Time-Optimal Top-k Document Retrieval*

Gonzalo Navarro
Department of Computer Science
University of Chile, Chile
gnavarro@dcc.uchile.cl

Yakov Nekrich
Department of Computer Science
University of Kansas, USA
yakov.nekrich@googlemail.com

Abstract

Let $\mathcal{D}$ be a collection of D documents, which are strings over an alphabet of size σ, of total length n. We describe a data structure that uses linear space and and reports k most relevant documents that contain a query pattern P, which is a string of length p packed in $p/\log_\sigma n$ words, in time $O(p/\log_\sigma n + k)$. This is optimal in the RAM model in the general case where $\log D = \Theta(\log n)$, and involves a novel RAM-optimal suffix tree search. Our construction supports an ample set of important relevance measures, such as the number of times P appears in a document (called term frequency), a fixed document importance, and the minimal distance between two occurrences of P in a document.

When $\log D = o(\log n)$, we show how to reduce the space of the data structure from $O(n \log n)$ to $O(n(\log \sigma + \log D + \log\log n))$ bits, and to $O(n(\log \sigma + \log D))$ bits in the case of the popular term frequency measure of relevance, at the price of an additive term $O(\log^\varepsilon_\sigma n)$ in the query time, for any constant $\varepsilon > 0$.

We also consider the dynamic scenario, where documents can be inserted and deleted from the collection. We obtain linear space and query time $O(p(\log\log n)^2/\log_\sigma n + \log n + k\log\log k)$, whereas insertions and deletions require $O(\log^{1+\varepsilon} n)$ time per symbol, for any constant $\varepsilon > 0$.

Finally, we consider an extended static scenario where an extra parameter $\mathtt{par}(P, d)$ is defined, and the query must retrieve only documents d such that $\mathtt{par}(P, d) \in [\tau_1, \tau_2]$, where this range is specified at query time. We solve these queries using linear space and $O(p/\log_\sigma n + \log^{1+\varepsilon} n + k\log^\varepsilon n)$ time, for any constant $\varepsilon > 0$.

Our technique is to translate these top-k problems into multidimensional geometric search problems. As a bonus, we describe some improvements to those problems.

1 Introduction

The design of efficient data structures for document (i.e., string) collections that can report those containing a query pattern P is an important problem studied in the information retrieval and pattern matching communities (see, e.g., a recent survey [52]). Due to the steadily increasing volumes of data, it is often necessary to generate a list $L(P)$ of the documents containing a string pattern P in decreasing order of relevance. Since the list $L(P)$ can be very large, in most cases we

*Partially funded by Fondecyt Grant 1-140796, Chile, and by Millennium Nucleus Information and Coordination in Networks ICM/FIC RC130003, Chile. An early partial version of this article appeared in *Proc. SODA 2012* [53].

are interested in answering *top-k queries*, that is, reporting only the first k documents from $L(P)$ for a parameter k given at query time.

Inverted files [15, 43, 9] that store lists of documents containing certain keywords are frequently used in practical implementations of information retrieval methods. However, inverted files only work when query patterns belong to a fixed pre-defined set of strings (keywords). The suffix tree [71], a handbook data structure known since 1973, uses linear space (i.e., $O(n)$ words, where n is the total length of all the documents) and finds all the *occ* occurrences of a pattern P in $O(p + occ)$ time, where $p = |P|$. Surprisingly, the general *document listing problem*, that is, the problem of reporting all the documents that contain an arbitrary query pattern P, was not studied until the end of the 90s. Suffix trees and other data structures for standard pattern matching queries do not provide a satisfactory solution for the document listing problem because the same document may contain many occurrences of P. Matias et al. [47] described the first data structure for document listing queries; their structure uses $O(n)$ words of space and reports all *docc* documents that contain P in $O(p \log D + docc)$ time, where D is the total number of documents in the collection. Muthukrishnan [51] presented a data structure that uses $O(n)$ words of space and answers document listing queries in $O(p + docc)$ time. Muthukrishnan [51] also initiated the study of more sophisticated problems in which only documents that contain P and satisfy some further criteria are reported. In the K-mining problem, we must report documents in which P occurs at least K times; in the K-repeats problem, we must report documents in which at least two occurrences of P are within a distance K. He described $O(n)$- and an $O(n \log n)$-word data structures that answer K-mine and K-repeats queries, respectively, both in $O(p + occ)$ time, where *occ* is the number of reported documents.

A problem not addressed by Muthukrishnan, and arguably the most important one for information retrieval, is the *top-k document retrieval* problem: report k most highly ranked documents for a query pattern P in decreasing order of their ranks. The ranking is measured with respect to the so-called *relevance* of a string P for a document d. A basic relevance measure is $tf(P, d)$, the number of times P occurs in d. Two other important examples are $mindist(P, d)$, the minimum distance between two occurrences of P in d, and $docrank(d)$, an arbitrary static rank assigned to a document d. Some more complex measures have also been proposed. Hon et al. [37] presented a solution for the top-k document retrieval problem for the case when the relevance measure is $tf(P, d)$. Their data structure uses $O(n \log n)$ words of space and answers queries in $O(p + k + \log n \log \log n)$ time. Later, Hon, Shah and Vitter [41] presented a general solution for a wide class of relevance measures. Their data structure uses linear space and needs $O(p + k \log k)$ time to answer a top-k query. A recent $O(n)$ space data structure [42] enables us to answer top-k queries in $O(p + k)$ time when the relevance measure is $docrank(d)$. However, that result cannot be extended to other more important relevance measures.

Our Results. Hon et al.'s results [41] are an important achievement, but their time is not yet optimal. In this paper we describe a linear space data structure that answers top-k document queries in $O(p/\log_\sigma n + k)$ time, where σ is the alphabet size of the collection and P comes packed in $p/\log_\sigma n$ words. This is optimal in the $\Theta(\log n)$-word RAM model we use, unless the collection has very few documents, that is, $\log D = o(\log n)$ (i.e., $D = o(n^\varepsilon)$ for any constant $\varepsilon > 0$). We support the same relevance measures as Hon et al. [41].

Theorem 1 *Let $\mathcal{D}$ be a collection of strings (called* documents*) of total length n over an integer alphabet $[1, \sigma]$, and let $w(S, d)$ be a function that assigns a numeric* weight *to string S in document d, so that $w(S, d)$ depends only on the set of starting positions of occurrences of S in d. Then there exists an $O(n)$-word space data structure that, given a string P of length p and an integer k, reports k documents d containing P with highest $w(P, d)$ values, in decreasing order of $w(P, d)$, in $O(p/\log_\sigma n + k)$ time. The time is online in k.*

Note that the weighting function is general enough to encompass measures $tf(P, d)$, $mindist(P, d)$ and $docrank(d)$. As stated, our solution is *online* in k: It is not necessary to specify k beforehand; our data structure can simply report documents in decreasing relevance order until all the documents are reported or the query processing is terminated by a user.

An online top-k solution using the *tf* measure solves the K-mining problem in optimal time and linear space. An online top-k solution using *mindist* measure solves the K-repeats problem in optimal time and linear space. We remind that Muthukrishnan [51] had solved the K-repeats problem using $O(n \log n)$-word space; later Hon et al. [41] reduced the space to linear. Now all these results appear as a natural corollary of our optimal top-k retrieval solution. Our results also subsume those on more recent variants of the problem [42], for example when the rank $docrank(d)$ depends only on d (we just use $w(P, d) = docrank(d)$), or where in addition we exclude those d where P appears less than K times for a fixed pre-defined K (we just use $w(P, d) = docrank(d)$ if $tf(P, d) \geq K$, or 0 otherwise).

Moreover, we can also answer queries for some relevance metrics not included in Theorem 1. For instance, we might be interested in reporting all the documents d with $tf(P, d) \times idf(P) \geq \tau$, where $idf(P) = \log(N/df(P))$ and $df(P)$ is the number of documents where P appears [9]. Using the $O(n)$-bit structure of Sadakane [63], we can compute $idf(P)$ in $O(1)$ time from the suffix tree locus of P. To answer the query, we use our data structure of Theorem 1 in online mode on measure *tf*: For every reported document d we find $tf(P, d)$ and compute $tf(P, d) \times idf(P)$; the procedure is terminated when a document d_l with $tf(P, d_l) \times idf(P) < \tau$ is encountered. Thus we need $O(p/\log_\sigma n + occ)$ time to report all occ documents with $tf \times idf$ scores above a threshold.

When $\log D = o(\log n)$, it is not clear that our time is RAM-optimal. Instead, we show that in this case the space of our data structures can be reduced from $O(n \log n)$ bits to $O(n(\log \sigma + \log D + \log\log n))$. This is $o(n \log n)$ bits unless $\log \sigma = \Theta(\log n)$ (in which case the linear-space data structure is already asymptotically optimal). For the most important *tf* relevance measure, where we report documents in which P occurs most frequently, we obtain a data structure that uses $O(n(\log \sigma + \log D))$ bits of space. The price of the space reduction is an additive term $O(\log^\varepsilon_\sigma n)$ in the query time, for any constant $\varepsilon > 0$.

We also consider the dynamic framework, where collection $\mathcal{D}$ admits insertions of new documents and deletions of existing documents. Those updates are supported in slightly superlogarithmic time per character, whereas the query times are only slightly slowed down. We note that measure C_w is just $O(1)$ for the typical relevance measures *tf* and *docrank*, and $O(\log n)$ for *mindist*.

Theorem 2 *Let $\mathcal{D}$ be a collection of documents of total length n over an integer alphabet $[1, \sigma]$, and let $w(S, d)$ be a function that assigns a numeric* weight *to string S in document d, so that $w(S, d)$ depends only on the set of starting positions of occurrences of S in d, and can be computed*

in $O(C_w|d|)$ time for all the nodes of the suffix tree of document d. Then there exists an $O(n)$-word space data structure that, given a string P of length p and an integer k, reports k documents d containing P with highest $w(P,d)$ values, in decreasing order of $w(P,d)$, in $O(p(\log\log n)^2/\log_\sigma n + \log n + k\log\log k)$ time, online in k. The structure can insert new documents and delete existing documents in $O(C_w + \log^{1+\varepsilon} n)$ time per inserted character and $O(\log^{1+\varepsilon} n)$ per deleted character, for any constant $\varepsilon > 0$.

We note that a direct dynamic implementation of the solution of Hon et al. [41] would require at least performing $p + k$ dynamic RMQs, which cost $\Omega(\log n/\log\log n)$ time [2]. Thus modeling the original problem as a geometric one pays off in the dynamic scenario as well.

Furthermore, we can extend the top-k ranked retrieval problem by allowing a further parameter $\texttt{par}(P,d)$ to be associated with any pattern P and document d, so that only documents with $\texttt{par}(P,d) \in [\tau_1,\tau_2]$ are considered. Some applications are selecting a range of creation dates, lengths, or PageRank values for the documents (these do not depend on P), bounding the allowed number of occurrences of P in d, or the minimum distance between two occurrences of P in d, etc.

Theorem 3 *Let $\mathcal{D}$ be a collection of documents of total length n over an integer alphabet $[1,\sigma]$, let $w(S,d)$ be a function that assigns a numeric* weight *to string S in document d, and let $\texttt{par}(S,d)$ be another parameter, so that w and $\texttt{par}$ depend only on the set of starting positions of occurrences of S in d. Then there exists an $O(n)$-word space data structure that, given a string P of length p, an integer k, and a range $[\tau_1,\tau_2]$, reports k documents d containing P and with $\texttt{par}(P,d) \in [\tau_1,\tau_2]$, with highest $w(P,d)$ values, in decreasing order of $w(P,d)$, in $O(p/\log_\sigma n + \log^{1+\varepsilon} n + k\log^{\varepsilon} n)$ time, online in k, for any constant $\varepsilon > 0$.*

Our solutions map these document retrieval problems into range search problems on multi-dimensional spaces, where points in the grids have associated weights. We improve some of the existing solutions for those problems.

An early partial version of this article appeared in *Proc. SODA 2012* [53]. It included basically the $O(p + k)$-time solution for the static case. This extended version includes, apart from more precise explanations and fixes, the improvement of the static result to achieve RAM-optimality on the suffix tree traversal, and the new results on the dynamic scenario. The paper is organized as follows. In Section 2 we review the top-k framework of Hon et al. [41] and reinterpret it as the combination of a suffix tree search plus a geometric search problem. We introduce a RAM-optimal suffix tree traversal technique that is of independent interest, and state our results on geometric grids, each of which is related to the results we achieve on document retrieval. Those can also be of independent interest. Sections 3 and 4 describe our basic static solution. Section 5 describes our dynamic solution. In Section 6 we show how the static solution can be modified to reduce its space requirements, and in Section 7 we show how it can be extended to support an additional restriction on the documents sought. Finally, Section 8 concludes and gives future work directions.

2 Top-k Framework

In this section we overview the framework of Hon, Shah, and Vitter [41]. Then, we describe a geometric interpretation of their structure and show how top-k queries can be reduced to a special case of range reporting queries on a grid.

Let T be the generalized suffix tree [71, 48, 69] for a collection of documents $d_1, \ldots, d_D$, each ending with the special terminator symbol "\$". T is a compact trie, such that all suffixes of all documents are stored in the leaves of T. We denote by path(v) the string obtained by concatenating the labels of all the edges on the path from the root to v. The *locus* of a string P is the highest node v such that P is a prefix of path(v). Every occurrence of P corresponds to a unique leaf that descends from its locus. We refer the reader to classical books and surveys [7, 36, 20] for an extensive description of this data structure.

We say that a leaf l is *marked* with document d if the suffix stored in l belongs to d. An internal node v is marked with d if at least two children of v contain leaves marked with d. While a leaf is marked with only one value d (equal suffixes of distinct documents are distinguished by taking all the string terminators as different from each other and ordering them arbitrarily), an internal node can be marked with many values d.

In every node v of T marked with d, we store a pointer $\mathsf{ptr}(v, d)$ to its lowest ancestor u such that u is also marked with d. If no ancestor u of v is marked with d, then $\mathsf{ptr}(v, d)$ points to a dummy node ν such that ν is the parent of the root of T. We also assign a weight to every pointer $\mathsf{ptr}(v, d)$. This weight is the relevance score of the document d with respect to the string path(v).

It is not hard to see that the nodes of T marked with d correspond precisely to the suffix tree T_d of document d, with the pointers $\mathsf{ptr}(v, d)$ playing the role of parent pointers in T_d. All the nodes of T_d in T are lowest common ancestors of consecutive leaves v_i and v_{i+1} of leaves marked with d in T, $\mathrm{lca}(v_i, v_{i+1})$. The following statements hold; we reprove them for completeness.

Lemma 1 ([41, Lem. 4]) *The total number of pointers* $\mathsf{ptr}(\cdot, \cdot)$ *in* T *is bounded by* $O(n)$.

Proof: The total number of pointers $\mathsf{ptr}(v, d)$, for all $v \in T$, does not exceed the number of nodes marked with d. The total number of internal nodes marked with d is smaller than the number of leaves marked with d. Since there are $O(|d|)$ leaves marked with d, the total number of pointers $\mathsf{ptr}(v, d)$ for a fixed document d is bounded by $O(|d|)$, and those $|d|$ add up to n. □

Lemma 2 ([41, Lem. 2]) *Assume that document* d *contains a pattern* P *and* v *is the locus of* P. *Then there exists a unique pointer* $\mathsf{ptr}(u, d)$, *such that* u *is in the subtree of* v *(which includes* v*) and* $\mathsf{ptr}(u, d)$ *points to an ancestor of* v.

Proof: If d contains P then there is at least one leaf u marked d below the locus of P, with a pointer $\mathsf{ptr}(u, d)$. If there are two maximal (in the sense of ancestorship) nodes u and u' below v with pointers $\mathsf{ptr}(u, d)$ and $\mathsf{ptr}(u', d)$, then their lowest common ancestor v' is also marked. Since v is an ancestor of u and u', v is v' or an ancestor of v' and then $\mathsf{ptr}(u, d)$ and $\mathsf{ptr}(u', d)$ must point to v', not to an ancestor of v. Therefore, there is a unique maximal node u marked d below v (note that u might be v). Thus, the pointer $\mathsf{ptr}(u, d)$ must point to an ancestor of v. □

Moreover, in terms of Lemma 2, it turns out that path(u) occurs in d at the same positions as path(v). Note that the starting positions of P and of path(v) in d are the same, since v is the locus of P, and those are the same as the starting positions of path(u) in d. Thus $w(P, d) = w(\mathrm{path}(u), d)$ for any measure $w(\cdot, \cdot)$ considered in Theorem 1.

RAM-optimal suffix tree traversal. To achieve time $O(p)$ for the locus search in the suffix tree while retaining linear space, one needs to organize the children of each node in a perfect hash function (phf) [29]. In order to reduce this time to the RAM-optimal $O(p/\log_\sigma n)$ when P comes packed in $p/\log_\sigma n$ words, we proceed as follows. Let $l(u,v)$ be the concatenation of string labels from node u to its descendant v. We collect in a phf $H(u)$, for the suffix tree root u, all the highest descendants v such that $|l(u,v)| \geq \ell = \log_\sigma n$. Those nodes v are indexed with a key built from the first ℓ symbols of $l(u,v)$ interpreted as a number of $\lg n$ bits (lg stands for $\log_2$). We build recursively phfs for all the identified descendants v. Since each suffix tree node is included in at most one hash table, the total size is $O(n)$ and the total deterministic construction time is $O(n \log\log n)$ [62].

Now P is searched for as follows. We take its first chunk of ℓ characters, interpret it as a number, and query the phf of the root. If no node v is found for that prefix of P, then P is not in the collection. Otherwise, the string depth of v is $\geq \ell$. We check explicitly the extra $|l(u,v)| - \ell$ symbols in the text, by comparing chunks of ℓ symbols. If there is a mismatch, then P does not appear in the collection. If the $|l(u,v)| - \ell$ symbols are sufficient to match all the rest of P, then the locus is v. Otherwise, we continue the search from v with the next ℓ unread symbols of P, and so on, until there are less than ℓ symbols to match from a node v in the remaining suffix P' of P.

At this point we switch to using a *weak prefix search (wps)* data structure [10]: For each node u holding a phf, we also store this wps data structure with all the nodes v that descend from u where $|l(u,v)| < \ell$, using $l(u,v)$ as their key. We must also include the children z of those nodes v, as well as the children z of u, with $|l(u,z)| \geq \ell$, using the first $\ell - 1$ symbols of $l(u,z)$ as their key (note that these nodes z are also stored in the phf structure of u). The wps data structure will return the lexicographic range of the nodes x where P' is a prefix of $l(u,x)$. The first in that range is the locus of P. One detail is that, if there is no such node (i.e., P has no locus), the wps structure returns an arbitrary value, but this can be easily checked in optimal time in the text.

The wps structure requires, for our length $|P'| < \ell$ and in the RAM model, $O(1)$ query time. For the strings of length $< \ell$ we store, the structure uses $O(\sqrt{\log n}\log\log n)$ bits, or $o(1)$ words, per stored node [10, Thm. 6]. Once again, each node is stored only in one wps structure, so the overall extra space is linear. The wps construction takes $O(n \log^\varepsilon n)$ randomized time. It can be made deterministic $O(n\,\mathrm{polylog}(n))$ time by using a phf inside the construction [1]. By replacing the wps structure by layered phfs for $(\log_\sigma n)/2^i$ symbols, we would have an additive term $O(\log\log_\sigma n)$ in the query time.

Geometric interpretation. We index the nodes of T in the following way: The nodes are visited in pre-order; we also initialize an index $i \leftarrow 0$. When a node v is visited, if v is marked with documents $d_{v_1}, \ldots, d_{v_j}$, we assign indexes $i+1, \ldots, i+j$ to v and set $i \leftarrow i+j$. We will denote by $[l_v, r_v]$ the integer interval bounded by the minimal and maximal indexes assigned to v or its descendants. Values l_v and r_v are stored in node v of T. Furthermore, for every d_{v_t}, $1 \leq t \leq j$, there is a pointer $\mathsf{ptr}(v, d_{v_t})$ that points to some ancestor u_t of v. We encode $\mathsf{ptr}(v, d_{v_t})$ as a point $(i+t, \mathrm{depth}(u_t))$, where *depth* denotes the depth of a node; $\mathrm{depth}(\nu) = 0$. Thus every pointer in T is encoded as a two-dimensional point on an integer $O(n) \times O(n)$ grid. The weight of a point p is that of the pointer it encodes. We observe that all the points have different x-coordinates. Thus we obtain a set S of weighted points with different x-coordinates, and each point corresponds to a

unique pointer.

For the final answers we will need to convert the x-coordinates of points found on this grid into document numbers. We store a global array of size $O(n)$ to do this mapping.

Answering queries. Assume that top-k documents containing a pattern P must be reported. We find the locus v of P in $O(p/\log_\sigma n)$ time. By Lemma 2, there is a unique pointer $\mathsf{ptr}(u,d)$, such that u is a descendant of v (or v itself) and $\mathsf{ptr}(u,d)$ points to an ancestor of v, for every document d that contains P. Moreover the weight of that point is $w(P,d)$. Hence, there is a unique point (x,y) with $x \in [l_v, r_v]$ and $y \in [0, \mathrm{depth}(v)-1]$ for every document d that contains P. Therefore, reporting top-k documents is equivalent to the following query: among all the points in the three-sided range $[l_v, r_v] \times [0, \mathrm{depth}(v))$, report k points with highest weights. We will call such queries *three-sided top-k queries*. In Sections 3 and 4 we prove the following result. Theorem 1 is an immediate corollary of it, as $h = \mathrm{depth}(v) - 1$ and $\mathrm{depth}(v) \leq p$, and we can choose $c \geq 1$.

Theorem 4 *A set of n weighted points on an $n \times n$ grid can be stored in a data structure using $O(n)$ words of space, built in $O(n \log n)$ time, so that for any $1 \leq k, h \leq n$ and $1 \leq a \leq b \leq n$, k most highly weighted points in the range $[a,b] \times [0,h]$ can be reported in decreasing order of their weights in $O(h/\log^c n + k)$ time, for any constant c.*

Top-k queries on dynamic collections. The static suffix tree is replaced by a dynamic one, with search time $O(p(\log\log n)^2/\log_\sigma n + \log n)$ and update time $O(\log n)$ per symbol. We must also update the grid, for which we must carry out lowest common ancestor queries on the dynamic suffix tree and also insert/delete points (and columns) in the grid. We split the grid of Theorem 4 into horizontal stripes of height $m = \text{polylog } n$ to obtain improved performance, and query the highest $\lceil p/m \rceil$ of those grids. In the most general case (i.e., the last grid) we carry out a three-sided top-k query. We address this in Section 5, where in particular we prove the following result on dynamic grids. Theorem 2 is then obtained by combining those results.

Theorem 5 *A set of n points, one per column with y-coordinates in $[1,n]$, and with weights in $[1, O(n)]$, can be stored in $O(n)$ words of space, so that for any $1 \leq k \leq n$, $1 \leq h \leq n$ and $1 \leq a \leq b \leq n$, k most highly weighted points in the range $[a,b] \times [0,h]$ can be reported in decreasing order of their weights in $O(h/\log^c n + \log n + k \log\log k)$ time, online in k, for any constant c. Points (and their columns) can be inserted and deleted in $O(\log^{1+\varepsilon} n)$ time, for any constant $\varepsilon > 0$.*

Parameterized top-k queries. We use the same geometric interpretation as described above, but now each pointer $\mathsf{ptr}(v,d)$ is also associated with the parameter value $\mathsf{par}(\mathrm{path}(v), d)$. We encode a pointer $\mathsf{ptr}(v, d_{v_t})$ as a three-dimensional point $(i+t, \mathrm{depth}(u_t), \mathsf{par}(\mathrm{path}(v), d_{v_t}))$, where i, t, and u_t are defined as in the case of nonparameterized top-k queries. All the documents that contain a pattern P (with locus v) and satisfy $\tau_1 \leq \mathsf{par}(P,d) \leq \tau_2$ correspond to unique points in the range $[l_v, r_v] \times [0, \mathrm{depth}(v)) \times [\tau_1, \tau_2]$. Hence, reporting top-$k$ documents with $\mathsf{par}(P,d) \in [\tau_1, \tau_2]$ is equivalent to reporting top-k points in a three-dimensional range. The following result is proved in Section 7, and Theorem 3 is an immediate corollary of it, choosing any $c \geq 1$.

Theorem 6 *A set of n weighted points on an $n \times n \times n$ grid can be stored in $O(n)$ words of space, so that for any $1 \leq k, h \leq n$, $1 \leq a \leq b \leq n$, and $1 \leq \tau_1 \leq \tau_2 \leq n$, k most highly weighted points in the range $[a, b] \times [0, h] \times [\tau_1, \tau_2]$ can be reported in decreasing order of their weights in $O(h/\log^c n + \log^{1+\varepsilon} n + k \log^{\varepsilon} n)$ time, for any constants c and $\varepsilon > 0$.*

3 An $O(m^\epsilon + k)$ Time Data Structure

In this section we give a first data structure for three-sided top-k queries $[a, b] \times [0, h]$ on a set S of n two-dimensional weighted points. It does not yet achieve the desired $O(h/\log n + k)$ time, but its time depends on the width m of the grid. This will be used in Section 4 to handle vertical stripes of the global grid, in order to achieve the final result.

We assume that a global array gives access to the points of S in constant time: if we know the x-coordinate $p.x$ of a point $p \in S$, we can obtain the y-coordinate $p.y$ of p in $O(1)$ time. Both $p.x$ and $p.y$ are in $[1, O(n)]$, thus the global array requires $O(n)$ words of space. We consider the question of how much additional space our data structure uses if this global array is available.

The result of this section is summed up in the following lemma, where we consider tall grids of m columns and n rows. The idea is to partition the points by weights, where the weights are disregarded inside each partition. Those partitions are further refined, forming a multi-ary range tree. Then we solve the problem by traversing the appropriate partitions and collecting all the points using classical range queries on unweighted points. A tree of arity $m^{\Theta(1)}$ yields constant height and thus constant space per point.

Note, however, that this is not just a standard multi-ary range tree. Range trees and similar geometric data structures can provide a solution for the threshold variant of the problem, where we must report all points in a three-sided range with weight exceeding a threshold. We can probably modify these methods and find a solution for the unsorted top-k problem, where k points with highest weights are reported in an arbitrary order. However, in this case we consider the sorted top-k problem, where points must be returned sorted by decreasing weight. It is not clear how this variant can be solved in optimal time and linear space using standard techniques.

Lemma 3 *Assume that $m \leq n$ and let $0 < \epsilon < 1$ be a constant. There exists a data structure that uses $O(m \log m)$ additional bits of space and construction time. It answers three-sided top-k queries for a set of m points on an $m \times n$ grid in $O(m^\epsilon + k)$ time.*

Proof: We partition S into classes $S_1, \ldots, S_r$, where $r = m^{\epsilon'}$ for a constant $0 < \epsilon' < \epsilon$. For any $1 \leq i < j \leq r$, the weight of any point $p_i \in S_i$ is larger than the weight of any point $p_j \in S_j$. For $1 \leq i < r$, S_i contains $m^{1-\epsilon'}$ points. Each class S_i that contains more than one element is recursively divided into $\min(|S_i|, r)$ subclasses in the same manner. This subdivision can be represented as a tree: If S_i is divided into subclasses $S_{i_1}, \ldots, S_{i_k}$, we will say that S_i is the parent of $S_{i_1}, \ldots, S_{i_k}$. This tree has constant height $O(1/\epsilon')$.

For every class S_j we store data structures that support three-sided range counting queries and three-sided range reporting queries, in $O(\log m)$ and $O(\log m + occ)$ time respectively. These structures will be described in Section 3.1 and require $O(m^{1-\epsilon'} \log m)$ construction time; note they do not involve weights. This adds up to $O((1/\epsilon')m \log m)$ construction time.

We will report k most highly weighted points in a three-sided query range $Q = [a, b] \times [0, h]$ using a two-stage procedure. During the first stage we produce an unsorted list L of k most highly weighted points. During the second stage the list L is sorted by weight.

Let Q^k denote the set of k most highly weighted points in $S \cap Q$. Then Q^k can be formed as the union of the result of the three-sided query over certain classes, at most $O(m^{\epsilon'})$ clases per level over a constant number of levels. More precisely, there are $O(m^{\epsilon'})$ classes S_c, such that $p \in Q^k$ if and only if $p \in S_c \cap Q$ for some S_c. During the first stage, we identify the classes S_c and report all the points in $S_c \cap Q$ using the following procedure. Initially, we set our current tree node to $\widetilde{S} = S$ and its child number to $i = 1$. We count the number of points inside Q in the i-th child S_i of $\widetilde{S}$. If $k_i = |S_i \cap Q| \leq k$, we report all the points from $S_i \cap Q$ and set $k = k - k_i$. If $k = 0$, the procedure is completed; otherwise we set $i = i + 1$ and proceed to the next child S_i of $\widetilde{S}$. If $k_i > k$, instead, we set $\widetilde{S} = S_i$, $i = 1$, and report k most highly weighted points in the children of $\widetilde{S}$ using the same procedure. During the first stage we examine $O(m^{\epsilon'})$ classes S_i and spend $O(m^{\epsilon'} \log m + k) = O(m^{\epsilon} + k)$ time in total.

When the list L is completed, we can sort it in $O(m^{\epsilon} + k)$ time. If L contains $k < m^{\epsilon'}$ points, L can be sorted in $O(k \log k) = O(m^{\epsilon})$ time. If L contains $k \geq m^{\epsilon'}$ points, then we can sort it in $O(k)$ time using radix sort: As the set S contains at most m distinct weights, we store their ranks in an array ordered by x-coordinate, and thus can sort the result using the ranks instead of the original values. By sorting $\epsilon' \lg m$ bits per pass the radix sort runs in time $O(k)$.

As for the space, the structures in Lemmas 4 and 5 require $O(\log m)$ bits per point. Each point of S belongs to $O(1/\epsilon')$ classes S_i. Hence, the total number of points in all classes is $O(m)$, giving $O(m \log m)$ bits of total space. The local array of weight ranks also uses $O(m \log m)$ bits. □

3.1 Counting and Reporting Points

It remains to describe the data structures that answer three-sided counting and reporting queries, with no weights involved. These are variants of standard geometric data structures [49, 18] that we adapt to the case of narrow grids, so as to use $O(\log m)$ bits per point instead of $O(\log n)$. We exploit the fact that we can obtain $p.y$ from $p.x$ for any point $p \in S$, and use compact data structures to reduce space.

Lemma 4 *Let $v \leq m \leq n$. There exists a data structure that uses $O(v \log m)$ additional bits and answers three-sided range counting queries for a set of v points on an $m \times n$ grid in $O(\log v)$ time. It can be built in $O(v \log v)$ time.*

Proof: We use the classic rank space technique [30] to reduce the problem of counting on an $m \times n$ grid to the problem of counting on a $v \times n$ grid. We apply ranking only on the x-coordinates, spending $O(\log v)$ additional query time and $v \lg m$ bits of space to perform and to store the mapping, respectively.

We store the mapped points on a variant of the *Wavelet Tree* data structure [34, 18]. Each node of this tree W covers all the points within a range of y-coordinates. The root covers all the nodes, and the two children of each internal node cover half of the points covered by their parent. The leaves cover one point. The y-coordinate limits of the nodes are not stored explicitly, to save space. Instead, we store the x-coordinate of the point holding the maximum y-coordinate in the

node. With the global array we can recover the y-coordinate in constant time. Each internal node v covering r points stores a bitmap $B_v[1..r]$, so that $B_v[i] = 0$ iff the i-th point, in x-coordinate order, belongs to the left child (otherwise it belongs to the right child). Those bitmaps are equipped with data structures answering operation $rank_b(B_v, i)$ in constant time and $r + o(r)$ bits of space [50], where $rank_b(B_v, i)$ is the number of occurrences of bit b in $B_v[1..i]$. Since W has $O(v)$ nodes and height $O(\log v)$, its bitmaps require $O(v \log v)$ bits and its pointers and x-coordinates need $O(v \log m)$ bits. The construction time is $O(v \log v)$.

We can easily answer range counting queries $[a, b] \times [0, h]$ on W [46]. After mapping $[a, b]$ to $[a', b']$ in rank-space, the procedure starts at the root node of W, with the range $[a', b']$ on its bitmap B. This range will become $[a_l, b_l] = [rank_0(B, a' - 1) + 1, rank_0(B, b')]$ on the left child of the root, and $[a_r, b_r] = [rank_1(B, a' - 1) + 1, rank_1(B, b')]$ on the right child. If the maximal y-coordinate of the left child is smaller than or equal to h, we count the number of points p with $p.x \in [a, b]$ stored in the left child, which is simply $b_l - a_l + 1$, and then visit the right child. Otherwise, the maximal y-coordinate in the left child is larger than h, and we just visit the left child. The time is $O(1)$ per tree level. □

Lemma 5 *Let $v \leq m \leq n$. There exists a data structure that uses $O(v \log m)$ additional bits and answers three-sided range reporting queries for a set S of v points on an $m \times n$ grid in $O(\log v + occ)$ time, to report the occ results. It can be built in $O(v \log v)$ time.*

Proof: We again reduce the problem of reporting on an $m \times n$ grid to the problem of reporting on a $v \times n$ grid, using the rank space technique [30]. The query time is increased by $O(\log v)$, and the space usage increases by $v \lg m$ bits. To solve this problem we sort the v points in x-coordinate order, build the sequence $Y[1..v]$ of their y-coordinates, and build a *Range Minimum Query (RMQ)* data structure on Y [27]. This structure requires only $O(v)$ bits of space, does not need to access Y after construction (so we do not store Y), and answers in constant time the query $r = \text{rmq}(x_1, x_2) = \arg\ \min_{x_1 \leq x \leq x_2} Y[x]$ for any x_1, x_2. We then compute $p.y$ for the point p such that the rank-mapped value of $p.x$ is r. If $p.y > h$ we can stop since there are no points to report. Otherwise we report p and continue recursively on the intervals $[a', r - 1]$ and $[r + 1, b']$. It is well known that this procedure retrieves all the *occ* points in the three-sided range $[a', b'] \times [0, h]$ in $O(occ)$ time; see, for example, Muthukrishnan [51]. The construction time is dominated by the sorting of the points. □

4 An Optimal Time Data Structure

The data structure of Lemma 3 gives us an $O(m^\epsilon + k)$ time solution for top-k queries on the three-sided range $[a, b] \times [0, h]$, for any constant ϵ, where m is the grid width. In this section we use it to obtain $O(h/\log n + k)$ time, where n is the number of points. The idea is to partition the space into vertical stripes, for different stripe widths, and index each stripe with Lemma 3. Then the query is run on the partition of width m so that the $O(m^\epsilon)$ time complexity is dominated by $O(h/\log n + k)$. The many partitions take total linear space because the size per point in Lemma 3 is $O(\log m)$, and our widths decrease doubly exponentially. As a query may span several stripes, a structure similar to the one used in the classical RMQ solution [14] is used: we precompute answers for sequences of doubling numbers of stripes, so that any query is covered by two overlapped precomputed answers;

the answers can then be merged. This doubling scheme uses linear space for stripes of width up to $\Omega(\log^2 n)$. Smaller stripes are solved with a smaller-scale replica of the main idea, and then using universal tables. The hierarchical structure of stripes is described in Section 4.1, and the way to query it in Section 4.2. Since the solution to smaller grids replicates the main idea at a smaller scale, it is postponed to Section 4.3.

In addition to the global array storing $p.y$ for each $p.x$, we use another array storing the weight corresponding to each $p.x$. As there are overall $O(n)$ different weights, those can be mapped to the interval $[1, O(n)]$ and still solve correctly any top-k reporting problem. Thus the new global array also requires $O(n)$ words of space.

4.1 Structure

Let $g_j = 1/2^j$ for $j = 0, 1, \ldots, r$. We choose r so that $n^{g_r} = O(1)$, thus $r = O(\log\log n)$. The x-axis is split into intervals of size $\Delta_j = n^{g_j} \lg^2 n$ and $j = 1, \ldots, r$. For convenience, we also define $\Delta_0 = n$ and $\Delta'_j = \Delta_j / \lg^2 n = n^{g_j}$. For every $1 \le j < r$ and for every interval $I_{j,t} = [(t-1)\Delta_j, t\Delta_j - 1]$, we store all the points p with $p.x \in I_{j,t}$ in a data structure $E_{j,t}$ implemented as described in Lemma 3. Then $E_{j,t}$ supports three-sided top-k queries in $O((\Delta_j)^\epsilon + k)$ time for any constant $0 < \epsilon < 1/4$. We also construct a data structure E_0 that contains all the points of S and supports three-sided top-k queries in $O(n^{1/4} + k)$ time. To simplify the description, we also assume that $I_{-1} = I_0 = [0, n-1]$ and $E_{-1} = E_0$.

The data structures $E_{j,t}$ for a fixed j contain $O(n)$ points overall, hence by Lemma 3 all $E_{j,t}$ use $O(n \log \Delta_j) = O(n \log(n^{g_j} \log^2 n)) = (1/2^j)O(n \log n) + O(n \log\log n)$ additional bits of space. Thus all $E_{j,t}$ for $0 \le j < r$ use $\sum_{j=0}^{r-1}[(1/2^j)O(n \log n) + O(n \log\log n)] = O(n \log n)$ bits, or $O(n)$ words. They also require $O(n \log n)$ total construction time. Since $\epsilon < 1/4$, a data structure $E_{j,t}$ supports top-k queries in time $O((\Delta_j)^\epsilon + k) = O((n^{g_j} \log^2 n)^\epsilon + k) = O(n^{g_{j+2}} / \log n + k) = O(\Delta'_{j+2} / \log n + k)$ time. For each of the smallest intervals $I_{r,t}$ we store data structures $\overline{E}_{r,t}$ that use $o(\log^2 n)$ words of space (adding up to $o(n)$) and support three-sided top-k queries in $O(h / \log n + k)$ time. This structure will be described in Section 4.3.

Note that our choice of writing $(n^{g_j} \lg^2 n)^\epsilon = O(n^{g_{j+2}} / \log n)$ was arbitrary, because ϵ is strictly less than $1/4$. We could have written $(n^{g_j} \lg^2 n)^\epsilon = O(n^{g_{j+2}} / \log^c n)$ for any constant c, and this would yield $O(h / \log^c n + k)$ query time. We have chosen to favor simplicity in the exposition, but will return to this point at the end of Section 4.3.

We also store structures to answer top-k queries on selected ranges of intervals. For $1 \le j \le r$, we consider the endpoints of intervals $I_{j,t}$. Let $\mathsf{top}_j(a, b, c, k)$ denote the list of top-k points in the range $[a \cdot \Delta_j, b \cdot \Delta_j - 1] \times [0, c]$ in descending weight order. We store the values of $\mathsf{top}_j(t, t + 2^v, c, \Delta'_{j+1})$ for any $t \in [0, n/\Delta_j]$, any $0 \le v \le \lg(n/\Delta_j)$, and any $0 \le c \le \Delta'_{j+1}$. All the lists $\mathsf{top}_j(\cdot, \cdot, \cdot, \cdot)$ use space $O((n/\Delta_j)(\Delta'_{j+1})^2 \log n) = O(n / \log n)$ words. Hence the total word space usage of all lists $\mathsf{top}_j(\cdot, \cdot, \cdot, \cdot)$ for $2 \le j \le r$ is $O(n \log\log n / \log n) = o(n)$. It can also be built in $o(n)$ time using dynamic programming.

4.2 Queries

We can carry out the query using a range of intervals $I_{j,t}$ of any width Δ_j. The key idea is to use a j value according to the height of the three-sided query, so that the search time in $I_{j,t}$ gives

the desired $O(h/\log n)$ time. More precisely, assume we want to report top-k points in the range $[a,b]\times[0,h]$. First, we find the index j such that $\Delta'_{j+1} > \max(h,k) \geq \Delta'_{j+2}$. The index j can be found in $O(\log\log(h+k))$ time by linear search[1]. If $[a,b]$ is contained in some interval $I_{j,t}$, then we can answer a query in $O(\Delta'_{j+2}/\log n + k) = O(h/\log n + k)$ time using $E_{j,t}$. If $[a,b]$ is contained in two adjacent intervals $I_{j,t}$ and $I_{j,t+1}$, we generate the lists of top-k points in $([a,b]\cap I_{j,t})\times[0,h]$ and $([a,b]\cap I_{j,t+1})\times[0,h]$ in $O(h/\log n + k)$ time, and merge them in $O(k)$ time.

To deal with the case when $[a,b]$ spans one or more intervals $I_{j,t}$, we use the pre-computed solutions for ranges of intervals. Assume that $[a,b]$ spans intervals $I_{j,t_1+1},\ldots,I_{j,t_2-1}$; $[a,b]$ also intersects with intervals I_{j,t_1} and I_{j,t_2}. Let $a'\Delta_j$ and $b'\Delta_j$ denote the left endpoints of I_{j,t_1+1} and I_{j,t_2}, respectively. The list L_m of top-k points in $[a'\Delta_j, b'\Delta_j - 1]\times[0,h]$ can be generated as follows. Intervals $[a'\Delta_j, (a'+2^v)\Delta_j - 1]$ and $[(b'-2^v)\Delta_j, b'\Delta_j - 1]$ for $v = \lfloor \lg(b'-a')\rfloor$ cover $[a'\Delta_j, b'\Delta_j - 1]$. Let L'_m and L''_m denote the lists of the first k points in $\mathsf{top}_j(a', a'+2^v, h, \Delta'_{j+1})$ and $\mathsf{top}_j(b'-2^v, b', h, \Delta'_{j+1})$ (we have k results because $k < \Delta'_{j+1}$; similarly we have the results for $c = h$ because $h < \Delta'_{j+1}$). We merge both lists (possibly removing duplicates) according to the weights of the points, and store in L_m the set of the first k points from the merged list. Let L_{t_1} and L_{t_2} denote the sets of top-k points in $[a, a'\Delta_j - 1]\times[0,h]$ and $[b'\Delta_j, b]\times[0,h]$. We can obtain L_{t_1} and L_{t_2} in $O(h/\log n + k)$ time using data structures E_{j,t_1} and E_{j,t_2} as explained above. Finally, we can merge L_m, L_{t_1}, and L_{t_2} in $O(k)$ time; the first k points in the resulting list are the top-k points in $[a,b]\times[0,h]$.

4.3 A Data Structure for an $O(\log^2 n)\times n$ Grid.

The data structures $\overline{E}_{r,t}$ for an interval $I_{r,t}$ use the same general approach as the data structures $E_{j,t}$, at a smaller scale. Note that these structures will be consulted only when $\max(h,k) < C = \Delta'_{r+1} = O(1)$. Each interval $I_{r,t}$ is subdivided into $\lg^{7/4} n$ intervals $\widetilde{I}_1, \widetilde{I}_2, \ldots$ of width $\lg^{1/4} n$. Let $\widetilde{S}$ denote the set that contains the endpoints of $\widetilde{I}_1, \widetilde{I}_2, \ldots$. For every $x\in\widetilde{S}$, each $1\leq v\leq 2\lg\lg n$ and each $h\leq C$, we store the lists $\mathsf{top}(x, x+2^v, h, C)$. All such lists use $O((n/\log^2 n)C^2\log^{7/4} n\log\log n) = o(n)$ space in total.

A query on $I_{r,t}$ is processed as follows. Suppose that $[a,b]$ intersects with intervals $\widetilde{I}_{g_1},\ldots,\widetilde{I}_{g_2}$ for some $g_1\leq g_2$. We find the top-k points from $(\widetilde{I}_{g_1+1}\cup\ldots\cup\widetilde{I}_{g_2-1})\times[0,h]$ in $O(k)$ time using lists $\mathsf{top}(\cdot,\cdot,\cdot,\cdot)$, as before. We also find top-k points from $(\widetilde{I}_{g_1}\cap[a,b])\times[0,h]$ and $(\widetilde{I}_{g_2}\cap[a,b])\times[0,h]$ in $O(k)$ time using data structures for $\widetilde{I}_{g_1}$ and $\widetilde{I}_{g_2}$, respectively, to be described next. We thus obtain three lists of points sorted by their weights, and merge them in $O(k)$ time as before.

Finally, we describe how to answer queries in $O(k)$ time in the grids $\widetilde{I}_g$, of width $\lg^{1/4} n$. We apply reduction to rank space [30] on the y-coordinates of points and on their weights. The resulting sequence X_g contains all mapped points in $\widetilde{I}_g$ and consists of $O(\log^{1/4} n\log\log n)$ bits, so all the descriptions of all sequences X_g require $O(n\log\log n)$ bits, or $o(n)$ words. There are $O(\log^{1/2} n)$ queries that can be asked (considering all the sensible values of $[a,b]$, h and k), and the answers require $O(k\log(\log^{1/4} n)) = O(\log\log n)$ bits. Thus we can store a universal look-up table of size $2^{O(\log^{3/4} n\log\log n)}O(\log\log n) = o(n)$ words common to all subintervals $\widetilde{I}_g$. This table contains pre-computed answers for all possible queries and all possible sequences X_g. Hence, we can answer a

[1]This is $O(h/\log n + k)$ for sure if $\max(h,k) = \Omega(\log n\log\log n)$; otherwise a small table can be used to perform the search in constant time.

top-k query on X_g in $O(k)$ time.

A query on $\widetilde{I}_g$ can be transformed into a query on X_g by reduction to rank space in the y coordinates. Consider a query range $Q = [a, b] \times [0, h]$ on $\widetilde{I}_g$. We can find the rank h' of h among the y-coordinates of points from $\widetilde{I}_g$ in $O(h) = O(1)$ time by linear search (remember that we store only the reordering of the local x-coordinates, and the actual y-coordinates are found in the global array). Then, we can identify the top-k points in $X_g \cap Q'$, where $Q' = [a, b] \times [0, h']$, using the look-up table and report those points in $O(k)$ time.

Thus our data structure uses $O(n)$ words of space and answers queries in $O(h/\log n + k)$ time. It can be built in $O(n \log n)$ time. As mentioned at the end of Section 4.1, we can obtain any query time of the form $O(h/\log^c n + k)$, for any constant c. This completes the proof of Theorem 4, which is given in this general form.

4.4 Online Queries

An interesting extension of the above result is that we can deliver the top-k documents in online fashion. That is, after the $O(p/\log_\sigma n)$ time initialization, we can deliver the highest weighted result, then the next highest one, and so on. It is possible to interrupt the process at any point and spend overall time $O(p/\log_\sigma n + k)$ after having delivered k results. That is, we obtain the same result without the need of knowing k in advance. This is achieved via an online version of Theorem 4, and is based on a known idea, see e.g. [16, 41]. We describe it for completeness.

Consider an arbitrary data structure that answers top-k queries in $O(f(n) + kg(n))$ time in the case when k must be specified in advance. Let $k_1 = \lceil f(n)/g(n) \rceil$, $k_i = 2k_{i-1}$, and $s_i = \sum_{j=1}^{i-1} k_j$ for $i \geq 2$. Let S be the set of points stored in the data structure and suppose that we must report top points from the range Q in the online mode. At the beginning, we identify top-k_1 points in $O(f(n) + g(n))$ time and store them in a list L_1. Reporting is divided into stages. During the i-th stage, we report points from a list L_i. L_i contains $\min(k_i, |Q \cap S| - s_i)$ top points that were not reported during the previous stages. Simultaneously we compute the list L_{i+1} that contains $\min(2k_i + s_i, |Q \cap S|) < 4k_i$ top points. We identify at most $2k_i + s_i$ top points in $O(f(n) + 4k_i \cdot g(n)) = O(k_i \cdot g(n))$ time. We also remove the first s_i points from L_{i+1} in $O(k_i)$ time. The resulting list L_{i+1} contains $2k_i = k_{i+1}$ points that must be reported during the next $(i+1)$-th stage. The task of creating and cutting the list L_{i+1} is executed in such a way that we spend $O(1)$ time when each point of L_i is reported. Thus when all the points from L_i are output, the list L_{i+1} that contains the next k_{i+1} top points is ready and we can proceed with the $(i+1)$-th stage.

This reporting procedure outputs the first k most highly weighted points in $O(f(n) + kg(n))$ time, and can be interrupted at any time.

5 A Dynamic Structure

We consider now a scenario where insertions and deletions of whole documents are interspersed with top-k queries. When a document d is inserted, each of its d suffixes must be inserted at the appropriate positions in the suffix tree. We must also maintain the pointers $\mathsf{ptr}(v, d)$ that describe the topology of the suffix tree T_d of d, and their connection with the grid, where weighted points

(and columns) are also inserted. All those structures must then support top-k queries. Document deletions must revert those updates.

Section 5.1 describes our dynamic suffix trees. It differs from the classical maintenance procedures in that we add to some selected nodes accelerator structures, similar to the phfs we used in the static case to jump by $\log_\sigma n$ characters of P. Those structures have to be maintained upon insertions and deletions of suffix tree nodes. We define which nodes hold accelerator structures in a way that facilitates maintaining them. The procedure to traverse the dynamic suffix tree to find the locus of the pattern is analogous to the static case, taking $O(p(\log\log n)^2/\log_\sigma n + \log n)$ time. Each document d is inserted or deleted in time $O(|d|\log n)$.

Section 5.2 describes how to maintain the relation between the suffix tree and the grid. The y-coordinates, that used to be the tree depth of the target nodes $u = \mathsf{ptr}(v, d)$, now become string depths, as these do not change upon updates in the suffix tree. Maintaining the x-coordinates is more complicated, as we have to insert and delete x-coordinates in the grid upon insertions and deletions of suffix tree nodes. We replace the integers of the grid by abstract labels, and use existing techniques to maintain order in a set of labels while we can insert and delete labels (this is the order maintenance problem [23, 13]). Then we describe how those labels are created or removed as we insert or delete each suffix of a document. For simplicity we consider the relevance measure *tf* in this section; in Section 5.5 we generalize to others. The additional time needed to maintain this relation is within $O(|d|\log n)$.

We then face the problem of storing those points on a dynamic grid, which can be efficiently queried. The most complex part is Section 5.3, where we consider grids of very small heights, $O(\log^\varepsilon n)$ for any constant $0 < \varepsilon < 1$. We describe an extension of B-trees where weighted points (using abstract labels as x-coordinates) can be inserted and deleted. By storing the heaviest point with each y-coordinate below each B-tree node, we carry out top-k queries in $O(\log n + k\log\log k)$ time. Insertion and deletion of all the points induced by a document d takes time $O(|d|\log^{1+\varepsilon} n)$, which dominates the overall update time. The height limit of the grids is raised to any $O(\mathrm{polylog}\, n)$ in Section 5.4, by decomposing the grid into successively more refined grids, each of height $O(\log^\varepsilon n)$. This decomposition has constant height, and thus it multiplies spaces and times by a constant only.

Section 5.5 wraps up, showing how a query for P starts on the suffix tree and then is translated into a query of the form $[a, b] \times [0, p-1]$, where a and b are abstract labels. This query is split into bands of height $O(\mathrm{polylog}\, n)$, for each of which we have stored a grid, and we obtain the k heaviest points from all the bands. The total search time is $O(p(\log\log n)^2/\log_\sigma n + \log n + k\log\log k)$. We also show how to accommodate other types of weighting schemes apart from *tf*, including those that yield real numbers.

5.1 Dynamic Suffix Trees

Compared to classical dynamic suffix tree data structures [3, 4], we aim to improve the time to traverse the suffix tree at query time, and also to support the operations that will maintain the connection with the grid. We build on a dynamic suffix tree maintenance algorithm where leaves and unary nodes can be inserted and deleted, and lowest common ancestors can be computed, all in constant worst-case time [19]. The updates on leaves and unary nodes are the operations we need to insert and delete all the suffixes of a document in the suffix tree, in time proportional to the length of the document inserted or deleted, whereas the lowest common ancestor queries are

necessary to compute the new $\mathsf{ptr}(\cdot,\cdot)$ pointers to insert in the grid. In any node we will maintain the up to σ children using a linear-space dynamic predecessor data structure that supports queries and updates in worst-case time $o((\log\log\sigma)^2)$ [5].

Upon insertion of a new document d of length $|d|$, we follow McCreight's procedure to insert a new string in a generalized suffix tree [48]. We describe it here for completeness. First we search for the whole document string d in the suffix tree, until we reach the point where it differs from any other suffix in the tree. If this point is a node v, we add a new leaf child z of v that represents the suffix $d_{1..|d|}$. If, instead, the point is in an edge linking v with its child u, we first split the edge at the proper point with a new node x, whose two children will be u and the new leaf z.

We must also maintain the *suffix links* of the tree, that is, the pointers from every node v representing a string $a \cdot X$ to the node $\mathrm{slink}(v)$ representing X, where a is a character and X is a string. To compute the value $\mathrm{slink}(z)$, we go to the node $v' = \mathrm{slink}(v)$, and descend from v' with the characters of $l(v,z)$ (although there are no nodes between v and z, we may go through several nodes from v'). We finally reach a situation similar to the one where we created z: we create z' and possibly split an edge to create its parent x'. Then we set $\mathrm{slink}(z) = z'$. If we had created a node x, the path from v' to z' is followed in two stages, one with the string $l(v,x)$, and another with $l(x,z)$. The path $l(v,x)$ from v' may lead to an existing node y, or we might have to split an edge to create y. Then we set $\mathrm{slink}(x) = y$ (if we had to create y, it will be that $x' = y$ is the parent of z'; we never create two internal nodes). The leaf z' represents the suffix $d_{2..|d|}$. We continue taking suffix links from v', x' (if we created it), and z', until we insert all the $|d|$ suffixes $d_{i..|d|}$. The whole process is known to require $O(|d|)$ operations on the suffix tree. Since we must update the predecessor structures when creating children, our total time is $O(|d|(\log\log\sigma)^2)$.

The deletion of a document d is symmetric to insertion. We find its corresponding string, delete its leaf z and possibly its parent x if it becomes unary, follow the suffix link $z' = \mathrm{slink}(z)$, and repeat the process until removing all the leaves and possibly their parents. This also takes $O(|d|(\log\log\sigma)^2)$ time.

Accelerating searches. On this dynamic suffix tree, finding the locus of P takes $O(p(\log\log\sigma)^2)$ time. In order to search faster we will use a technique analogous to the one used with the static suffix tree in Section 2. We define $\ell = \log_\sigma n$, and the *level* of a node v as $\mathrm{lev}(v) = \lfloor |l(\mathrm{root}, v)|/\ell \rfloor$. Note that the level of a node depends on its string depth, and thus it does not change upon updates. Each suffix tree node v with parent u such that $\mathrm{lev}(v) > \mathrm{lev}(u)$ will maintain a predecessor data structure called an *accelerator*, storing all its highest descendant nodes z such that $\mathrm{lev}(z) > \mathrm{lev}(v)$. The key used for the predecessor data structure are the ℓ characters ($\lg n$ bits) formed by $l(\mathrm{root}, z)[\mathrm{lev}(v)\cdot\ell + 1, (\mathrm{lev}(v)+1)\cdot\ell]$. Note these keys do not depend precisely on v being the node holding the accelerator; any other ancestor of z of the same level of v yields the same key. Note also that the nodes z stored in the accelerator of v are owners of subsequent accelerators.

The predecessor structures hold $O(n)$ nodes, and thus they require $o((\log\log n)^2)$ time and linear space [5]. The total extra space is linear because each suffix tree node belongs to at most one predecessor structure.

Upon searches, we start at the root and use the accelerators of successive nodes, using consecutive chunks of ℓ symbols in P. In some cases we may arrive at nodes whose string-depth difference with the previously visited node is more than ℓ; in those cases we check the missing symbols di-

rectly in the text, also in chunks of ℓ characters. When, finally, there are less than ℓ remaining characters to compare in P, we switch to the character-based search. Thus the total search time is $O(p(\log\log n)^2/\log_\sigma n + (\log_\sigma n)(\log\log\sigma)^2) = O(p(\log\log n)^2/\log_\sigma n + \log n)$.

Those accelerators must be updated upon insertions and deletions of suffix tree nodes. Note that we always know $l(\text{root}, v)$ when we insert or delete a node v. Upon insertion of a leaf z as a child of a node x, it may turn out that the leaf must be inserted into an accelerator (because $\text{lev}(z) > \text{lev}(x)$). We can simply find the nearest ancestor holding an accelerator via at most ℓ parent operations from z. We must also initialize an empty accelerator for z. Symmetrically, when a leaf z is removed, we may have to remove it from its ancestor's accelerator. When an edge from v to u is split with a new node x, it may be that $\text{lev}(v) < \text{lev}(x) = \text{lev}(u)$. In this case, x takes the role of u, "stealing" the accelerator from u (which needs no change, as explained). We must also replace u by x in the accelerator stored at the proper ancestor of x. Another case that requires care is when $\text{lev}(v) < \text{lev}(x) < \text{lev}(u)$. In this case u is replaced by x in the proper ancestor of x, but u retains its accelerator and x creates a new accelerator holding only u. Other cases require no action. Upon deletions, the obvious reverse actions are necessary. The total update time can be bounded by $O(|d|\log n)$ for both insertions and deletions.

5.2 Relating the Suffix Tree and the Grid

Since grid columns will appear and disappear upon document insertions and deletions, we will not associate integers to columns, but just abstract labels. The mapping between the suffix tree and the grid columns will be carried out via a dynamic technique to maintain order in a list X of such abstract labels [23, 13]. The data structure supports the operations of creating a new label y as the immediate successor of a given label $x \in X$, deleting a label $y \in X$, and determining which of two labels comes first in X, all in constant time. In addition, each suffix tree node v will hold a (classical) doubly-linked list $\mathit{list}(v)$ storing consecutive labels of X, each label corresponding to a grid column where this node induces points, and will maintain pointers to the first and the last node in $\mathit{list}(v)$. Finally, v will maintain special labels $\mathit{first}(v), \mathit{last}(v) \in X$ that do not represent any column, but are the predecessor (resp. successor) in X of the first (resp. last) label in its subtree.

Inserting suffixes. As we insert a new leaf z as the child of v, we must create a successor of $\mathit{last}(w)$ to assign to $\mathit{first}(z)$, where w is the previous sibling of z, and then create a successor of $\mathit{first}(z)$ to assign to $\mathit{last}(z)$. If z is the first child of v, instead, we create a successor of $\mathit{first}(v)$ to assign to $\mathit{first}(z)$. The same is done to compute $\mathit{first}(z)$ when we create a new node z that splits an edge from v to u, but this time $\mathit{last}(z)$ is obtained by creating a new successor of $\mathit{last}(u)$. When a node z is removed, its labels $\mathit{first}(z), \mathit{last}(z)$ are also removed from X.

As we insert a new document d, we must associate new grid columns to the new and existing suffix tree nodes traversed. Each newly created pointer $\mathsf{ptr}(v, d)$ will require creating a new label $t(v, d) \in X$ as the successor of the last node in $\mathit{list}(v)$ (it will also be stored at the end of $\mathit{list}(v)$), or as the successor of $\mathit{first}(v)$ if $\mathit{list}(v)$ is empty. The pointer $\mathsf{ptr}(v, d)$ will be stored associated with the label $t(v, d)$.

Computing the new weights. As we insert new leaves in the suffix tree, we collect them in an array $L[1, |d|]$. We also create the first label $t(v, d)$ of such leaves v. Now we sort L by the labels

$t(v,d)$, and as a result the new leaves become sorted by their suffix tree preorder. All the internal suffix tree nodes that must be labeled with d are obtained as $u = \mathrm{lca}(v, v')$ for consecutive leaves $v = L[i]$ and $v' = L[i+1]$. We create pointers $\mathsf{ptr}(v,d)$ and $\mathsf{ptr}(v',d)$ towards node u, associated with the labels $t(v,d)$ and $t(v',d)$, respectively, and with weights $w(v,d) = w(v',d) = 1$ (as said, we are considering term frequency weights for now). For each new internal suffix tree node $u = \mathrm{lca}(v, v')$ obtained, we create a new label $t(u,d)$ for the new grid column that u will originate, and associate weight $w(u,d) = 2$ to it (at the end, $w(u,d)$ will be the number of leaves labeled d in the subtree of u). Each time u is obtained again (which we know because the last element of $\mathit{list}(u)$ is already $t(u,d)$), we increase $w(u,d)$ by 1.

The previous procedure already visits all the internal nodes u of T that are labeled with d, that is, all the nodes of the suffix tree T_d of document d. Now we have to propagate weights and pointers from those internal nodes to their nearest ancestors labeled with d (i.e., the nodes that would be their parent in T_d). For this sake, the internal nodes $u = \mathrm{lca}(v, v')$ obtained are collected in a new array I, of size up to $|d| - 1$, and I is sorted by the labels $t(u,d)$, so that the nodes become sorted by preorder. We traverse I left to right, simulating a recursive preorder traversal of the suffix tree of document d, although the nodes are in the generalized suffix tree. Along this simulated recursive traversal, each node identifies its parent in T_d, setting the pointer ptr to it and increasing its weight. Let $u = I[i]$ and $v = I[j]$, initially for $i = 1$ and $j = 2$. If $\mathrm{lca}(u, v) = u$, then u is the parent of v in T_d. Thus we recursively traverse the subtree that starts in $v = I[j]$, which finishes at a node $v' = I[j']$ that is not anymore a descendant of v. Now we check whether $\mathrm{lca}(u, v') = u$ (i.e., v' is the second child of u in T_d), and so on. At some point, it will hold that $I[j']$ does not descend from u, and we have finished the traversal of the subtree of u. Then we set $i = j'$, $j = i + 1$, and restart the process. Along this recursive traversal we will identify the nearest ancestor u labeled d of each node v labeled d, that is, the parent u of each v in T_d. For each such pair, and after having processed v and computed $w(v,d)$, we increase $w(u,d) = w(u,d) + w(v,d)$ and generate the pointer $\mathsf{ptr}(v,d)$ pointing to u, associated with label $t(v,d)$ and weight $w(v,d)$.

All the labels created when inserting a document d are additionally chained in a (classical) list $\mathit{list}(d)$, to facilitate deletion of the document.

Creating the grid points. Finally, we will create new columns and points in the grid associated with all the pointers $\mathsf{ptr}(v,d) = u$ created. The label $t(v,d)$ will be an identifier for the x-coordinate of the point (we remark that these are not integers, but just labels that can be compared). The y-coordinate will be the string depth of the target node, $|l(\mathrm{root}, u)|$. This value is stored in the suffix tree node when the node is created and, unlike the tree depth, does not change upon suffix tree updates. The document associated with the new point is the new one, d, and the weight is the value $w(v,d)$ associated with the source node of the pointer.

The overall time of inserting d is $O(|d| \log |d|)$, dominated by the sorting via comparisons of labels in X.

Deleting documents. Handling the deletion of a document d is simple. After deleting all the corresponding suffix tree nodes, we follow the chain of labels $t(v,d)$ in $\mathit{list}(d)$, delete them from X and remove their nodes from the doubly-linked list $\mathit{list}(v)$. This takes $O(|d|)$ additional time. We also remove the columns in the grid corresponding to the labels deleted, and the associated points.

Both insertion and deletion times are superseded by those of Section 5.1.

5.3 Slim Grids

To achieve faster searches, the grid will be divided into horizontal slices of small height r. For every slice, we maintain a structure that reports k most highly weighted points from a horizontal range of labels $[a, b)$ intersected with a vertical range of integers $[0, y)$. We describe here how those slices are updated and queried for a sublogarithmic value of r, and in Section 5.4 we extend the solution to grids of polylogarithmic height.

Each slice is represented with a B-tree ordered by the labels (i.e., x-coordinates) of the points, of arity r to $2r-1$, for some $r = \lg^{\varepsilon} n$ and a constant $0 < \varepsilon < 1/2$ (as usual, the root can have arity as low as 2). Thus the B-tree has height $O(\log_r n)$. At each internal node u with $a(u)$ children $v_1, \ldots, v_{a(u)}$, we will store $a(u)$ arrays $W_{v_1}[0..r-1], \ldots, W_{v_{a(u)}}[0..r-1]$. In these arrays W_v, $W_v[y]$ is the point p with maximum weight among all points (1) whose x-coordinates belong to the subtree of v, (2) with y-coordinate equal to y, and (3) not stored in $W_u[y]$ for ancestors u of v (some $W_v[y]$ cells can be empty, if no point with y-coordinate y exists below v). We will also store a structure W_{root} for the root node. Thus $W_{\text{root}}[y]$ contains the point p_r of maximum weight among all points with y-coordinate y; for a child v of the root, $W_v[y]$ contains the point p_v of maximum weight among all points $p \neq p_r$ in the subtree of v with y-coordinate y. In general, all points already stored in ancestors are excluded from consideration. We store $W_v[y] = (x, w, d)$, where x is the x-coordinate, w is the weight, and d is the document of the point. Each point is also stored in the corresponding leaf node of the B-tree. Those points in W_v are not used to separate the x-coordinates of the points in the tree. Instead, new labels $x(v_1) \ldots x(v_{a(u)-1}) \in X$ will be created and stored at node u, to split the points between its $a(u)$ consecutive children. That is, the x-coordinate of any point stored below v_i will be between $x(v_{i-1})$ and $x(v_i)$. The size of the list X stays $O(n)$.

The leaves of the B-tree will store r to $2r-1$ points. Leaves store the actual points, even if they are also mentioned in some previous W_v structure. The points in leaves l are arranged in an array W'_l, which is similar to the arrays W_v and lists the points in increasing y-coordinate order, except that W'_l has no empty cells and some y-coordinates can be repeated in the points. Therefore the W'_l cells store the full point data, $W'_l[j] = (x, y, w, d)$.

To each internal node u with children $v_1, \ldots, v_{a(u)}$ we will also associate structures $Yx_u[0..a(u)r-1]$ and $Yw_u[0..a(u)r-1]$, where the child numbers and the y-coordinates of the (up to) r points of the $a(u)$ arrays W_{v_i} are sorted by their x-coordinate label (in Yx_u) and by their weight (in Yw_u). That is, in Yx_u and Yw_u we store the pair (i, y) for each entry $W_{v_i}[y]$, ordered by $W_{v_i}[y].x$ (in Yx_u) or by $W_{v_i}[y].w$ (in Yw_u). Each value stored in Yx_u and Yw_u requires $\lg(2r^2)$ bits, thus all the values in these two structures add up to at most $4r^2 \lg(2r^2)$ bits. Each time we modify a value in a W_{v_i} array, we rebuild from scratch the Yx_u and Yw_u structures of the parent u of v_i.

We will also maintain structures Yx'_l and Yw'_l on the (up to) $2r-1$ points of leaves l, analogous to the Yx_u and Yw_u structures of internal nodes. Instead of the pairs (i, y), structures Yx'_l and Yw'_l will just store positions j of the array W'_l (those positions would coincide with y-coordinates in internal nodes). Leaves will also store an array $Y_l[0..r-1]$ where $Y_l[y] = j$ if j is the last position where $W'_l[j] < y$. Finally, leaves will store bitmaps Q_l marking in $Q_l[j]$ whether the point in $W'_l[j]$ also appears in the W_v array of an ancestor v of l.

Since $4r^2 \lg(2r^2) = o(\log n)$, universal tables of $2^{4r^2 \lg(2r^2)} \cdot O(\text{polylog } (r^2)) = o(n)$ bits will be

used to query and update the arrays Yx_u and Yw_u, in constant time. Similarly, leaves will use even smaller universal tables of $2^{4r\lg(2r)} \cdot O(\text{polylog}\ (r)) = o(n)$ bits.

The whole data structure requires linear space, because the leaves contain $\Theta(r)$ points. The W_v arrays of internal nodes spend $\Theta(r)$ words and can be almost empty (if all the descendants have the same y-coordinate, say), but there are only $O(n/r)$ internal nodes. If the whole grid contains less than r points, we just store the space for them in a leaf.

5.3.1 Insertions

Consider the insertion of a new point (x, y, w, d), with label $x \in X$, y-coordinate $y \in [0, r)$, weight w and document d. While following the normal insertion procedure on the B-tree (where we compare the labels $x(v_i)$ of the nodes with x to decide the insertion path), we look for the highest node v with $W_v[y].w < w$ or with $W_v[y]$ empty. For the first (i.e., highest) such v we find, we set $W_v[y] \leftarrow (x, w, d)$, and then we continue the classical insertion procedure (not looking at $W_v[y]$ entries anymore) until adding the point (x, y, w, d) in a leaf l. In the leaf we mark in the corresponding Q_l entry whether we had updated an entry $W_v[y]$ in some ancestor v.

If we updated some $W_v[y]$, and it already had a previous value $W_v[y] = (x', w', d')$, we perform a process we call *reinsertion* of (x', w', d'). We restart the process of inserting the point (x', y, w', d') from node v (note that this point already exists in a leaf; reinsertion will not alter the structure of the tree, but just rewrite some W and Q values). In the reinsertion path, if we arrive at a node v' where $W_{v'}[y].w < w'$, we set $W_{v'}[y] \leftarrow (x', w', d')$. If there was a previous value $W_{v'}[y] = (x'', w'', d'')$, we continue the reinsertion process for point (x'', w'', d'') from node v', and so on until either we find an empty space in some $W_u[y]$ or we reach the leaf l where the point being reinserted is actually stored. In this latter case, we clear the corresponding bit in Q_l, indicating that this point is not stored anymore in an ancestor structure.

Thus we traverse two paths, one for inserting the point, and another for reinserting the point(s) possibly displaced from some $W_v[y]$ structure. This part of the operation requires, in the worst case, $O(\log_r n)$ updates to the structures Yx_u and Yw_u of the parents u of nodes v where W_v is modified, plus an insertion in a leaf.

Rebuilding structures Yx_u and Yw_u. Upon an assignment $W_{v_i}[y] \leftarrow (x, w, d)$, we must rebuild the structures Yx_u and Yw_u of the parent u of v_i. We binary search Yx_u for x, and binary search Yw_u for w, both in $O(\log(r^2)) = O(\log r)$ time. In these binary searches we obtain the actual label and weight of each element of Yx_u and Yw_u, respectively, using its (i', y') pair, as $W_{v_{i'}}[y'].x$ and $W_{v_{i'}}[y'].w$. These binary searches give the insertion positions $0 \le e < 2r^2$ and $0 \le g < 2r^2$, respectively, of the pair (i, y) in Yx_u and Yw_u. Note that the new contents of Yx_u and Yw_u depend only on their current contents, on the values e and g, and on the incoming pair (i, y) (the existing occurrence of (i, y), if any, must be removed). Thus, the new content of Yx_u and Yw_u for each (e, g, i, y) can be precomputed in a universal table of $o(n)$ bits, as explained, so that they are updated in constant time. Therefore the time to update the structures is $O(\log(r^2)) = O(\log r)$, and the cost of a full reinsertion process is $O(\log_r n \log r) = O(\log n)$.

The process to update Yx_u and Yw_u upon removal of the value of a cell $W_{v_i}[y]$ is analogous: we find with binary search the positions e and g of the pair (i, y) to remove, and then compute the new structures Yx_u and Yw_u in constant time with a universal table.

When a new internal node of the B-tree is created, or when an internal node is removed, we have to create a whole new W_v array, or remove a whole array W_v. We can insert or remove all such cells one by one in Yx_u and Yw_u, in time $O(r \log r)$. In those cases, we must rename all the labels i in the pairs (i, y) stored in those structures, but those updates can also be precomputed in universal tables of sublinear size.

Insertion in leaves. In leaves l, we must actually insert the point, possibly displacing all the entries in W'_l and also inserting the point data in Y_l, Yx'_l, Yw'_l and Q_l, all in $O(r)$ time. When a leaf overflows to $2r$ points, we must split it into two leaves l' and l'' of r points each. We first remove the array W_l from the parent u of l, clearing the corresponding bits in Q_l. Now we distribute the points of W'_l into the new arrays $W'_{l'}$ and $W'_{l''}$, and make l' and l'' children of u, replacing the old l. We create a new label $x(l') \in X$ as the successor of the largest x-coordinate in l', and add it to u separating l' and l''.

Next we build new arrays $W_{l'}$ and $W_{l''}$. Those arrays, as well as the Y, Yx', Yw' and Q structures of l' and l'', are built in $O(r)$ time from W'_l, Yx'_l, Yw'_l and Q_l. We also set in $Q_{l'}$ and $Q_{l''}$ the points that have been included in $W_{l'}$ and $W_{l''}$ (we cannot choose any point for $W_{l'}$ and $W_{l''}$ that is already marked in Q_l). Finally, we update the tables Yx_u and Yw_u according to the new tables $W_{l'}$ and $W_{l''}$.

The overall time is $O(r)$, but this is dominated by the $O(r \log r)$ time needed to update the Yx_u and Yw_u arrays upon the $O(r)$ changes induced by substituting W_l by $W_{l'}$ and $W_{l''}$.

Overflows in internal nodes. The insertion of a new child in the parent u can trigger an overflow in this internal node, if its arity reaches $2r$. We must split u, with children $v_1, \ldots, v_{2r}$, into two nodes, u' with children $v_1, \ldots, v_r$ and u'' with children $v_{r+1}, \ldots, v_{2r}$. The process is analogous to the case of leaves, but slightly more complicated. We create a new x-coordinate $x(u') \in X$ as a successor of $x(v_r)$, to separate the points of u' and u''. We create the two nodes u' and u'' with their corresponding arrays $W_{v_1}, \ldots, W_{v_r}$ and $W_{v_{r+1}}, \ldots, W_{v_{2r}}$, and build the tables Yx and Yw of u' and u'', in $O(r^2)$ time from Yx_u and Yw_u.

Now we must create new arrays $W_{u'}$ and $W_{u''}$ to replace W_u in the parent of u. First, we move each point in $W_u[y]$ to $W_{u'}[y]$ or $W_{u''}[y]$, according to its x-coordinate. Now we can get rid of W_u, but we still have several empty cells in $W_{u'}[y]$ and $W_{u''}[y]$. Those are filled with a process we call *uninsertion*: To fill some cell $W_{u'}[y]$ (analogously for u''), we take the maximum weight in cells $W_{v_1}[y], \ldots, W_{v_r}[y]$. The maximum $W_{v_i}[y].w$ is found in constant time using a universal table on $Yw_{u'}$ that returns the first pair with y-coordinate equal to y. Then we copy $W_{u'}[y] \leftarrow W_{v_i}[y]$, and continue the uninsertion process for $W_{v_i}[y]$. When we finally arrive at uninserting a point from a leaf l, all we have to do is to clear the corresponding entry in Q_l. Note that uninsertion does not alter the structure of the tree; it just rewrites some W and Q values. The cost of one uninsertion is $O(\log_r n \log r) = O(\log n)$, to rebuild the affected structures Yx and Yw. Thus the $O(r)$ uninsertions in u' and u'' add up to $O(r \log n)$ time, which subsumes the $O(r \log r)$ cost to replace W_u by $W_{u'}$ and $W_{u''}$ in the parent of u.

Note that the insertion of a single point could produce one split per level of the B-tree, which would be too costly. To avoid this, we use a deamortization technique by Fleischer [28]. This allows nodes have from $r/2$ to $2r$ children. It maintains a *cursor* per leaf, which all the time is

navigating upwards to the root and then returns to the leaf. Each update on the leaf also moves its cursor one step upwards. The node where the cursor lies is split if it has r children or more (for leaves, r elements or more). If the leaves are of size $\log_r n$ (the tree height) or more, then one can ensure [28] that all the nodes contain $r/2$ to $2r$ elements, and only one split per update is performed. We can easily accommodate this wider freedom for the node arities. The problem is that our leaves are of size $r = \lg^\varepsilon n$, too small for the cursor to return to the leaf in time for the next overflow. We then use a hybrid scheme: the lower nodes of the tree, of height up to $1/\varepsilon$ (taking a leaf as of height 1) are organized as weight-balanced B-trees (WBB-trees) [8], which differ from classical B-trees only in the policy to split nodes. Our WBB-tree leaves contain r to $2r$ elements, whereas nodes of height $h > 1$ have arity $r/4$ to $4r$ and contain $r^h/2$ to $2r^h$ elements in their leaves. The nodes in WBB-trees perform the splits as soon as they are needed, whereas the higher nodes use Fleischer's scheme. Then an update may trigger $O(1/\varepsilon)$ splits at lower nodes, each of which costs $O((1/\varepsilon)r \log n)$, for a total of $O(r \log n)$ time. The lowest nodes of the higher levels act at the same time as the root of the WBB-trees and as the leaves in Fleischer's scheme. They contain at least $(\lg n)/2$ elements in their subtree, and therefore they can only overflow once every $(\lg n)/2$ insertions due to the WBB-tree splitting policies [8, Lem. 7]. Therefore Fleischer's cursors associated with those leaves have time to return to them and carry out the necessary splits.

5.3.2 Deletions

Deletion of a point (x, y) starts by searching the B-tree for the x-coordinate x. The point will be found in its leaf, and also possibly in some cell $W_v[y]$ of some internal node v. The search takes $O(\log n)$ time because, for internal nodes u, we binary search the coordinates $x(v_i)$ stored in u for the correct child v, in $O(\log r)$ time, and then only have to check if $W_v[y].x = x$. In leaves l, we binary search for x in Yx'_l in $O(\log r)$ time.

If the point has to be deleted from some $W_v[y]$, we carry out the *uninsertion* process already described, in $O(\log_r n \log(r^2)) = O(\log n)$ time. We also remove the point (x, y) itself from leaf l. When a leaf l underflows, we merge it with a neighbor leaf and, if necessary, split it again. The merging process is analogous to the splitting and can be easily carried out in $O(r)$ time, plus $O(r \log r)$ to update the structures Yx and Yw in the parent.

If an internal node underflows, we also merge it with its neighbor and re-split it if necessary. The merging of two sibling nodes v and v' is carried out in $O(r)$ time, including the construction of the Yx and Yw structures for the merged node, u. The difficult part is, again, to get rid of the arrays W_v and $W_{v'}$ at the parent node, replacing them by a new W_{v^*} table for the merged node v^*. For this sake, we choose the maximum weight between each $W_v[y]$ and $W_{v'}[y]$ and assign it to $W_{v^*}[y]$. The point that was not chosen among $W_v[y]$ and $W_{v'}[y]$ must be *reinserted*, as before. Finally, we must rebuild the Yx and Yw structures of the parent of v^*. The total cost is $O(r \log n)$, just as for insertions.

Note that, upon leaf or internal node merges, a separating label $x(v)$ becomes unused, and it is removed from X. Like Fleischer [28], to avoid excessive merging work, we delay these merges and use global deamortized rebuilding [59].

5.3.3 Queries

Identifying the relevant nodes. To solve a top-k query with label restriction $[a, b)$ and y-coordinate restriction $[0, y)$ on the slice, we first identify the $O(\log_r n)$ ranges of siblings of the B-tree tree that exactly cover the interval of labels $[a, b)$; plus up to 2 leaf nodes that partially overlap the interval. For each node u that is the parent of a range of children $v_s, \ldots, v_e$ included in the cover, we find the maximum weight in $W_{v_s}[0, y-1], \ldots, W_{v_e}[0, y-1]$ and insert the result in a max-priority queue $\mathcal{Q}$ sorted by the weights of the points. Such maximum across $W_{v_i}[0, y-1]$ arrays is obtained in constant time using universal tables on Yw_u that find the first pair (i, y') with $s \leq i \leq< e$ and $y' < y$. For the leaves l partially or fully overlapping $[a, b)$, the points fall in the interval $W'_l[0, Y_l[y]]$. In addition, if the leaf partially overlaps $[a, b)$, we must binary search Yx'_l for the range $[x_a, x_b]$ corresponding to the interval $[a, b)$. Furthermore, we can only return points whose Q_l bit is not set, to avoid repeated answers. Knowing the range in $Yx'_l[x_a, x_b]$ and the range $W'_l[0, Y_l[y]]$, the maximum weight can be obtained from Yx'_l, Yw'_l and Q_l with a universal table, in constant time. Identifying the cover nodes and finding their $O(\log_r n)$ maxima takes $O(\log n)$ time, and leaves add only $O(\log r)$ time.

Each element inserted in $\mathcal{Q}$ coming from a range of siblings will be a tuple (u, s, e, i, z, k), where u is the parent node of the range of children $v_s, \ldots, v_e$ in the cover, (i, z) means that the maximum was found at $W_{v_i}[z]$ $(s \leq i \leq e)$, and k indicates that the point $W_{v_i}[z]$ is the kth in the range of interest for u. All the nodes initially inserted have $k = 1$.

The elements inserted in $\mathcal{Q}$ coming from leaves l are of the form $[l, j, x_a, x_b, k]$, meaning that the maximum was found in $W'_l[j]$, that the range of interest is $W'_l[0, Y_l[y]]$ and $Yx'[x_a, x_b]$, and that the point is the kth in the range of interest. The first insertions use $k = 1$.

We also insert in $\mathcal{Q}$ a third kind of tuples, namely, the maximum-weight point in $W_v[0, y-1]$ with x-coordinate in $[a, b)$, for each of the $O(\log_r n)$ ancestors v of the cover nodes, as they may also hold relevant points. To find those maxima we consider the parent u of v and binary search Yx_u for a and b, to find a mapped interval $Yx_u[x_a, x_b]$, in $O(\log(r^2)) = O(\log r)$ time. Note that this area of Yx_u corresponds to nodes in W_v. Then we use universal tables on Yx_u and Yw_u to find the maximum weight of y-coordinate below y and in the range $Yx_u[x_a, x_b]$. For these nodes we insert tuples of the form $\langle u, v, z, x_a, x_b, k\rangle$ in $\mathcal{Q}$, meaning that the maximum was obtained from $W_v[z]$, the range of interest is $Yx_u[x_a, x_b]$, and the point is the kth in its range of interest. Since all these ancestors also amount to $O(\log_\rho n)$, the initial computation on these nodes requires $O(\log_r n \log r) = O(\log n)$ time. Recall that the root node of the B-tree will also have a W structure computed (this is easily treated as a special case).

We implement $\mathcal{Q}$ as a Thorup's priority queue [67] on the universe of weights $[1, O(n)]$. Note that we do not need to insert the whole initial set of $O(\log_r n)$ tuples in $\mathcal{Q}$ if this number exceeds k: if a tuple is not among the first k, it cannot contribute to the answer. Then we use linear-time selection to find the kth largest weight in the tuples and then insert only the first k tuples in $\mathcal{Q}$. This structure supports insertions in constant time, thus the initialization of $\mathcal{Q}$ takes time $O(\log_r n)$.

Extracting the top-k points. The first answer to the top-k query is among the $O(k)$ tuples we have inserted in $\mathcal{Q}$. Therefore, to obtain the first result, we extract the tuple with maximum weight from $\mathcal{Q}$. If it is of the form $[l, j, x_a, x_b, k]$, that is, it comes from a leaf l, we report the point $W'_l[j]$, compute the $(k+1)$th highest-weight point $W'_l[j']$ within $W'_l[0, Y_l[y]]$ and $Yx'_l[x_a, x_b]$

using universal tables, and reinsert tuple $[l, j', x_a, x_b, k+1]$ in $\mathcal{Q}$. (To these universal tables we give in addition the latest point reported, and they find the next one in decreasing weight order.) If, instead, the maximum tuple extracted from $\mathcal{Q}$ is of the form $\langle u, v, z, x_a, x_b, k\rangle$, that is, it becomes from an ancestor of a cover node, we report the point $W_v[z]$, compute the $(k+1)$th highest-weight point $W_v[z']$ with y-coordinate below y and within $Yx_u[x_a, x_b]$ using universal tables, and reinsert tuple $\langle u, v, z', x_a, x_b, k+1\rangle$. Finally, if the maximum tuple extracted from $\mathcal{Q}$ is of the form (u, s, e, i, z, k), we report the point $W_{v_i}[z]$, where v_i is the ith child of u, compute the $(k+1)$th highest-weight point $W_{v_{i'}}[z']$ in $W_{v_s}[0, y-1], \ldots, W_{v_e}[0, y-1]$ using universal tables, and reinsert tuple $(u, s, e, i', z', k+1)$. If the extracted point had $k = 1$, however, it is possible that the next highest-weight element comes from the child v_i. Therefore, if v_i is an internal node, we compute the highest-weight point in Yw_{v_i} with y-coordinate below y. Let it be the pair (i'', z''), then we insert a new tuple $(v_i, 1, a(v_i), i'', z'', 1)$ in $\mathcal{Q}$. If, instead, v_i is a leaf $l = v_i$ with $r(l)$ elements, then we find the maximum-weight point $W'_l[j']$ in $W'_l[0, Y_l[y]]$ using Yw'_l, and insert the tuple $[l, j', 1, r(l), 1]$ in $\mathcal{Q}$. In all cases the cost to compute and insert the new tuples is constant.

If we carry out k extractions from $\mathcal{Q}$, we will also carry out up to $2k$ insertions, thus the size of $\mathcal{Q}$ will be $O(k)$ and minima extractions will cost $O(\log\log k)$ [67]. The cost of this part is then $O(k \log\log k)$, and the total query time is $O(\log n + \log r + \log n + \log_r n + k\log\log k)$. Recalling that $r = \lg^{\varepsilon} n$, the query time is $O(\log n + k\log\log k)$ and the update time is $O(r \log n) = O(\log^{1+\varepsilon} n)$. Note that the process is not online: We must know k in advance so as to initially limit the size of $\mathcal{Q}$ to k. We use the technique of Section 4.4 to make the process online in k. That is, k is not specified in advance and the process can be interrupted after having produced any number k of results, and the total cost paid will be $O(\log n + \log\log k)$.

5.4 Multiresolution Grids

We extend the result of Section 5.3 to grids of polylogarithmic height r^c, for some constant c. We will represent the grid at various resolutions and split it into slim grids for each resolution. Consider a virtual perfect tree of arity r and n leaves, so that the ith left-ro-right node of height j covers the rows $(i-1)\cdot r^j + 1$ to $i \cdot r^j$. The tree is of height c.

For each node v of this tree we store a slim grid of r rows, one per child. All the points whose row belongs to the area covered by the ith child of v will be represented as having y-coordinate i in the slim grid of v.

When a new point (x, y, w, d) is inserted in the grid, we insert it into the c slim grids that cover it, giving it the appropriate row value in each slim grid, and similarly when a point is deleted. The x-coordinate labels are shared among all the grids. This arrangement multiplies space and insertion and deletion times by the constant c.

Now consider a 3-sided top-k query with the restriction $[a, b)$ on the x-coordinates and $[0, y)$ on the y-coordinates. The range $[0, y)$ is covered with the union of one range in one slim grid per level of the tree. Let $y_c, \ldots, y_1$ be the child numbers of the path from the root to the y-th row of the grid. Then we take the 3-sided query $[a, b) \times [0, y_j)$ at the node of height j in the path.

We start the searches in the c slim grids, and extract the first result from each grid. We insert those local maxima into a new global queue $\mathcal{Q}$. Now we repeat k times the process of extracting the next result from $\mathcal{Q}$, reporting it, requesting the next result from the grid where the result came

from, and inserting it in $\mathcal{Q}$. Note that $\mathcal{Q}$ can be implemented naively because it contains at most c elements and c is a constant.

Initializing the searches will then require $O(c \log n)$ time, and extracting k results from the slim grids will require $O(k \log\log k)$ time. Managing $\mathcal{Q}$ will require $O(ck)$ time even if done naively. Therefore the total time is still $O(\log n + k \log\log k)$. The update time per element stays $O(c \log^{1+\varepsilon} n)$. The process is also online. Then we obtain the following lemma.

Lemma 6 *A set of n points, one per column on an $n \times r^c$ grid, for $r = \lg^\varepsilon n$ and any constant $0 < \varepsilon < 1$ and $c \geq 1$, with weights in $[1, O(n)]$, can be stored in $O(n)$ words of space, so that for any $1 \leq k \leq n$, $1 \leq h \leq r^c$ and $1 \leq a \leq b \leq n$, k most highly weighted points in the range $[a, b] \times [0, h]$ can be reported in decreasing order of their weights in $O(\log n + k \log\log k)$ time, online in k. Points (and their columns) can be inserted and deleted in $O(\log^{1+\varepsilon} n)$ time.*

5.5 The Final Result

We find the locus v of P in the suffix tree in time $O(p(\log\log n)^2/\log_\sigma n + \log n)$. Then the x-coordinate range of labels to search for in the grid is $[a, b)$, where a is the first label in $list(v)$ and $b = last(v)$. Since we store string depths in the grid, the y-coordinate range of the query is $[0, p)$.

Our dynamic grid is horizontally split into bands of r^c rows, for a constant c, which are handled as explained in Section 5.4. Therefore, our 3-sided query is translated into 3-sided queries on the first $\lceil p/r^c \rceil$ bands. All but the last will query for the whole row interval $[0, r^c)$, whereas the latter will query for the row interval $[0, (p-1) \bmod r^c]$.

We start the searches in all the bands, and extract the first result from each. If there are more than k bands, we use linear-time selection to keep only the k highest weights. Then we insert the local maxima into a new global queue $\mathcal{Q}$. Now we repeat k times the process of extracting the first result from $\mathcal{Q}$, and if it came from the ith band, then we request the next result from that band and insert it in $\mathcal{Q}$ (unless it has no more results, in which case we continue with the remaining bands). Again, $\mathcal{Q}$ will be implemented with Thorup's priority queue [67].

Initializing the searches will then require $O(\lceil p/r^c \rceil \log n)$ time, and extracting k (and inserting other k) results in $\mathcal{Q}$ will take time $O(k \log\log k)$. We choose $r = \lg^\varepsilon n$ for some $0 < \varepsilon < 1/2$, as explained, and rename c as $(c+1)/\varepsilon$. Therefore, we obtain a query time of $O(p/\log^c n + \log n + k \log\log k)$ for the grid. Once again, the scheme can be made online with the technique of Section 4.4. Updating grid points, including extending the grid downwards, requires $O(\log^{1+\varepsilon} n)$ time. This yields Theorem 5.

We developed our result for *tf* as the relevance measure. It is very easy to support others like *docrank*, but if the weights are not integer numbers, then Thorup's priority queues [67] cannot be used. In this case we insert all the new weights that appear in a data structure for monotonic list labeling, which assigns them integers in a polynomial universe $[1, n^{O(1)}]$. This adds at most $O(\log n)$ time per symbol inserted [22] (deletions can be handled by deamortized periodic rebuildings [59]). In general we can support any measure that can be computed in time $O(|d| C_w)$ over the suffix tree T_d of the document d to insert: We explicitly build T_d, compute the relevance measure for all the nodes, and then use them to assign the weights as we insert the nodes in our suffix tree. At the end we delete T_d. This suffix tree can be built (and deleted) in $O(|d|)$ time on integer alphabets [24]. Therefore we simply charge $O(C_w)$ time per character inserted in our text collection. Note

that C_w is $O(1)$ for measures *tf* and *docrank*. Hon et al. [41] show how to compute *mindist* from T_d in $O(|d| \log |d|)$ time, so $C_w = O(\log n)$ in this case.

Our scheme works as long as $\lg n$ has a fixed value (plus $O(1)$). We use standard techniques to incrementally rebuild the structure for larger or smaller $\lg n$ values as more insertions or deletions are processed [58].

6 A Space-Efficient Data Structure

We now show how the space of our static structure can be reduced to $O(n(\log \sigma + \log D + \log\log n))$ bits, where σ is the alphabet size and D is the number of documents, and retain almost the same query time. Our approach is to partition the tree into *minitrees*, which are connected subtrees containing $O(\sigma^2 D \log n)$ nodes, so that their grids can be represented within the given space. Pointers ptr that cross over minitrees can be safely replaced by shorter ones going from some minitree leaf to a fake node above the minitree root, without altering the answers to top-k queries. The minitrees are seen as nodes in a so-called *contracted tree* T^c, which requires only $O(n)$ bits. Some queries are solved locally within a minitree, while others are solved on T^c. To find the locus in the right minitree or in T^c, we use a *compressed suffix tree* data structure, which uses $O(n \log \sigma)$ bits. This is the responsible of the slight increase in query times compared to the linear-space version of Theorem 1. In Section 6.1 we show how to remove the $O(n \log\log n)$ term from the space when the relevance measure is the term frequency, by exploiting particularities of this measure.

Partitioning the suffix tree. We define $z = \Theta(\sigma D \log n)$. We say that a suffix tree node $v \in T$ is *heavy* if the subtree rooted at v has at least z leaves, otherwise it is *light*. A heavy node is *fat* if it has at least two heavy children, otherwise it is *thin*.

All the non-fat nodes of T are grouped into minitrees as follows. We traverse T in depth-first order. If a visited node v has two heavy children, we mark v as fat and proceed. If v has no heavy children, we mark v as thin or light, and make v the root of a minitree T_v that contains all the descendants of v (which need not be traversed). Finally, if v has one heavy child v_1, we mark v as thin and make it the root of a minitree T_v. The extent of this minitree is computed as follows. If v_i, $i \geq 1$, is a thin node with one heavy child v_{i+1}, we visit nodes $v_1, v_2, \ldots v_{j-1}$ and include v_i and all the descendants of its other children, until either v_{j-1} has no heavy children or is fat, or T_v contains more than σz nodes after considering v_j. Then we continue our tree traversal from v_j. Note that T_v contains at the very least the descendants of v by children other than v_1.

With the procedure for grouping nodes described above, the leaves of minitrees can be parents of nodes not in the minitree. Those child nodes can be either fat nodes or roots of other minitrees. However, at most one leaf of a minitree can have children in T.

Note that the size of a minitree is at most $O(\sigma z)$. On the other hand, as two heavy children have disjoint leaves, there are $O(n/z)$ fat nodes in T. Finally, minitrees can contain as little as one node (e.g., for leaves that are children of fat nodes). However, note that a minitree root is either a child of a fat node (and thus there are $O(\sigma n/z)$ minitrees of this kind), or a child of a leaf of another minitree such that the sum of both minitree sizes exceeds σz (otherwise we would have included the root v_j of the child minitree as part of the parent minitree). Moreover, as said, at most one of the leaves of a minitree can be the parent of another minitree, so these minitrees

that are "children" of others form chains where two consecutive minitrees cover at least σz nodes of T. Thus there are $O(n/(\sigma z))$ minitrees of this second kind. Adding up both cases, there are $O(\sigma n/z) = O(n/(D \log n))$ minitrees in T.

Contracted tree and minitrees. The pointers in a tree T are defined in the same way as in Section 2. Since we cannot store T without violating the desired space bound, we store a *contracted tree* T^c and the minitrees T_v.

The contracted tree T^c contains all fat nodes of T, plus one node v^c for each minitree T_v. Each pointer $\mathsf{ptr}(u, d) = u'$ of T is mapped to a pointer $\mathsf{ptr}^c(u^c, d) = (u')^c$ of T^c as follows. If u is a fat node, then $u^c = u$. Otherwise, if u belongs to minitree T_v, then $u^c = v^c$. Similarly, if u' is a fat node then $(u')^c = u'$; otherwise, if u' belongs to minitree $T_{v'}$ then $(u')^c = (v')^c$. In other words, nodes of a minitree are mapped to the single node that represents that minitree in T^c and pointers are changed accordingly.

For each minitree T_v, we store one additional dummy node ν that is the parent of v. If a leaf u_h of T_v has a heavy child $u' \notin T_v$, we store an additional dummy node $\nu' \in T_v$ that is the only child of u_h. Pointers of T_v are modified as follows. Each pointer $\mathsf{ptr}(u, d)$, $u \in T_v$, that points to an ancestor of v is transformed into a pointer $\mathsf{ptr}(u, d)$ that points to ν. Every pointer $\mathsf{ptr}(u'', d)$ that starts in a descendant u'' of u_h and points to a node $u \in T_v$, $u \neq u_h$, (respectively to an ancestor of v) is transformed into $\mathsf{ptr}(\nu', d)$ that starts in ν' and points to u (respectively to ν). By Lemma 2, there are at most D such pointers $\mathsf{ptr}(u'', d)$. We observe that there is no need to store pointers to the node u_h in the minitree T_v because such pointers are only relevant for the descendants of u_h that do not belong to T_v.

Compressed suffix trees. The contracted tree T^c consists of $O(n/(D \log n))$ nodes, and thus it would require just $O(n/D)$ bits. The minitrees contain $O(\sigma z)$ nodes, but still an edge of a minitree can be labeled with a string of length $\Theta(n)$. Instead of representing the contracted tree and the minitrees separately, we use Sadakane's compressed suffix tree (CST) [63] to represent the topology of the whole T in $O(n)$ bits, and a compressed representation [35] of the global suffix array (SA) of the string collection, which takes $O(n \log \sigma)$ bits. This SA representation finds the suffix array interval $[l, r]$ of P in time $O(p/\log_\sigma n + \log^\varepsilon_\sigma n)$ for any constant $\varepsilon > 0$, and a lowest-common-ancestor query for the l-th and r-th leaves of T finds the locus u of P in $O(1)$ additional time. A bitmap $M[1, n]$ marks which nodes are minitree roots, and another bitmap $C[1, n]$ marks which nodes are fat or minitree roots. Both are indexed with preorder numbers of T, which are computed in constant time on the CST. With a simple $O(n)$-bit structure for constant-time marked ancestor queries that is compatible with our CST representation [61, Sec. 4.1], we can find the lowest ancestor v of u marked in M or in C. With bitmap M we can identify whether u belongs to a minitree rooted at v (with local preorder $preorder_{T_v}(u) = preorder_T(u) - preorder_T(v)$ and depth $depth_{T_v}(u) = depth_T(u) - depth_T(v)$; depths are also computed in constant time). Similarly, with C and M we can identify whether u is a fat node, and find out its preorder in T^c as $preorder_{T^c}(u) = rank_1(C, preorder_T(u))$, in constant time. Its depth in T^c can be stored in an array indexed by $preorder_{T^c}$ in $O(n/D)$ bits.

Contracted grid. We define the grid of the contracted tree T^c as in Section 2, considering all pointers ptr^c. Those are either ptr pointers leaving from fat nodes, or leaving from inside some minitree T_v and pointing above v. For every fat node and for every minitree T_v, and for each document d, there is at most one such pointer by Lemma 2. Thus each node of T^c contributes at most D pointers ptr^c. As there are $O(n/(D \log n))$ nodes, there are $O(n/\log n)$ pointers ptr^c in T^c.

Therefore, the grid associated with T^c is of width $O(n/\log n)$ and height $O(n/(D \log n))$. As there are $O(n/\log n)$ distinct weights among the ptr^c pointers, we only store their ranks. This change does not alter the result of any top-k query. Therefore the data structure of Theorem 4 on T^c occupies $O(n/\log n)$ words, or $O(n)$ bits.

Local grids. The local grid for a minitree T_v collects the pointers ptr local to T_v. It also includes at most D pointers towards its dummy root ν, and at most D pointers coming from its node ν', if it has one. Overall T_v contains $O(\sigma z)$ pointers and $O(\sigma z)$ nodes, so its grid is of size $O(\sigma z) \times O(\sigma z)$. The weights are also replaced by their ranks, so they are also in the range $[1, O(\sigma z)]$. Using Theorem 4 the minitree requires $O(\log(\sigma z))$ bits per node. Added over all the nodes of T that can be inside minitrees, the total space is $O(n \log(\sigma z)) = O(n(\log \sigma + \log D + \log\log n))$. Note that the tree topology is already stored in the CST, so information associated with nodes $u \in T_v$ such as the intervals $[l_u, r_u]$ can be stored in arrays indexed by preorder numbers.

Queries. Given a query pattern P, we find the locus u of P and determine whether u is a fat node or it belongs to a minitree in $O(p/\log_\sigma n + \log^\varepsilon_\sigma n)$ time, as explained. If u is fat, we solve the query on the contracted grid of T^c. Note that this grid does not distinguish among different nodes in the same minitree. But since u is an ancestor either of all nodes in a minitree or of none of them, such distinction is not necessary.

If u belongs to a minitree T_v, we answer the query using the corresponding local grid. This grid does not distinguish where exactly the pointers pointing to ν lead, nor where exactly the pointers that originate in ν' come from. Once again, however, this information is not important in the case where the locus u of P belongs to T_v.

Note that we still need to maintain the global array mapping x-coordinates to document identifiers. This requires $O(n \log D)$ bits.

Theorem 7 *Let $\mathcal{D}$ be a collection of D documents over an integer alphabet $[1, \sigma]$ with total length n, and let $w(S, d)$ be a function that assigns a numeric weight to string S in document d, that depends only on the set of starting positions of occurrences of S in d. Then there exists an $O(n(\log D + \log\sigma + \log\log n))$-bit data structure that, given a string P and an integer k, reports k documents d containing P with highest $w(P, d)$ values, in decreasing order of $w(P, d)$, in $O(p/\log_\sigma n + \log^\varepsilon_\sigma n + k)$ time, for any constant $\varepsilon > 0$.*

In case $p < \lg^{1+\varepsilon}_\sigma n$, we can use a different compressed suffix array [11], which gives $O(p)$ search time, and the overall time becomes $O(p + k)$.

6.1 A Smaller Structure when using Term Frequencies

In this section we show that the space usage can be further improved if $w(P, d) = tf$, i.e., when the data structure must report k documents in which P occurs most frequently.

Our improvement is based on applying the approach of Theorem 7 to each minitree. The nodes of a minitree are grouped into microtrees; if the structure for a microtree still needs too much space, we store them in a compact form that will be described below.

Let $z' = \sigma D \lg m$, where m is the number of nodes in a minitree $\mathcal{T}$. Using the same method as in Theorem 7, we divide the nodes of $\mathcal{T}$ into $O(m/z')$ minifat nodes and $O(m/(D \log m))$ microtrees, so that each microtree contains $O(\sigma z')$ nodes. We construct the contracted minitree and the contracted grid for $\mathcal{T}$ as in Theorem 7. Both the contracted minitree and the structure for the contracted grid use $O(m)$ bits. We can traverse a path in the microtree using the implementation of the global suffix tree described in the previous section, as well as compute local preorders and depths, and attach satellite information to microtree nodes.

For every microtree $\mathcal{T}_v$, we define the dummy nodes ν and ν'. Pointers in $\mathcal{T}_v$ are transformed as in the proof of Theorem 7 with regard to ν and ν'.

Let m' denote the number of nodes in a microtree. If $\log m' = O(\log \sigma + \log D)$, we implement the local grid data structure described in Theorem 7 for a microtree. In this case we can store a data structure for a microgrid in $O(\log(m' + D)) = O(\log \sigma + \log D)$ bits per node.

If, instead, $\log m' = \omega(\log \sigma + \log D)$, since $\log m' = O(\log(\sigma z')) = O(\log \sigma + \log D + \log\log m)$, it follows that $\log m' = O(\log\log m)$. Hence, the size of the microtree is $m' = \log^{O(1)} m = (\log \sigma + \log D + \log\log n)^{O(1)} = (\log\log n)^{O(1)}$. The total number of pointers in the microtree is also $m'' = m' + O(D) = (\log\log n)^{O(1)}$ (since $\log D = o(\log m')$). Since all the grids in $m'' \times m'$, with one point per x-coordinate, and weights in $[1, m'']$, can be expressed in $m''(\lg m' + \lg m'') = o(\log n)$ bits, we can store pre-computed answers for all possible queries on all possible small microtrees. The only technical difficulty is that weights of some pointers in a microtree can be arbitrarily large. However, as explained below, it is not necessary to know the exact weights of pointers to answer a query on a small microtree.

All pointers $\mathsf{ptr}(u_l, d)$ where u_l is a leaf node and $u_l \neq \nu'$ have weight 1. The weights of $\mathsf{ptr}(\nu', d)$ can be arbitrarily large. The weight of a pointer $\mathsf{ptr}(u, d)$ for an internal node u equals to the sum of weights of all pointers $\mathsf{ptr}(u', d)$ for the same document d that lead to u. Thus the weight of $\mathsf{ptr}(u, d)$ can also be large. We note that there is at most one pointer $\mathsf{ptr}(\nu', d)$ for each d. Therefore the weight of each pointer $\mathsf{ptr}(u, d)$ can be expressed as the sum $w_1(u) + w_2(u)$, where $w_1(u)$ is the weight of $\mathsf{ptr}(\nu', d)$ or 0 and $w_2(u) \leq m'$. In other words, the weight of $\mathsf{ptr}(u, d)$ differs from the weight of $\mathsf{ptr}(\nu', d)$ by at most m'.

Let the set $\mathcal{N}$ contain the weights of all pointers $\mathsf{ptr}(u_l, d)$ and $\mathsf{ptr}(\nu', d)$. Let $\mathcal{N}' = \{\lfloor w/m' \rfloor, \lfloor w/m' \rfloor + 1 \mid w \in \mathcal{N}\}$. To compare the weights of any two pointers it is sufficient to know (i) the tree topology (ii) for every leaf u_l, the document d whose suffix is stored in u_l (iii) for every $\mathsf{ptr}(\nu', d)$, the pair $(\mathrm{rank}(\lfloor w/m' \rfloor, \mathcal{N}'), w \mod m')$ where w is the weight of $\mathsf{ptr}(\nu', d)$. There are $o(n/\log n)$ possible combinations of tree topologies and possible pairs $(\mathrm{rank}(\lfloor w/m' \rfloor, \mathcal{N}'), w \mod m')$. Hence, we can store answers to all possible queries for all microtrees in a global look-up table of size $o(n)$ bits.

The topology of a microtree can be stored in $O(m')$ bits. We can specify the index of the document d stored in a leaf u_l with $\lg D$ bits. We can specify each pair $(\mathrm{rank}(\lfloor w/m' \rfloor, \mathcal{N}'), w \mod m')$ with $O(\log m')$ bits. Since $D = O(m'/\log m')$, information from item (iii) can be stored in $O(m')$ bits. Thus each microtree can be stored in $O(m' \log D)$ bits if $\log m' = \omega(\log \sigma + \log D)$. Summing up, our data for a minitree uses $O(m(\log \sigma + \log D))$ bits. Therefore the total space usage

is $O(n(\log\sigma + \log D))$ bits.

A query for a pattern P is answered by locating the locus u of P. If u is a fat node in T, the query is answered by a data structure for the contracted grid. If u belongs to a minitree $\mathcal{T}$ and u is a minifat node, we answer the query by employing the data structure for the contracted grid of $\mathcal{T}$. If u belongs to a microtree $\mathcal{T}_v$, the query is answered either by a microgrid data structure or by a table look-up.

Theorem 8 *Let $\mathcal{D}$ be a collection of strings over an integer alphabet $[1,\sigma]$ with total length n, and let $tf(P,d)$ denote the number of occurrences of P in d. Then there exists an $O(n(\log D + \log\sigma))$ bit data structure that, given a string P and an integer k, reports k documents d containing P with highest $tf(P,d)$ values, in decreasing order of $tf(P,d)$, in $O(p/\log_\sigma n + \log^\varepsilon_\sigma n + k)$ time, for any constant $\varepsilon > 0$.*

Again, we can obtain $O(p+k)$ query time when $p < \lg^{1+\varepsilon}_\sigma n$.

7 Parameterized Top-k Queries

In this section we consider the extended problem where a parameter $\texttt{par}(P,d)$ is associated with the documents in relation to the pattern P, and we want to recover only documents d where $\texttt{par}(P,d) \in [\tau_1,\tau_2]$. We include this extension in our framework by adding one more dimension to our slim grids of Section 5.3. Since the grids are slim, one of the dimensions is small, which we exploit to obtain good query times.

We first show that two-dimensional top-k queries can be solved in linear space and $O((k+\log n)\log^\varepsilon n)$ time, for any constant $\varepsilon > 0$, on $n \times n$ grids with weights in $[1, O(n)]$. Then we extend the result (although we do not build on it) to three-dimensional grids where one of the dimensions has an extension polylogarithmic in n, obtaining the same space and time as in two dimensions. Finally, in Section 7.3, we show how this geometric structure is used to solve our parameterized top-k queries by replacing our original two-dimensional grids.

7.1 Faster Two-Dimensional Queries

In this section we improve a recent data structure that supports two-dimensional top-k queries [54, Sec. 5]. The structure is similar to our wavelet tree W described in the proof of Lemma 4. In addition, for the points stored at any node of W, it stores an RMQ data structure that gives in constant time the position of the point with maximum weight within any interval. As explained, this structure [27] uses $O(t)$ bits if the node of W handles t points, and thus the total space of this extended wavelet tree W is $O(n)$ words for an $O(n) \times O(n)$ grid.

They [54] show how to support top-k queries in a general interval $[a,b]\times[c,d]$ by first identifying the $O(\log n)$ nodes $v \in W$ that cover $[c,d]$, mapping the interval $[a,b]$ to $[a_v,b_v]$ in all those nodes v, and setting up a priority queue with the maximum-weight point of each such interval. Now, they repeat k times the following steps: (i) extract the maximum weight from the queue and report it; (ii) replace the extracted point, say $x \in [a_v,b_v]$, by two points corresponding to the maxima in the ranges $[a_v, x-1]$ and $[x+1, b_v]$, prioritized by those maximum weights.

Their total time is $O((k+\log n)\log n)$ if using linear space. The $O(\log n)$ extra factor is due to the need to traverse W in order to find out the real weights, so as to compare weights from

different nodes. However, those weights can be computed in time $O(\log^\varepsilon n)$ and using $O(n \log n)$ extra bits [18, 57, 17]. The operations on the priority queue can be carried out in $O(\log \log n)$ time if the weights are integers in $[1, O(n)]$ [67]. Thus we have the following result.

Lemma 7 *Given a grid of $n \times n$ points, there exists a data structure that uses $O(n)$ words of space and reports k most highly weighted points in a range $Q = [a, b] \times [c, d]$ in $O((k + \log n) \log^\varepsilon n)$ time, for any constant $\varepsilon > 0$. The structure is built in $O(n \log n)$ time.*

Note this technique automatically admits being used in online mode (i.e., without knowing k in advance), since we have not made use of k to speed up the priority queue as in previous sections. We can easily stop the computation at some k and resume it later.

7.2 Limited Three-Dimensional Queries

In this section we slightly extend the scenario considered above. We assume that each point has an additional coordinate, denoted z, and that $z \leq \lg^\alpha n$ for a constant $\alpha > 0$. Top-k points in a three-dimensional range $[a, b] \times [c, d] \times [\beta, \gamma]$ must be reported sorted by their weights. Such queries will be further called *limited three-dimensional top-k queries*. We can obtain the same result as in Lemma 7 for these queries.

Instead of a binary wavelet tree, we use a multiary one [25], with node degree $\lg^\varepsilon n$ and height $O(\log n / \log \log n)$. Now each node $v \in W$ has associated a vector B_v so that $B_v[i]$ contains the index of the child in which the i-th point of v is stored. Each element in B_v needs $O(\log(\log^\varepsilon n)) = O(\log \log n)$ bits, adding up to $O(n \log \log n)$ per wavelet tree level and $O(n \log n)$ bits in total. Using B_v and some auxiliary data structures, we can obtain the weight of any point at any node in $O(\log^\varepsilon n)$ time [57]. These structures also require $O(n \log n)$ bits.

We regard the t points of each node v as lying in a two-dimensional grid of x- and z-coordinates. Instead of one-dimensional RMQs on the x-coordinates $[a_v, b_v]$, we issue two-dimensional RMQs on $[a_v, b_v] \times [\beta, \gamma]$. The wavelet tree of the basic two-dimensional RMQ data structure [54] (not our result of Lemma 7) handles $n \times m$ grids in $O(n \log m)$ bits of space and answers RMQs in time $O(\log^2 m)$. In our case $m < \lg^\alpha n$ and thus the space is $O(n \log \log n)$ bits and the query time is $O((\log \log n)^2)$. Thus the space of the two-dimensional data structures is of the same order of that used for vectors B_v, adding up to $O(n \log n)$ bits. As one-dimensional RMQs are built in linear time, the total construction time is $O(n \log n)$.

Now we carry out a procedure similar to that of the two-dimensional version. The range $[a, b]$ is covered by $O(\log^{1+\varepsilon} n / \log \log n)$ nodes, since the wavelet tree has $O(\log n / \log \log n)$ levels and there can be $O(\log^\varepsilon n)$ covering nodes per level. We obtain all their (two-dimensional) range maxima, insert them in a priority queue, and repeat k times the process of extracting the highest weight and replacing the extracted point $x \in [a_v, b_v]$ by the next highest weighted point in $[a_v, b_v]$ (thus we are running these range maxima queries in online mode).

The two-dimensional RMQ structures at nodes v cannot store the absolute weights within overall linear space. Instead, when they obtain the x-coordinate of their local grid, this coordinate x_v is mapped to the global x-coordinate in $O(\log^\varepsilon n)$ time, using the same technique as above. Then the global array of weights is used. Hence these structures find a two-dimensional maximum weight in time $O(\log^\varepsilon n \log \log n)$, not $O((\log \log n)^2)$. This is repeated over $O(\log^{1+\varepsilon} n / \log \log n)$

nodes, and then iterated k times. The overall time is $O((k + \log^{1+\varepsilon} n/\log\log n)\log^{\varepsilon} n \log\log n)$, which is of the form $O((k + \log n)\log^{\varepsilon} n)$ by adjusting ε. The times to handle the priority queue are negligible [67].

Lemma 8 *Given a grid of $n \times n \times \lg^{\alpha} n$ points, for a constant $\alpha > 0$, there exists a data structure that uses $O(n)$ words of space and reports k most highly weighted points in a range $Q = [a, b] \times [c, d] \times [\beta, \gamma]$ in $O((k + \log n)\log^{\varepsilon} n)$ time, for any constant $\varepsilon > 0$. It is built in $O(n \log n)$ time.*

Again, this result holds verbatim in online mode.

7.3 The Final Result

We divide the grid into horizontal stripes of height $r = \lceil \lg^{c+1+\varepsilon} n \rceil$ for any constant c, much as in Section 5.5. We store a data structure for limited three-dimensional top-k queries for each slim grid, taking y as the limited coordinate. A query $[a, b] \times [0, h] \times [\tau_1, \tau_2]$ is processed just as in Section 5.5, with the only difference that the queries $[a, b] \times [\tau_1, \tau_2] \times [0, y']$ to the local grids now require $O(\log^{1+\varepsilon} n)$ initialization time and then $O(\log^{\varepsilon} n)$ time per element retrieved, according to Lemma 8. Then, we initialize our global query $\mathcal{Q}$ in time $O(\lceil h/r \rceil \log^{1+\varepsilon} n) = O(h/\log^{c} n + \log^{1+\varepsilon} n)$, and then extract each new result in time $O(\log^{\varepsilon} n)$. The time of the priority queue is blurred by adjusting ε. Hence, the total query time is $O(h/\log^{c} n + (k + \log n)\log^{\varepsilon} n)$, and Theorem 6 is proved.

8 Conclusions

We have presented an optimal-time and linear-space solution to top-k document retrieval, which can be used on a wide class of relevance measures and subsumes in an elegant and uniform way various previous solutions to other ranked retrieval problems. We have also presented dynamic variants, space-reduced indexes, and structures that solve extensions of the basic problem. The solutions reduce the problem to ranked retrieval on multidimensional grids, where we also present improved results, some tailored to this particular application, some of more general interest.

After the publication of the conference version of this article [53], Shah et al. [65] showed how to achieve the optimal $O(k)$ time once the locus of P is known. This is in contrast to our original result, where we used time $O(p + k)$ after having spent time $O(p)$ to find the locus. Their improvement allows one to use these techniques in other scenarios where the locus is obtained in some other way, without the need to search for it directly using P. They also extend the results to the important case of external memory. The new results we obtain in this article about how to find the locus in RAM-optimal time $O(p/\log_{\sigma} n)$, and how to handle the dynamic scenario, nicely complement those results and add up to a rather complete solution to the problem.

It is also worth mentioning that the journal version of the original paper of Hon et al. has recently appeared as well [40]. Here they show how to obtain $O(p + k)$ time if the top-k results are not to be returned sorted by relevance. Our original [53] and present solutions do obtain the results in decreasing relevance order.

There are several relevant research directions, on which we comment next.

RAM optimality. In our previous conference version we had achieved time $O(p+k)$, which was optimal only in the comparison model (although we used RAM-based techniques). Now we have improved this result to $O(p/\log_\sigma n+k)$, which is optimal in general in the RAM model (considering the case $\log D = \Theta(\log n)$), because it is the size in words of the input plus the output of the query. Achieving $O(p/\log_\sigma n)$ time on the suffix tree, without any polylogarithmic additive penalty, is an interesting result by itself, and we have obtained it without altering the topology of the suffix tree (which is crucial for the invariants of Hon et al. [41] to work). However, we do not know if our solution is optimal when there are very few distinct documents, $\log D = o(\log n)$. The question of whether $O(p/\log_\sigma n + k/\log_D n)$ time can be achieved is still open.

Construction time. Without considering the cost to compute weights $w(\mathrm{path}(u), d)$ for all pointers ptr in the suffix tree, the construction time of Hon et al. [41] (which achieves suboptimal query time) is $O(n)$. The time to build our grid structure is $O(n \log n)$, to which we must add $O(n \log^\varepsilon n)$ randomized time to achieve RAM-optimal search time in the suffix tree traversal (or $O(n \operatorname{polylog} n)$ deterministic time). Is it possible to achieve linear, or at least $O(n \log n)$, deterministic construction time for our data structures?

Dynamic optimality. In our dynamic variant, the static RAM-optimal search time in the suffix tree becomes $O(p(\log\log n)^2/\log_\sigma n + \log n)$. There are schemes that do better for large σ, for example $O(p+(\log\log\sigma)^2)$ time [26]. Although they do not support deletions yet, this seems to be possible. On the other hand, we obtained $O(\log^{1+\varepsilon} n)$ update time per symbol. A general question is, which is the best search time we can obtain in the dynamic scenario?

Practical results. Our solutions are not complex to implement and do not make use of impractical data structures. A common pitfall to practicality, however, is space usage. Even achieving linear space (i.e., $O(n \log n)$ bits) can be insufficient. We have shown that our structure can use, instead, $O(n(\log\sigma + \log D))$ bits for the *tf* measure (and slightly more for others), but the constants are still large. There is a whole trend of reduced-space representations for general document retrieval problems with the *tf* measure [64, 70, 41, 21, 32, 38, 12, 31, 68, 39, 56]. The current situation is as follows [52]: One trend aims at the least space usage. It has managed to use just $D\lg(n/D) + O(D) + o(n)$ bits on top of a compressed suffix array of the collection, and the best time complexity it has achieved is $O(p + k\log^2 k \log^{1+\varepsilon} n)$ for any constant $\varepsilon > 0$ [56]. Another trend adds to the space the so-called *document array* [51], which uses $n\lg D + o(n\log D)$ bits and enables faster solutions. Currently the fastest one achieves time $O(p + k\log^* k)$ [56]. This is very close to optimal, but not yet our $O(p/\log_\sigma n + k)$ time.

In practice, the most compact implementation in this trend [55] reaches about 1–2 times the text size (including a representation of the text) and retrieves each of the top-k results within milliseconds. Implementations of the ideas we propose in this article [44, 33] make use of the fact that, under very general probabilistic models, the average height of the suffix tree (and hence of our grids) is $O(\log n)$ [66]. This enables simple implementations of our grid-based index that use up to 2.5–3.0 times the text size (including the text) and, although not reaching optimal query time, return each answer within microseconds.

More complex queries. In the long term, the most interesting open questions are related to extending the one-pattern results to the bag-of-words paradigm of information retrieval. Our model easily handles single-word searches, and also phrases (which is quite complicated with inverted indexes [72, 9], particularly if their weights have to be computed). Handling a set of words or phrases, whose weights within any document d must be combined in some form (for example using the $tf \times idf$ model) is more challenging. Recent results [45] give $\Omega(\sqrt{n})$-time lower bounds for some basic and natural queries that combine two patterns, unless there is a breakthrough on the boolean matrix multiplication problem. Instead, one can aim at complexities related to the results achieved with inverted lists on the simpler natural language model. It is interesting to note that our online result allows simulating the left-to-right traversal, in decreasing weight order, of the virtual list of occurrences of any string pattern P. Therefore, for a bag-of-word queries, we can emulate any algorithm designed for inverted indexes that stores those lists in explicit form [60, 6], therefore extending any such technique to the general model of string documents.

Acknowledgements

We thank Djamal Belazzougui and Roberto Grossi for helpful pointers.

References

[1] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4/5):434–449, 1996.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 534–544, 1998.

[3] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994.

[4] A. Amir, M. Farach, R. M. Idury, J. A. Lapoutre, and A. A. Schaffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

[5] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):article 13, 2007.

[6] V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 372–379, 2006.

[7] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.

[8] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[9] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011.

[10] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*, LNCS 6346, pages 427–438 (part I), 2010.

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

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

[13] M. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th Annual European Symposium on Algorithms (ESA)*, LNCS 2461, pages 152–164, 2002.

[14] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Theoretical Informatics Symposium (LATIN)*, LNCS 1776, pages 88–94, 2000.

[15] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[16] G. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz. Online sorted range reporting. In *Proc. 20th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 5878, pages 173–182, 2009.

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

[18] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[19] R. Cole and R. Hariharan. Dynamic lca queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[20] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

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

[22] P. Dietz and R. Raman. Persistence, randomization and parallelization: On some combinatorial games and their applications (abstract). In *Proc. 3rd Workshop on Algorithms and Data Structures (WADS)*, LNCS 709, pages 289–301, 1993.

[23] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[24] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.

[25] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

[26] J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 160–171, 2015.

[27] 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.

[28] R. Fleischer. A simple balanced search tree with O(1) worst-case update time. *International Journal of Foundations of Computer Science*, 7(2):137–149, 1996.

[29] M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[30] H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

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

[32] 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.

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

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

[35] 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.

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

[37] W.-K. Hon, M. Patil, R. Shah, and S.-Bin Wu. Efficient index for retrieving top-k most frequent documents. *Journal of Discrete Algorithms*, 8(4):402–417, 2010.

[38] 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)*, LNCS 7354, pages 173–184, 2012a.

[39] 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.

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

[41] 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.

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

[43] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

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

[45] K. G. Larsen, J. I. Munro, J. S. Nielsen, and S. V. Thankachan. On hardness of several string indexing problems. In *Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 8486, pages 242–251, 2014.

[46] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[47] Y. Matias, S. Muthukrishnan, S.C. Sahinalp, and J. Ziv. Augmenting suffix trees, with applications. In *Proc. 6th European Symposium on Algorithms (ESA)*, LNCS 1461, pages 67–78, 1998.

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

[49] E. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

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

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

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

[53] 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.

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

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

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

[57] Y. Nekrich. A linear space data structure for orthogonal range reporting and emptiness queries. *International Journal of Computational Geometry and Applications*, 19(1):1–15, 2009.

[58] M. H. Overmars. *The Design of Dynamic Data Structures*. LNCS 156. Springer, 1983.

[59] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[60] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Sicence*, 47(10):749–764, 1996.

[61] L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):article 53, 2011.

[62] M. Ružić. Constructing efficient dictionaries in close to sorting time. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 5125, pages 84–95 (part I), 2008.

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

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

[65] R. Shah, C. Sheng, S. V. Thankachan, and J. Vitter. Top-k document retrieval in external memory. In *Proc. 21st Annual European Symposium on Algorithms (ESA)*, LNCS 8125, pages 803–814, 2013.

[66] W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM Journal on Computiing*, 22(6):1176–1198, 1993.

[67] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences*, 69(3):330–353, 2004.

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

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

[70] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 205–215, 2007.

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

[72] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):article 6, 2006.