits. Thus the overall space for maintaining $F(x, z)$ (in encoded form) for all $x \in S_{g_h}$ can be bounded by $O(|S_{g_h}| z \lg^{(h)} n) = O(n/\lg^{(h)} n)$ bits. Instead of using pointers as in the base case, we mark in a bitmap $B^h[1..2n]$ the node preorders of GST that belong to S_{g_h}. Therefore the list $F(x,z)$ of a node $x \in S_{g_h}$ is stored in an array at offset $rank_{B^h}[x]$. Since we will only compute *rank* on positions x where $B^h[x]=1$, an "indexed dictionary" is sufficient [23], which requires $O((n/g_h)\lg g_h + \lg\lg n) = o(n/\lg^{(h)} n)$ bits and computes *rank* in time $O(1)$. We now show how to encode the list $F(x,z)$, for $x \in S_{g_h}$, in $O(\lg^{(h)} n)$ bits per element, and how to decode it in $t_{h-1}+O(z)$ time.

We will maintain a structure STR_h, using s_h bits, for each grouping factor g_h, and will decode $F(x,z)$ for $x \in S_{g_h}$ recursively, using $O(z)$ time in addition to the time needed to decode $F(y,z)$ for some $y \in S_{g_{h-1}}$, as suggested in Lemma 4. As we cannot afford to sort the documents within the targeted query time, it is important to assume a fixed arrangement of documents within any particular decoded list $F(\cdot,\cdot)$. That is, each time we decode a specific list, the decoding algorithm must return the elements in the same order.

Let x be a node in S_{g_h} and y (if it exists) be its highest descendant node in $S_{g_{h-1}}$. We show how to encode and decode $F(x,z)$. To decode $F(x,z)$, we first decode the list $F(y,z)$ using STR_{h-1} in time t_{h-1}. From now onwards we have constant-time access to any element the list $F(y,z)$. The the list $F(x,z)$ will be partitioned into the following two disjoint lists:

(i) D_{old}, the documents that are common to $F(x,z)$ and $F(y,z)$.

(ii) D_{new}, the documents that are present in $F(x,z)$, but not in $F(y,z)$.

Encoding and decoding document identifers in D_{old}. We maintain a bit vector $B'[1..z]$, where $B'[i]=1$ iff the ith document in $F(y,z)$ is present in $F(x,z)$. Therefore D_{old} can be decoded by listing those elements in $F(y,z)$ (in the same order as they appear) at positions i where $B'[i]=1$. Thus space for maintaining the encoded information is z bits and the time for decoding is $O(z)$.

Encoding and decoding document identifers in D_{new}. For each document $d_r \in D_{new}$, there exists at least one leaf in $L(x\backslash y)$ that is marked with d_r (otherwise $score(r,x) = score(r,y)$ and d_r could not be in $F(x,z)$ and not in $F(y,z)$). Therefore, instead of explicitly storing r, it is sufficient to refer to such a leaf. For this we shall store a bit vector $B''[1..|L(x\backslash y)|]$ with all its bits in 0, except for $|D_{new}|$ 1's: for every document $d_r \in D_{new}$, we set one bit, say $B''[i]=1$, where the ith leaf in $L(x\backslash y)$ is marked with d_r. Since $|B''| = |L(x\backslash y)| < 2g_{h-1}$ and the number of 1's is at most z, B'' can be encoded in $O(z\lg(g_{h-1}/z)) = O(z\lg^{(h)} n)$ bits with constant time *select* support [22] ($select_{B''}(j)$ is the position of the j-th 1 in B''). Now, given B'', the documents in D_{new} can be identified in $O(z)$ time as follows: Find all those (at most z) increasing positions i where $B''[i]=1$ using *select* queries. Then, for each such i, find the ith leaf $\ell_{i'} \in L(x\backslash y)$ in constant time using the tree operations[3]. Finally, report $d_{DA[i']}$ as a document in D_{new} for each such i' using a constant-time *access* operation on the document array.

As mentioned before, it is important for our (recursive) encoding/decoding algorithm to assume a fixed permutation of elements within any list $F(\cdot,\cdot)$. We

[3] Compute the leftmost leaves ℓ_{i_x} and ℓ_{i_y}, respectively, of x and y, then $\ell_{i'}$ is ℓ_{i_x+i-1}, if $i_x+i-1 < i_y$, and $\ell_{j_y+i-(i_y-i_x)}$ otherwise, where ℓ_{j_y} is the rightmost leaf of y.

use the convention that, in $F(x,z)$, the documents in D_{old} come before the documents in D_{new}. Moreover the documents within D_{old} and D_{new} are in the same order as the decoding algorithm identified them. In conclusion, the list of identifiers of documents in $F(x,z)$ can be encoded in $O(z\lg^{(h)} n)$ bits and decoded in $O(z)$ time, assuming constant-time access to any element in $F(y,z)$. If node y does not exist, we proceed as if $F(y,z) = \emptyset$ and $F(x,z) = D_{new}$. We now consider how to encode the *score*'s associated with the elements in $F(x,z)$ (i.e., $\mathit{score}(r,x)$ for all $d_r \in F(x,z)$).

Encoding and decoding of scores. Let d_{r_i}, for $i \in [1..z]$, be the ith document in $F(x,z)$, and $f_i = \mathit{score}(r_i,x)$. Then, define $\delta_i = f_i - f'_i \geq 0$, where

$$f'_i = \begin{cases} \mathit{score}(r_i,y) & \text{if } i \leq |D_{old}| \text{ (i.e., if } r_i \in D_{old}), \\ \tau = \min\{\mathit{score}(r,y), r \in F(y,z)\} & \text{if } i > |D_{old}| \text{ (i.e., if } r_i \in D_{new}). \end{cases}$$

The following is an important observation: The number of leaves in $L(x\backslash y)$ marked with document d_{r_i} is $\mathit{score}(r_i,x) - \mathit{score}(r_i,y)$, which is same as δ_i for $i \leq |D_{old}|$. For $i > |D_{old}|$, $\mathit{score}(r_i,x) - \mathit{score}(r_i,y) \geq \delta_i$, otherwise $\mathit{score}(r_i,y) > \tau$ and d_{r_i} would have qualified as a top-z document in $F(y,z)$ (which is a contradiction as $d_{r_i} \in D_{new}$). By combining with the fact that each leaf node is marked with a unique document, we have the inequality $\sum_{i=1}^{z} \delta_i \leq |L(x\backslash y)| < 2g_{h-1}$. Therefore, δ_i for all $i \in [1..z]$ can be encoded using a bit vector $B''' = 10^{\delta_1}10^{\delta_2}10^{\delta_3}\ldots10^{\delta_z}$ of length at most $2g_{h-1}+z$ with z 1's, in $O(z\lg(g_{h-1}/z)) = O(z\lg^{(h)} n)$ bits with constant-time *select* support [22].

The decoding algorithm is described as follows: compute the f'_i's for $i = 1\ldots z$ in the ascending order of i. For $i \leq |D_{old}|$, f'_i is given by score associated with the $(\mathit{select}_{B'}[i])$th document (which is same as d_{r_i}) in $F(y,z)$. This takes only $O(z)$ time as the number of constant-time *select* operations is $O(z)$, and we have constant-time access to any element and score in $F(y,z)$. Next, $\tau = \min\{\mathit{score}(r,y), r \in F(y,z)\}$ can be obtained by scanning the list $F(y,z)$ once. Thus all the f'_i's are computed in $O(z)$ time. Next we decode each δ_i and add it to f'_i to obtain f_i, for $i = 1\ldots z$ in $O(z)$ time, where $\delta_i = \mathit{select}_{B'''}(i) - \mathit{select}_{B'''}(i-1) - 1$ is computed in $O(1)$ time. Thus the space for maintaining the scores is $O(z\lg^{(h)} n)$ bits and the time for decoding them is $O(z)$.

Adding over the h levels, the total space is $s_h = s_{h-1} + O(n/\lg^{(h)} n) = O(n/\lg^{(h)} n)$ bits and the total decoding time is $t_h = t_{h-1} + O(z) = O(zh)$ (note that $s_1 = O(n/\lg n)$ and $t_1 = O(z)$). This completes the proof. □

4 Completing the Picture

Let $\pi \in [1..\lg^* n)$ be an integer such that $\lg^{(\pi-1)} n \geq \sqrt{\lg^* n} > \lg^{(\pi)} n$, then $\lg^{(\pi)} n = \omega(1)$ (note that $\pi = \lg^* n - \lg^* \sqrt{\lg^* n} = \Theta(\lg^* n)$). Then, by choosing g_π as the grouping factor, the space s_π is $O(n/\lg^{(\pi)} n) = o(n)$ bits. We maintain $\lg D$ such structures corresponding to $z = 1, 2, 4, 8, ..., 2^{\lfloor \lg D \rfloor}$, in $o(n \lg D)$ bits total space. By combining the space bounds of all the components, we obtain the following lemma.

Lemma 5 *The total space requirement of our data structure is* $|\mathsf{CSA}| + n \lg D + o(n \lg D)$ *bits.*

The next lemma gives the total time to extract the sampled results and hints how we will use them.

Lemma 6 *Given any node* q *in* GST *and an integer* k*, our data structure can report the list* $F(q', k)$ *in* $O(k \lg^* n)$ *time, where* q' *is a node in the subtree of* q *with* $|L(q \backslash q')| = O(k\sqrt{\lg^* n})$*.*

Proof. As the first step, round k to $z = 2^{\lceil \lg k \rceil}$, which is the next highest power of 2. Then identify the highest node q', in the subtree of q, that is marked with respect to the grouping factor g_π: Let ℓ_i and ℓ_j be the leftmost and rightmost leaves of q in GST, then $q' = lca(\ell_{i'}, \ell_{j'})$ where $i' = g_\pi \cdot \lceil i/g_\pi \rceil$ and $j' = g_\pi \cdot \lfloor j/g_\pi \rfloor$ (there is no q' if $i' \geq j'$). This takes constant time on our representation of the GST topology.

Since $g_\pi = z \lg^{(\pi)} n < z\sqrt{\lg^* n}$, from Lemma 3 it holds $|L(q \backslash q')| = O(g_\pi) = O(z \lg^{(\pi)} n) = O(k\sqrt{\lg^* n})$. As $q' \in S_{g_\pi}$, the list $F(q', z)$ can be decoded in time $t_\pi = O(z\pi) = O(z \lg^* n)$ from the precomputed lists (from Lemma 4). The final $F(q', k)$ can be obtained by filtering those documents in $F(q', z)$ with score at least θ by a single scan of the list, where θ is the kth highest score in $F(q', z)$ (which can be computed in $O(z) = O(k)$ time using the linear-time selection algorithm [4]). In case q' does not exist, we report $F(q', k) = \emptyset$, and even in such a case the inequality condition $|L(q)| < 2g_\pi$ is guaranteed (from Lemma 3). □

4.1 Query Answering

The query answering algorithm consists of the following steps:

1. Find the locus node q of the input pattern P in GST by first obtaining the suffix range $[sp, ep]$ of P using CSA in $O(p)$ time, and then computing the lowest common ancestor of ℓ_{sp} and ℓ_{ep} in $O(1)$ time.
2. Using Lemma 6, find the node q' in the subtree of q, where $|L(q \backslash q')| = O(k\sqrt{\lg^* n})$ and retrieve the list $F(q', k)$ in $O(k \lg^* n)$ time.
3. Every document d_r in the final output $F(q, k)$ must either belong to $F(q', k)$, or it must be that $r = DA[i]$ for some leaf $\ell_i \in L(q \backslash q')$. Let us call S_{cand} the union of both sets of candidate documents. Then we compute $score(r, q)$ of each document $d_r \in S_{cand}$.
4. Report k documents in S_{cand} with the highest $score(r, q)$ value. In this step, we first compute the kth highest score θ using the selection algorithm, and then use θ as a threshold for a document to be an output (more precisely, we report the $k' < k$ documents $d_r \in S_{cand}$ with $score(r, q) < \theta$ in a first pass, and then report the first $k - k'$ documents $d_r \in S_{cand}$ we find with $score(r, q) = \theta$ in a second pass). The time is $O(|S_{cand}|) = O(k\sqrt{\lg^* n})$.

The overall time for Steps 1, 2, and 4 is $O(p + k \lg^* n)$. In the remaining part of this section we show how to handle Step 3 efficiently as well, for the documents $r = DA[i]$ we find in $L(q \backslash q')$. Note that $score(r, q)$ can be computed as $rank_{DA}(r, ep) - rank_{DA}(r, sp - 1)$ using two *rank* queries on the document array, but those *rank* queries are expensive. Instead, we use a more sophisticated scheme where only the faster *select*, *access*, and partial *rank* queries are used. This is described next.

4.2 Computing Scores Online

Firstly, we construct a supporting structure, SUP, in $O(k \lg^* n)$ time and occupying $o(n \lg D) + O(z \lg n)$ bits, capable of answering the following query in $O(\lg \lg^* n)$ time: for any given r, return $score(r, q')$ if $r \in F(q', k)$, otherwise return -1. Let $\Delta = \Theta(\lg^* n)$, then structure SUP is a forest of D/Δ balanced binary search trees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{D/\Delta}$. Initially each $\mathcal{T}_i$ is empty, hence the initial space is $O(\lg n)$ bits per tree (for maintaining a pointer to the location where it is stored), adding up to $O((D/\Delta) \lg n) = o(n \lg D)$ bits, which we consider a part of index. Next we shall insert each document $d_r \in F(q', k)$, along with its associated score, into tree $\mathcal{T}_{\lceil r/\Delta \rceil}$ of SUP. The size of each search tree can grow up to Δ, hence the total insertion time is $O(k \lg \Delta)$. These insertions will increase the space of SUP by $O(k \lg n)$ bits, which can be justified as it is the size of the output. Now we can search for any d_r in $\mathcal{T}_{r/\Delta}$ and, if $d_r \in F(q', k)$, we will retrieve $score(r, q')$ in $O(\lg \Delta)$ time. Once we finish Step 3, these binary search trees can be set back to their initial empty state by visiting each document $d_r \in F(q', k)$ and deleting it from the corresponding tree in total $O(k \lg \Delta)$ time. This does not impact the total asymptotic query processing time.

An outline of Step 3 follows: We scan each leaf $\ell_i \in L(q \backslash q')$, and compute $score(\cdot, q)$ of the corresponding document $d_{DA[i]}$. Note that there can be many leaves in $L(q \backslash q')$ marked with the same document, but we compute $score(\cdot, q)$ of a document only once (i.e., when we encounter it for the first time). After this, we also scan the documents $d_r \in F(q', k)$ and compute $score(r, q)$ if we have not considered this document in the previous step. However, the scanning of leaves is performed in a carefully chosen order. Let $\ell_{sp'}$ and $\ell_{ep'}$ be the leftmost and rightmost leaves in the subtree of q', and $B[1..D]$ be a bit vector initialized to all 0's (its size is D bits and can be considered a part of index). A detailed description of Step 3 follows:

3.1 Start scanning the leaves ℓ_i for $i = sp, sp + 1, \ldots, sp' - 1$, in the *ascending* order of i, then for $i = ep, ep - 1, \ldots, ep' + 1$, in the *descending* order of i, and do the following: if $B[DA[i]] = 0$, then set it to 1, compute $score(DA[i], q)$, and store the result $(DA[i], score(DA[i], q))$ for Step 4. Note that each time we compute $score(DA[i], q)$, i is either the first or the last occurrence of $DA[i]$ in $DA[sp, ep]$. Assume it is the first (the other case is symmetric). We use a constant-time partial *rank* query, $x = rank_{DA}(DA[i], i)$. Then, by performing successive $select_{DA}(DA[i], j)$ queries for $j = x + 1, x + 2, \ldots, y$, where $select_{DA}(DA[i], y) > ep \geq select_{DA}(DA[i], y - 1)$, we compute

$score(DA[i], q) = y - x$. The number of *select* queries required is precisely $y - x = score(DA[i], q)$, which can be further reduced as follows:

- If $d_{DA[i]} \in F(q', k)$, retrieve $score(DA[i], q')$ from SUP in time $O(\lg\lg^* n)$. As we know that $score(DA[i], q') \leq score(DA[i], q)$, we start *select* queries from $j = x + score(DA[i], q')$, so the number of *select* queries used to find y is reduced to $score(DA[i], q) - score(DA[i], q') = score(DA[i], L(q \backslash q'))$, that is, the number of leaves in $L(q \backslash q')$ marked with $d_{DA[i]}$.
- If $d_{DA[i]} \notin F(q', k)$, compute $x' = select_{DA}(DA[i], x + \tau - 1)$, where we remind that $\tau = \min\{score(r, q'), r \in F(q', k)\}$. If $x' > ep$, we conclude that $score(DA[i], q) < \tau$, and hence $d_{DA[i]}$ can be discarded from being a candidate for the final output. On the other hand, if $x' \leq ep$, the *select* queries can be started from $j = x + \tau$, which reduces the number of *select* queries to $score(DA[i], q) - \tau \leq score(DA[i], L(q \backslash q'))$ (since $d_{DA[i]} \notin F(q', k)$, it holds $score(DA[i], q') \leq \tau$).

The query time for executing this step can be analyzed as follows: for each i, we perform a query on SUP. The computation of $score(DA[i], q)$ requires at most $score(DA[i], L(q \backslash q'))$ *select* queries. As we do this computation only once per distinct document, the total number of *select* queries is at most $\sum_r score(r, L(q \backslash q')) = |L(q \backslash q')|$. By choosing the cost $f(n, D) = \sqrt{\lg^* n}$ for *select* queries, the total time is $O(|L(q \backslash q')|(f(n, D) + \lg\lg^* n)) = O(k \lg^* n)$.

3.2 Now scan the documents $d_r \in F(q', k)$. If $B[r] = 0$, then there exists no leaf in $L(q \backslash q')$ marked with d_r. Thus $score(r, q) = score(r, q')$ and the pair $(r, score(r, q'))$ is stored for Step 4. If $B[r] = 1$ then d_r has already been dealt with in the previous pass. The time for accessing $score(r, q')$ using SUP is $O(\lg\lg^* n)$, hence this step takes $O(k \lg\lg^* n)$ time.

3.3 Reset B to its initial state (all bits set to 0) for supporting queries in future. By revisiting the leaves in $L(q \backslash q')$ and the list $F(q', k)$, we can exactly find out those locations in B where the corresponding bit is 1. The time for this step can be bounded by $O(|L(q \backslash q')| + k) = O(k\sqrt{\lg^* n})$.

Thus the time for Step 3 is $O(k \lg^* n)$, and the result follows.

5 Reducing the Time to $O(p + k \lg^* k)$

Note that, when p or k is at least $\lg\lg n$, it already holds $O(p + k \lg^* n) = O(p + k \lg^* k)$. Therefore, we now concentrate on the case when $\max(p, k) < \lg\lg n$. We use the following result [8].

Lemma 7 *Given a fixed κ, an array $A[1..n]$ of n indices can be indexed in $O(n \lg^2 \kappa)$ bits for answering the following query in $O(k)$ time, without accessing A and for any $1 \leq k \leq \kappa$: given i, j, and k, output the positions of the k highest elements in $A[i, j]$.*

Let S_δ be the set of nodes in GST with node depth equal to δ. We start with the description of an $O(n \lg^2 \kappa)$-bit structure for a fixed $\kappa = \lg\lg n$ and a fixed

$\delta < \lg\lg n$, for answering top-k queries for any $1 \leq k \leq \kappa$ and those patterns with their locus node belonging to S_δ. First, we construct an array $A[1..n]$ (with all its elements initialized to zero) as follows: For $i = 1 \ldots n$, if the first occurrence of document $DA[i]$ in $DA[a,b]$ is at position i, where $[a,b]$ is the suffix range corresponding to a unique node $u \in S_\delta$, then set $A[i] = \mathit{score}(DA[i],u)$. We do not store this array explicitly, instead we maintain the structure of Lemma 7 over it, requiring $O(n \lg^2 \kappa)$ bits space. Now the list of documents $F(u,k)$ for any locus node $u \in S_\delta$ can be reported in $O(k)$ time as follows: First perform a top-k query on the structure of Lemma 7 with the suffix range $[sp, ep]$. The output will be a set of k locations $j_1, j_2, \ldots, j_k \in [sp, ep]$, and then the identifiers of the top-k documents are $DA[j_1], DA[j_2], \ldots, DA[j_k]$. By maintaining similar structures for all the $\delta \in [1, \lg\lg n)$, any such top-k query with $p < \lg\lg n$ can be answered in $O(p+k)$ time. The additional space required is $o(n(\lg\lg n)^3)$ bits, which can be bounded by $o(n \lg \sigma)$ bits if, say, $\lg \sigma \geq \sqrt{\lg n}$. Otherwise, we shall explicitly maintain the top-κ documents corresponding to all patterns of length at most $\lg\lg n$, in decreasing frequency order, using a table of $O(\sigma^{\lg\lg n} \lg\lg n \lg D) = o(n)$ bits. The query time in this case is just $O(k)$.

Thus, by combining the cases, we achieve $O(p + k \lg^* k)$ query time and Theorem 2 follows.

Theorem 2 *There exists a compact index of* $|\mathsf{CSA}| + n \lg D + o(n(\lg \sigma + \lg D))$ *bits and near-optimal* $O(p + k \lg^* k)$ *query time time, for the (unsorted) top-*k *frequent document retrieval problem.*

6 Conclusions

We have shown that it is possible to obtain almost optimal time for top-k document retrieval, $O(p+k \lg^* n)$, using compact space, $|\mathsf{CSA}|+n \lg D+o(n \lg D)$ bits. By adding $o(n \lg \sigma)$ bits, the time decreases to $O(p+k \lg^* k)$. This is an important step towards answering the question of which is the minimum space that is necessary to obtain the optimal time, $O(p+k)$. The other important open question is which is the minimum time that can be obtained by using the asymptotically optimal space, $|\mathsf{CSA}| + o(n)$ bits. Right now this time is $O(p + k \lg^2 k \lg^{1+\epsilon} n)$ [20], and it is not clear which is the lower bound.

References

1. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. 19th ESA*, pages 748–759, 2011.
2. D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. 20th ESA*, LNCS 7501, pages 181–192, 2012.
3. D. Belazzougui, G. Navarro, and D. Valenzuela. Improved compressed indexes for full-text document retrieval. *J. Discr. Alg.*, 18:3–13, 2013.
4. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comp. Sys. Sci.*, 7(4):448–461, 1973.

5. S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, 2010.
6. S. Culpepper, G. Navarro, S. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. 18th ESA*, LNCS 6347, pages 194–205 (part II), 2010.
7. T. Gagie, J. Kärkkäinen, G. Navarro, and S. Puglisi. Colored range queries and document retrieval. *Theo. Comp. Sci.*, 483:36–50, 2013.
8. R. Grossi, J. Iacono, G. Navarro, R. Raman, and S. S. Rao. Encodings for range selection and top-k queries. In *Proc. 21st ESA*, LNCS 8125, pages 553–564, 2013.
9. W.-K. Hon, M. Patil, R. Shah, and S.-B. Wu. Efficient index for retrieving top-k most frequent documents. *J. Discr. Alg.*, 8(4):402–417, 2010.
10. W.-K. Hon, R. Shah, S. Thankachan, and J. Vitter. Document listing for queries with excluded pattern. In *Proc. 23rd CPM*, LNCS 7354, pages 185–195, 2012.
11. W.-K. Hon, R. Shah, S. Thankachan, and J. Vitter. Faster compressed top-k document retrieval. In *Proc. 23rd DCC*, pages 341–350, 2013.
12. W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th FOCS*, pages 713–722, 2009.
13. W.-K. Hon, R. Shah, and S.-B. Wu. Efficient index for retrieving top-k most frequent documents. In *Proc. 16th SPIRE*, pages 182–193, 2009.
14. R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. 23rd 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. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc 13th SODA*, pages 657–666, 2002.
17. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):art. 2, 2007.
18. G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd SODA*, pages 1066–1078, 2012.
19. G. Navarro, S. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Proc. 10th SEA*, LNCS 6630, pages 193–205, 2011.
20. G. Navarro and S. Thankachan. Faster top-k document retrieval in optimal space. In *Proc. 20th SPIRE*, LNCS 8214, pages 255–262, 2013.
21. G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *Proc. 11th SEA*, pages 307–319, 2012.
22. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th ALENEX*, 2007.
23. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Alg.*, 3(4):art. 43, 2007.
24. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discr. Alg.*, 5:12–22, 2007.
25. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st SODA*, pages 134–149, 2010.
26. D. Tsur. Top-k document retrieval in optimal space. *Inf. Proc. Lett.*, 113(12):440–443, 2013.
27. N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th CPM*, LNCS 4580, pages 205–215, 2007.
28. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.