Top-k Document Retrieval in Optimal Time and Linear Space [*]

Gonzalo Navarro[†] Yakov Nekrich[‡]

Department of Computer Science, University of Chile

Abstract

We describe a data structure that uses $O(n)$-word space and reports k most relevant documents that contain a query pattern P in optimal $O(|P| + k)$ time. Our construction supports an ample set of important relevance measures, such as the frequency of P in a document and the minimal distance between two occurrences of P in a document. 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, where σ is the alphabet size and D is the total number of documents.

1 Introduction

Design of efficient data structures for document (i.e., string) collections is an important problem studied in the information retrieval and pattern matching communities. Due to the steadily increasing volumes of data, it is often necessary to generate a list $L(P)$ of documents containing a string pattern P in decreasing order of relevance. Since the list $L(P)$ can be very large, in most cases we are interested in answering *top-k queries*, i.e., reporting only the first k documents from $L(P)$ for a parameter k given at query time. In this paper we show that a document collection can be stored in $O(n)$ words (where n is the sum of the string lengths in the collection), so that top-k queries are answered in optimal $O(|P|+k)$ time. Our construction supports a number of relevance measures that will be defined later in this section.

Previous Work. Inverted files [7, 23] that store lists of documents containing certain keywords are frequently used in practical implementations of information retrieval methods. However, inverted files only work in the situation when query patterns belong to a fixed pre-defined set of strings (keywords). Surprisingly, the general *document listing problem*, i.e., the problem of reporting all documents that contain an arbitrary query pattern P, was not studied until the end of the 90s. Matias et al. [24] 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 $O(|P| \log D + \text{docc})$ time, where D is the total number of documents in the collection. Muthukrishnan [27] presented a data structure that uses $O(n)$ words of space and answers document listing queries in $O(|P| + \text{docc})$ time. Muthukrishnan [27] 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 $O(|P| + \texttt{occ})$ time, where $\texttt{occ}$ 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 $\mathit{tf}(P, d)$, the number of times P occurs in d. Two other important examples are $\mathit{mind}(P, d)$, the minimum distance between two occurrences of P in d, and $\mathit{docrank}(d)$, an arbitrary static rank assigned to a document d. Some more complex measures have also been proposed. Hon et al. [19] presented a solution for the top-k document retrieval problem for the case when the relevance measure is $\mathit{tf}(P, d)$. 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 [21] presented a general solution for a wide class of relevance measures. Their data structure uses linear space and needs $O(|P| + k \log k)$

[*]Partially funded by Fondecyt Grant 1-110066, Chile, and by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

[†]Email gnavarro@dcc.uchile.cl

[‡]Email yakov.nekrich@googlemail.com

time to answer a top-k query. A recent $O(n)$ space data structure [22] 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 Result. Hon et al.'s results [21] 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 optimal $O(|P| + k)$ time. Our solution supports the same relevance measures as Hon et al. [21].

THEOREM 1.1. *Let $\mathcal{D}$ be a collection of strings (called* documents*) of total length n, 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 and an integer k, reports k documents d containing P with highest $w(P, d)$ values, in decreasing order of $w(P, d)$, in $O(|P| + k)$ time.*

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

We also show how the space can be reduced from $O(n \log n)$ bits to $O(n(\log \sigma + \log D + \log \log n))$, where σ is the size of the alphabet of the strings. 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.

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 *mind* measure solves the K-repeats problem in optimal time and linear space. We remind that Muthukrishnan [27] had solved the K-repeats problem using $O(n \log n)$-word space; later Hon et al. [21] 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 [22], for example when the rank *docrank*(d) repends only on d (we just use $w(P, d) = \textit{docrank}(d)$), or where in addition we exclude those d where P appears less than K times for a fixed pre-defined K (we just use $w(P, d) = \textit{docrank}(d)$ if $\textit{tf}(P, d) \geq K$, else 0).

Moreover, we can also answer queries for some relevance metrics not included in Theorem 1.1. For instance, we might be interested in reporting all documents d with $\textit{tf}(P, d) \times \textit{idf}(P) \geq \tau$, where $\textit{idf}(P) = \log(N/\textit{df}(P))$ and $\textit{df}(P)$ is the number of documents where P appears [3]. Using the $O(n)$-bit structure of Sadakane [34], we can compute *idf*(P) in $O(|P|)$ time. To answer the query, we use our data structure of Theorem 1.1 in online mode on measure *tf*: For every reported document d we find $\textit{tf}(P, d)$ and compute $\textit{tf}(P, d) \times \textit{idf}(P)$; the procedure is terminated when a document d_l with $\textit{tf}(P, d_l) \times \textit{idf}(P) < \tau$ is encountered. Thus we need optimal $O(|P| + \texttt{occ})$ time to report all $\texttt{occ}$ documents with $\textit{tf} \times \textit{idf}$ scores above a threshold.

Furthermore, we can extend the top-k ranked retrieval problem by allowing a further parameter $\texttt{par}(P, d)$ to be associated to 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 1.2. *Let $\mathcal{D}$ be a collection of documents of total length n, a 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, 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 $O(|P| + (k + \log n) \log^{\varepsilon} n)$ time, for any constant $\varepsilon > 0$.*

Our solutions map these document retrieval problems into range search problems on multidimensional spaces, where points in the grids have associated weights.

2 Top-k Framework

In this section we overview the framework of Hon, Shah, and Vitter [21]. 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 for a collection of documents $d_1, \ldots, d_D$; 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 labels of all 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, e.g., Gusfield [18] for an extensive description of the generalized suffix tree and related notions.

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 ordering the string terminators 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 $path(v)$. The following statements hold.

Lemma 2.1. ([21], Lemma 4) *The total number of pointers $\mathsf{ptr}(\cdot, \cdot)$ in T is bounded by $O(n)$.*

Lemma 2.2. ([21], Lemma 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.*

Moreover, in terms of Lemma 2.2, 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 $w(P, d) = w(path(u), d)$ for any measure $w(\cdot, \cdot)$ considered in Theorem 1.1.

Geometric interpretation. We index the nodes of T in the following way: All nodes of T are visited in pre-order; we also initialize an index $i \leftarrow 0$. When a node v is visited, if v is marked with values $d_{v_1}, \dots, d_{v_j}$, we assign indexes $i+1, \dots, i+j$ 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}, $1 \le t \le j$, 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 $(i+t, depth(u_t))$, where $depth$ denotes the depth of a node; $depth(\nu) = 0$. 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 p encodes. We observe that all 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 $O(|P|)$ time. By Lemma 2.2, 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 $y \in [0, depth(v) - 1]$ for every document d that contains P. Therefore, reporting top-k documents is equivalent to the following query: among all points in the three-sided range $[l_v, r_v] \times [0, depth(v) - 1]$, 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 1.1 is an immediate corollary of it, as $h = depth(v) - 1$ and $depth(v) \le |P|$.

Theorem 2.1. *A set of n weighted points on an $n \times n$ grid can be stored in $O(n)$ words of space, so that for any $1 \le k, h \le n$ and $1 \le a \le b \le n$, k most highly weighted points in the range $[a, b] \times [0, h]$ can be reported in decreasing order of their weights in $O(h + k)$ time.*

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 $\mathtt{par}(path(v), d)$. We encode a pointer $\mathsf{ptr}(v, d_{v_t})$ as a three-dimensional point $(i+t, depth(u_t), \mathtt{par}(path(v), d_{v_t}))$, where i, t, and u_t are defined as in the case of nonparameterized top-k queries. All documents that contain a pattern P (with locus v) and satisfy $\tau_1 \le \mathtt{par}(P, d) \le \tau_2$ correspond to unique points in the range $[l_v, r_v] \times [0, depth(v) - 1] \times [\tau_1, \tau_2]$. Hence, reporting top-$k$ documents with $\mathtt{par}(P, d) \in [\tau_1, \tau_2]$ is equivalent to reporting top-k points in a three-dimensional range. The following result is proved in Section 7, and Theorem 1.2 is an immediate corollary of it.

Theorem 2.2. *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 $1 \le k, h \le n$, $1 \le a \le b \le n$, and $1 \le \tau_1 \le \tau_2 \le n$, 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 $O(h + (k + \log n) \log^{\varepsilon} n)$ time.*

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

In this section we give a data structure that does not yet achieve the desired $O(h + k)$ 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 a set 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.

LEMMA 3.1. *Assume that $m \le n$ and let $0 < f < 1$ be a constant. There exists a data structure that uses $O(m \log m)$ additional bits of space and answers three-sided top-k queries for a set of m points on an $m \times n$ grid in $O(m^f + k)$ time.*

Proof. The idea is to partition the points of S by weights, where the weights are disregarded inside each partition. Those partitions are also refined into a 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 $m^{\Theta(f)}$ yields constant height and thus constant space per point.

More precisely, we partition S into classes $S_1, \ldots, S_r$, where $r = m^{f'}$ for a constant $0 < f' < f$. For any $1 \le i < j \le r$, the weight of any point $p_i \in S_i$ is larger than the weight of any point $p_j \in S_j$. For $1 \le i < r$, S_i contains $m^{1-f'}$ 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 $O(1/f')$.

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 Lemmas 3.2 and 3.3; note they do not involve weights. 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 $S \cap Q$. There are $O(m^{f'})$ 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 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 $k_i = |S_i \cap Q| \le k$, we report all 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 $O(m^{f'})$ classes S_i and spend $O(m^{f'} \log m + k) = O(m^f + k)$ time in total.

When the list L is completed, we can sort it in $O(m^f + k)$ time. If L contains $k < m^{f'}$ points, L can be sorted in $O(k \log k) = O(m^f)$ time. If L contains $k \ge m^{f'}$ 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 $f' \log_2 m$ bits per pass the radix sort runs in time $O(k)$.

As for the space, the structures in Lemmas 3.2 and 3.3 require $O(\log m)$ bits per point. Each point of S belongs to $O(1/f')$ 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.

Counting and reporting points. It remains to describe the data structures that answer three-sided counting and reporting queries, with no weights involved.

LEMMA 3.2. *Let $v \le m \le n$. There exists a data structure that uses $O(v \log m)$ additional bits and answers three-sided range counting queries for a set S of v points on an $m \times n$ grid in $O(\log v)$ time.*

Proof. Using the rank space technique [14], we reduce the problem of counting on an $m \times n$ grid to the problem of counting on a $v \times n$ grid: let

$\tau(p.x, p.y) = (\mathrm{rank}(p.x, S_x), p.y)$, where the set S_x consists of the x-coordinates of all points in S and $\mathrm{rank}(a, S) = |\{ e \in S \,|\, e \leq a \}|$. Then the mapped set is $S' = \{\tau(p.x, p.y), (p.x, p.y) \in S\}$. Let $\mathrm{pred}(a, S) = \max\{ e \in S \,|\, e \leq a \}$ and $\mathrm{succ}(a, S) = \min\{ e \in S \,|\, e \geq a \}$. Then a query $[a, b] \times [0, h]$ on S is equivalent to a query $[a', b'] \times [0, h]$ on S', where $a' = \mathrm{rank}(\mathrm{succ}(a, S_x), S_x)$ and $b' = \mathrm{rank}(\mathrm{pred}(b, S_x), S_x)$. Using standard binary search on an array, we can find a' and b' in $O(\log v)$ time and $v \log m$ bits of space.

We store the points on a variant of the wavelet tree data structure [17]. 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 p points stores a bitmap $B_v[1..p]$, 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 $p + o(p)$ bits of space [26], 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.

We can easily answer range counting queries $[a', b'] \times [0, h]$ on W [25]. 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, if the maximal y-coordinate in the left child is larger than h, we just visit the left child. The time is $O(1)$ per tree level.

Lemma 3.3. *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.*

Proof. We can reduce the problem of reporting on an $m \times n$ grid to the problem of reporting on a $v \times n$ grid, as described above. The query time is increased by an additive $O(\log v)$ factor, and the space usage increases by $O(v \log m)$ bits of space. Now 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 [13]. 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 $rmq(c, d) = \arg\ \min_{c \leq i \leq d} Y[i]$ for any c, d. It is well known that with such queries one can recursively retrieve all the points in the three-sided range in $O(occ)$ time; see, for example, the paper of Muthukrishnan [27].

4 An Optimal Time Data Structure

The data structure of Lemma 3.1 gives us an $O(m^f + k)$ time solution, for any constant f, where m is the grid width. In this section we use it to obtain $O(h+k)$ time. The idea is to partition the space into vertical stripes, for different stripe widths, and index each stripe with Lemma 3.1. Then the query is run on the partition of widths m so that the $O(m^f)$ time complexity is dominated by $O(h + k)$. The many partitions take total linear space because the size per point in Lemma 3.1 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 [6] is used. This gives linear space for stripes of width up to $\Omega(\log^\delta n)$. Smaller ones are solved with universal tables.

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.

Structure. Let $f = 1/2$ and $g_j = f^j$ 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 $\Delta_j = n^{g_j} \log^\delta n$ for a constant $1/2 < \delta < 1$ and $j = 1, \ldots, r$. For convenience, we also define $\Delta_0 = n$ and $\Delta'_j = \Delta_j / \log^\delta n = n^{g_j}$. For every $1 \leq j < r$ and for every interval $I_{j,t} = [(t-1)\Delta_j, t\Delta_j - 1]$, we store all points p with $p.x \in I_{j,t}$ in a data structure $E_{j,t}$ implemented as described in Lemma 3.1. $E_{j,t}$ supports three-sided top-k queries in $O((\Delta_j)^{f'} + k)$ time for any constant f', which we set for convenience to any $f' < 1/4$. 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 3.1 all $E_{j,t}$ use $O(n \log(n^{g_j} \log^{\delta} n)) = (1/2^j)O(n \log n) + O(n \log\log n)$ 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. Since $f' < 1/4$, a data structure $E_{j,t}$ supports top-k queries in time $O((\Delta_j)^{f'} + k) = O((n^{g_j} \log^{\delta} n)^{f'} + k) = O(n^{g_{j+2}} + k) = O(\Delta'_{j+2} + k)$ time. For each of the smallest intervals $I_{r,t}$ we store data structures $\overline{E}_{r,t}$ that use $o(\log^{\delta} n)$ words of space (adding up to $o(n)$) and support three-sided top-k queries in $O(h + k)$ time. This data structure will be described at the end of this section.

Queries. Assume we want to report k points with highest weights in the range $[a, b] \times [0, h]$. First, we find the index j, such that $\Delta'_{j+1} > \max(h, k) \ge \Delta'_{j+2}$. The index j can be found in $O(\log\log(h + k))$ time by linear search. If $[a, b]$ is contained in some interval $I_{j,t}$, then we can answer a query in $O(\Delta'_{j+2} + k) = O(h + k)$ 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 $O(h + k)$ 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 store precomputed solutions for some 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 $0 \le v \le \log_2(n/\Delta_j)$, and any $0 \le c \le \Delta'_{j+1}$. All lists $\mathsf{top}_j(\cdot,\cdot,\cdot,\cdot)$ use space $O((n/\Delta_j)(\Delta'_{j+1})^2 \log n) = O(n/\log^{2\delta-1} n)$ words. Hence the total word space usage of all lists $\mathsf{top}_j(\cdot,\cdot,\cdot,\cdot)$ for $2 \le j \le r$ is $O(n \log\log n/\log^{2\delta-1} n) = o(n)$.

Assume that $[a, b]$ spans intervals $I_{j,l+1}, \ldots, I_{j,r-1}$; $[a, b]$ also intersects with intervals $I_{j,l}$ and $I_{j,r}$. Let $a'\Delta_j$ and $b'\Delta_j$ denote the left endpoints of $I_{j,l+1}$ and $I_{j,r}$, 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 $v = \lceil \log_2((b' - a')/\Delta_j) \rceil$ 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_l and L_r 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_l and L_r in $O(h + k)$ time using data structures $E_{j,l}$ and $E_{j,r}$ as explained above. Finally, we can merge L_m, L_l, and L_r in $O(k)$ time; the first k points in the resulting list are the top-k points in $[a, b] \times [0, h]$.

A data structure for an $O(\log^{\delta} n) \times n$ grid. We store a data structure $\overline{E}_{r,j}$ that answers top-k queries on $I_{r,j}$ in $O(k)$ time. We replace the y-coordinates of points by their ranks; likewise, the weights of points are also replaced by their ranks. The resulting sequence X_j contains all mapped points in $I_{r,j}$ and consists of $O(\log^{\delta} n \log\log n)$ bits. There are $O(\log^{4\delta} n)$ queries that can be asked, and the answers require $O(k \log(\log^{\delta} n)) = O(\log^{\delta} \log\log n)$ bits, which is less than one word. Thus we can store a universal look-up table of size $2^{O(\log^{\delta} n \log\log n)} O(\log^{4\delta} n) = o(n)$ words common to all subintervals $I_{r,j}$. This table contains precomputed answers for all possible queries and all possible sequences X_j. Hence, we can answer a top-k query on X_j in $O(k)$ time.

A query on $I_{r,j}$ can be transformed into a query on X_j by reduction to rank space in the y coordinates. This requires in total $O(n \log(\log^{\delta} n))$ bits, or $o(n)$ words, because only the reordered x-coordinates need to be stored; the global array gives the corresponding y-coordinates. Consider a query range $Q = [a, b] \times [0, h]$ on $I_{r,j}$. We can find the rank h' of h among the y-coordinates of points from $I_{r,j}$ in $O(h)$ time by linear search. Then, we can identify the top-k points in $X_j \cap Q'$, 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 $O(h+k)$ time. This completes the proof of Theorem 2.1.

4.1 Online Queries

An interesting extension of the above result is that we can deliver the results in online fashion. That is, after the $O(|P|)$ 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 $O(|P| + k)$ 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 on Theorem 2.1, and is based on the idea used, for example, in [8, 21]. For completeness, we describe it here.

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 $O(f(n))$ 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 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.

The reporting procedure described above outputs the first k most highly weighted points in $O(f(n) + kg(n))$ time. It can be interrupted at any time.

5 A Space-Efficient Data Structure

We show now how the space of our 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 the same query time. Our approach is to partition the tree into minitrees, which can be represented using narrower grids.

Partitioning the tree. We define $z = \Theta(\sigma D \log n)$. We say that a 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 $u^c = u$. Else, if u belongs to minitree T_v, then $u^c = v^c$. Similarly, if u' is a fat node then $(u')^c = u'$; else 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 2.2, 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.

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) [35] to represent the topology of the whole T in $O(n)$ bits, and a recent compressed representation [5] 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 $O(|P|)$, 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 at most $|P|$ constant-time *parent* operations in T we traverse all the nodes from the locus u of P to the root of T. With bitmap M we can identify, along this path, whether u belongs to a minitree rooted at v (with local preorder $\mathit{preorder}_{T_v}(u) = \mathit{preorder}_T(u) - \mathit{preorder}_T(v)$ and depth $\mathit{depth}_{T_v}(u) = \mathit{depth}_T(u) - \mathit{depth}_T(v)$; 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 $\mathit{preorder}_{T^c}(u) = \mathit{rank}_1(C, \mathit{preorder}_T(u))$, in constant time. Its depth in T^c can be stored in an array indexed by $\mathit{preorder}_{T^c}$ 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 2.2. 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 to 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 2.1 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 2.1 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 to 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 $O(|P|)$ 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 actually takes $O(n \log D)$ bits.

Theorem 5.1. *Let $\mathcal{D}$ be a collection of D documents over an alphabet of size σ 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 $O(|P| + k)$ time.*

6 A Data Structure for Top-k Most Frequent Documents

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 5.1 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 $z' = \sigma D \log m$, where m is the number of nodes in a minitree $\mathcal{T}$. Using the same method as in Theorem 5.1, 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 5.1. 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 5.1 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 5.1 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 $m''(\log m' + \log m'') = o(\log n)$ 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 $w_2(u) \le m'$. 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 $\mathcal{N}' = \{\, \lfloor w/m' \rfloor, \lfloor w/m' \rfloor + 1 \,|\, w \in \mathcal{N}\}$. 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 $\log D$ 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 6.1. *Let $\mathcal{D}$ be a collection of strings over an alphabet of size σ 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 $O(|P| + k)$ time.*

7 Parameterized Top-k Queries

In this section we improve a recent data structure that supports two-dimensional top-k queries [30, Sec. 3.2]. The structure is similar to our wavelet tree W described in the proof of Lemma 3.2. 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 [13] uses $O(p)$ bits if the node of W handles p 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 [30] 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 $[a_v, x-1]$ and $[x+1, b_v]$, prioritized by the maximum weight in those ranges.

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 [10, 31, 9]. The operations on the priority queue can be carried out in $O((\log\log n)^2)$ time [1]. Thus we have the following result.

Lemma 7.1. *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 $\varepsilon > 0$.*

Note this technique automatically admits being used in online mode (i.e., without knowing k in advance). We can easily stop the computation at some k and resume it later.

7.1 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 $p.z \leq \log^{\alpha} n$ 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 7.1 for these queries.

Instead of a binary wavelet tree, we use a multiary one [12], with node degree $\log^{\varepsilon} n$ 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. 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 [31]. All vectors B_v and the extra data structures use $O(n)$ words.

We regard the p points of each node v as lying in a two-dimensional grid of x- and z-coordinates. Instead of one-dimensional RMQ queries on the x-coordinates $[a_v, b_v]$, we issue two-dimensional RMQ queries on $[a_v, b_v] \times [\beta, \gamma]$. The wavelet tree of the basic two-dimensional RMQ data structure [30] handles $n \times m$ grids in $O(n \log m)$ bits of space and answers RMQ queries in time $O(\log^2 m)$. In our case $m < \log^{\alpha} n$ and thus the space is $O(n \log\log n)$ bits and the 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 $O(n)$ words.

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. 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 $O(\log^{\varepsilon} n (\log\log n)^2)$. This is repeated over $O(\log^{1+\varepsilon} n / \log\log n)$ nodes, and then iterated k times. The overall time is $O((k + \log^{1+\varepsilon} n / \log\log n) \log^{\varepsilon} n (\log\log n)^2)$, which is of the form $O((k + \log n) \log^{\varepsilon} n)$ by adjusting ε. The times to handle the priority queue are negligible [1].

Lemma 7.2. *Given a grid of $n \times n \times \log^{\alpha} n$ 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 $\varepsilon > 0$.*

Once again, this results holds verbatim in online mode.

7.2 Proof of Theorem 2.2 Now we are ready to prove Theorem 2.2. We divide the grid into horizontal stripes H_j of height $r = \lceil \log^{1+\varepsilon} n \rceil$. The j-th stripe H_j contains all points $p \in S$ such that $(j-1)r \le p.y < jr$. We store a data structure for limited three-dimensional top-k queries for each H_i, taking y as the limited coordinate. A query $[a,b] \times [0,h] \times [\tau_1,\tau_2]$ is processed as follows. We find the largest j such that $(j-1)r \le h$. For each $i < j$, we find the top point p_i in the range $Q_i = [a,b] \times [\tau_1,\tau_2] \times [(i-1)r, ir-1]$ of H_i; we also find the top point p_j in $Q_j = [a,b] \times [\tau_1,\tau_2] \times [(j-1)r, h]$. If $j > k$, we select k top points among $p_1, \ldots, p_j$ and store them in a priority queue $\mathcal{Q}$. Otherwise all p_i are stored in $\mathcal{Q}$. The following steps are repeated at most k times: we extract and report the maximal weight point p_m from $\mathcal{Q}$, and replace it in $\mathcal{Q}$ by the next highest weighted point p'_m (using online queries on H_m). According to Lemma 7.2, we can initialize $\mathcal{Q}$ in $O(\lceil h/r \rceil \log^{1+\epsilon} n) = O(h + \log^{1+\varepsilon} n)$ time, and operations on $\mathcal{Q}$ can be supported in $O((\log\log n)^2)$ time [1]. We can extract the next most highly weighted point from H_i in $O(\log^{\varepsilon} n)$ time. Hence, the total query time is $O(h + (k + \log n)\log^{\varepsilon} n)$.

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. The solution reduces the problem to ranked retrieval on multidimensional grids, where we also present improved solutions, some tailored to this particular application, some of more general interest.

Without considering the cost to compute weights $w(path(u), d)$ for all pointers `ptr` in the suffix tree, the construction time of our data structure is $O(n \log n)$, whereas that of Hon et al. [21] (which achieves suboptimal query time) is $O(n)$. This raises the challenge of achieving linear construction time for our structure.

Our solutions are not complex to implement and do not make use of impractical data structures. Moreover, our worst-case analysis does not consider the average-case behavior of some parameters, which could significantly simplify or speed up our data structures. For example, the height of our grids was bounded by $O(n)$, but it corresponds to the height of the suffix tree. This is $O(\log n)$ on average for any text generated from a Markov model [36], and indeed small in most practical cases.

A common pitfall to practicality 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, $O(n(\log\sigma + \log D + \log\log n))$ bits. There is a whole trend of reduced-space representations for general document retrieval problems [34, 37, 16, 21, 15, 11, 4]. Most of them make use of the so-called *document array* [27]. This approach has been shown to be very competitive in space and time for top-k problems, even using heuristic solutions [11, 29]. The space of this solution, essentially $n \log D$ bits, is of the same order of our reduced-space variant (usually D is much larger than σ and $\log n$). Although our constants are much higher in theory, this suggests that our data structure is amenable of a practical implementation, as recently achieved with the (more space-consuming) structure of Hon et al. [32].

A farther-reached challenge is to move on towards 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 [39, 3], 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. We are only aware of some very preliminary results for this case [28, 20]. 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 which stores those lists in explicit form [33, 2], therefore extending any such technique to the general model of string documents.

References

[1] A. Andersson, M. Thorup, *Dynamic Ordered Sets with Exponential Search Trees*, Journal of the ACM 54(3), 13 (2007).

[2] V. Anh and A. Moffat, *Pruned Query Evaluation using Pre-computed Impacts*, Proc. 29th SIGIR, 372-379 (2006).

[3] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, 2nd edition, Addison-Wesley (2010).

[4] D. Belazzougui and G. Navarro, *Improved Compressed Indexes for Full-Text Document Retrieval*, Proc. 18th SPIRE, 386–397 (2011).

[5] D. Belazzougui and G. Navarro, *Alphabet-Independent Compressed Text Indexing*, Proc. 19th ESA, 748–759 (2011).

[6] M. Bender and M. Farach-Colton, *The LCA Problem Revisited*, Proc. 4th LATIN, 88-94 (2000).

[7] 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 (1987).

[8] G. S. Brodal, R. Fagerberg, M. Greve, A. Lopez-Ortiz, *Online Sorted Range Reporting*, Proc. 20th ISAAC, 173-182 (2009).

[9] T. M. Chan, K. Larsen, M. Pătraşcu, *Orthogonal Range Searching on the RAM, Revisited*, Proc. SoCG, 1-10 (2011).

[10] B. Chazelle, *A Functional Approach to Data Structures and its Use in Multidimensional Searching*, SIAM Journal on Computing 17, 427-462 (1988).

[11] S. Culpepper, G. Navarro, S. Puglisi, and A. Turpin. *Top-k Ranked Document Search in General Text Databases*, Proc. 18th ESA, 194-205 (part II) (2010).

[12] 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).

[13] J. Fischer, *Optimal Succinctness for Range Minimum Queries*, Proc. 9th LATIN, 158-169 (2010).

[14] H. N. Gabow, J. L. Bentley, R. E. Tarjan, *Scaling and Related Techniques for Geometry Problems*, Proc. 16th STOC, 135-143 (1984).

[15] T. Gagie, G. Navarro, and S. J. Puglisi, *Colored Range Queries and Document Retrieval*, Proc. 17th SPIRE, 67-81 (2010).

[16] T. Gagie, S. J. Puglisi, A. Turpin, *Range Quantile Queries: Another Virtue of Wavelet Trees*, Proc. 16th SPIRE, 1-6 (2009).

[17] R. Grossi, A. Gupta, and J. S. Vitter, *High-order Entropy-Compressed Text Indexes*, Proc. 14th ACM-SIAM SODA, 841-850 (2003).

[18] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

[19] W.-K. Hon, M. Patil, R. Shah, and S.-B. Wu, *Efficient Index for Retrieving Top-k most Frequent Documents*, Journal of Discrete Algorithms 8(4), 402-417 (2010).

[20] W.-K. Hon, R. Shah, S. Thankachan, and J. Vitter, *String Retrieval for Multi-pattern Queries.* Proc. 17th SPIRE, 55-66 (2010).

[21] W.-K. Hon, R. Shah, J. S. Vitter, *Space-Efficient Framework for Top-k String Retrieval Problems*, Proc. 50th IEEE FOCS, 713-722 (2009).

[22] M. Karpinski and Y. Nekrich, *Top-K Color Queries for Document Retrieval*, Proc. 22nd ACM-SIAM SODA, 401-411 (2011).

[23] D. Knuth, *The Art of Computer Programming*, Vol. 3. Reading, Massachusetts: Addison-Wesley (1973).

[24] Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv, *Augmenting Suffix Trees, with Applications*, Proc. 6th ESA, 67-78 (1998).

[25] V. Mäkinen and G. Navarro, *Rank and Select Revisited and Extended*, Theoretical Computer Science, 387(3), 332-347 (2007).

[26] I. Munro, *Tables*, Proc. 16th FSTTCS, 37-42 (1996).

[27] S. Muthukrishnan, *Efficient Algorithms for Document Retrieval Problems*, Proc. 13th ACM-SIAM SODA, 657-666 (2002).

[28] G. Navarro and S. Puglisi, *Dual-Sorted Inverted Lists*, Proc. 17th SPIRE, 310-322 (2010).

[29] G. Navarro, S. J. Puglisi, and D. Valenzuela, *Practical Compressed Document Retrieval*, Proc. 10th SEA, 193-205 (2011).

[30] G. Navarro and L. Russo, *Space-Efficient Data-Analysis Queries on Grids*, Proc. 22nd ISAAC, 321–330 (2011).

[31] Y. Nekrich, *A Linear Space Data Structure for Orthogonal Range Reporting and Emptiness Queries*, International Journal of Computational Geometry and Applications 1-15 (2009).

[32] M. Patil, S. Thankachan, R. Shah, W.-K. Hon, J. Vitter, and S. Chandrasekaran, *Inverted Indexes for Phrases and Strings*, Proc. 34th ACM SIGIR (2011).

[33] M. Persin, J. Zobel, and R. Sacks-Davis, *Filtered Document Retrieval with Frequency-Sorted Indexes*, Journal of the American Society for Information Sience 47(10), 749-764 (1996).

[34] K. Sadakane, *Succinct Data Structures for Flexible Text Retrieval Systems*, Journal of Discrete Algorithms, 5(1), 12-22 (2007).

[35] K. Sadakane, *Compressed Suffix Trees with Full Functionality*, Theory of Computing Systems 41(4), 589-607 (2007).

[36] W. Szpankowski, *Probabilistic Analysis of Generalized Suffix Trees*, Proc. 3rd CPM, 1-14 (1992).

[37] N. Välimäki and V. Mäkinen, *Space-Efficient Algorithms for Document Retrieval*, Proc. 18th CPM, 205-215 (2007).

[38] P. Weiner, *Linear Pattern Matching Algorithms*, Proc. 14th IEEE Symp. on Switching and Automata Theory, 1-11 (1973).

[39] J. Zobel and A. Moffat, *Inverted Files for Text Search Engines*, ACM Computing Surveys 38(2), article 6 (2006).