Top-k Document Retrieval in Compressed Space

Gonzalo Navarro[*] Yakov Nekrich[†]

Abstract

Let $\mathcal{D}$ be a collection of D strings of total length n over an alphabet of size σ. We consider the so-called top-k document retrieval problem: given a short string P and an integer k, list the identifiers of k strings in $\mathcal{D}$ most relevant to P, in decreasing order of relevance. Relevance may be a fixed value associated with the strings where P occurs, or the number of times P occurs in the strings. While RAM-optimal solutions using $O(n \log n)$ bits and $O(|P|/\log_\sigma n + k)$ time exist, solving the problem optimally within space close to $O(n \log \sigma)$ bits is open.

We describe a data structure for the top-k document retrieval problem that uses $O(\log \log n)$ bits per symbol on top of any compressed suffix array (CSA) of $\mathcal{D}$, and supports queries in essentially optimal time, in the following sense. Given a CSA using $|\mathrm{CSA}|$ bits of space, that finds the suffix array range of a query string P in time t_{cnt}, and accesses a suffix array entry in time t_{SA}, listing any k pattern occurrences would take time $O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}})$. Our top-$k$ data structure uses $|\mathrm{CSA}| + O(n \log \log n)$ bits and reports k most relevant documents that contain P in time $O(t_{\mathrm{cnt}} + k\,(t_{\mathrm{SA}} + \log \log n))$. On every known CSA using $O(n \log \sigma)$ bits, t_{SA} is $\Omega(\log \log n)$ in virtually all cases, thus our time is $O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}})$ in most situations.

When the query string P is sufficiently long, some CSAs reach time $O(t_{\mathrm{cnt}} + k)$ to list any k occurrences of P. Our structure achieves similar performance in this case, obtaining time $O(t_{\mathrm{cnt}} + t_{\mathrm{sort}}(k, n))$ on every known CSA, where $t_{\mathrm{sort}}(k, n)$ is the time to sort k integers in $[1, n]$. This time is already $O(t_{\mathrm{cnt}} + k)$ in the typical regimes, $k = O(\mathrm{polylog}\, n)$ and $k = \Omega(n^{\varepsilon})$ for any constant $\varepsilon > 0$. If we can deliver the results in unsorted order of relevance, then the time for long patterns is always $O(t_{\mathrm{cnt}} + k)$, which is optimal with respect to the CSA, and reaches the RAM-optimal time $O(|P|/\log_\sigma n + k)$ on a particular CSA. No top-k solution using $o(n \log D)$ bits of space has achieved this before.

1 Introduction

The *top-k document retrieval problem* consists of retrieving the k documents that are "most relevant" to a given query. The version of the problem where both the documents and the query are strings are a natural extension of the original natural-language-based information retrieval problem, and has received considerable attention from the community, as it arises in various applications [29]. The fact that any substring of the string collection can be queried makes the problem fundamentally distinct from the one addressed in natural-language information retrieval. The problem is also related, but again fundamentally different, from the classic pattern matching where one seeks to count or retrieve all the occurrences of the query pattern.

Let $\mathcal{D}$ be a text collection formed by a set of D strings—the documents—, over an alphabet of size σ, and let n be the sum of their lengths. We want to preprocess $\mathcal{D}$ so that, later, given a (comparatively short) query string $P[1..p]$ and a threshold k, we want to obtain the identifiers of k documents in $\mathcal{D}$ that are *most relevant* to P, among those where P appears (if P appears in fewer than k documents, we just report them all), in decreasing order of relevance. In this paper we consider two measures of relevance:

- Document rank, where each document d has a fixed relevance $\mathit{docrank}(d)$ independent of P.
- Text frequency, that is, the number $\mathit{freq}(P, d)$ of times P appears in document d.

We will refer to both measures generically as $\mathit{rel}(P, d)$. Those can be considered to be the most basic relevance measures, for example the well-known PageRank measure is a form of document rank, whereas the text frequency is the basis of the famous tf-idf relevance model in information retrieval [29]. Other more sophisticated measures will be discussed in the conclusions.

[*]Department of Computer Science, CeBiB and IMFD, University of Chile. Email `gnavarro@uchile.cl`. Supported by Basal Funds FB0001, Millennium Science Initiative Program – Code ICN17_002, and Fondecyt Grant 1-230755, ANID, Chile.

[†]Michigan Technological University. Email `yakov@mtu.edu`. Supported by the National Science Foundation under NSF grant 2203278.

Both problems can be solved in optimal time, $O(p+k)$ and even $O(p/\log_\sigma p + k)$, using $O(n \log n)$ bits of space [21, 23, 31, 18, 32], for those and many other measures of relevance. Using that space is not so satisfactory, however, if one considers that $\mathcal{D}$ can be represented in just $n \log \sigma$ bits (our logarithms are base 2) and the huge sizes of modern string collections. Given that pattern matching was successfully solved within much less space, that is, the entropy of $\mathcal{D}$ (which is at most $n \log \sigma$) plus some small redundancy [30], it was natural to look for solutions to the top-k problem that used less space by resorting to the same tools. Most compressed-space pattern matching solutions replace suffix trees and arrays by *compressed suffix arrays (CSAs)*, which use space close to the text entropy (at most $|\mathrm{CSA}| \in O(n \log \sigma)$ bits) and provide two operations [30]:

- Count, which in time $t_{\mathrm{cnt}} = t_{\mathrm{cnt}}(p)$ retrieves the lexicographical range of all the suffixes starting with P. This time is typically in $O(p \log n)$ or $O(p \log \sigma)$, but also $O(p)$ and even $O(p/\log_\sigma n + \log_\sigma^\varepsilon n)$ within $O(n \log \sigma)$ bits of space, for any constant $\varepsilon > 0$ [2, 15].

- Access, which in time $t_{\mathrm{SA}} = t_{\mathrm{SA}}(n)$ retrieves the starting text position of the ith lexicographically smallest suffix, given i. This is typically $O(\log n)$ if $O(n)$ extra bits of space are allowed, but can also be $O(\log_\sigma^\varepsilon n)$, and even $O(\log \log_\sigma n)$ if we exceed the $O(n \log \sigma)$ bits [14].

Top-k with document rank. The document rank measure is simpler to handle because it is fixed at indexing time. Karpinski and Nekrich [23] solved it in $\mathrm{O}(p+k)$ time and $O(n \log n)$ bits.

They also gave the first solution aiming at using *compact* space, that is, the entropy of $\mathcal{D}$ plus a redundancy in $O(n \log D)$ bits. Their compact solution builds on a CSA and uses $|\mathrm{CSA}| + O(n \log D)$ bits, reporting the k most important documents where P appears in time $O(t_{\mathrm{cnt}} + k)$. The space was slightly tightened to $|\mathrm{CSA}| + n \log D + o(n \log D)$ bits by Gagie et al. [13], at the price of increasing the time to $O(t_{\mathrm{cnt}} + k \log(D/k))$. We note that, unless there are just a few documents, the space $n \log D$ can be very significant, typically much larger than $n \log \sigma$ and close to the $n \log n$ bits needed by a plain suffix array. This space comes from the use of a *document array*, a variant of the suffix array that records only the documents where the suffixes occur.

More ambitious than compact solutions are *compressed* ones, which aim at using $O(|\mathrm{CSA}| + n)$ bits, avoiding large terms of the form $O(n \log D)$. Belazzougui et al. [3] showed how to use $|\mathrm{CSA}| + O(n)$ bits of space and solve the problem in time $O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}} \log k \log^\varepsilon n)$.

Top-k with text frequency. The top-k problem using text frequency has received much more attention. The foundational work of Hon et al. [21, 18] showed that this problem could be solved in time $O(p + k \log k)$ and $O(n \log n)$ bits of space, not only for $\mathit{freq}(P, d)$ but for any relevance measure that depended on the document and the suffix tree locus of P. Navarro and Nekrich [31] recasted their framework into a geometric problem, achieving the optimal time $O(p+k)$ and even the RAM-optimal time $O(p/\log_\sigma n + k)$ if P is given packed in $O(p/\log_\sigma n)$ computer words of $\Theta(\log n)$ bits [32] (Hon et al. [18] also obtained $O(p+k)$ time, though delivering the results in unsorted order). They also offered compact versions achieving the same optimal times plus $O(\log_\sigma^\varepsilon n)$ for any constant $\varepsilon > 0$, using $O(n \log \sigma + n \log D)$ bits for $\mathit{freq}(P, d)$, and $O(n \log \sigma + n \log D + n \log \log n)$ bits for the more general measures supported by Hon et al. After some developments [19], the best compact result is by Navarro and Thankachan [35], which uses $|\mathrm{CSA}| + n \log D + o(n \log D + n \log \sigma)$ bits and answers in near-optimal time $O(t_{\mathrm{cnt}} + k \log^* k)$.

In their seminal paper [21, 18], Hon et al. also provided the first compressed solution for the problem using text frequency (not a more general measure). Their techniques, based on storing the query answers for selected suffix tree nodes, also pioneered the subsequent work on compressed solutions. Using $2|\mathrm{CSA}| + O(n)$ bits of space (note the factor 2), they solve the problem in time $O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}} \log^{3+\varepsilon} n)$ for any constant $\varepsilon > 0$. Their result triggered significant activity on the compressed version of the problem [12, 3, 40, 20], culminating with Navarro and Thankachan [35], who using $|\mathrm{CSA}| + O(n)$ bits of space achieved query time $O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}} \log^2 k \log^\varepsilon n)$.

1.1 Our contribution Since a CSA needs in general $O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}})$ time to report *any* k occurrences of P in $\mathcal{D}$, this time can be regarded as an optimality standard for any top-k solution that builds on CSAs. While solutions using $O(n \log D)$ bits can reach lower times, compressed solutions using just $O(n)$ additional bits have approached this "optimal" complexity only up to polylogarithmic multiplicative gaps.

We explore a new category of space, which is between the compact one that allows $O(n \log D)$ extra bits and the compressed one that allows only $O(n)$. We allow $O(n \log \log n)$ extra bits, which is well below what we call compact space (unless there are just a few large documents, $D = O(\mathrm{polylog}\, n)$), and slightly above what we call

All measures	Space	Time
Linear space [32]	$O(n \log n)$	$O(p/\log_\sigma n + k)$
Compact space [32]	$O(n \log \sigma + n \log D)$	$O(p/\log_\sigma n + \log_\sigma^\varepsilon n + k)$
Document rank	Space	Time
Compact space [23]	$\lvert\mathrm{CSA}\rvert + O(n \log D)$	$O(t_{\mathrm{cnt}} + k)$
Compact space [13]	$\lvert\mathrm{CSA}\rvert + n \log D + o(n \log D)$	$O(t_{\mathrm{cnt}} + k \log(D/k))$
Our space, unsorted [35]	$\lvert\mathrm{CSA}\rvert + O(n \log\log n)$	$O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}})$
Compressed space [3]	$\lvert\mathrm{CSA}\rvert + O(n)$	$O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}} \log k \log^\varepsilon n)$
Text frequency	Space	Time
Compact space [35]	$\lvert\mathrm{CSA}\rvert + n \log D + o(n \log D)$	$O(t_{\mathrm{cnt}} + k \log^* k)$
Compressed space [35]	$\lvert\mathrm{CSA}\rvert + O(n)$	$O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}} \log^2 k \log^\varepsilon n)$
Ours (both measures)	Space	Time
General	$\lvert\mathrm{CSA}\rvert + O(n \log\log n)$	$O(t_{\mathrm{cnt}} + k\,(t_{\mathrm{SA}} + \log\log n))$
Unsorted	$2\lvert\mathrm{CSA}\rvert + O(n \log\log n)$	$O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}})$
Long $p = \Omega(\log^{4+\varepsilon} n)$	$\lvert\mathrm{CSA}\rvert + O(n \log\log n)$	$O(t_{\mathrm{cnt}} + t_{\mathrm{SA}} \log^{3+\varepsilon} n + t_{\mathrm{sort}}(k, n))$
Long p & unsorted	$\lvert\mathrm{CSA}\rvert + O(n \log\log n)$	$O(t_{\mathrm{cnt}} + t_{\mathrm{SA}} \log^{3+\varepsilon} n + k)$

Table 1: Best space-time tradeoff for top-k document retrieval on strings and our contribution.

compressed space. This range of space was only considered before [35] to solve the problem for document rank in the "optimal" time $O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}})$. However, the latter solution does not deliver the results in sorted order.

We introduce a new index that uses $\lvert\mathrm{CSA}\rvert + O(n \log\log n)$ bits of space and solves top-k queries for our two measures of relevance, delivering the results in decreasing order of relevance, in time $O(t_{\mathrm{cnt}} + k\,(t_{\mathrm{SA}} + \log\log n))$, which is faster than all the compressed solutions. We remark that most CSAs using $O(n \log \sigma + n \log\log n)$ bits offer $t_{\mathrm{SA}} = \Omega(\log n / \log\log n)$ [30], in which case our time is $O(t_{\mathrm{cnt}} + k\, t_{\mathrm{SA}})$.[1] We show how to obtain that time for any CSA if we spend $\lvert\mathrm{CSA}\rvert$ further bits of space and deliver the results in unsorted order.

Further, for long patterns with $p = \Omega(\log^{4+\varepsilon} n)$, our time on all existing CSAs becomes $O(t_{\mathrm{cnt}} + t_{\mathrm{sort}}(k, n))$, where $t_{\mathrm{sort}}(k, n)$ is the time to sort k integers in $[1, n]$ within $O(k \log n + n \log\log n)$ bits. For example, we can obtain $t_{\mathrm{sort}} = O(k \log\log k)$ [1, 17], $t_{\mathrm{sort}} = O(k \log\log_k n)$ [24], and $t_{\mathrm{sort}} = O(k(1 + \log k / \log\log n))$ using dynamic predecessor data structures [38]. This yields $O(t_{\mathrm{cnt}} + k)$ for the most interesting regimes of k. If we can deliver the results in unsorted order, the time for long patterns becomes $O(t_{\mathrm{cnt}} + k)$ on all known CSAs and the RAM-optimal $O(p/\log_\sigma n + k)$ on one particular CSA [15]. No top-k solution using $o(n \log D)$ bits of space has achieved this before.

We thus show that, using this slightly higher redundancy space, we can essentially reach the optimality standard when using CSAs. Table 1 puts our main contributions in context. As most previous results, ours hold in the RAM model of computation with a machine word of $\Theta(\log n)$ bits.

Techniques. We depart from the line of work followed so far to obtain compact or compressed space, and get closer to the classical geometric structure of Navarro and Nekrich [31, 32]. In the case of nodes with polylogarithmic string depth, we obtain variants of the geometric structures that use $O(\log\log n)$ bits per element. To handle longer patterns, however, we need other novel ideas.

One new idea is to exploit the recent discovery that the total string length of all the suffix tree leaves starting equal-letter runs in the Burrows-Wheeler Transform (BWT) [4] adds up to $O(n \log n)$ [22]. We call "irreducible" the ancestors of those leaves. The amount of nodes, over some minimum string length, that are "near" irreducible nodes via suffix links, can be bounded, which allows us to store sufficient information to answer the top-k queries from those nodes.

A second novel idea is that reducible nodes, which have a unique Weiner link, must be isomorphic to the

[1]The only exception offers $t_{\mathrm{SA}} = O(\log_\sigma^\varepsilon n)$ [15], which is also $\Omega(\log\log n)$ unless $\sigma \geq n^{1/\mathrm{polylog}\, n}$. Yet, if $\sigma = n^{\Theta(1)}$, then $n \log \sigma = \Theta(n \log n)$ and compressed solutions make no (asymptotic) sense, so the range is narrow.

subtree they reach via that link. This allows us to sample the chains of reducible nodes and store information only on the sampled nodes. Nodes that are "far", via suffix links, from irreducible nodes, are then handled using those samples. Various novel insights of independent interest on properties of suffix trees are obtained in the way.

Those ideas are combined with novel results on geometric data structures, such as finding heaviest vertical segments that stab a horizontal segment in 2D, or finding heaviest points in 3D. Those problems are not new, but we find new solutions using little space or particular time complexities.

2 Preliminaries

2.1 Suffix Trees A *suffix tree* [42, 25, 41] over a string $S[1..n]$ is essentially a trie storing all n suffixes $S[i..n]$, where unary paths are compressed into edges that are labeled with substrings of S (the labels are represented in constant space as pointers to S, or by converting the trie into a Patricia tree [26]). The suffix tree has then n leaves and less than n internal nodes, so it can be represented in linear space, $O(n \log n)$ bits. For technical convenience we assume that S is terminated with a special symbol, so suffixes are distinguished from other substrings. Every suffix tree leaf then represents a suffix $S[i..n]$ and every internal node represents a substring $S[i..j]$ that occurs more than once in S.

We sometimes refer to the *suffix array*, which is an array aligned to the leaves of the suffix tree. At position j, the suffix array contains the position i of the suffix $S[i..n]$ represented by the jth suffix tree leaf in left-to-right order (we assume the children of suffix tree nodes are ordered left to right by increasing lexicographic order of their edge labels).

Every suffix tree node v represents the string $str(v)$ obtained by concatenating the string labels of the edges that lead from the root to v; we say that $strdepth(v) = |str(v)|$ is the *string depth* of v. The basic function of a suffix tree is to find the occurrences of patterns $P[1..p]$ in S. This is done by computing the *locus* of P, $v = locus(P)$, which is the unique highest node v such that P is a prefix of $str(v)$. By using perfect hashing to store the first symbol of the labels of all the edges leaving each node, the suffix tree finds $locus(P)$ in time $O(p)$. At the locus, we can for example store the number of descendant leaves to know the number of times P appears in S, or we can traverse all those leaves in order to report the positions where P appears in T (i.e., the starting positions i of the suffixes $S[i..n]$ descending from the locus) in optimal time.

To solve more sophisticated problems (and to build them in linear time), suffix trees store additional information. The *suffix link* of a node v representing string $str(v) = a \cdot \alpha$, for a an alphabet symbol, points to the node $w = slink(v)$ representing the string $str(w) = \alpha$; we say that $slabel(v) = a$ labels $slink(v)$. Suffix links always exist because, if v is a node, then $str(v) = a \cdot \alpha$ appears followed by at least two different symbols in S, and so does α. The reverse links are called *Weiner links*: $v = wlink(w, a)$ is the Weiner link of w by symbol a. Not every node has a Weiner link by every possible symbol, and it does not always point to a suffix tree node: $str(w) = \alpha$ may appear followed by more than one symbol in S, but still $a \cdot \alpha$ may not appear (so $wlink(w, a)$ does not exist) or appear always followed by the same symbol (so $wlink(w, a)$ would "point to the middle of an edge" in the suffix tree). We will call locations those points in the middle of edges.

Given a string collection $\mathcal{D} = \{S_1, \ldots, S_D\}$ of D distinct strings of lengths $n_d = |S_d|$, we define their concatenation $S[1..n] = S_1 \cdots S_D$ (so $n = \sum_d n_d$) and call T the suffix tree of S (we also say T is the suffix tree of $\mathcal{D}$). For convenience, we assume that each string S_d has its own special terminator symbol $\$_d$, and so assume that suffixes in S reach only up to the next terminator without risk of having duplicate suffixes. We refer to each document S_d simply by its identifier d, and call T_d the suffix tree of S_d (or just of d).

We will use $v \in T_d$ or $v' \in T$ to indicate which tree the nodes belong. The trees T_d can be embedded in T: for each $v \in T_d$ there is a *corresponding* node $v' \in T$ such that $str(v') = str(v)$, but the reverse is not always true. Further, a node in T may correspond to several nodes in distinct T_ds. Since all the suffixes in $\mathcal{D}$ are different, however, leaves of T_d have one-to-one correspondences in T. Note that the node corresponding to $slink(v) \in T_d$ is $slink(v') \in T$, and the node corresponding to $wlink(v, a) \in T_d$ is $wlink(v', a) \in T$. Ancestorship in T_d is also preserved in T, and vice versa, but parent/child relations are not.

Figure 1 illustrates some of these concepts. Identifying nodes v with $str(v)$, we note that node a has a Weiner link $wlink(\mathsf{a}, \mathsf{c})$ that points to the explicit node ca, a Weiner link $wlink(\mathsf{a}, \mathsf{b})$ that points to the middle of the edge between nodes b and bar, and no Weiner link $wlink(\mathsf{a}, \mathsf{a})$.

When using relevance measure $rel(P, d) = docrank(d)$, we assume that the strings S_d are ordered by increasing relevance, so $docrank(d)$ is just d and we want the k highest document identifiers d such that P appears in S_d

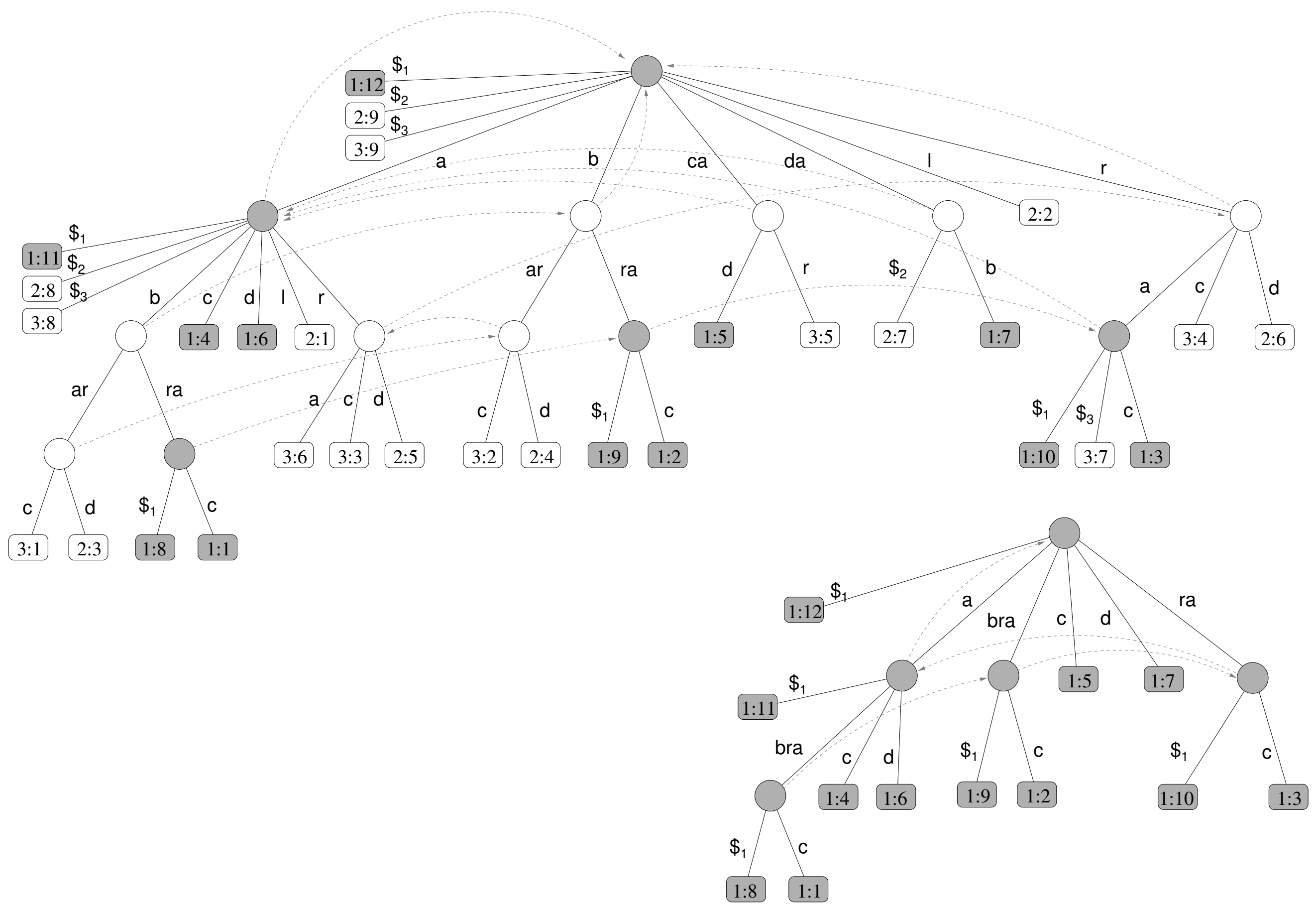

Figure 1: The suffix tree T of the collection $\mathcal{D} = \{\mathsf{abracadabra}, \mathsf{alabarda}, \mathsf{abarcara}\}$. The edges to leaves show only the first letter of their labels. Leaves indicate positions $S_d[i]$ as $d : i$. Dashed arrows show the suffix links from internal nodes; their reverse arrows are Weiner links. The grayed nodes show the embedding of the suffix tree T_1 of $S_1 = \mathsf{abracadabra\$}$ in T, which is shown on the bottom right.

(i.e., $locus(T_d, P)$ exists).[2] When using $rel(P, d) = freq(P, d)$, the measure is the number of leaves descending from $locus(T_d, P)$ and we want the k documents d so that $locus(T_d, P)$ exists and has the largest number of leaves descending from it. Note that, if $v = locus(T_d, P)$, then $freq(P, d) = freq(str(v), d)$.

2.2 The Pointer Framework The *pointer framework* of Hon et al. [21, 18] (also followed by Navarro and Nekrich [31, 32]) solves top-k queries by introducing upward *pointers* in the suffix tree T of $\mathcal{D}$. We say that a leaf $v \in T$ is *marked* with document d if the suffix represented by v belongs to S_d. An internal suffix tree node v, instead, is *marked* with d if at least two children of v are ancestors of leaves marked with d. While a leaf is marked with only one value d, an internal node can be marked with many values d. From every node $v \in T$ marked with d, a *pointer* $ptr(v, u, d)$ leads to its lowest ancestor $u \in T$ that is also marked with d. If no such ancestor exists, then a pointer $ptr(v, \top, d)$ leads to a dummy node $\top$ that is assumed to be the parent of the root of T. We also assign a *weight* to every pointer $ptr(v, u, d)$, which is the relevance score $rel(str(v), d)$. We will say that a pointer $ptr(v, u, d)$ *starts* in the node v, *ends* in the node u, and is *marked* with the document d.

An important property is that the set of pointers marked with a specific document d form a tree that is embedded in T and is isomorphic to T_d.

Lemma 2.1. *[21] The set of pointers $ptr(v, u, d)$, interpreted as edges from u to a child v, is isomorphic to the*

[2]If the documents cannot be reordered in that way, then we assume a constant-time function computes $docrank(d)$.

suffix tree T_d of d, that is, the parent-child relations are preserved and the leaves correspond to the same suffixes.

We will call this set of edges the *pointer tree* of d and, per Lemma 2.1, will identify it with T_d. We will say that a pointer $ptr(v,u,d)$ *straddles* a node w if v is a descendant of w (or w itself) and u is a proper ancestor of w. The key to solving the top-k retrieval problem is the following lemma.

LEMMA 2.2. *[21] For every node w of T and every document d, there is at most one pointer $ptr(v,u,d)$ that straddles w, and it exists only if $str(w)$ occurs in S_d. If $w = locus(P)$ and $ptr(v,u,d)$ straddles w, then P appears in S_d and $rel(P,d) = rel(str(w),d)$, which is the weight of the pointer $ptr(v,u,d)$.*

The task of finding k most relevant documents that contain P is then reduced to finding k heaviest pointers that straddle the locus of P.

We can illustrate some concepts using again Figure 1. Regarding document S_1, each gray node $v \in T$ has a pointer $ptr(v,u,1)$ towards its lowest grayed ancestor. The set of those pointers is isomorphic to T_1. Identifying nodes v with their string $str(v)$, we can see that $ptr(\mathsf{ra},\varepsilon,1)$ straddles ra and r. The documents where $P = \mathsf{ra}$ appears are S_1 and S_3; consequently, exactly one pointer per document straddles the locus ra of P: $ptr(\mathsf{ra},\varepsilon,1)$ for S_1 and $ptr(\mathsf{ra\$_3},\mathsf{r},3)$ for S_3. If we use frequencies, their weights are 2 and 1, respectively.

3 On the Structure of Suffix Trees

3.1 Reducible and Irreducible Nodes and Locations At the core of our solution is the idea that some nodes v of suffix trees T_d have only one Weiner link $wlink(v,a)$ leaving them. We say that such nodes are *reducible* and define $wlabel(v) = a$. Certainly all leaves v are reducible, as they represent a unique suffix $S_d[i..]$ and $wlabel(v) = S_d[i-1]$,[3] but we are more interested in the internal nodes of T_d. We start with the following observation.

LEMMA 3.1. *If a node $u \in T_d$ is reducible and v descends from u, then v is also reducible and $wlabel(v) = wlabel(u)$.*

Proof. Let $str(u) = \alpha$ and $str(v) = \alpha\beta$. Since u is reducible, all occurrences of α in S_d are preceded by the same symbol $a = wlabel(u)$. Then all the occurrences of $\alpha\beta$ in S_d are preceded by a, thus v is also reducible and $wlabel(v) = a = wlabel(u)$. ☐

The *(w-)irreducible* nodes of T_d are then those (internal) nodes v having more than one Weiner link leaving them. Per Lemma 3.1, all the ancestors of irreducible nodes are also irreducible. We define *lowest irreducible* nodes as the irreducible nodes whose descendants are all reducible.

Note that every irreducible node v must have two consecutive descendant leaves v'_l, v_l such that $wlabel(v'_l) \neq wlabel(v_l)$, as otherwise all the descendant leaves would have the same *wlabel* and v would be reducible. We call (v'_l, v_l) a *switching* pair of leaves.

A key property of irreducible nodes is the following. To prove it, let us define $lcp(v)$ for a leaf $v \in T_d$. If v is the leftmost leaf in T_d, then $lcp(v) = 0$. Otherwise, let $v' \in T_d$ be the leaf preceding v and $u = lca(v',v)$ be the lowest common ancestor of v' and v; then $lcp(v) = strdepth(u)$.

LEMMA 3.2. *Let I_d be the set of all lowest irreducible nodes in T_d. Then the sum of the string depths of all the nodes in I_d is $O(n_d \log n_d)$.*

Proof. By definition of irreducibility, a switching pair of leaves (v'_l, v_l) must descend from each node v in I_d, and this switching pair must be unique for each $v \in I_d$ because lowest irreducible nodes cannot descend from one another. Since both v'_l and v_l descend from v, it follows that $lcp(v_l) = strdepth(lca(v'_l,v_l)) \geq strdepth(v)$. The sum of the *lcp* values for *all* the leaves v_l belonging to switching pairs (v'_l, v_l) of T_d is bounded by $O(n_d \log n_d)$ [22, Thm. 1], therefore $\sum_{v \in I_d} strdepth(v) \leq \sum_{v \in I_d} lcp(v_l) = O(n_d \log n_d)$. ☐

Later in the paper, we will consider *locations* in a suffix tree, which correspond to (virtual) nodes in the middle of an edge. Let $str(u') = \alpha$ and $str(v') = \alpha\beta$, then a location μ on (v',u') is defined by its represented string, $str(\mu) = \alpha\beta'$, where β' is a strict nonempty prefix of β.

[3]The leaf representing the whole string S_d has no Weiner link; we also call it reducible and set $wlabel(v) = \$_d$.

We note that Weiner links from nodes may lead to locations: if $str(x) = \alpha$ and $a \cdot \alpha$ appears in $\mathcal{D}$, then $wlink(x, a)$ might not exist as a node, but it still corresponds to a location μ with $str(\mu) = a \cdot \alpha$. We can also take suffix and Weiner links of locations. Given locations μ on edge (x, y) and ν on edge (x', y'), we say that $\nu = slink(\mu)$ iff $slink(y)$ (is or) descends from y' and $strdepth(\nu) = strdepth(\mu) - 1$. We also say that $\mu = wlink(\nu, a)$, where $a = slabel(y)$.

3.2 Near and Far Nodes and Locations We now define a notion of nodes v being *near* or *far* irreducible nodes, via suffix or Weiner links. If a node $v \in T_d$ is reducible, we say that its Weiner link $wlink(v, a)$ is *lonely* (this is a property of the Weiner link pointer, not of the target node). The *w-distance* of a node $v \in T_d$, $w\text{-}dist(v)$, is its distance, via lonely Weiner links, to an irreducible node. That is, $w\text{-}dist(v)$ is the maximum s such that there is a chain $v = v_0 \to \cdots \to v_s$ of lonely Weiner links from each v_i to v_{i+1}, and v_s is irreducible. Note that v is irreducible iff $w\text{-}dist(v) = 0$. The w-distance is undefined for leaves of T_d.

Locations interact with irreducible nodes in the following interesting way.

Lemma 3.3. *If $u \in T_d$ is a node and $\mu = wlink(u, a)$ is a location, then this Weiner link is not lonely and u is irreducible, that is, $w\text{-}dist(u) = 0$.*

Proof. Let $str(u) = \alpha$, so $str(\mu) = a \cdot \alpha$. Since μ is a location, every occurrence of $a \cdot \alpha$ in S_d is followed by the same symbol c. But since u is a node, there must be occurrences of α followed by a symbol $d \neq c$. Those occurrences of $\alpha \cdot d$ cannot be preceded by a, so there must be some symbol $b \neq a$ such that $b \cdot \alpha$ occurs in S_d, and therefore $wlink(u, b)$ exists and u is irreducible. ∎

Analogously, we say that the suffix link $v = slink(u)$ corresponding to a lonely Weiner link $u = wlink(v, a)$ is lonely, and define the *s-distance* of a node $u \in T_d$, $s\text{-}dist(u)$, as the length of the maximal chain of lonely suffix links leaving u. Note that $s\text{-}dist(u) = 0$ iff its suffix link pointer $slink(u)$ is not lonely, or equivalently, its suffix link target is w-irreducible. We will say that u is s-irreducible if $s\text{-}dist(u) = 0$, and define *lowest s-irreducible* nodes u as nodes that satisfy $s\text{-}dist(u) = 0$ and $s\text{-}dist(w) > 0$ for every descendant w of u. We prove an analogue of Lemma 3.2 for s-distances.

Lemma 3.4. *Let V_d be the set of all internal lowest s-irreducible nodes $u \in T_d$. Then the total string depth of all nodes in V_d is $O(n_d \log n_d)$.*

Proof. Let $u \in V_d$. Since $s\text{-}dist(u) = 0$, its suffix link leads to an internal node $v = slink(u)$ such that $w\text{-}dist(v) = 0$. Let $a = slabel(u)$. Then there must exist a switching pair (v'_l, v_l) descending from v, and a symbol $b \neq a$, such that either $wlabel(v'_l) = a$ and $wlabel(v_l) = b$, or vice versa. We say that the switching pair (v'_l, v_l) *justifies* u.

We now show that every switching pair (v'_l, v_l) in T_d can justify at most two nodes in V_d. Indeed, if their *wlabel* values are a and b, then there can be two nodes in V_d, u_a and u_b, where $u_a = wlink(v, a)$ and $u_b = wlink(v, b)$, justified by (v'_l, v_l). If the pair justifies another node u'_a by the symbol a (analogously, u'_b by the symbol b), then $v' = slink(u'_a)$ must be an ancestor of v'_l and v_l, and thus descend from v or be an ancestor of v. If v' descends from v, then $u'_a = wlink(v', a)$ descends from $u_a = wlink(v, a)$, and u_a cannot belong to V_d by definition since $s\text{-}dist(u'_a) = 0$. Otherwise, v descends from v', so u_a descends from u'_a, and similarly u'_a cannot belong to V_d.

Now, note that $strdepth(v) \leq strdepth(lca(v'_l, v_l)) = lcp(v_l)$ and then, if u is justified by (v'_l, v_l), then $strdepth(u) = 1 + strdepth(v) \leq 1 + lcp(v_l)$. Let J_d be the set of all switching pairs in T_d. Every $u \in V_d$ must be justified, and every pair (v'_l, v_l) justifies at most two nodes $u \in V_d$, in which case $strdepth(u) \leq 1 + lcp(v_l)$. Therefore, $\sum_{u \in V_d} strdepth(u) \leq 2\,n_d + 2 \sum_{(v'_l, v_l) \in J_d} lcp(v_l)$. The result follows again from the fact that $\sum_{(v'_l, v_l) \in J} lcp(v_l) = O(n_d \log n_d)$ [22]. ∎

Lemma 3.3 justifies defining the *s-distance* of locations, $s\text{-}dist(\mu)$, as the minimum length of a chain of suffix links from μ reaching a node v (because v must be irreducible). We can define *s-irreducible* locations μ as those with $s\text{-}dist(\mu) = 0$, and *lowest s-irreducible* locations as those irreducible locations without irreducible locations descending from them. We can also prove an analogue of Lemma 3.4 for locations.

Lemma 3.5. *Let $\overline{V}_d$ be the set of all lowest s-irreducible locations $\mu \in T_d$. Then the total string depth of all locations in $\overline{V}_d$ is $O(n_d \log n_d)$.*

Proof. Let $\mu \in \overline{V}_d$ be such a location. By Lemma 3.3, the node $v = \mathit{slink}(\mu)$ has $\mathit{w\text{-}dist}(v) = 0$. Then exactly the same argument used in Lemma 3.4 for nodes in V_d applies to locations in $\overline{V}_d$, because we are also considering lowest locations and thus every switching pair can justify at most two. ∎

We now prove an extension of Lemma 3.1, yet for s-distances.

Lemma 3.6. *Let $u \in T_d$ be an ancestor of $v \in T_d$. Then $\mathit{s\text{-}dist}(u) \le \mathit{s\text{-}dist}(v)$.*

Proof. If $\mathit{s\text{-}dist}(u) = 0$ we are done. Otherwise, we proceed by induction on $\mathit{s\text{-}dist}(v)$. If $\mathit{s\text{-}dist}(v) = 0$, then $w = \mathit{slink}(v)$ is w-irreducible, and by Lemma 3.1, so is its ancestor $\mathit{slink}(u)$; thus $\mathit{s\text{-}dist}(u) = 0$. If not, both s-distances are positive and $\mathit{s\text{-}dist}(u) = 1 + \mathit{s\text{-}dist}(\mathit{slink}(u))$, $\mathit{s\text{-}dist}(v) = 1 + \mathit{s\text{-}dist}(\mathit{slink}(v))$. The result then follows by the inductive hypothesis because $\mathit{slink}(u)$ is an ancestor of $\mathit{slink}(v)$. ∎

A key property we will use is that, if v is reducible, then the subtree of T_d rooted at v must be isomorphic to the one rooted at $\mathit{wlink}(v, a)$; cf. [16, Thm. 7.7.1].

Lemma 3.7. *Let $u, v \in T_d$ where $\mathit{s\text{-}dist}(u) > 0$ and $v = \mathit{slink}(u)$. Then the subtrees of T_d rooted at u and v are isomorphic.*

Proof. Since $\mathit{s\text{-}dist}(u) > 0$, v is reducible and so are all its descendants, by Lemma 3.1. Let $a = \mathit{wlabel}(v)$, so $u = \mathit{wlink}(v, a)$. Any descendant u' of u also satisfies $\mathit{s\text{-}dist}(u') > 0$ by Lemma 3.6, and since $v' = \mathit{slink}(u')$ descends from v, it follows that $u' = \mathit{wlink}(v', a)$. Analogously, for every descendant v' of v, we have that $u' = \mathit{wlink}(v', a)$ descends from u and it holds $v' = \mathit{slink}(u')$. It then follows that $\mathit{slink}(\cdot)$ and $\mathit{wlink}(\cdot, a)$ are inverse functions, and hence bijections, between the subtrees of u and v. For $x = u, v$, let $S_x = \{\alpha,\ \mathit{str}(x') = \mathit{str}(x)\alpha$ and x' descends from $x\}$. By the bijections, it follows that $S_u = S_v$, and since the topology and edge labels of the subtrees of u and v are functions of S_u and S_v, respectively, the subtrees of u and v are isomorphic. We note that the suffix tree positions of the leaves are also the same, all shifted by one. ∎

We will say that an internal node $v \in T_d$ is *δ-near*, or just *near*, if its s-distance does not exceed δ, for a parameter δ that will be defined later, $\mathit{s\text{-}dist}(v) \le \delta$. The *($\delta$-)near subtree* of a document d is the subtree T_d^δ formed by the δ-near nodes of T_d. By Lemma 3.6, T_d^δ consists of the top part of T_d, that is, if a node v belongs to T_d^δ, then its ancestors belong as well. We note that the leaves of T_d^δ are always internal nodes in T_d. Note also that a node of T might correspond to a near node in some T_d and to a not-near node in another $T_{d'}$. Let us call slink^j the iterated application of slink. The following is a key result on near subtrees.

Lemma 3.8. *The sum of the string depths of all the leaves in a near subtree T_d^δ, and thus the total length of all the edge labels, is $O(\delta n_d \log n_d)$.*

Proof. For any $0 \le j \le \delta$, let V^j be the set of all leaves v of T_d^δ such that $\mathit{s\text{-}dist}(v) = j$. Note that V^0 is a subset of the set V_d defined in Lemma 3.4, as nodes in V_d might not be leaves in T_d^δ. By Lemma 3.4, then, the total string depth of all nodes in V^0 is $O(n_d \log n_d)$.

For every $1 \le j \le \delta$, there is an injective mapping between V^j and V_d: if we follow j suffix links from a node $v^j \in V^j$, we reach a distinct s-irreducible node $\mathit{slink}^j(v^j) = v \in V_d$. Indeed: (1) The node v is unique for v^j because there is a path of lonely Weiner links from v to v^j. (2) The node v belongs to V_d because, first, $\mathit{s\text{-}dist}(v) = 0$, so v is s-irreducible. Second, it is lowest s-irreducible: by Lemma 3.7, if there were a descendant w of v with $\mathit{s\text{-}dist}(w) = 0$, then there would be a descendant w^j of v^j with $\mathit{slink}^j(w^j) = w$, that is, it would hold $w^j \in V^j$ and then v^j would not be a leaf of T_d^δ.

Note that $\mathit{strdepth}(v^j) = \mathit{strdepth}(v) + j$. Hence the total string depth of all nodes in V^j does not exceed $O(n_d \log n_d + j\,|V^j|) \subseteq O(n_d \log n_d + \delta |V^j|)$. Added over all the sets V^j, since they are disjoint, yields the bound $O(\delta n_d \log n_d + \delta n_d) = O(\delta n_d \log n_d)$ for the string depths of all the leaves in T_d^δ, and hence also for the sum of all edge labels in T_d^δ. ∎

A consequence of this result is that there cannot be too many nodes in near subtrees whose string depth is large enough, and therefore we can spend some space on those nodes. Their total string depth is also bounded.

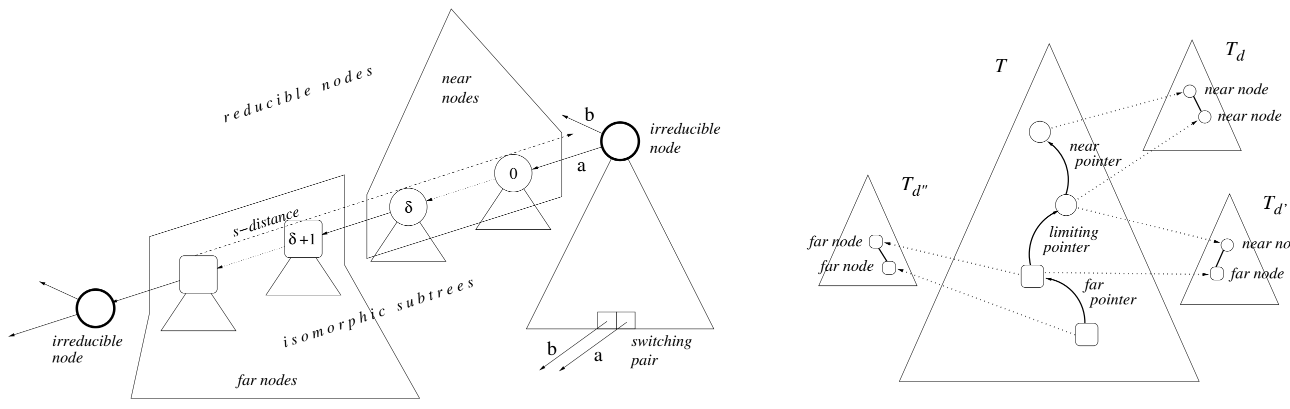

Figure 2: On the left, the main concepts on the nodes of a suffix tree T_d. All leftward arrows are Weiner links. The numbers inside the reducible nodes are their s-distance. On the right, types of pointers in T and relation with nodes in T_d.

Lemma 3.9. *The sum of the string depths of all the leaves of all the near subtrees T_d^δ, as well as the total length of all their edge labels, is $O(\delta n \log n)$. There are $O(n/\log n)$ leaves and branching nodes with string depth at least $\Delta = \delta \log^2 n$ across all the near subtrees T_d^δ.*

Proof. Adding the $O(\delta n_d \log n_d)$ bounds of Lemma 3.8 over all documents d yields the claimed limit $O(\delta n \log n)$. In particular, this is also a limit on the sum of the string depths of all the leaves with string depth at least Δ in near subtrees. It follows that there are only $O((\delta n \log n)/\Delta) = O(n/\log n)$ such leaves, as well as branching internal nodes, of string depth at least Δ in all near subtrees. $\square$

Nodes $v \in T_d$, both internal and leaves, with $s\text{-}dist(v) > \delta$, are called *$\delta$-far*, or just *far*. To handle far nodes v, we will exploit Lemma 3.7. Since a δ-far node v induces a chain of isomorphic subtrees rooted at $slink^j(v)$, we will *sample* one out of δ nodes in those chains, and infer the answer on every node of the chain from the information stored at the next sampled node.

Analogously as for nodes, we say that a location μ on an edge (v, u) of T_d is *(δ-)near* if either the node v is δ-near or $s\text{-}dist(\mu) \le \delta$. We will also bound the string depths of all δ-near locations.

Lemma 3.10. *Let $\overline{V}$ denote the set of all lowest δ-near locations in T_d. Then the total string depth of all locations in $\overline{V}$ is bounded by $O(\delta n_d \log n_d)$.*

Proof. For $0 \le j \le \delta$, let $\overline{V}^j$ be the set of locations $\nu \in T_d$ such that $s\text{-}dist(\nu) = j$ and every location ν' below ν has $s\text{-}dist(\nu') > j$ (thus $\overline{V}^0 = \overline{V}_d$). An argument analogous to that of Lemma 3.8, now applied on locations instead of nodes, shows that $slink^j(\cdot)$ is an injective mapping from $\overline{V}^j$ to $\overline{V}^0$: if $\nu \in \overline{V}^j$, then $\mu = slink^j(\nu)$ is unique and belongs to $\overline{V}_d$ because (1) one reaches ν via j lonely Weiner links from μ; (2) μ is s-irreducible and, if there existed an irreducible location μ' descending from μ, then by Lemma 3.7 there would exist ν' below ν with $s\text{-}dist(\nu') \le j$, and thus ν would not belong to $\overline{V}^j$. From the injectiveness, we can use exactly the same counting argument of Lemma 3.8 to conclude that $|\overline{V}| = O(\delta n_d \log n_d)$. $\square$

Figure 2 (left) illustrates the concepts. We miss leaves v with $s\text{-}dist(v) \le \delta$. Since for leaves v it holds $s\text{-}dist(v) = strdepth(v)$, those leaves will be handled by "shallow" pointers, see next.

4 General Structure of our Solution

We will classify the pointers $ptr(v, u, d)$ in the global suffix tree T as follows (see Fig. 2, right):

- If v is a near node, then u is also near by Lemma 3.6 and we say that $ptr(v, u, d)$ is *near*.
- If u is a far node, then v is also far by Lemma 3.6 and we say that $ptr(v, u, d)$ is *far*.

- The other possibility is that v is far and u is near; in this case we say that $ptr(v, u, d)$ is *limiting*.
- Independently of the above classes, if $strdepth(u) \le \Delta$, we say that $ptr(v, u, d)$ is *shallow*.

Our data structure consists of

(i) Any CSA on the sequence $S[1..n]$, using $|\mathrm{CSA}|$ bits. It counts in time $t_{\mathrm{cnt}} = t_{\mathrm{cnt}}(p)$ and accesses any suffix array position in time $t_{\mathrm{SA}} = t_{\mathrm{SA}}(n)$.

(ii) The topologies of the global suffix tree T of S and the local suffix trees T_d of S_d, using $O(n)$ bits overall and supporting in constant time a number of navigation operations [34].

(iii) A compressed suffix tree (CST) representation of T, which builds on top of the CSA and, adding $O(n)$ bits, supports most suffix tree operations in time $O(t_{\mathrm{SA}})$ [9].

(iv) A bitvector $L[1..n]$ that stores 1s at the positions where the different documents S_d start in their concatenation S. It supports operation $rank(L, i)$, which counts the 1s in $L[1..i]$, in constant time, using $n + o(n)$ bits in total [7, 27].

(v) Other structures using $O(n \log\log n)$ bits that are specific of each kind of pointer we handle.

We define $\delta = \log^{2+\varepsilon} n$ for any constant $\varepsilon > 0$, and $\Delta = \delta \log^2 n = \log^{4+\varepsilon} n$. To answer a top-$k$ query for a pattern string $P[1..p]$, we first find the suffix array interval $[sp, ep]$ of P using the CSA, compute the sp-th and ep-th leaves l and r of T using its topology, and then the locus $w = locus(P) = lca(l, r)$. We also compute the parent $w_p = parent(w)$ of w in T using the tree topology, and its string depth $strdepth(w_p)$ using the CST; note $p > strdepth(w_p)$. All this takes time $O(t_{\mathrm{cnt}} + t_{\mathrm{SA}})$. Now we proceed as follows.

1. If $strdepth(w_p) > \Delta$ and $ep - sp < \delta \log n$, the query is solved by brute force, in time $O(\delta \log n + k\, t_{\mathrm{SA}})$, as described next in Section 5. Since $p > \Delta$, this time is in $O(p/\log n + k\, t_{\mathrm{SA}})$.
2. Otherwise, if $strdepth(w_p) > \Delta$, we form our answer from the first three pointer categories:
   (a) We identify k heaviest near pointers that straddle w in $O(t_{\mathrm{sort}}(k, n))$ time, as described in Section 6. If we can deliver the results in unsorted order, the time is $O(k)$.
   (b) We identify k heaviest far pointers that straddle w, as described in Section 7, in time $O(k + \delta(t_{\mathrm{SA}} + \log n))$. Again, since $p > \Delta$, this is in $O(k + p\, t_{\mathrm{SA}}/\log^2 n + p/\log n)$.
   (c) Limiting pointers will be assimilated to the near and far pointers, as described in Section 8, without additional costs.

   Those sets of candidates come sorted by decreasing relevance, so from the $2k$ candidates we obtain in $O(k)$ additional time the final k documents to be returned.
3. Otherwise, $strdepth(w_p) \le \Delta$ and all pointers straddling w are shallow. We then complete the query in time $O(k(t_{\mathrm{SA}} + \log\log n))$ as described in Section 9.

The total time is then $O(t_{\mathrm{cnt}} + p/\log n + p\, t_{\mathrm{SA}}/\log^2 n + k(t_{\mathrm{SA}} + \log\log n))$. In the RAM model we need time $\Omega(p/\log_\sigma n)$ just to read the pattern even if it comes in packed form, thus the term $p/\log n$ must be in $O(t_{\mathrm{cnt}})$. Further, all known CSAs offer $t_{\mathrm{SA}} = O(\log n)$ if they are allowed to use $O(n)$ bits of space, in which case the same argument applies to $O(p\, t_{\mathrm{SA}}/\log^2 n)$ as well. We can extend the bound to any CSA offering $t_{\mathrm{SA}} = O(\mathrm{polylog}\, n)$ by redefining $\Delta = \delta \log n(\log n + t_{\mathrm{SA}})$ so that $\delta\, t_{\mathrm{SA}} \le \Delta/\log n < p/\log n$; our results only need that Δ is polylogarithmic.

Theorem 4.1. *Let a document collection $\mathcal{D}$ have total length n and a CSA built on it use $|\mathrm{CSA}|$ bits, finding the suffix array range of a string of length p in time $t_{\mathrm{cnt}}(p)$ and retrieving any suffix array cell in time $t_{\mathrm{SA}}(n) = O(\mathrm{polylog}\, n)$. Then there exists a data structure that uses $|\mathrm{CSA}| + O(n \log\log n)$ bits of space and supports top-k queries on $\mathcal{D}$ for a string of length p in time $O(t_{\mathrm{cnt}}(p) + k \cdot (t_{\mathrm{SA}}(n) + \log\log n))$. The relevance measures supported by the data structure are document rank and text frequency. The working space of our query is $O(k \log n)$ bits on top of the index space bounds.*

For example, using $O(n \log \sigma + n \log \log n)$ bits of space, particular CSAs yield times $O(p + k \log n / \log \log n)$ [2], or if, say, $\sigma = O(\text{polylog}\, n)$, $O(p/\log_\sigma n + k \log_\sigma^\varepsilon n)$ [15].

Note that, in case 1, k is bounded by $ep - sp + 1 = O(\delta \log n)$. Thus, we can return to $\Delta = \delta \log^2 n$ and write the time in the following way for long patterns.

Corollary 4.1. *The time of Theorem 4.1, when $p = \Omega(\log^{4+\varepsilon} n)$ for any constant $\varepsilon > 0$, can be reduced to $O(t_{\text{cnt}}(p) + t_{\text{SA}} \log^{3+\varepsilon} n + t_{\text{sort}}(k, n))$. On all the known CSAs, the time is $O(t_{\text{cnt}}(p) + t_{\text{sort}}(k, n))$.*

Results in unsorted order. If we can return unsorted results, the time in case 2 is bounded by $O(t_{\text{cnt}} + k)$: to merge the unsorted results of 2(a) with those of 2(b), we use a linear-time algorithm to find the kth most relevant element, then scan the list to collect the $k' < k$ elements more relevant than the kth, and finally scan again to collect $k - k'$ elements as relevant as the kth.

Corollary 4.2. *The time of Theorem 4.1, when $p = \Omega(\log^{4+\varepsilon} n)$ for any constant $\varepsilon > 0$, can be reduced to $O(t_{\text{cnt}}(p) + t_{\text{SA}} \log^{3+\varepsilon} n + k)$ if the results can be delivered in unsorted order. On all the known CSAs, the time is $O(t_{\text{cnt}}(p) + k)$.*

For example, using the CSA of Grossi and Vitter [15], the time for long patterns is the RAM-optimal $O(p/\log_\sigma n + k)$. No such a result was obtained before in $o(n \log D)$ bits of space.

Another corollary is that, if we can return the results in unsorted order and store additional CSAs for all the documents S_d, then we can reduce the time for short patterns as follows; see the end of Section 9.1.

Corollary 4.3. *If we use $2|\text{CSA}| + O(n \log \log n)$ bits of space and the results can be delivered in unsorted order, then the time of Theorem 4.1 can be reduced to $O(t_{\text{cnt}}(p) + k\, t_{\text{SA}})$.*

5 Brute Force Solution

For the case where $strdepth(w_p) > \Delta$ and $ep - sp < \delta \log n$, we store a mapped document array $M[1..n]$ that assigns local document identifiers: inside each maximal node $v \in T$ having at most $\delta \log n$ descendant leaves, we give new identifiers to the distinct documents that mark the leaves. These identifiers are in $[1..\delta \log n]$ and preserve the *docrank* of the original documents. Array M requires $O(n \log \log n)$ bits of space.

At query time, since $ep - sp + 1 \leq \delta \log n$ and $[sp, ep]$ is the leaf range of $w = locus(P)$, it follows that w descends from some of the maximal nodes v defined above, and that the document identifiers are mapped consistently within $M[sp..ep]$. We can then operate on this subarray, finding the k maximum values (for *docrank*) or the k most frequent values (for *freq*). This is easily done in $O(\delta \log n)$ time with basic counting techniques because $\delta \log n$ bounds both the size of the range to traverse and the size of the universe of values. For *freq* we require to sort the frequencies, which is also done in linear time with radix sort because the universe size is polylogarithmic.

Together with each distinct value m_i we find in $M[sp..ep]$ we maintain a position $sp \leq p_i \leq ep$ where $M[p_i] = m_i$. Once we have the final k mapped document values m_i, we must convert them to actual document identifiers d_i. For this sake we use bitvector L: if j is the suffix array value in position p_i, $rank(B, j)$ is the identifier of the document that marks the m_ith leaf in T. Suffix array values are computed in time t_{SA} with the CSA, so we compute the k identifers in time $O(k\, t_{\text{SA}})$.

6 Near Pointers

We divide the global suffix tree T into horizontal *slices* H_i of height Δ, where a slice H_i corresponds to the interval $[(i-1)\Delta, i\Delta - 1]$. For each slice H_i we store a data structure D_i representing all the pointers $ptr(v, u, d)$ that intersect the string depths of H_i: D_i contains one vertical segment for each pointer $ptr(v, u, d)$ such that either (1) $strdepth(v)$ is in H_i or (2) $strdepth(u)$ is in H_i or (3) $strdepth(v)$ is below H_i and $strdepth(u)$ is above H_i; in the latter case we say that $ptr(v, u, d)$ *spans* H_i. Pointer $ptr(v, u, d)$ is represented in D_i by a vertical segment (x, y_l, y_h) where x is unique for the pointer and will be defined later, $y_l = \max(i\Delta - strdepth(v), 0)$ and $y_h = \min(i\Delta - strdepth(u), \Delta)$. Note that a pointer that spans H_i is represented by a segment $(x, 0, \Delta)$. Further, the segment is assigned the weight $rel(str(v), d)$ of $ptr(v, u, d)$.

Let $w = locus(P)$ and $i = \lceil strdepth(w)/\Delta \rceil$. Then, every pointer $ptr(v, u, d)$ that straddles w is represented in D_i, and moreover its corresponding segment (x, y_l, y_h) must satisfy $y_l \leq i\Delta - strdepth(w) < y_h$. The second condition needed to straddle w will be that $l_w \leq x \leq r_w$ for some $[l_w, r_w]$ that will be defined later. Therefore, the answer to our top-k query is a set of k heaviest segments in D_i that satisfy the given conditions. The data structure D_i is structured to support those queries, as described in the following lemma.

LEMMA 6.1. *For any $\Delta = O(\text{polylog}\, n)$, we can store m weighted vertical segments on an $m \times \Delta$ grid in an $O(m \log\log n)$-bit data structure, so that for any a, b, and h the heaviest segment (x, y_l, y_h) satisfying $a \leq x \leq b$ and $y_l \leq h < y_h$ can be found in $O(t_{\text{wght}})$ time, where t_{wght} is the time required to compute the weight of a segment.*

Proof. We maintain a variant of the segment tree with node degree $\log^\gamma n$ for a constant $\gamma < 1$. Leaves of the tree contain all possible y-coordinates between 0 and Δ, so the tree has constant height. The range $[\min(u), \max(u)]$ of a node u is the interval containing the smallest to the largest y-coordinates in the leaf descendants of u. The set $S(u)$ associated with an internal node u contains all segments (x, y_l, y_h) such that $[\min(u_i), \max(u_i)] \subseteq [y_l, y_h - 1]$ for at least one child u_i of u but $[\min(u), \max(u)] \not\subseteq [y_l, y_h - 1]$. Every segment (x, y_l, y_h) is then associated with at most two sets $S(u)$ in the same level: if it were associated with three nodes (covering consecutive leaf ranges), then it would contain the middle one. Over all levels, then, each segment is stored in $O(1)$ sets.

A segment (x, y_l, y_h) is stored in $S(u)$ as (x', y'_l, y'_h), where x' is the rank of x in $S(u)$, and y'_l (y'_h) is the index of the smallest (largest) child node u_l (u_r) so that $\min(u_l) \geq y_l$ ($\max(u_r) < y_h$). The coordinate x' is not stored as it corresponds to the position of the segment in $S(u)$; the values y'_l and y'_h are stored in $O(\log\log n)$ bits. Each node u also stores $\log^\gamma n$ bitvectors $B_i(u)$, where the x'-coordinates of segments in $S(u)$ that also belong to $S(u_i)$ are marked with a 1. A coordinate range $[a, b]$ in u is then mapped to the coordinate range $[rank(B_i(u), a-1)+1, rank(B_i(u), b)]$ in u_i, and a coordinate x' inside u_i is mapped to coordinate $select_1(B_i(u), x')$ in u. In this way we map in constante time the query ranges downwards and the answer positions upwards in the tree.

Note that every segment (x, y_l, y_h) where $y_l \leq c < y_h$ is stored in some ancestor u of the leaf c, and in $S(u)$ the segment contains the child u_i of u that contains c. Then, to find the heaviest segment (x, y_l, y_h) such that $x \in [a, b]$ and $y_l \leq c < y_h$, we visit all ancestors u of the leaf c. In each node we find $O(1)$ heaviest segments such that $a_u \leq x' \leq b_u$ and $y'_l \leq i \leq y'_h$, where $[a_u, b_u]$ is the range $[a, b]$ mapped to u and the leaf c descends from the child u_i of u. We then map to the root the $O(1)$ heaviest segments found in each level and return the result, in total time $O(t_{\text{wght}})$.

It remains to explain how queries on $S(u)$ are answered. A set $S(u)$ contains m_u segments (x', y'_l, y'_h) on an $m_u \times \log^\gamma n$ grid. We divide $S(u)$ into $m_u/\log^\gamma n$ blocks of $\log^\gamma n$ consecutive segments. For each $1 \leq j \leq \log^\gamma n$, consider the sequence $S_j(u)$ with the heaviest segment (x', y'_l, y'_h) satisfying $y'_l \leq j \leq y'_h$ from each block. We only keep a range-maximum data structure [8] on the weights of each sequence $S_j(u)$, which consumes $O(|S_j(u)|) = O(m_u/\log^\gamma n)$ bits, adding up to $O(m_u)$ bits over all the values j. On the other hand, each block contains $\log^\gamma n$ segments with values in $[1, \log^\gamma n]$. Hence, the answers to all the possible queries with values (a_u, b_u, i) confined to every possible segment can be precomputed in a universal table of $2^{(\log^\gamma n) \cdot 2\gamma \log\log n} \cdot \log^{3\gamma} n \cdot \gamma \log\log n \in o(n)$ bits. A query range is divided into a middle part that is aligned with block boundaries and is solved with the range maximum query data structure, and left and right extremes that fit inside one block (if not, then the query is confined in a single block, too). We then find up to three candidates to be the heaviest, as promised.

Since each segment belongs to $O(1)$ sets $S(u)$, the sum of all m_u is $O(m)$. The space is then $O(m)$ for the range maximum queries and $O(m \log\log n)$ to store the mapped values y'. The bitvectors $B_i(u)$ of u sum to length $m_u \log^\gamma n$, adding up to $O(m \log^\gamma n)$ across the tree. Each segment stored in some $S(u_i)$ induces a 1 in the bitvector $B_i(u)$ of its parent u, and so the total number of 1s of all bitvectors is $O(m)$. It is then possible to represent them using $O(m \log\log^\gamma n) + o(m) = O(m \log\log n)$ bits and support *rank* and *select* in constant time [36]. ◻

We can solve the top-k query by applying Lemma 6.1 iteratively, as follows (cf. [18]). Frederickson [11] showed how, given an abstract binary tree where each element is not smaller than its children, one can obtain k largest elements with $O(k)$ traversal operations: visit the root or, given any already visited node, visit its children. We define our binary tree as follows: the root corresponds to the range $[l_w, r_w]$, and its value to the maximum weight Lemma 6.1 yields on this range. If this maximum is at position t in the range, then the left and right children of the root correspond to the ranges $[l_w, t-1]$ and $[t+1, r_w]$, respectively. Empty intervals correspond to leaves of the tree.

This algorithm delivers the k heaviest segments in unsorted order, however, so we must sort them by decreasing weight, in the time $t_{\mathrm{sort}}(k,n)$ of any algorithm sorting k integers in $O(n)$. The algorithm must use $O(k \log n)$ bits of working space, plus up to $O(n \log\log n)$ extra bits of space that can be charged to the index. For example, we can obtain $t_{\mathrm{sort}} = O(k \log\log k)$ [17] and $O(\sqrt{n})$ extra space [1, Sec. 5], $t_{\mathrm{sort}}(k,n) = O(k \log\log_k n)$ [24] and $O(\sqrt{n})$ extra space (by sorting in rounds of $\frac{1}{2}\log_2 n$ bits), and $t_{\mathrm{sort}}(k,n) = O(k(1+\log k/\log\log n)$ using dynamic predecessor data structures [38], with just $O(k)$ extra space. In all these cases, $O(k \log n)$ bits of working space suffices [10]. Those choices reach sorting time $O(k)$ for $k = \Omega(n^{\varepsilon})$ for any constant $\varepsilon > 0$, or $k = O(\mathrm{polylog}\, n)$.

To analyze space usage, we first bound the number of segments represented for all the slices.

Lemma 6.2. *The number of segments stored in all the structures D_i, $i > 1$, is $O(n/\log n)$.*

Proof. Every near pointer $ptr(v,u,d)$ may start in a slice and end in another slice, but it may span many slices. Lemma 3.9 states that there are $O(n/\log n)$ near nodes v with depth over Δ, and hence there are $O(n/\log n)$ near pointers that are not shallow. Therefore, there are $O(n/\log n)$ segments starting or ending in some D_i for some $i > 1$. To account for the number of segments spanning slices, we define the *length* of $ptr(v,u,d)$ as $strdepth(v) - strdepth(u)$. By Lemma 2.1, the set of all pointers $ptr(v,u,d)$ is isomorphic to T_d. The set of all near pointers $ptr(v,u,d)$ is then isomorphic to T_d^{δ}, and the length of $ptr(v,u,d)$ corresponds to the length of the string labeling the corresponding edge in T_d^{δ}. By Lemma 3.9, the sum of all edge label lengths in T_d^{δ} is $O(\delta n \log n)$. The same bound then holds for the sum of the lengths of all near pointers $ptr(v,u,d)$ that are not shallow. Every slice spanned by a pointer uses Δ units of its length that are not used for other slices, so the total number of segments spanning slices is $O((\delta n \log n)/\Delta) = O(n/\log n)$. ☐

Therefore, the space required by the structures of Lemma 6.1 across all the slices is $O(n)$ bits.

We list segments in D_i according to the preorder ranks of the nodes v in their corresponding pointers $ptr(v,u,d)$. For every segment s that represents a pointer $ptr(v,u,d)$, we explicitly store its weight and the preorder rank of v in T. All the preorder ranks of those nodes v are also kept in a predecessor data structure [37] that uses constant space per segment and supports queries in time $O(\log\log n)$. The predecessor data structure, as well as all the extra space, uses $O(\log n)$ bits per segment, which summed over all slices yields $O(n)$ bits by Lemma 6.2.

The x-coordinate of a segment (x, y_l, y_h) representing $ptr(v,u,d)$ is then its position in the sequence of segments of D_i where we sort them by the preorder rank of v. Several segments can have the same preorder rank if they have the same node v associated with various pointers with different documents; we give them consecutive x-coordinates in arbitrary order. Thus for every node $w \in T$ there is a range $[l_w, r_w]$ so that the x-coordinate of a segment representing $ptr(v,u,d)$ is in $[l_w, r_w]$ iff v is a descendant of w. We can find l_w and r_w in $O(\log\log n)$ time by querying our predecessor data structure. A pointer $ptr(v,u,d)$ straddles w iff the segment (x, y_l, y_h) representing this pointer satsifies $l_w \le x \le r_w$ and $y_h \le i\Delta - strdepth(w) < y_h$. Hence we can find the top-k near pointers that straddle w using Lemma 6.1.

Lemma 6.3. *If $strdepth(w_p) > \Delta$, we can find k most relevant near pointers that straddle w, in decreasing relevance order, in time $O(t_{\mathrm{sort}}(k,n))$ and using $O(n \log\log n)$ additional bits.*

7 Far Pointers

We cannot bound the number or total string length of far nodes, but as Lemma 3.7 shows, they induce isomorphic subtrees. We now show that, as a consequence, many far pointers are equivalent.

Lemma 7.1. *Let $ptr(v_0,u_0,d)$ be a far pointer. Then there are pointers $ptr(v_1,u_1,d),\ldots,ptr(v_\delta,u_\delta,d)$ such that $v_i = slink(v_{i-1})$ and all the pointers have the same weight.*

Proof. By definition, v_0 is a (δ-)far node, thus $s\text{-}dist(v'_0) > \delta$ holds for its corresponding node $v'_0 \in T_d$. Then there is a chain of lonely suffix links $v'_0 \to \cdots \to v'_\delta$, all with $s\text{-}dist(v'_i) > 0$. By Lemma 3.7, all the subtrees of $v'_i \in T_d$ are isomorphic. Since all belong to the same document d and the weight of $ptr(v_i,u_i,d)$ is a function of the subtree rooted at the node $v'_i \in T_d$, the weights are the same. In particular, *freq* is the same because the subtrees rooted at v'_i have the same number of leaves, and *docrank* is the same because d is the same. ☐

Since there are many similar "consecutive" far nodes, we select one out of δ from those. The pointers corresponding to selected nodes will be called *special*. The selection of special pointers is done as follows. Inside every tree T_d, we select all the nodes v' where $\mathit{s\text{-}dist}(v')$ is a nonzero multiple of δ, and declare special the corresponding pointers $\mathit{ptr}(v,u,d)$, where $v \in T$ corresponds to v'. For each selected node v', there are $\delta - 1$ unique non-selected ones, reached via lonely suffix links from v'. Thus, we select at most n_d/δ nodes in each T_d, and there are $O(n/\delta)$ special pointers.

By our construction, for every far node $v' \in T_d$, there is a selected node $\mathit{slink}^i(v') \in T_d$, with $i < \delta$. We then make use of the following property.

Lemma 7.2. *Let $w \in T$ be such that $\mathit{strdepth}(w) \geq \delta$. Then, for every far pointer $p_0 = \mathit{ptr}(v,u,d)$ that straddles w, there is some $0 \leq i < \delta$ and a special pointer $p_i = \mathit{ptr}(\mathit{slink}^i(v), \mathit{slink}^i(u), d)$ that straddles $\mathit{slink}^i(w)$ and has the same weight of p_0.*

Proof. Nodes $v, u \in T$ are far by definition. Since suffix links of T_d are preserved in T, by our construction there is a special pointer (v_i, u_i, d), where $v_i = \mathit{slink}^i(v)$ for some $0 \leq i < \delta$. By Lemma 2.1 and the existence of p_0, there is an edge in T_d between the nodes u' and v' corresponding to u and v, precisely u' is the parent of v'. Then, $v'_i = \mathit{slink}^i(v')$ descends from $u'_i = \mathit{slink}^i(u')$. By Lemma 3.7, the subtree of the node $u'_i \in T_d$ is isomorphic to the subtree of node $u' \in T_d$. Therefore, u'_i is the parent of v'_i. By Lemma 2.1, then, there exists a pointer from the node corresponding to v'_i to that corresponding to u'_i, marked with document d. Those nodes are precisely $v_i = \mathit{slink}^i(v)$ and $u_i = \mathit{slink}^i(u)$, because suffix links are preserved in the embedding of T_d in T. Thus the special pointer is p_i. The suffix link preserves the property of being straddled, so p_i straddles $\mathit{slink}^i(w)$. Further, by Lemma 7.1, the weight of p_i is the same as that of p_0. □

The idea to find k heaviest far pointers that straddle w is as follows. Although within each T_d the chains of lonely suffix links have a regular structure (one node out of δ is selected), those intermingle in arbitrary patterns in T. Further, lonely suffix link chains refer to trees T_d, but they can join others in T. As a result, we can find special pointers straddling every node w_i in the chain $w_i = \mathit{slink}^i(w)$. Lemma 7.2 guarantees that, if we explore all $0 \leq i < \delta$, we will find special pointers representing every pointer that straddles w. To determine if a special pointer $\mathit{ptr}(v_i, u_i, d)$ straddles w_i, we simply check that v_i is or descends from w_i and that $\mathit{strdepth}(u_i) < \mathit{strdepth}(w_i)$. However, we must also determine if the special pointer represents some pointer $\mathit{ptr}(v,u,d)$ that straddles w, that is, if w has a corresponding node in T_d. Note that this check must be done for all the possible documents d simultaneously. We do this by associating, with each special pointer $\mathit{ptr}(v_i, u_i, d)$, the string of δ symbols that sprout via lonely Weiner links from the node $v'_i \in T_d$ corresponding to v_i, and then requiring that the symbols by which we go from w_i to w form a suffix of that string.

The special pointers are then stored in a data structure for three-dimensional points. For every special pointer $\mathit{ptr}(v,u,d)$, we store a point (x,y,z), where x is the preorder index of v in T, $y = \mathit{strdepth}(u)$, and z identifies the chain of (up to) δ lonely Weiner links that begins in the node $v' \in T_d$ corresponding to v: $v_1 = \mathit{wlink}(v', a_1), v_2 = \mathit{wlink}(v_1, a_2), \ldots, v_\delta = \mathit{wlink}(v_{\delta-1}, a_\delta)$. (The sequence may be shorter because we stop if we hit an irreducible node v_i in the way; for simplicity we assume its length is δ.) Note that $a_\delta \cdots a_2 a_1$ is a substring of at least one string from $\mathcal{D}$. Let T^R be the global suffix tree of the *reverse* documents. We then set z to the preorder index of the node $v^R \in T^R$ such that $v^R = \mathit{locus}(a_1 \cdots a_\delta)$. We assign to the point (x,y,z) the same weight of $\mathit{ptr}(v,u,d)$.

To report k heaviest far pointers that straddle a node w, we then visit the nodes $w_0 = w$, $w_1 = \mathit{slink}(w_0)$, $w_2 = \mathit{slink}(w_1)$, …, $w_{\delta-1} = \mathit{slink}(w_{\delta-2})$. In the process, we incrementally form the string s_i with the sequence of Weiner links traversed, appending every new symbol. For each visited node w_i, we find the range $[\mathit{revleft}(s_i), \mathit{revright}(s_i)]$ in T^R, which contains the preorders of all the nodes in T^R below the locus of s_i^{rev} (i.e., s_i read backwards), and find in our data structure all the points (x,y,z) such that (1) $x \in [\mathit{left}(w_i), \mathit{right}(w_i)]$ (the range of preorders in the subtree of $w_i \in T$), (2) $y < \mathit{strdepth}(w_i) = \mathit{strdepth}(w) - i$, and $z \in [\mathit{revleft}(s_i), \mathit{revright}(s_i)]$ (that is, s_i^{rev} is a prefix of the string $a_1 \cdots a_\delta$ associated with the pointer, or equivalently, s_i is a suffix of $a_\delta \cdots a_1$).

In the following subsections we describe how we simulate the reverse suffix tree T^R (Section 7.1), how the three-dimensional range queries are carried out (Section 7.2), and how the lists $L(w_i)$ of k heaviest pointers straddling w_i are generated and merged (Section 7.3).

The total time required to generate and merge the lists $L(w_i)$ is $O(k + \delta \log n)$. In addition, we perform $O(\delta)$ suffix tree operations like taking suffix links, computing string depths, and preorder ranges, which adds $O(\delta\, t_{\mathrm{SA}})$ time.

LEMMA 7.3. *If $strdepth(w_p) > \Delta$, we can find k most relevant far pointers that straddle w, in decreasing relevance order, in time $O(k + \delta(t_{\mathrm{SA}} + \log n))$ and using $O(n)$ additional bits.*

7.1 Reverse Suffix Tree Preorders Instead of representing a full suffix tree T^R, we use the following data structure. For every node $v \in T$ such that $strdepth(v) \geq \delta$, $strdepth(parent(v)) < \delta$, and v having at least $\delta \log n$ leaf descendants, we store the string $str(v)[1..\delta]$ in an uncompressed trie. In each node of the trie representing string α, we store the range $[revleft(\alpha), revright(\alpha)]$ of all the preorders in the subtree of the node $v^R = locus(\alpha^{rev}) \in T^R$. The trie stores $O(n/(\delta \log n))$ strings of length δ, so it has $O(n/\log n)$ nodes and requires $O(n)$ bits in total.

If, in our suffix link chain $w, \ldots, w_i$, we have read the string $s_i = c_1 \cdots c_i$, then $str(w) = s_i\alpha$ for some α, and therefore string s_i is indexed in the trie. We can thus descend in the trie by each new symbol c_i, in constant time per suffix link taken, and directly access the range $[revleft(s_i), revright(s_i)]$.

Note that $w[1..\delta]$ must be in the trie, because $strdepth(v) \geq \Delta$; otherwise w is shallow and not addressed here. On the other hand, if w has fewer than $\delta \log n$ leaf descendants, then the query was solved by brute force in Section 5. Otherwise, every w_i must have at least $\delta \log n$ leaf descendants because it is reached by (a chain of lonely) suffix links from w.

7.2 Three-dimensional Queries Our data structure holding the points (x, y, z) must report k heaviest points (x, y, z), such that $x \in [a, b]$, $y \in [c, d]$, and $z \in [e, f]$, in decreasing weight order. We now describe such a data structure.

LEMMA 7.4. *There exists a data structure that stores $m \leq n$ three-dimensional weighted points with coordinates and weights in $O(n)$, uses $O(m \log^{2+\varepsilon} n)$ bits of space for any constant $\varepsilon > 0$ and can report k heaviest points in a three-dimensional orthogonal range in time $O(k + \log m)$.*

Proof. The data structure from Chan et al. [6, Thm. 7] uses $O(m \log n)$ bits of space and works in the special case when the two-dimensional query range is bounded on two sides: for any a and b, it reports k heaviest two-dimensional points (x, y) such that $x \leq a$ and $y \leq b$ in time $O(k + \log \log n)$, for any $1 \leq a, b, k \leq n$. Using the non-uniform grids technique [5, Sec. 3], we can extend this result to a data structure that uses $O(m \log^{1+\varepsilon} n)$ bits of space and reports k heaviest points in any two-dimensional query range in $O(k + \log \log n)$ time. Then we extend this result to a data structure that supports three-dimensional queries, using range trees with node degree $\log^{\varepsilon} n$. The space usage increases by a factor of $O(\log^{1+2\varepsilon} n)$ factor and the query time increases by a factor of $O(\log m / \log \log n)$. We refer to Appendix A for details. The resulting data structure uses $O(m \log^{1+\varepsilon} n \log^{1+2\varepsilon} n) = O(m \log^{2+3\varepsilon} n)$ bits of space and can report k heaviest points in an arbitrary three-dimensional query range in time $O(k + \log n)$. If we replace ε with $\varepsilon/3$, we obtain the result. ◻

In our case $m = O(n/\delta)$ is the number of special pointers, so the space usage is $O((n/\delta) \log^{2+\varepsilon} n) = O(n)$ bits and the query time is $O(k + \log n)$.

7.3 Merging Lists Now we explain how the list of k heaviest points in $\cup_{i=0}^{\delta-1} L(w_i)$ can be generated. Let $L(w_i, g)$ denote the list of $2^g \log n$ heaviest points in $L(w_i)$. We start by setting $l_i = 0$ and generating $L(w_i, l_i)$ for all i, $0 \leq i < \delta$. We extract the heaviest point from each $L(w_i, l_i)$, and add these to a heap H. Then we repeat the following steps until the list of top-k pointers is generated:

1. We extract the heaviest point p from H.
2. If the extracted point comes from the list $L(w_i, l_i)$, we remove the next point from $L(w_i, l_i)$ and add it to H.
3. If $L(w_i, l_i)$ is empty, we generate $L(w_i, l_i + 1)$, remove the first $2^{l_i} \log n$ points from $L(w_i, l_i + 1)$ (i.e., we remove the points that were already reported), and increment l_i.

The time of this procedure can be analyzed as follows. Every list $L(w_i, l_i)$ is generated in time $O(2^{l_i} \log n + \log n) = O(2^{l_i} \log n)$ using the structure of Section 7.2. Let f_i denote the final value of l_i, so we queried the structure f_i times, asking for increasing numbers $2^j \log n$ of results. The sum of those query times is

$O(\sum_{j=0}^{f_i} 2^j \log n) = O(2^{f_i} \log n)$. Let k_i denote the number of points from $L(w_i)$ reported by the procedure. Since $2^{f_i - 1} \log n < k_i \le 2^{f_i} \log n$, the cost incurred on $L(w_i)$ is $O(k_i + \log n)$, the second term owing to the first call with $l_i = 0$. Added over the values $0 \le i < \delta$, we get $O(k + \delta \log n)$. The operations on the heap H take $O(1)$ time because it contains at most δ elements, which is polylogarithmic in n [38]. Thus the heap only adds $O(k)$ additional time.

8 Limiting Pointers

Limiting pointers $ptr(v,u,d)$ start in a far node v and end in a near node u. The main idea to handle them is to "split" $ptr(v,u,d)$ into two "consecutive" pointers, $ptr(v,x,d)$ and $ptr(x,u,d)$. The new pointers inherit the weight of the original limiting pointer. We handle the pointers $ptr(x,u,d)$ in the same way as near pointers and the pointers $ptr(v,x,d)$ as far pointers.

Let $ptr(v,u,d)$ correspond to the edge (v',u') in T_d (i.e., u' is the parent of v'). The new node x is then inserted in the middle of the edge (v',u'), that is, in a location. Concretely, x will be the lowest possible near location on the edge, if it exists, recall Section 3.2.

Consider a pointer $ptr(v,u,d)$ from a far node v to a near node u. Since v is far, it must be $strdepth(v) > \delta$. If there are near locations on the corresponding edge (v',u') of T_d, we select the lowest one μ and conceptually insert a new *dummy* node x' splitting (v',u'), with $strdepth(x') = \max(strdepth(\mu), \delta)$. The purpose of this maximization is to ensure that x' can also be treated as a far node, as one can follow δ suffix links from it. If no near location μ exists on the edge, but $strdepth(u) < \delta$, we also insert a dummy node x' on (v',u') with $strdepth(x') = \delta$. If $strdepth(u) \ge \delta$, we do not create any dummy node.

We call edge (x',u') a *dummy edge*. The *extended near subtree* E_d^δ is obtained from the near subtree T_d^δ by adding all dummy edges. Lemma 3.9 is also true for extended short subtrees E_d^δ.

Lemma 8.1. *The sum of the string depths of all the leaves of all the near extended subtrees E_d^δ, as well as the total length of all the edge labels, is $O(\delta n \log n)$. There are $O(n/\log n)$ leaves and branching nodes with string depth at least $\Delta = \delta \log^2 n$ across all the extended near subtrees E_d^δ.*

Proof. Every dummy leaf corresponds to a unique location μ in $\overline{V}$, and its string depth does not exceed $strdepth(\mu) + \delta$. By Lemma 3.10 the total string depth of all dummy leaves in E_d^δ is $O(\delta n_d \log n_d + \delta n_d) = O(\delta n_d \log n_d)$. By the same arguments as in Lemma 3.9, the total string length of all edges, and the total string depth of all branching nodes with string depth at least Δ, are also bounded by $O(\delta n_d \log n_d)$. Thus there are $O(n_d/\log n)$ such branching nodes. Summing over all documents, we obtain the claim. ◻

By Lemma 8.1, we can store the dummy pointers $ptr(x,u,d)$ corresponding to the dummy edges (x',u) in the same data structures D_i described in Section 6, without exceeding our space budget. The other kind of created pointers is $ptr(v,x,d)$, where v is far and x is dummy. They are stored and handled as far pointers exactly as described in Section 7. The next lemma proves the correctness of our scheme.

Lemma 8.2. *Suppose that a node $w \in T$ is straddled by a limiting pointer $ptr(v,u,d)$. Then either (1) w is straddled by a dummy pointer $ptr(x,u,d)$ or (2) $w_i = slink^i(w)$ is straddled by a special pointer $ptr(v_i,u_i,d)$ with the same weight, for some $0 \le i < \delta$.*

Proof. Suppose we added a dummy node x' on the edge (v',u') of T_d corresponding to $ptr(v,u,d)$. If $strdepth(w) \le strdepth(x')$, then w is straddled by $ptr(x,u,d)$. If $strdepth(w) > strdepth(x') \ge \delta$, there exists by Lemma 7.2 an index $0 \le i < \delta$ such that $ptr(v_i,u_i,d)$ is special and $v_i = slink^i(v)$. Let $v_i' \in T_d$ correspond to $v_i \in T$, and $v_p' \in T_d$ be its parent. No location μ' below x' on the edge (v',u') can be reached by following i Weiner links from v_p', otherwise μ' would have been near as well. Hence $strdepth(v_i') - strdepth(v_p') \ge strdepth(v') - strdepth(x') > strdepth(v) - strdepth(w)$. Thus, the pointer $ptr(v_i, v_p, d)$, corresponding to the edge (v_i', v_p'), straddles w_i. If there is no dummy node x' on the edge (v',u'), then $strdepth(u') \ge \delta$, so $u_i' = slink^i(u')$ exists and must be the parent of v_i', $u_i' = v_p'$, so the pointer $ptr(v_i,u_i,d)$ straddles w_i. The reason is that, if $v_p' \ne u_i'$, then v_p' is between v_i' and u_i', thus there must be a near location μ' on (v',u') such that $slink^i(\mu') = v_p'$ and node x' would have been created. ◻

9 Shallow Pointers

Now we consider the case when the string depth of the parent of the locus node w is at most Δ. All the pointers that straddle w are then shallow (independently of being near, far, or limiting, which is irrelevant here). The techniques in the other sections assume $p > \Delta$.

Our solution is similar in spirit to the structure for near pointers in Section 6. We represent each shallow pointer $ptr(v, u, d)$ with a vertical segment (x, y_l, y_h), where x is an index related to the rightmost leaf marked with d in the subtree of v. We set $y_h = \Delta - strdepth(u)$ and $y_l = \max(\Delta - strdepth(v), 0)$. For each pointer that straddles w, there is a segment (x, y_l, y_h) where x refers to some leaf in the subtree of w and $y_l \leq \Delta - strdepth(w) < y_h$.

Note that the "rightmost leaf marked with d in the subtree of v" might be associated with several segments. Segments are ordered by the index of their associated leaf, breaking ties arbitrarily. The x-coordinate of a segment is its position in this ordering. To map between x-coordinates and leaves, we store a bitvector B where we traverse the leaves of T left to right and append a 0 for every new leaf and then a 1 for every segment associated with that leaf. Since all the segments associated with the same leaf are consecutive, the leaf rank of the jth segment is computed in constant time as $select_1(B, j) - j$. Further, let w be the ancestor of the lth to the rth leaves of T. Then the range of all the x-coordinates of nodes below w and its ancestors is $[l_w, r_w] = [select_0(B, l) - l + 1, select_0(B, r + 1) - r - 1]$. The segments are stored in the data structure of Lemma 6.1, which takes $O(n \log \log n)$ bits of space; B takes $O(n)$ bits.

At query time, given $strdepth(w)$ and the leftmost and rightmost leaves l and r descending from w, we compute $[l_w, r_w]$ and query the data structure of Lemma 6.1 $O(k)$ times, following the same procedure described in Section 6. The k final answers come in the form of segment positions; we convert them to leaf ranks in T as explained, and from those ranks we use the CSA to obtain the document identifiers with bitvector L, as explained in Section 5. The total time is dominated by k times the cost to compute a weight, the time to sort the result (the time $t_{\text{sort}}(k, n) = O(k \log \log n)$ is convenient here), and the $O(k\, t_{\text{SA}})$ time to obtain the actual document identifiers. We describe next how to compute the weights in $O(\log \log n)$ time and $O(n \log \log n)$ bits of space.

Lemma 9.1. *If $strdepth(w_p) \leq \Delta$, we can find k most relevant shallow pointers that straddle w, in decreasing relevance order, in time $O(k(t_{\text{SA}} + \log \log n))$ and using $O(n \log \log n)$ additional bits.*

9.1 Computing Weights We have shown how to identify the document d of a segment representing $ptr(v, u, d)$, so computing $docrank(d)$ is immediate. We now show how to compute $freq(P, d)$ in $O(\log \log n)$ time and using $O(n \log \log n)$ bits of space.

The first component of our solution builds on a node marking scheme that has been used several times for compressed top-k retrieval [21, 33, 35]. We mark $O(n_d / \log n)$ nodes $x' \in T_d$, by marking every $\log(n)$-th leaf of T_d and all the lowest marked ancestors of consecutive marked leaves [33]. The preorders of the $O(n_d / \log n)$ nodes $x \in T$ corresponding to marked nodes $x' \in T_d$ are stored in a successor data structure F_d that takes $O(n_d)$ bits and answers in time $O(\log \log n)$ [37].

Given a segment belonging to document d, we know there is exactly one node v below the locus w such that a pointer $ptr(v, u, d)$ straddles w. Let $v' \in T_d$ be the node corresponding to $v \in T$ (i.e., v' is the locus of P in T_d). By the marking scheme, there is at most one maximal marked node $x' \in T_d$ below v', and the preorder of its corresponding node $x \in T$ will be the smallest one following the preorder of w in F_d. Therefore, the successor of the preorder of w in F_d yields the preorder of x, with which we associate the corresponding node $x' \in T_d$ (there is space to store x' because there are $O(n / \log n)$ marked nodes overall).

The second component to find v' is to associate with each segment (x, y_l, y_r) representing $ptr(v, u, d)$, an additional field storing the *tree* depth of the node $v' \in T_d$ corresponding to node $v \in T$, $depth(v')$. Since $depth(u) \leq strdepth(u) \leq \Delta$, the tree depth of its corresponding node $u' \in T_d$ is also $depth(u') \leq \Delta$, and since v' is the child of u' in T_d by Lemma 2.1, it holds $depth(v') \leq \Delta + 1$. The tree depth of v' can then be stored in $O(\log \log n)$ bits with the segments.

We then know a node $x' \in T_d$ that descends from v', and the tree depth of v'. A level ancestor query on the topology of T_d yields then v' in constant time. From v' we compute $freq(P, d)$ as the number of leaves below v', also in constant time.

We have not yet covered the case, however, that v' is so low in T_d that it has no marked descendants. In this case it has only $O(\log n)$ leaves descending from it. The number of those leaves is another component that can be stored directly with the segment (x, y_l, y_r) associated with $ptr(v, u, d)$, as it is a function of v and d and requires only $O(n \log \log n)$ extra bits in total.

Computing weights with an additional CSA. Instead of the successor data structures F_d, we can use one CSA for each document d, CSA_d, which computes the lexicographic rank of any text position in its document (i.e., the inverse of the suffix array permutation) also in time t_{SA}. Given the leaf rank i of the leaf associated with the segment, we use the global CSA to compute its corresponding position t in S, and then the position $t' = t - select_1(L, d) + 1$ within document d. Finally, CSA_d yields in t_{SA} time the rank i' of the corresponding leaf in T_d. We then proceed as above to find v' from that leaf instead of from x'.

10 Conclusions

We have shown that, by adding $O(n \log \log n)$ bits of space on top of a compressed suffix array (CSA), we can solve top-k queries in essentially the time needed to output with the CSA any k positions where the pattern occurs. This time is arguably optimal if we must build on a CSA; we actually achieve RAM-optimal time on long enough patterns if the results can be delivered in unsorted order. In this sense, our index closes this "optimality" gap that is present in the structures that use $O(n)$ bits of extra space, while still using in most practical cases much less space than that of a document array, using which faster solutions are possible. We develop several new insights on suffix trees and geometric data structures that can be of independent interest.

We focused on two relevance measures: (pattern independent) document rank and, especially, the more challenging text frequency. Those can be easily combined in our data structure, so that for example the text frequency is multiplied by some measure of document relevance. Other measures that depend only on the pattern P and can be efficiently computed from its locus are also easy to combine in our index without extra cost. An example is document frequency (i.e., the number of documents where P appears), which can be computed in constant time and $O(n)$ extra bits given the locus of P [39]. On the contrary, a measure that is easy to include in frameworks that use $O(n \log n)$ bits of space, but that has evaded the attempts to be computed on the fly in compact space, is the minimum distance between two occurrences of the pattern in the document; the only scheme supporting this measure in compressed space [28] is much slower than our index.

References

[1] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

[2] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):article 23, 2014.

[3] D. Belazzougui, G. Navarro, and D. Valenzuela. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms*, 18:3–13, 2013.

[4] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[5] T. M. Chan, K. G. Larsen, and M. Puatracscu. Orthogonal range searching on the ram, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[6] T. M. Chan, Y. Nekrich, S. Rahul, and K. Tsakalidis. Orthogonal point location and rectangle stabbing queries in 3-d. *Journal of Computational Geometry*, 13(1), 2022.

[7] D. R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[8] 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.

[9] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[10] G. Franceschini, S. Muthukrishnan, and M. Puatracscu. Radix sorting with no extra space. In *Proc. 15th Annual European Symposium on Algorithms (ESA)*, pages 194–205, 2007.

[11] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214, 1993.

[12] T. Gagie, J. Kärkkäinen, G. Navarro, and S. Puglisi. Colored range queries and document retrieval. *Theoretical Computer Science*, 483:36–50, 2013.

[13] 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.

[14] 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.

[15] 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, 2006.

[16] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[17] Y. Han. Deterministic sorting in $o(n \log \log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004.

[18] W. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. Space-efficient frameworks for top-k string retrieval. *Journal of the ACM*, 61(2):9:1–9:36, 2014.

[19] W.-K. Hon, R. Shah, and S. Thankachan. Towards an optimal space-and-query-time index for top-k document retrieval. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 173–184, 2012.

[20] W.-K. Hon, R. Shah, S. Thankachan, and J. Vitter. Faster compressed top-k document retrieval. In *Proc. 23rd Data Compression Conference (DCC)*, pages 341–350, 2013.

[21] W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 713–722, 2009.

[22] J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2009.

[23] M. Karpinski and Y. Nekrich. Top-k color queries for document retrieval. In *Proc. 22nd Symposium on Discrete Algorithms (SODA)*, pages 401–411, 2011.

[24] D. G. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28:263–276, 1984.

[25] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[26] D. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[27] J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[28] J. I. Munro, G. Navarro, J. S. Nielsen, R. Shah, and S. V. Thankachan. Top-k term-proximity in succinct space. *Algorithmica*, 78(2):379–393, 2017.

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

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

[31] 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.

[32] G. Navarro and Y. Nekrich. Time-optimal top-k document retrieval. *SIAM Journal on Computing*, 46(1):89–113, 2017.

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

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

[35] G. Navarro and S. Thankachan. New space/time tradeoffs for top-k document retrieval on sequences. *Theoretical Computer Science*, 542:83–97, 2014.

[36] M. Pătraşcu. Succincter. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

[37] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[38] M. Pătraşcu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 166–175, 2014.

[39] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[40] D. Tsur. Top-k document retrieval in optimal space. *Information Processing Letters*, 113(12):440–443, 2013.

[41] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

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

A Geometric Data Structure for Reporting Heaviest Points

In this section we describe a data structure for reporting k heaviest points in an orthogonal range.

Lemma A.1. *There exists a data structure that uses $O(m \log^2 n)$ bits of space to store m points in $[1..n]^2$, for $m \leq n$, and reports k heaviest points in a three-sided range in time $O(k + \log \log n)$. There exists a data structure that uses $O(m \log^3 n)$ bits of space and reports k heaviest points in a general (four-sided) range in time*

$O(k + \log\log n)$.

Proof. We construct a range tree on x-coordinates: all points are stored in leaf nodes in increasing order of their x-coordinates (every leaf contains one point). Every internal node u is associated with a point set $S(u)$. $S(u)$ consists of all points stored in leaf descendants of u. Given a query range $[a, b] \times (-\infty, c]$, we find the leaves that hold the predecessor of b and the successor of a. Then we find the lowest common ancestor u of these two leaves. Let u_l and u_r denote the left and right children of u. All points with x-coordinates in $[a, b]$ can be divided into two groups: points in $S(u_l)$ that satisfy $p.x \geq a$ and points in $S(u_r)$ that satisfy $p.x \leq b$. Hence we can find the k heaviest points in $[a, b] \times (-\infty, c]$ by (1) finding k heaviest points in $([a, +\infty) \times (-\infty, c]) \cap S(u_l)$, (2) finding k heaviest points in $((-\infty, b] \times (-\infty, c]) \cap S(u_r)$, and (3) merging the lists of points obtained in steps (1) and (2) and reporting the k points with the largest weights. The total query time is $O(\log\log n + k)$. Points (1) and (2) can be done in time $O(k + \log\log n)$ and linear space [6, Thm. 7]. The space usage is $O(m \log n \log m)$ bits because each point p is stored in $O(\log m)$ nodes of the range tree.

Using the same method we can transform a data structure for three-sided queries into a data structure for four-sided queries at a cost of increasing the space usage by $O(\log m)$ factor. Since $m \leq n$ the lemma follows. ∎

We now show that the space usage of the data structure for three-sided queries can be reduced to $O(m \log n \log^{\varepsilon} m)$ by using the non-uniform grid approach [5, Sec. 3].

Lemma A.2. *Suppose that there exists a data structure that uses $O(m \log n \log^{1+1/f} n)$ bits to store m points in $[1..n]^2$, with $m \leq n$, and can report k heaviest points in a three-sided range in time $O(k + \log\log n)$ for some constant $f \geq 1$. Then there exists a data structure that uses $O(m \log n \log^{1+1/(f+1)} n)$ bits and can report k heaviest points in a three-sided range in time $O(k + \log\log n)$.*

Proof. All points are stored in a range tree, but every internal node has t children for a a parameter t that will be defined later. For every internal node u we store all points of $S(u)$ in the following data structures:
(1) A data structure that can report k heaviest points in a two-sided range of the form $(-\infty, a] \times (-\infty, b]$ or $[a, +\infty) \times (-\infty, b]$.
(2) We divide the set $S(u)$ into $O(|S(u)|/t^4)$ horizontal slabs (rows) so that every row contains $O(t^4)$ points from $S(u)$. Additionally we divide the grid into $O(t)$ vertical slabs (columns) so that the j-th column contains all points from $S(u_j)$ where u_j is the j-th child of u. Thus every vertical slab contains $O(|S(u)|/t)$ points. Let a cell denote an intersection of a column and a row. There are $O(|S(u)|/t^3)$ cells. A data structure $D_t(u)$ contains t^2 heaviest points from each cell. This data structure reports k heaviest points and is implemented as in Lemma A.1. For every row R, we also keep all its points in a data structure for three-sided heaviest-points queries that uses $O(n \log^{1+1/f} t)$ bits and has $O(\log\log n + k)$ query time. Points in R are reduced to rank space (i.e., coordinates of points are positive integers bounded by $|R|$). Existence of such a data structure is the pre-condition of this Lemma.
(3) A data structure that supports one-dimensional range counting queries with weight restrictions: Given two parameters a and w, find the number of points in $S(u)$ with $p.y \leq a$ such that the weight of p does not exceed w. Such queries are equivalent to two-dimensional range counting queries. Hence the data structure uses $O(|S(u)| \log n)$ bits and supports queries in $O(\log n)$ time.

To report k heaviest points in a range, we find the node u such that u is the lowest common ancestor of leaves l_a and l_b that hold the successor of a and the predecessor of b respectively. We find k heaviest points in $S(u_r) \cap ((-\infty, b] \times (-\infty, c])$ where u_r is the child of u that is an ancestor of l_b. We find k heaviest points in $S(u_l) \cap ([a, +\infty) \times (-\infty, c])$ where u_l is the child of u that is an ancestor of l_a. Using the method described below, we find the k heaviest points in $S(u_{l+1})$, …, $S(u_{r-1})$ such that their y-coordinates do not exceed c. Finally we merge the three lists and find k heaviest points in $O(k)$ time using the linear-time selection algorithm.

It remains to show how we find the k heaviest points in $S(u_{l+1})$, …, $S(u_{r-1})$ such that their y-coordinates do not exceed c. This is equivalent to finding the k heaviest points in the range $Q' = [a', b'] \times (-\infty, c]$ where a' is the x-coordinate of the point in the leftmost leaf descendant of u_{l+1} and b' is the x-coordinate of the point in the rightmost leaf descendant of u_{r-1}. If $k \leq t^2$, we divide Q' into two sub-ranges, Q_1 and Q_2 and find the k heaviest points in both ranges. Let R denote the horizontal slab of $S(u)$ that contains c and let c' denote the smallest y-coordinate of a point in R. Then $Q_1 = [a', b'] \times (-\infty, c' - 1]$ and $Q_2 = [a', b'] \times [c', c]$ We can answer a query Q_1 using the data structure $D_t(u)$. The query Q_2 is entirely contained in one horizontal slab R of $S(u)$. Hence $Q_2 = [a', b'] \times [c', c] = ([a', b'] \times (-\infty, c]) \cap R$ and we can answer the query Q_2 on $S(u)$ using the data structure for three-sided queries on R. We merge the answers to queries Q_1 and Q_2 and find the k heaviest points in Q' using selection algorithm in $O(k)$ time. Since queries on Q_1 and Q_2 are answered in $O(\log\log n + k)$ time, the total query time is $O(\log\log + k)$. If $k > t^2$, we find a value $w_{\max}$ such that there are exactly k points with weight at most $w_{\max}$ in Q'. For every value x we can find the number of points in Q' with weight at most x in time $O(t \log n)$: each point from Q' is stored in a set $S(u_i)$ for $l < i < r$; hence we can calculate the number of points in Q' with weight at most x by answering $O(t)$ range counting queries with weight restriction. Using binary search on the weights of points in $S(u)$, we can find the w_{max} in $O(t \log^2 n)$ time. When $w_{\max}$ is found, reporting k heaviest points in Q' is equivalent to reporting all points in Q' with weight at most $w_{\max}$. The latter task can be accomplished by answering a three-dimensional four-sided query (on the global set of points). Using the data structure from [5], we can answer such queries in time $O(\log\log n + k)$. Since $k > t^2$, the total query time is $O(t \log^2 n + k + \log\log n) = O(k)$ provided that $t \geq \log^2 n$.

All data structures on the set $S(u)$ use $O(n(\log n + \log^{1+1/f} t)$ bits of space: Every point is stored in one row. A row data structure stores $O(t)$ points and uses $O(\log^{1+1/f} t)$ bits per point. Hence all row data structures for a set $S(u)$ use $O(|S(u)|(\log n + \log^{1+1/f} t)$ bits. The data structure for counting queries with weight restrictions and the data structure for reporting heaviest points in a two-sided range use $O(|S(u)| \log n)$ bits. Thus the total space usage of all data structures associated with a node u of the range tree is $O(|S(u)|(\log n + \log^{1+1/f} t))$ bits. We set $t = 2^\alpha$ for $\alpha = \log^{f/(f+1)} n$ so that

$$\log^{1+1/f} t = (\log t)^{\frac{f+1}{f}} = \alpha^{\frac{f+1}{f}} = (\log n)^{\frac{f}{f+1}\frac{f+1}{f}} = \log n$$

Hence all data structures associated with a node u use $O(|S(u)| \log n)$ bits. The height of the range tree with node degree t is $O(\log_t n)$; hence every point is stored $O(\log_t n)$ times. Since

$$\log_t n = \frac{\log n}{\alpha} = \log^{1-f/(f+1)} n = \log^{1/(f+1)} n\,,$$

the overall space usage is $O(n \log n \log^{1/(f+1)} n)$ bits. ◻

Lemma A.3. *Given m points in $[1..n]^2$, where $m \leq n$, there exists a data structure that uses $O(m \log^{1+\varepsilon} n)$ bits and supports three-sided heaviest points queries in time $O(k + \log\log n)$.*

Proof. For any integer constant g there exists a data structure that uses $O(n \log^{1+1/g} n)$ bits and supports queries in $O(k + \log\log n)$ time. To prove this we start with the data structure from Lemma A.1 and apply Lemma A.2 $(g-1)$ times. The statement of this Lemma follows if we set $g = \lceil 1/\varepsilon \rceil$. ◻

We can prove analogues of Lemma A.2 and Lemma A.4 for four-sided queries using exactly the same method.

Lemma A.4. *There exists a data structure that stores m points in $[1..n]^2$, with $m \leq n$, using $O(m \log^{1+\varepsilon} n)$ bits and supports four-sided heaviest points queries in time $O(k + \log\log n)$.*

Finally, we show how this result extends to three dimensions.

Lemma A.5. *There exists a data structure that stores m points in $[1..n]^3$, with $m \leq n$, using $O(m \log^{2+\varepsilon} n)$ bits and supports heaviest points queries in time $O(k + \log n)$.*

Proof. We use range trees on the third coordinate, with node degree $\log^\varepsilon n$, where for every range of children $i, \ldots, j$ we store the data structure of Lemma A.4. Since each element is stored $O(\log^{2\varepsilon} n \log m)$ times in those structures, the total space is $O(m \log^{2+3\varepsilon} n)$; we replace ε by $\varepsilon/3$ to obtain the claimed space.

The query for a range $[a, b] \times [c, d] \times [e, f]$ proceeds as follows: we find in the range tree the first and last leaves, l_1 and l_2, included in $[e, f]$. We take the lowest common ancestor u of l_1 and l_2. For every node v on the path from l_1 to u (resp. on the path from l_2 to u), we consider the set of its children $v_{j(v)}, \ldots, v_{j'(v)}$, such that all points in $S(v_{j(v)}) \cup \ldots \cup S(v_{j'(v)})$ have their third coordinate in $[e, f]$. Using the data structure for $S(v_{j(v)}) \cup \ldots \cup S(v_{j'(v)})$, we can generate the list $L_H(v)$ of k' heaviest points p that are stored in $v_{j(v)}, \ldots, v_{j'(v)}$ and whose projections on the first two coordinates are in $[a,b] \times [c,d]$. By Lemma A.4, this can be done in time $O(k' + \log\log n)$ for any $k' \leq k$. Moreover, if the third coordinate of some point p is in $[e,f]$, then p is stored in exactly one node v_r, so that $j(v) \leq r \leq j'(v)$ and the parent node v of v_r is on the path from l_1 to v or on the path from l_2 to v. Hence, we can obtain the list of k heaviest points in $[a,b] \times [c,d] \times [e,f]$ by merging (the prefixes of) lists $L_H(v)$. Every list $L_H(v)$ or its length-k' prefix can be generated in time $O(k' + \log\log n)$ by Lemma A.4. As there are $O(\log m / \log\log n)$ nodes v in the paths, using the same method as in Section 7.3, we can also obtain the list of k heaviest points from all lists $L_H(v)$ in time $O(k + \log n)$. $\square$