A

Space-Efficient Frameworks for Top-k String Retrieval

Wing-Kai Hon, National Tsing Hua University
Rahul Shah, Louisiana State University
Sharma V. Thankachan, Louisiana State University
Jeffrey Scott Vitter, The University of Kansas

The inverted index is the backbone of modern web search engines. For each word in a collection of web documents, the index records the list of documents where this word occurs. Given a set of query words, the job of a search engine is to output a ranked list of the most relevant documents containing the query. However, if the query consists of an arbitrary string — which can be a partial word, multiword phrase, or more generally any sequence of characters — then word boundaries are no longer relevant and we need a different approach. In string retrieval settings, we are given a set $\mathcal{D} = \{d_1, d_2, d_3, \ldots, d_D\}$ of D strings with n characters in total taken from an alphabet set $\Sigma = [\sigma]$, and the task of the search engine, for a given query pattern P of length p, is to report the "most relevant" strings in $\mathcal{D}$ containing P. The query may also consist of two or more patterns. The notion of relevance can be captured by a function $score(P, d_r)$, which indicates how relevant document d_r is to the pattern P. Some example score functions are the *frequency* of pattern occurrences, *proximity* between pattern occurrences, or pattern-independent *PageRank* of the document.

The first formal framework to study such kinds of retrieval problems was given by Muthukrishnan [SODA 2002]. He considered two metrics for relevance: frequency and proximity. He took a threshold-based approach on these metrics and gave data structures that use $O(n \log n)$ words of space. We study this problem in a somewhat more natural top-k framework. Here, k is a part of the query, and the top k most relevant (highest-scoring) documents are to be reported in sorted order of score. We present the first linear-space framework (i.e., using $O(n)$ words of space) that is capable of handling arbitrary score functions with near-optimal $O(p + k \log k)$ query time. The query time can be made optimal $O(p + k)$ if sorted order is not necessary. Further, we derive compact space and succinct space indexes (for some specific score functions). This space compression comes at the cost of higher query time. At last, we extend our framework to handle the case of multiple patterns. Apart from providing a robust framework, our results also improve many earlier results in index space or query time or both.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Trees*; E.4 [**Data**]: Coding and Information Theory—*Data Compaction and Compression*; F.2.2 [**Nonnumerical Algorithms and Problems**]: Pattern Matching; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing Methods*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: String Matching, Document Retrieval, Top-k Queries

ACM Reference Format:
Hon, W. K., Shah, R., Thankachan, S. V., and Vitter, J. S. 2012. Space-Efficient Frameworks for Top-k String Retrieval. J. ACM V, N, Article A (January YYYY), 37 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

This work is supported in part by Taiwan NSC Grants 99-2221-E-007-123 and 102-2221-E-007-068 (W. K. Hon), and US NSF Grant CCF-1017623 (R. Shah and J. S. Vitter). This work builds on a preliminary version that appeared in the *Proceedings of the IEEE Foundations of Computer Science* (FOCS), 2009 [Hon et al. 2009] and more recent material in [Hon et al. 2010; Hon et al. 2012; Hon et al. 2013].

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

In string retrieval settings, we are given a collection of D string documents $\mathcal{D} = \{d_1, d_2, d_3, \ldots, d_D\}$ of total length n. Each document is a (possibly long) string whose characters are drawn from an alphabet set $\Sigma = [\sigma]$, and the end of each document is marked with a unique symbol \$ not appearing elsewhere in the document. We can pre-process this collection and build a data structure on it, so as to answer queries. A query is of the form (P, k) that consists of a pattern P of p characters and a numeric parameter k. We are required to output k most relevant documents (with respect to the pattern P) in sorted order of "relevance". The measure of relevance between pattern P and a document d_r is captured by the function $score(P, d_r)$. The score depends on the set of occurrences (given by their locations) of pattern P in document d_r. For example, $score(P, d_r)$ can simply be the term frequency $\mathtt{TF}(P, d_r)$ (i.e., the number of occurrences of P in d_r), or it can be the term proximity $\mathtt{TP}(P, d_r)$ (i.e., the distance between the pair of closest occurrences of P in d_r), or a pattern-independent *importance* score of d_r such as PageRank [Page et al. 1999]. We refer to this problem as the *top-k document retrieval problem*.

Top-k document retrieval is the most fundamental task done by modern-day search engines. To handle the task, inverted indexes are applied and form the backbone. For each word w of the document collection, an inverted index maintains a list of documents in which that word appears, in the descending order of $score(w, \cdot)$. Top-k queries for a single word are easily answered using an inverted index. However, when querying phrases that consist of multiple words, inverted indexes are not as efficient [Patil et al. 2011]. Also, in the cases of biological databases as well as eastern language texts where the usual word boundary demarcation may not exist, the documents are best modeled as strings. In this case, the query pattern can be a contiguous sequence of words, and we are interested in those documents that contain the pattern as a *substring*. The usual inverted index approach might require us to index the list of documents for each possible substring, which can possibly take quadratic space. This approach is neither theoretically interesting nor practically sensible. Hence, pattern matching-based data structures need to be taken into account.

In text pattern matching, the most basic problem is to find all the locations in the text where this pattern matches. Earlier work has focused on developing linear-time algorithms for this problem [Knuth et al. 1977]. In a data structural sense, the text is known in advance and the pattern queries arrive in an online fashion. The suffix tree [McCreight 1976; Weiner 1973] is a popular data structure to handle such queries; it takes linear-space and answers pattern matching queries in optimal $O(p + \mathtt{occ})$ time, where $\mathtt{occ}$ denotes the number of occurrences of the pattern in the text. Most string databases consist of a collection of multiple text documents (or strings) rather than just one single text. In this case, a natural problem is to retrieve all the documents in which the query pattern occurs. This problem is known as the *document listing* problem, which can be seen as a particular case of the top-k document retrieval problem. One challenge is that the number of such qualifying documents (denoted by $\mathtt{ndoc}$) may be much smaller than the actual number of occurrences of the pattern over the entire collection. Thus, a simple suffix-tree-based search might be inefficient since it might involve browsing through a lot more occurrences than the actual number of qualifying documents. This problem was first addressed by Matias et al. [1998], where they gave a linear-space and $O(p \log D + \mathtt{ndoc})$ time solution. Later, Muthukrishnan [2002] improved the query time to optimal $O(p + \mathtt{ndoc})$.

Muthukrishnan [2002] also initiated a more formal study of document retrieval problems with various relevance metrics. The two problems considered by Muthukrishnan were *K-mine* and *K-repeats*. In the K-mine problem, the query asks for all the documents which have at least K occurrences of the pattern P. This basically amounts to thresholding by term frequency. In the K-repeats problem, the query asks for all the documents in which there is at least one pair of occurrences of the pattern P such that these occurrences are at most distance K apart. This relates to another popular relevance measure in information retrieval called term proximity. He gave $O(n \log n)$-word data structures for these problems that can answer the queries in optimal $O(p + \texttt{output})$ time. Here, output represents the number of qualifying documents for the given threshold. These data structures work by augmenting suffix trees with additional information. Based on Muthukrishnan's index for K-mine problem, Hon et al. [2010] designed an $O(n \log n)$-word index for top-k *frequent* document retrieval problem (i.e., the relevance metric is term frequency) with near-optimal $O(p + k + \log n \log \log n)$ query time. The main drawback of the above indexes was the $\Theta(\log n)$ factor of space blow-up when compared with the "linear-space" suffix tree.

In modern times, even suffix trees are considered space-bloated, as its space occupancy can grow to 15 – 50 times the size of the text. In the last decade, with advances in succinct data structures, compressed alternatives of suffix trees have emerged that use an amount of space close to the entropy of the compressed text. The design of succinct/compressed text indexing data structures has been a field of active research with great practical impact [Grossi and Vitter 2005; Ferragina and Manzini 2005]. Sadakane [2007b] showed how to solve the document listing problem using succinct data structures that take space very close to that of the compressed text. He also showed how to compute the TF-IDF scores [Witten et al. 1999] of each document with such data structures. However, one limitation of Sadakane's approach is that it needs to first retrieve all the documents where the pattern (or patterns) occurs, and then find their relevance scores. The more meaningful task from the information retrieval point of view, however, is to get only some of the highly relevant documents. In this sense, it is very costly to retrieve all the documents first. Nevertheless, Sadakane did show some very useful tools and techniques for deriving succinct data structures for these problems.

Apart from fully succinct data structures, the document listing problem has also been considered in the compact space model, where an additional $n \log D$ bits of space is allowed [Välimäki and Mäkinen 2007; Gagie et al. 2009; Gagie et al. 2010]. Typically, fully succinct data structures take space that is less than or comparable to the space taken by the original text collection; compact data structures are shown to take about 3 times the size of the original text. Both succinct and compact data structures are highly preferable over the usual suffix-tree-based implementations, but they are typically slower in query time.

The document listing problem has also been studied for multiple patterns. For the case of two patterns P_1 and P_2 (of lengths p_1 and p_2, respectively), an index proposed by Muthukrishnan's [2002] takes $\widetilde{O}(n^{3/2})$ space (which is prohibitively expensive) and report those ndoc documents containing both P_1 and P_2 in $O(p_1 + p_2 + \sqrt{n} + \texttt{ndoc})$ time.[1] Cohen and Porat [2010] gave a more space-efficient version taking $O(n \log n)$ words of space while answering queries in $O(p_1 + p_2 + \sqrt{(\texttt{ndoc} + 1) \times n \log n \log^2 n})$ time.

In our paper, we introduce various frameworks, by which we provide improved solutions for some of the known document retrieval problems, and also provide efficient solutions for some new problems. In the remaining part of this section, we first list our main contributions, and then briefly survey the work that happened in this line of research after our initial conference paper [Hon et al. 2009].

[1]The notation $\widetilde{O}(\cdot)$ ignores polylogarithmic factors.

1.1. Our Contributions

The following summarizes our contributions (throughout this paper ϵ represents any small positive constant):

(1) We provide a framework for designing linear-space (i.e., using $O(n)$ words) data structures for top-k string retrieval problems, when the query consists of a single pattern P of length p. Our framework works with any arbitrary relevance score function which depends on the set of locations of occurrences of P in the document. Many popular metrics like term frequency, term proximity, and importance are covered by this model. We achieve query time of $O(p + k \log k)$ for retrieving the top k documents in sorted order of their relevance score, and optimal $O(p + k)$ time if sorted order is not necessary.
(2) We reduce the space requirement of our linear-space index to achieve compact space when the score function is term frequency. We achieve an index of size $|\mathsf{CSA}| + 2n \log D + o(n \log D + n \log \sigma)$ bits with query time $O(t_s(p) + p + \log^4 \log n + k(\log \log n + \log k))$. The space is further improved to $|\mathsf{CSA}| + n \log D + o(n \log D + n \log \sigma)$ bits with slightly more query time of $O(t_s(p) + p + \log^6 \log n + k((\log \sigma \log \log n)^{1+\epsilon} + \log^2 \log n + \log k))$.
(3) We provide a framework for designing succinct/compressed data structures for the single-pattern top-k string retrieval problems. Our main idea is based on sparsification, which allows us to achieve better space (at the cost of somewhat worse query time). Our framework is applicable to any score function that can be calculated on-the-fly in compressed space. In the specific case when we score by term frequency, we derive the first succinct data structure that occupies $2|\mathsf{CSA}^*| + o(n) + D \log \frac{n}{D} + O(D)$ bits and answers queries in $O(t_s(p) + k \times t_{sa} \log k \log^\epsilon n)$ time. When we score by importance, the above space can be reduced to $|\mathsf{CSA}| + o(n) + D \log \frac{n}{D} + O(D)$ bits with the same query time. Here, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, and $t_s(p)$ is the time for computing the suffix range of P using CSA (We defer to Section 2.2 and Section 2.3 for the definitions of suffix array, suffix range and compressed suffix array).
(4) We provide a framework to answer top-k queries for two or more patterns. For two patterns P_1 and P_2 of lengths p_1 and p_2 respectively, we derive linear-space (i.e., using $O(n)$ words) indexes with query time $O(p_1 + p_2 + \sqrt{nk \log n} \log \log n)$ for various score functions.

1.2. Postscript

After our initial conference paper [Hon et al. 2009], many results on top-k retrieval have appeared with improvements in index space or query time or both (see Table I for the summary of results). Karpinski and Nekrich [2011] derived an optimal $O(p + k)$-time linear-space index for sorted top-k document retrieval problem when $p = \log^{O(1)} n$. Navarro and Nekrich [2012b] gave the first linear-space index achieving optimal query time; they also showed that it is possible to maintain their index in $O(n(\log \sigma + \log D + \log \log n))$ bits of space. Recently, Shah et al. [2013] proposed an alternative linear-space index that can answer the top-k queries in $O(k)$ time, once the locus of the query pattern is given; they also studied the problem in the external memory model [Vitter 2008; Aggarwal and Vitter 1988], and presented an I/O-optimal index occupying almost linear-space of $O(n \log^* n)$ words.

Let $\mathsf{T} = d_1 d_2 d_3 \cdots d_D$ be the text (of n characters from an alphabet set $\Sigma = [\sigma]$) obtained by concatenating all the documents in $\mathcal{D}$. For succinct indexes (which take space close to the size of T in its compressed form), existing work focussed on the case where the relevance metric is term frequency or static importance score. Most of the succinct indexes used a key idea from an earlier paper by Sadakane [2007b], where he showed how to compute the TF-IDF score of each document, by maintaining a compressed suffix array CSA of T along with a compressed suffix array CSA_r of each document d_r (see Section 2.3 for the definition of CSA).

Table I. Indexes for Top-k Frequent Document Retrieval
(assuming $\sigma \leq D = o(n/\log n)$, with $\log D$ and $\log(D/k)$ simplified to the worst-case bound of $\log n$ in the reporting time)

Source	**Space (in bits)**	**Per-Document Reporting Time**
Hon et al. [2010]	$O(n\log n + n\log^2 D)$	$O(1)$
Ours (Theorem 3.8)	$O(n\log n)$	$O(\log k)$
Navarro and Nekrich [2012b]	$O(n(\log D + \log\sigma))$	$O(1)$
Shah et al. [2013]	$O(n\log n)$	$O(1)$
Hon et al. [2009]	$2\lvert\mathsf{CSA}^*\rvert + o(n)$	$O(t_{sa}\log^{3+\epsilon} n)$
Gagie et al. [2010]	$2\lvert\mathsf{CSA}^*\rvert + o(n)$	$O(t_{sa}\log^{3+\epsilon} n)$
Belazzougui et al. [2013]	$2\lvert\mathsf{CSA}^*\rvert + o(n)$	$O(t_{sa}\log k\log^{1+\epsilon} n)$
Ours (Theorem 5.6)	$2\lvert\mathsf{CSA}^*\rvert + o(n)$	$O(t_{sa}\log k\log^{\epsilon} n)$
Tsur [2013]	$\lvert\mathsf{CSA}\rvert + o(n)$	$O(t_{sa}\log k\log^{1+\epsilon} n)$
Navarro and Thankachan [2013]	$\lvert\mathsf{CSA}\rvert + o(n)$	$O(t_{sa}\log^2 k\log^{\epsilon} n)$
Gagie et al. [2010]	$\lvert\mathsf{CSA}\rvert + n\log D(1+o(1))$	$O(\log^{2+\epsilon} n)$
Belazzougui et al. [2013]	$\lvert\mathsf{CSA}\rvert + n\log D(1+o(1))$	$O(\log k\log^{1+\epsilon} n)$
Gagie et al. [2010]	$\lvert\mathsf{CSA}\rvert + O(\frac{n\log D}{\log\log D})$	$O(t_{sa}\log^{2+\epsilon} n)$
Belazzougui et al. [2013]	$\lvert\mathsf{CSA}\rvert + O(\frac{n\log D}{\log\log D})$	$O(t_{sa}\log k\log^{1+\epsilon} n)$
Belazzougui et al. [2013]	$\lvert\mathsf{CSA}\rvert + O(n\log\log\log D)$	$O(t_{sa}\log k\log^{1+\epsilon} n)$
Ours (Theorem 4.5)	$\lvert\mathsf{CSA}\rvert + 2n\log D(1+o(1))$	$O(\log\log n + \log k)$
Ours (Theorem 4.6)	$\lvert\mathsf{CSA}\rvert + n\log D(1+o(1))$	$O(\log^2\log n + (\log\sigma\log\log n)^{1+\epsilon} + \log k)$

In our initial conference paper [Hon et al. 2009], we proposed the first succinct index for top-k *frequent* document retrieval. The index occupies $2\lvert\mathsf{CSA}^*\rvert + o(n) + D\log\frac{n}{D} + O(D)$ bits of space and answers a query in $O(t_s(p) + k \times t_{sa}\log^{3+\epsilon} n)$ time. While retaining the same space complexity, Gagie et al. [2010] improved the query time to $O(t_s(p) + k \times t_{sa}\log D\log(D/k)\log^{1+\epsilon} n)$, and Belazzougui et al. [2013] further improved it to $O(t_s(p) + k \times t_{sa}\log k\log(D/k)\log^{\epsilon} n)$. Our result of Theorem 5.6 in this paper (initially appeared in [Hon et al. 2013]) achieves an even faster query time of $O(t_s(p) + k \times t_{sa}\log k\log^{\epsilon} n)$. An open problem of designing a space-optimal index is positively answered by Tsur [2013], where he proposed an index of size $\lvert\mathsf{CSA}\rvert + o(n) + O(D) + D\log(n/D)$ bits with $O(t_s(p) + k \times t_{sa}\log k\log^{1+\epsilon} n)$ query time; very recently, Navarro and Thankachan [2013] improved the query time further to $O(t_s(p) + k \times t_{sa}\log^2 k\log^{\epsilon} n)$. Top-$k$ *important* document retrieval (i.e., the score function is document importance) is also a well-studied problem, and the best known succinct index appeared in [Belazzougui et al. 2013]. This index takes $\lvert\mathsf{CSA}\rvert + o(n) + O(D) + D\log(n/D)$ bits of space, and answers a query in $O(t_s(p) + k \times t_{sa}\log k\log^{\epsilon} n)$ time.

The *document array* (refer to Section 2.4 for the definition) is a powerful data structure for solving string retrieval problems, and its space occupancy is $n\lceil\log D\rceil$ bits. This was first introduced in [Välimäki and Mäkinen 2007] for solving the document listing problem. Later, Culpepper et al. [2010] showed how to efficiently handle top-k frequent document retrieval queries using a simple data structure, which is essentially a wavelet tree maintained over the document array. Although their query algorithm is only a heuristic (no worst-case bound), it works well in practice, with space occupancy roughly $1 - 3$ times the text size. From now onwards, an index that allows a space term of roughly $n \log D$ bits, corresponding to the document array, will be called a *compact* index. Gagie et al. [2010] proposed two compact indexes of sizes $|\mathsf{CSA}| + n \log D(1 + o(1))$ bits and $|\mathsf{CSA}| + O(n \log D / \log\log D)$ bits, with query time bounds of $O(t_s(p) + k \times \log D \log(D/k) \log^{\epsilon} n)$ and $O(t_s(p) + k \times t_{sa} \log D \log(D/k) \log^{\epsilon} n)$, respectively. Belazzougui et al. [2013] showed that the $\log D$ factor in the query time of both results by Gagie et al. can be converted to $\log k$ without increasing the index space; they also showed an index of size $|\mathsf{CSA}| + O(n \log\log\log D)$ bits with $O(t_s(p) + k \times t_{sa} \log k \log^{1+\epsilon} n)$ query time. In terms of per-document reporting time, the compact indexes proposed in our paper (results in Theorem 4.5 and Theorem 4.6) achieve $(\log\log n)^{O(1)}$ time as opposed to the $O(\log^{O(1)} n)$ time of the other compact indexes mentioned above. Navarro and Nekrich [2012b] gave an index of size $O(n(\log \sigma + \log D))$ bits index with optimal $O(p + k)$ time; however, the hidden constants within the big-O notations are not small in practice [Konow and Navarro 2013]. It has been shown that, compact space indexes provide the best practical performance [Konow and Navarro 2013; Culpepper et al. 2010] compared to linear space indexes [Patil et al. 2011] (which are less efficient in terms of space occupancy) and the succinct space indexes [Culpepper et al. 2012; Navarro et al. 2011] (which are less efficient in terms of query processing time). See also [Hsu and Ottaviano 2013] for a related result on top-k completion.

Fischer et al. [2012] introduced a new variation of two-pattern queries (known as forbidden pattern queries), which is defined as follows: Given input patterns P_1 and P_2, report those $\mathtt{ndoc}$ documents containing P_1, but not P_2. The authors gave an $O(n^{3/2})$-bit data structure with query time $O(p_1 + p_2 + \sqrt{n} + \mathtt{ndoc})$. Later, Hon et al. [2012] improved the index space to $O(n)$ bits, however the query time is increased to $O(p_1 + p_2 + \sqrt{(\mathtt{ndoc} + 1) \times n \log n} \log^2 n)$.

Although most of the relevant results on the central problem is summarized in this section, there are still many related problems which we have excluded. See the recent surveys [Navarro 2013; Hon et al. 2013] for further reading.

1.3. Organization of the Paper

Section 2 gives the preliminaries. Next, we describe our linear-space, compact-space, and succinct-space solutions for the top-k document retrieval problem in Section 3, Section 4, and Section 5, respectively. Section 6 describes the data structure for multi-pattern queries. Finally, we conclude in Section 7 with some open problems.

2. PRELIMINARIES

2.1. Generalized Suffix Tree (GST)

Let $\mathsf{T} = d_1 d_2 d_3 \cdots d_D$ be the text (of n characters from an alphabet set $\Sigma = [\sigma]$) obtained by concatenating all the documents in $\mathcal{D}$. The last character of each document is $\$$, a special symbol that does not appear anywhere else in T. Each substring $\mathsf{T}[i..n]$, with $i \in [1, n]$, is called a *suffix* of T. The *generalized suffix tree* (GST) of $\mathcal{D}$ is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in GST is labeled by a string, and $path(x)$ of a node x is the concatenation of edge labels along the path from the $root$ of GST to node x. Let ℓ_i for $i \in [1, n]$ represent the ith leftmost leaf in GST. Then $path(\ell_i)$ represents the ith lexicographically smallest suffix of T. Corresponding to each node, a perfect hash function [Fredman et al. 1984] is maintained such that, given any node u and any character $c \in \Sigma$, we can compute the child node v of u (if it exists) where the first character on the edge connecting u and v is c. A node x is called the *locus* of a pattern P, if it is the highest node $path(x)$ prefixed by P. The total space consumption of GST is $O(n)$ words and the time for computing the locus node of P is $O(p)$. When $\mathcal{D}$ contains only one document d_r, the corresponding GST is commonly known as the *suffix tree* of d_r [Weiner 1973].

2.2. Suffix Array (SA)

The *suffix array* $\mathsf{SA}[1..n]$ is an array of length n, where $\mathsf{SA}[i]$ is the starting position (in T) of the ith lexicographically smallest suffix of T [Manber and Myers 1993]. In essence, the suffix array contains the leaf information of GST but without the tree structure. An important property of SA is that the starting positions of all the suffixes with the same prefix are always stored in a contiguous region of SA. Based on this property, we define the *suffix range* of P in SA to be the maximal range $[sp, ep]$ such that for all $i \in [sp, ep]$, $\mathsf{SA}[i]$ is the starting point of a suffix of T prefixed by P. Therefore, ℓ_{sp} and ℓ_{ep} represents the first and last leaves in the subtree of the locus node of P in GST.

2.3. Compressed Suffix Arrays (CSA)

A compressed representation of suffix array is called a *compressed suffix array* (CSA) [Grossi and Vitter 2005; Ferragina and Manzini 2005; Grossi et al. 2003]. We denote the size (in bits) of a CSA by $|\mathsf{CSA}|$, the time for computing $\mathsf{SA}[\cdot]$ and $\mathsf{SA}^{-1}[\cdot]$ values by t_{sa}, and the time for finding the suffix range of a pattern of length p by $t_s(p)$. There are various versions of CSA in the literature that provide different performance tradeoffs (see [Navarro and Mäkinen 2007] for an excellent survey). For example, the space-optimal CSA by Ferragina et al. [2007] takes $nH_h + o(n \log \sigma)$ bits space, where $H_h \leq \log \sigma$ denotes the empirical hth-order entropy of T.[2] The timings t_{sa} and $t_s(p)$ are $O(\log^{1+\epsilon} n \log \sigma)$ and $O(p(1 + \log \sigma / \log \log n))$, respectively. Recently, Belazzougui and Navarro [2011] proposed another CSA of space $nH_h + O(n) + o(n \log \sigma)$ bits with $t_s(p) = O(p)$ and $t_{sa} = O(\log n)$.

2.4. Document Array (E)

The document array $\mathsf{E}[1..n]$ is defined as $\mathsf{E}[j] = r$ if the suffix $\mathsf{T}[\mathsf{SA}[j]..n]$ belongs to document d_r. Moreover, the corresponding leaf node ℓ_j is said to be *marked* with document d_r. By maintaining E using the structure described in [Golynski et al. 2006], we have the following result.

LEMMA 2.1. *The document array* E *can be stored in* $n \log D + o(n \log D)$ *bits and support* $rank_{\mathsf{E}}$, $select_{\mathsf{E}}$ *and* $access_{\mathsf{E}}$ *operations in* $O(\log \log D)$ *time, where*

— $rank_{\mathsf{E}}(r, i)$ *returns the number of occurrences of* r *in* $\mathsf{E}[1..i]$*;*
— $select_{\mathsf{E}}(r, j)$ *returns the location of* j*th leftmost occurrence of* r *in* E*; and*
— $access_{\mathsf{E}}(i)$ *returns* $E[i]$*.*

Define a bit-vector $\mathsf{B_E}[1..n]$ such that $\mathsf{B_E}[i] = 1$ if and only if $\mathsf{T}[i] = \$$. Then, the suffix $\mathsf{T}[i..n]$ belongs to document d_r if $r = 1 + rank_{\mathsf{B_E}}(i)$, where $rank_{\mathsf{B_E}}(i)$ represents the number of 1s in $\mathsf{B_E}[1..i]$. The following is another useful result.

LEMMA 2.2. *Using* CSA *and an additional structure of size* $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ *bits, the document array* E *can be simulated to support* $rank_{\mathsf{E}}$ *operation in* $O(t_{sa} \log \log n)$ *time, and* $select_{\mathsf{E}}$ *and* $access_{\mathsf{E}}$ *operations in* $O(t_{sa})$ *time.*

PROOF. The document array E can be simulated using the following structures: **(i)** compressed suffix array CSA of T (of size $|\mathsf{CSA}|$ bits), where $\mathsf{SA}[\cdot]$ and $\mathsf{SA}^{-1}[\cdot]$ represent the suffix array and inverse suffix array values in CSA; **(ii)** compressed suf-

[2]The space bound holds for all $h < \alpha \log n / \log \sigma$, where α is any fixed constant with $0 < \alpha < 1$.

fix array CSA_r of document d_r (of size $|\mathsf{CSA}_r|$ bits) corresponding to every $d_r \in \mathcal{D}$, where $\mathsf{SA}_r[\cdot]$ and $\mathsf{SA}_r^{-1}[\cdot]$ represent the suffix array and inverse suffix array values in CSA_r; and **(iii)** the bit-vector $\mathsf{B_E}$ maintained in $D \log \frac{n}{D} + O(D) + o(n)$ bits with constant-time rank/select supported [Raman et al. 2007]. Hence the total space is bounded by $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits in addition to the $|\mathsf{CSA}|$ bits of CSA, where $|\mathsf{CSA}^*| = \max\{|\mathsf{CSA}|, \sum_{r=1}^{D} |\mathsf{CSA}_r|\}$ [3].

The function $access_{\mathsf{E}}(i) = 1 + rank_{\mathsf{B_E}}(\mathsf{SA}[i])$ can be computed in $O(t_{sa})$ time. For computing $select_{\mathsf{E}}(r, j)$, we first compute the jth smallest suffix in CSA_r and obtain the position pos of this suffix within document d_r, from which we can easily obtain the position pos' of this suffix within T as $select_{\mathsf{B_E}}(r-1) + pos$, where $select_{\mathsf{B_E}}(x)$ is the position of the xth 1 in $\mathsf{B_E}$. After that, we compute $\mathsf{SA}^{-1}[pos']$ in CSA as the desired answer for $select_{\mathsf{E}}(r, j)$. This takes $O(t_{sa})$ time. The function $rank_{\mathsf{E}}(r, i) = j$ can be obtained in $O(t_{sa} \log n)$ time using a binary search on j such that $select_{\mathsf{E}}(r, j) \le i < select_{\mathsf{E}}(r, j+1)$. Belazzougui et al. [2013] showed that the time for computing $rank_{\mathsf{E}}(r, i)$ can be improved to $O(t_{sa} \log\log n)$ as follows: At every $(\log^2 n)$th leaf of each CSA_r, we explicitly maintain its corresponding position in CSA and a predecessor search structure over it [Willard 1983]; the size of this additional structure is $o(n)$ bits. Now, when we answer the query, we can first search this predecessor structure for an approximate answer, and the exact answer can be obtained by a binary search on a smaller range of only $\log^2 n$ leaves. □

By choosing the CSA by Grossi and Vitter [2005] of size $O(n \log \sigma \log\log n)$ bits with $t_{sa} = O(\log\log_\sigma n)$, the above lemma can be restated as follows.

COROLLARY 2.3. *The document array* E *can be encoded in* $O(n \log \sigma \log\log n)$ *bits and support* $rank_{\mathsf{E}}$ *operation in* $O(\log^2 \log n)$ *time, and* $select_{\mathsf{E}}$ *and* $access_{\mathsf{E}}$ *operations in* $O(\log\log n)$ *time.*

LEMMA 2.4. *Let* E *be the document array corresponding to a document collection* $\mathcal{D}$*. Then, for any document* $d_r \in \mathcal{D}$*,* $\mathtt{TF}(P, d_r) = rank_{\mathsf{E}}(r, ep) - rank_{\mathsf{E}}(r, sp-1)$*, where* $[sp, ep]$ *represents the suffix range of* P*.*

2.5. Succinct Representation of Ordinal Trees

Any n-node ordered rooted tree can be represented in $2n + o(n)$ bits, such that if each node is labeled by its preorder rank in the tree, each of the following operations can be supported in constant time [Sadakane and Navarro 2010]: $parent(u)$, which returns the parent of node u; $child(u, q)$, which returns the qth child of node u; $child\_rank(u)$, which returns the number of siblings to the left of node u; $lca(u, v)$, which returns the lowest common ancestor of two nodes u and v; and $lmost\_leaf(u)/rmost\_leaf(u)$, which returns the leftmost/rightmost leaf of node u.

2.6. Score Functions

Given a pattern P and a document d_r, let S denote the set of all positions in d_r where P matches. We study a class of score functions $score(P, d_r)$ that depend only on the set S. Popular examples in the class include: (1) term frequency $\mathtt{TF}(P, d_r)$, which is the cardinality of S; (2) term proximity $\mathtt{TP}(P, d_r)$, which is the minimum distance between any two positions in S; (3) $docrank(P, d_r)$, which is simply a static "importance" value associated with document d_r.

[3] Notice that in the case of some specific versions of CSA, for example the version by Ferragina et al. [2007] occupying $\log \sigma + o(\log \sigma)$ bits per symbol, $|\mathsf{CSA}^*|$ can be bounded by $|\mathsf{CSA}| + O(D \log n)$.

The functions $\texttt{TF}(P, d_r)$ and $\texttt{TP}(P, d_r)$ are directly associated with K-mine and K-repeats problems, respectively. The importance metric captured by $docrank(P, d_r)$ can be realized in practice by the PageRank [Page et al. 1999] of the document d_r, which is static and invariant of P.

In our paper, we focus on obtaining the top k highest-scoring documents given the pattern P. In the design of our succinct/compressed solutions, some of the score calculation will be done on the fly. We call a score function *succinctly calculable* if there exists a data structure on document d_r of $O(|d_r| \log \sigma)$ bits that can calculate $score(P, d_r)$ in $O(\text{ poly }(p, \log |d_r|))$ time; here, $|d_r|$ denotes the number of characters in d_r. Note that $\texttt{TF}(P, d_r)$ and $docrank(P, d_r)$ are succinctly calculable (see Lemma 2.2 and Lemma 2.4), but it is yet unknown if $\texttt{TP}(P, d_r)$ is succinctly calculable.

2.7. Top-k using RMQs and Wavelet Trees

Let A be an array of length n. A range maximum query (RMQ) on A asks for the position of the maximum between two specified array indices i and j. That is, the RMQ should return an index k such that $i \le k \le j$ and $\mathsf{A}[k] \ge \mathsf{A}[x]$ for all $i \le x \le j$. Although solving RMQs is as old as Chazelle's original paper on range searching [Chazelle 1988], many simplifications [Bender and Farach-Colton 2000] and improvements have been made, culminating in the index of size $2n + o(n)$ bits by Fischer and Heun [2011]. Even our results shall extensively use RMQ as a tool to obtain the top k items in a given set of ranges.

LEMMA 2.5. *Let $\mathsf{A}[1..n]$ be an array of n numbers. We can preprocess A in linear time and associate A with an RMQ data structure of size $2n + o(n)$ bits, such that given a set of z non-overlapping ranges $[L_1, R_1], [L_2, R_2], \ldots, [L_z, R_z]$, we can find (i) all those* $\texttt{output}$ *numbers in $\mathsf{A}[L_1..R_1] \cup \mathsf{A}[L_2..R_2] \cup \cdots \cup \mathsf{A}[L_z..R_z]$ which are greater (or less) than a given threshold value in $O(z + \texttt{output})$ time, or (ii) the largest (or smallest) k numbers in $\mathsf{A}[L_1..R_1] \cup \mathsf{A}[L_2..R_2] \cup \cdots \cup \mathsf{A}[L_z..R_z]$ in unsorted order in $O(z + k)$ time.*

PROOF. We use the following result of Frederickson [1993]: the kth largest number from a set of numbers maintained in a binary max-heap Δ can be retrieved in $O(k)$ time by visiting $O(k)$ nodes in Δ. In order to solve our problem, we may consider a conceptual binary max-heap Δ as follows: Let Δ' denote the balanced binary subtree with z leaves that is located at the top part of Δ (with the same root). Each of the $z - 1$ internal nodes in Δ' holds the value ∞. The ith leaf node ℓ_i in Δ' (for $i = 1, 2, \ldots, z$) holds the value $\mathsf{A}[M_i]$, which is the maximum element in the interval $\mathsf{A}[L_i..R_i]$. The values held by the nodes below ℓ_i will be defined recursively as follows: For a node ℓ storing the maximum element $\mathsf{A}[M]$ from the range $\mathsf{A}[L..R]$, its left child stores the maximum element in $\mathsf{A}[L..(M - 1)]$ and its right child stores the maximum element in $\mathsf{A}[(M + 1)..R]$. Note that this is a conceptual heap which is built on the fly, where the value associated with a node is computed in constant time based on the RMQ structures only when needed.

For (i), we simply perform a preorder traversal of Δ, such that if the value ($\neq \infty$) associated with a node satisfies the threshold condition, we then report it and move to the next node; otherwise, we discard it and do not check the nodes in its subtree. The query time can be bounded by $O(z + \texttt{output})$. For (ii), we first find the $(z - 1 + k)$th largest element X in this heap by visiting $O(z + k)$ nodes (with $O(z + k)$ RMQs) using Frederickson's algorithm [1993]. Then, we obtain all those numbers in Δ that are $\ge X$ in $O(z + k)$ time by a preorder traversal of Δ, such that if the value associated with a node is $< X$, we do not check the nodes in its subtree. However, if the number of values $\ge X$ is $\omega(z + k)$, we may end up visiting $\omega(z + k)$ nodes, resulting in $\omega(z + k)$ query time. To avoid this pitfall, we do the following: First, we report all those n_X values which are strictly greater than X (note that $n_X < z + k$); then, we run the algorithm a second time to report up to $z - 1 + k - n_X$ values equal to X. □

While the above lemma dealt with a query range with three constraints (two from range boundaries and one from top-k), the next lemma shows how to extend this to one more dimension so as to obtain top-k among 4-sided rectangular ranges.[4] Instead of RMQs, here we shall use wavelet trees and use RMQs in each node of the wavelet tree.

LEMMA 2.6. *Let* $\mathsf{A}[1..n]$ *be an array of* n *numbers taken from an alphabet set* $\Pi = [\pi]$ *where each number* $\mathsf{A}[i]$ *is associated with a score* (*which may be stored separately and can be computed in* t_{score} *time*)*. Then, the array* A *can be maintained in* $O(n \log \pi)$ *bits, such that given two ranges* $[x', x'']$*,* $[y', y'']$*, and a parameter* k*, we can search among those entries* $\mathsf{A}[i]$ *with* $x' \leq i \leq x''$ *and* $y' \leq \mathsf{A}[i] \leq y''$*, and report the* k *highest-scoring entries in unsorted order in* $O((\log \pi + k)(\log \pi + t_{score}))$ *time.*

PROOF. To answer the above query, we maintain A in the form of a *wavelet tree* [Grossi et al. 2003], which is an ordered balanced binary tree of n leaves. Each leaf is labeled with a symbol in Π, and the leaves are sorted alphabetically from left to right. Each internal node w represents an alphabet set Π_w and is associated with a bit-vector B_w. In particular, the alphabet set of the root is Π, and the alphabet set of a leaf is the singleton set containing its corresponding symbol. Each node partitions its alphabet set among the two children (almost) equally, such that all symbols represented by the left child are lexicographically (or numerically) smaller than those represented by the right child.

For a node w, let A_w be a subsequence of A by retaining only those symbols that are in Π_w. Then B_w is a bit-vector of length $|\mathsf{A}_w|$, such that $\mathsf{B}_w[i] = 0$ if $\mathsf{A}_w[i]$ is a symbol represented by the left child of w, else $\mathsf{B}_w[i] = 1$. Indeed, the subtree from w itself forms a wavelet tree of A_w. To reduce the space requirement, the array A is not stored explicitly in the wavelet tree. Instead, we only store the bit-vectors B_w, each of which is augmented with Raman et al.'s scheme [2007] to support constant-time rank/select operations. The total size of the bit-vectors and the augmented structures in a particular level of the wavelet tree is $n(1 + o(1))$ bits. We maintain an additional *range maximum query* (RMQ) [Fischer and Heun 2007; Sadakane 2007a] structure over the scores of all elements of the sequence A_w (in $O(|\mathsf{A}_w|)$ bits). As there are $\log \pi$ levels in the wavelet tree, the total space is $O(n \log \pi)$ bits. Note that the value of any $\mathsf{A}_w[i]$ for any given w and i can be computed in $O(\log \pi)$ time by traversing $\log \pi$ levels in the wavelet tree. Similarly, any range $[x', x'']$ can be translated to w as $[x'_w, x''_w]$ in $O(\log \pi)$ time, where $\mathsf{A}[x'_w..x''_w]$ is a subsequence of $A[x'..x'']$ with only those elements in Π_w.

The desired k highest-scoring entries can be obtained as follows: First the given range $[y', y'']$ can be split into at most $2 \log \pi$ disjoint subranges, such that each subrange is represented by Π_w associated with some internal node w. All the numbers in the subsequence A_w associated with such an internal node w will satisfy the condition $y' \leq \mathsf{A}_w[i] \leq y''$. And for all such (at most $2 \log \pi$) A_ws, the range $[x', x'']$ can be translated into the corresponding range $[x'_w, x''_w]$ in $O(\log \pi)$ time [Gagie et al. 2012]. Then, we can apply Lemma 2.5 (where $z \leq 2 \log \pi$) to obtain the desired entries. However, retrieving a node value in the conceptual max-heap (in the proof of Lemma 2.5) requires us to compute the score of $\mathsf{A}_w[i]$ for some w and i on the fly, which shall be done by first finding the entry $\mathsf{A}[i']$ that corresponds to $\mathsf{A}_w[i]$, and then retrieving the

[4]See Lemma 7.1 in [Navarro and Nekrich 2012b] for a better solution for this problem.

score of $\mathsf{A}[i']$. This takes $O(\log \pi + t_{score})$ time, so that the total time will be bounded by $O(\log \pi + (2\log \pi + k)(\log \pi + t_{score})) = O((\log \pi + k)(\log \pi + t_{score}))$. □

2.8. Differentially Encoding a Sorted Array

Let $\mathsf{A}[1..m]$ be an array of integers such that $1 \le \mathsf{A}[i] \le \mathsf{A}[i+1] \le n$. The array A can be encoded using a bit vector $\mathsf{B} = 10^{c_1}10^{c_2}10^{c_3}\cdots 10^{c_n}$, where c_i denotes the number of entries $\mathsf{A}[\cdot] = i$. The length of B is $m+n$, and hence B can be maintained in $(m+n)(1+o(1))$ bits (along with constant-time rank/select structures [Munro et al. 2001; Clark 1996]). Then, for any given $j \in [1, m]$, we can compute $A[j]$ in constant time by first finding the location of the jth 0 in B, and then counting the number of 1s up to that position.

3. LINEAR SPACE STRUCTURES

In this section, we describe our linear-space data structures with near optimal query times. Although we describe our result in terms of the frequency metric (i.e., $score(\cdot,\cdot) = \mathtt{TF}(\cdot,\cdot)$), it works directly for any other score function. First, we build a generalized suffix tree (GST) of $\mathcal{D}$ and augment with the following structures described below. The number of leaves in the subtree of a node v in GST that are marked with document d_r is represented by $\mathtt{freq}(v, r)$. Note that $\mathtt{freq}(v, r) = \mathtt{TF}(path(v), d_r)$.

3.1. N-structure

At any node v of GST, we store an N-structure N_v, which is an array of 5-tuples $\langle$document id r, score s, pointer t, first depth δ_f, last depth $\delta_l\rangle$. First, N_v for any leaf node ℓ_i (recall that ℓ_i represents the ith leftmost leaf node in GST) will contain exactly one entry with document id $\mathsf{E}[i]$ and score 1. For an internal node v, an entry with document id r occurs in N_v if and only if at least two children of v contain leaves marked with document d_r in their subtrees. In case the entry of document d_r is present in N_v then its corresponding score value s denotes the number of leaves in the subtree of v marked with document d_r (i.e., $\mathtt{freq}(v, r)$). The pointer t stores two attributes: origin and target. The origin is set to node v, while the target is set to the lowest ancestor of v that also has an entry of document d_r. Note that such an ancestor always exists, unless v is the root (in this case, the target of t is a dummy node which is regarded as the parent of the root). For δ_f and δ_l, we shall give their definitions and describe their usage later. See Figure 1 for an illustration of some N-structure entries, each showing the first three fields of the 5-tuple.

OBSERVATION 1. *Let ℓ_i and ℓ_j be two leaves belonging to the same document d_r. If v is the lowest common ancestor $lca(\ell_i, \ell_j)$, then N_v contains an entry for document d_r.*

PROOF. Leaf ℓ_i and leaf ℓ_j must be in the subtree of different children of v (otherwise, v cannot be their lowest common ancestor). Thus, at least two children of v contain leaves marked with document d_r, so that N_v contains an entry for d_r. □

OBSERVATION 2. *If for two nodes u and w, both N_u and N_w contain an entry for document d_r, then the node $z = lca(u, w)$ also has an entry for document d_r in N_z.*

PROOF. Nodes u and w must be in the subtree of different children of z (otherwise, z cannot be their lowest common ancestor). Since N_u and N_w both contain an entry for document d_r, the subtree of u and the subtree of w must each contain some leaf marked by d_r. Consequently, at least two children of z contain leaves marked with d_r, so that N_z contains an entry for d_r. □

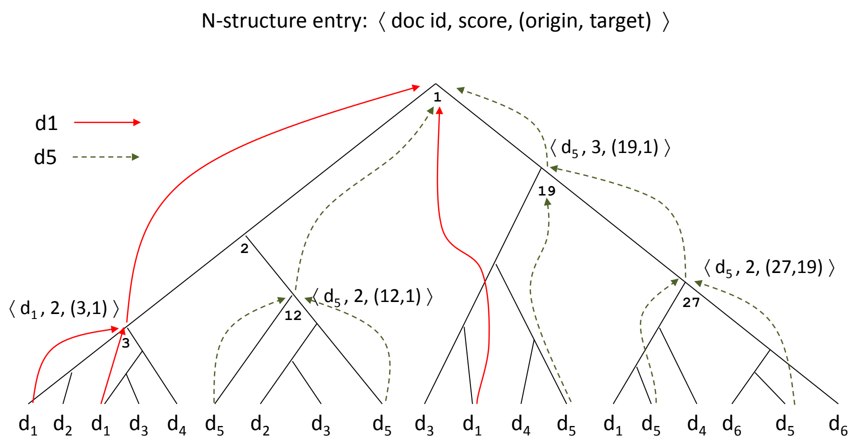

Fig. 1. Example of N-structure entries (without δ fields), with *score* assumed to be TF for illustration purpose

LEMMA 3.1. *For any document d_r which occurs at some leaf in the subtree of a node v, there is exactly one pointer t such that (i) t corresponds to document d_r, (ii) t originates at some node in the subtree of v (including v), and (iii) t targets to some proper ancestor of v.*

PROOF. It is easy to check that at least one pointer t will simultaneously satisfy the three properties. To show that t is unique, suppose on the contrary that two nodes u and w in the subtree of v both contain an entry of document d_r and with the corresponding pointers targeting to some nodes outside the subtree of v. By Observation 2, $z = lca(u, w)$ also has an entry for d_r. Consequently, the pointers originated from u and w must target to some nodes in the subtree of z. On the other hand, since both u and w are in subtree of v, z must be in the subtree of v. The above statements immediately imply that the pointers originated from u and w are targeting to some nodes *in the subtree of* v. Thus, contradiction occurs and the lemma follows. □

LEMMA 3.2. *The total number of internal nodes that have an entry for document d_r is at most $|d_r| - 1$, where $|d_r|$ denotes the number of characters in document d_r.*[5]

PROOF. By construction, each internal node with an entry for d_r has at least two branches, where the subtree of each branch contains some leaf marked by d_r. Indeed, these internal nodes, together with all the leaves marked by d_r, form an induced subtree of the original GST (and is equivalent to the suffix tree for document d_r); moreover, there is no degree-1 internal node. Thus, it follows that the number of internal nodes is bounded by $|d_r| - 1$, since the number of leaves is $|d_r|$. □

3.2. I-structure

Based on the pointer field in the N-structure, we are now ready to describe another structure I_v that is stored at every internal node v. For each pointer t in some N-structure whose target is v, I_v contains a corresponding entry that stores information about the origin of t. Specifically, let $\langle r, s, t, \cdot, \cdot \rangle$ be an entry in an N-structure N_w associ-

[5]Here, we exclude the dummy node.

ated with a node w. If the target of t is v, then I_v stores a triplet $\langle$document id r, score s, origin $w\rangle$.

The entries in the I-structure I_v are sorted, in ascending order, by the preorder ranks of the origins. We store I_v by three separate arrays Doc_v, Sco_v, and Ori_v such that the jth entry of I_v is denoted by $\mathsf{I}_v[j]$, and has the value $\langle \mathsf{Doc}_v[j], \mathsf{Sco}_v[j], \mathsf{Ori}_v[j]\rangle$. Note that some entries in I_v may have the same origin value (say w), if N_w contains more than one entriy targeting v; in this case these entries are further ordered by the document ids within I_v. Also, an I_v may have entries with the same document id (when these entries have different origins). Finally, we store a Range Maximum Query (RMQ) structure on the array Sco_v [Fischer and Heun 2011].

LEMMA 3.3. *The total number of entries $\sum_v |\mathsf{I}_v|$ in all I-structures is at most $2n$.*

PROOF. The total number of entries in I-structures is the same as the total number of pointers. This in turn is the total number of entries in N-structures. By Lemma 3.2, the total number of such entries inside all internal nodes is at most $\sum_{r=1}^{D}(|d_r|-1) \leq n$. On the other hand, the number of such entries inside all leaves is exactly n, so that the total number is at most $2n$. $\square$

3.3. Answering Queries

To answer a query, we match the pattern P in GST in $O(p)$ time and reach at locus node v_P. By the property of GST, all the leaves in the subtree of v_P correspond to the occurrences of P. Now, by Lemma 3.1, for each document d_r that appears in the subtree of v_P, there will be a unique pointer, originating from some node in the subtree of v_P, that targets to some ancestor node of v_P. Note that the *score* value s associated with that pointer is exactly the same as $\mathtt{TF}(P, d_r)$. Thus, the top k documents can be reported by first identifying such pointers, selecting those k highest-scoring ones among them, and then reporting the corresponding document ids.

By the definition of the I-structure, we know that each of these pointers must target to one of the ancestors of v_P. For the locus v_P we have just reached, let v'_P be the rightmost leaf in the subtree of v_P (i.e., v'_P is the highest preorder rank of any node in the subtree of v_P). Note that all the nodes in subtree of v_P have contiguous preorder ranks. Thus, nodes in the subtree of v_P can be represented by the range $[v_P, v'_P]$.

Now, for each ancestor u of v_P, the entries in the I-structure I_u are sorted according to the preorder ranks of the origins. The contiguous range in the origin array Ori_u, with values from $[v_P, v'_P]$, will correspond to pointers originating from the nodes in the subtree of v_P (that point to u). Suppose such a range can be found in the array I_u for each ancestor u. That is, we can find L_u and R_u such that $\mathsf{Ori}_u[L_u]$ and $\mathsf{Ori}_u[R_u]$, respectively, are the first and last entries in Ori_u that are at least v_P and at most v'_P. Then we can examine the score array $\mathsf{Sco}_u[L_u..R_u]$ for each I_u and apply Lemma 2.5 to return those k documents with the highest score (See Figure 2.).

The range $[L_u, R_u]$ for each I_u can be found in $O(\log\log n)$ time if we have maintained a predecessor search structure [Willard 1983] over the array Ori_u for each u. The number of ancestors of v_P is at most $depth(v_P)$, where $depth(v_P)$ denotes the number of nodes in GST from root to v_P. Since $depth(v_P) \leq p$, this range translation takes at most $O(p\log\log n)$ time overall. The subsequent step by using Lemma 2.5 then takes $O(p+k)$ time. So in total, the top k frequent documents can be returned in $O(p\log\log n+k)$ time. The outputs can be obtained by score by spending another $O(k\log k)$ time.

3.4. Improvement: Reducing $O(p\log\log n)$ to $O(p)$

To achieve optimal query time, the main bottleneck comes from querying the predecessor structure for range translation, which costs us $O(p\log\log n)$ time. We shall see

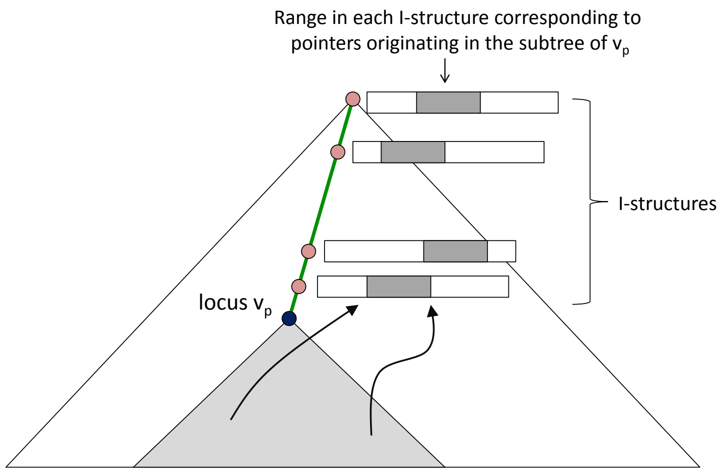

Fig. 2. Answering queries with I-structures

how we can convert the $O(p \log \log n)$ term to $O(p)$. Notice that the $\log \log n$ factor comes from the need of translating the range $[v_P, v'_P]$ in the I-structure of each of the ancestor of v. Next, we shall show how we can use the two fields δ_f and δ_l to directly map the range without having to resort to the predecessor query.

Intuitively, to speed up the range translation, we check for each N-structure entry e whether e corresponds to a left boundary (in the I-structure of its target) in some pattern query. If so, we store e with the locus nodes of all those corresponding patterns. Given the locus node v_P of an online pattern query, we can find all those entries e that are stored with the locus node v_P, and obtain the desired left boundaries in each of the corresponding I-structures.

Using Figure 3 as illustration, observe that if an N-structure entry e is to be stored, the corresponding locus nodes must form a path from the origin x of e to some ancestor w of x. Indeed, such an ancestor w must be the highest one, such that among all entries originating from the subtree of w with the same target y as e, e is the first entry in preorder rank. (In other words, among all entries originating from the subtree of $w' = parent(w)$, there will be an entry e' smaller than e in preorder rank with the same target y as e.) This motivates us to define the δ_f value for an entry in the N-structure N_x of any node x as follows.

Definition 3.4. Consider an entry e in N_x whose target is y. Let w be the highest node on the path from x to y, such that among all entries whose origins are from the subtree of w, e is the first one whose target is y. (In other words, the corresponding I-structure entry of e is the leftmost one in I_y, among those with origins from the subtree of w.) Then, δ_f of the entry e is defined to be the value $depth(w)$. If no such node w exists, then δ_f of e is defined to be ∞. We also define δ_l, symmetrically, with respect to the last entry targeting y.

The δ_f value of an entry e can be determined in another way, as shown in the following lemma. This lemma will be useful in the construction algorithm in Section 3.5, where we need to compute the δ_f value for each N-structure entry.

LEMMA 3.5. *Let e by an* N*-structure entry in N_x whose target is y. Let $\mathsf{I}_y[a]$ be the corresponding I-structure entry of e in I_y. Suppose that the origin of the entry $\mathsf{I}_y[a-1]$*

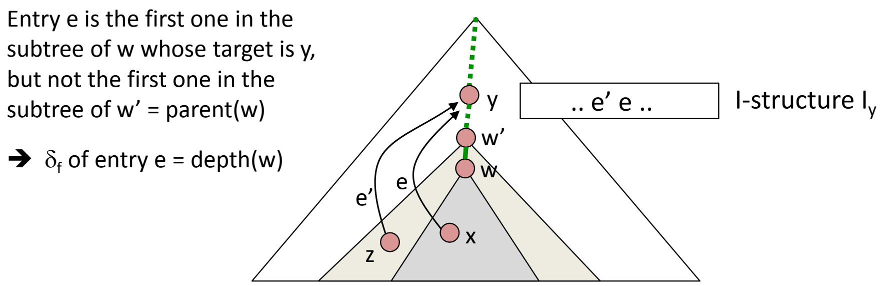

Fig. 3. δ_f field

is z, and the corresponding N*-structure entry is e'. Then, δ_f of e is $1 + depth(lca(z, x))$ if $z \neq x$; else, it is ∞.*

PROOF. If $z = x$, the entry e cannot be the first one originating from the subtree of any ancestor of x with target y (since e' will always appear before e), so that δ_f is ∞.

If $z \neq x$, let w' denote the lca node $lca(z, x)$. Since I_y entries are sorted by the preorder ranks of the origins, $z < x$ in the preorder rank. Let w be the child of w' whose subtree contains x. Then, $\delta_f \leq depth(w)$ since e is the first entry in the subtree of w with target y. However, $\delta_f > depth(w')$ since e cannot be the first one originating from the subtree of w' with target y (since e' will always appear before e). Because w' is the parent of w, $depth(w') = 1 + depth(w)$, so that δ_f must be equal to $1 + depth(w') = 1 + depth(lca(z, x))$ as claimed (See Figure 3). □

Remark. In the above lemma, we differentiate the case of $z = x$ from the other cases, where we set $\delta_f = \infty$. Indeed, we may as well adopt a unified approach by setting $\delta_f = 1 + \delta(lca(z, x))$ for all cases. Note that there is no information loss, since $\delta_f > depth(x)$ if and only if the original δ_f is ∞; also, *no* change is needed with the query answering algorithm. Nevertheless, we shall stick to the original definition of δ_f as it is more intuitive.

Let us now get back to the original problem of finding the left and right boundaries in I_u of each ancestor u of v_P. Based on the definitions of δ_f and δ_l, we have the following lemma:

LEMMA 3.6. *Consider all the pointers originating from the subtree of v (i.e., the pointers that are in the* N*-structure of some descendant of v). If one such pointer satisfies $\delta_f \leq depth(v)$ (resp. $\delta_l \leq depth(v)$), then there exists an ancestor u of v such that this pointer is the first (resp. last) among all the pointers in the I-structure I_u that originate in the subtree of v.*

Conversely, for any ancestor u of v, if a pointer t is the first (resp. last) pointer among all the pointers in I_u that originate in the subtree of v, then t satisfies $\delta_f \leq depth(v)$ (resp. $\delta_l \leq depth(v)$).

PROOF. For the first part of the lemma, consider a pointer t originating in the subtree of v that satisfies $\delta_f \leq depth(v)$. Suppose that t points to I-structure I_u for some ancestor u of v. Now assume to the contrary that t is not the first pointer originating in subtree of v that reaches I_u. Then, there exists another pointer q originating in the subtree of v also reaching I_u, and the preorder rank of the origin of q is just less than that of t. In this case, δ_f of t must be strictly more that the depth of the lca of these two originating nodes (Lemma 3.5). Since both nodes are in the subtree of v, the lca is also in the subtree of v. Thus, δ_f of t is strictly more than $depth(v)$. For the converse, suppose t is the first pointer to reach I_u from the subtree of v. Then, consider a pointer q that appears just before t in I_u. The origin of q must be outside the subtree of v. Thus, δ_f of t is strictly more than the depth of the lca of the origins of q and t. Since this lca must be some proper ancestor of v, δ_f of t is at most $depth(v)$. Similar arguments work for the case of δ_l. □

By the above lemma, if we can search for all the pointers originating in the subtree of v_P that satisfy $\delta_f \leq depth(v_P)$ (resp. $\delta_l \leq depth(v_P)$), we can find the desired left (resp. right) boundary in I_u for each ancestor u of v_P. To facilitate the above search, we shall visit each node of the GST in preorder, concatenate the N-structures for all the nodes in one single array N, and construct two RMQ data structures (Lemma 2.5) over the δ_f entries and δ_l entries, respectively. Thus, there is a contiguous range in N corresponding to the subtree of v_P. Now we find all the δ_f and δ_l values in this range that are less than $depth(v_P)$ using Lemma 2.5, thus obtain the desired leftmost and rightmost pointers. As there are at most $2 \times depth(v_P)$ such pointers reported, the total time is $O(depth(v_P))$, which is $O(p)$.

LEMMA 3.7. *There exists a data structure of size $O(n)$ words for the top-k frequent document retrieval problem with query time $O(p + k \log k)$. If the outputs need not be reported in sorted order, the query time can be made optimal $O(p + k)$.* □

Although we described our result in described in terms of the term frequency metric, it can be easily generalized for handling arbitrary score functions, simply by replacing the $\texttt{freq}(\cdot, r)$ values by $score(path(\cdot), d_r)$ (We remark that only the construction algorithm may be affected, which depends on how easy it is to evaluate the given score function).

THEOREM 3.8. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$-word space, such that whenever a pattern P (of p characters) and an integer k come as a query, the index returns those k documents with the highest $score(P, \cdot)$ values in decreasing order of $score(P, \cdot)$ in $O(p + k \log k)$ time, where $score(P, d_r)$ of a document d_r is a predefined function dependent on the set of occurrences of P in d_r. If the outputs need not be sorted, the query time can be made optimal $O(p + k)$.*

3.5. Construction Algorithms

Although our data structure framework is very general for arbitrary score functions, the running time of our construction algorithm depends on how easily we can calculate the score for a given set of positions.

In the case of term frequency as the score function, we do the following: First, we construct a GST in $O(n)$ time [Farach 1997]. Next, we construct the LCA data structure of Bender and Farach-Colton [2000], also in $O(n)$ time, so that the lca of any two nodes in the GST can be reported in $O(1)$ time. Then, for each document d_r, we traverse all the leaves corresponding to d_r in GST and add an entry for d_r in each node that is an lca of successive leaves from document d_r; this is done in a total of $O(|d_r|)$ time. In this way, we have identified those nodes in the GST which are in the induced subtree formed by the leaves marked by d_r, and the transitive closure of their lca's. After that, we construct a suffix tree for document d_r in $O(|d_r|)$ time, then traverse this tree in postorder. Note that there is a one-to-one correspondence between the nodes in this suffix tree and the lca's found in the GST. Consequently, the pointer values of all entries can be determined while the frequency counts can be calculated by maintaining subtree sizes along the traversal. In total, the first three tuples of all entries in all N-structures (i.e., document id r, frequency score s, and pointer t) can be initialized in $O(n)$ time.

Next, we traverse the GST in preorder, and corresponding to each pointer in the N-structure encountered, we add an entry to the I-structure of the respective node. Once the entries in each I-structure are ready, we visit each I-structure and construct an RMQ data structure over it. This overall takes $O(n)$ time.

Now, it remains to show how to calculate the δ_f (similarly δ_l) values. For this, we traverse each of the I-structures I_w sequentially and get the list of origin nodes (they appear in preorder). Now, we take successive lca queries between consecutive origin nodes. The δ_f value for a particular node v is exactly equal to 1 plus the depth of the lca of v and its previous node in I_w, which can be computed in $O(1)$ time (see Lemma 3.5).[6] After computing all the δ_f values in all entries, we traverse all the N-structures in preorder and construct an RMQ structure over δ_f values. All of this can be accomplished in $O(n)$ time.

In the case of term proximity as the score function, we need more time to evaluate the score function; this is the only change. Precisely, the scores are first calculated over the suffix tree of each document d_r. For this, we do a recursive computation. Say at a node v, we have two children v_1 and v_2. Also assume that the following is available at v_1 (and v_2): (1) $mindist(v_1)$;[7] (2) a list $\mathcal{L}_1$ of text positions appearing in the subtree of v_1 in sorted format (stored as a binary search tree). Then, we first merge the list $\mathcal{L}_1$ at v_1 and the list $\mathcal{L}_2$ at v_2 to obtain the list $\mathcal{L}$ at v, and also during this merge operation we find out the closest pair of positions with one coming from the list at v_1 and the other from v_2. Now we compare the distance of this pair with $mindist(v_1)$ and $mindist(v_2)$ and obtain $mindist(v)$ for v. This merging step can be done in $O\left(|\mathcal{L}_1| \log(|\mathcal{L}_2|/|\mathcal{L}_1|)\right)$ time (assuming that $|\mathcal{L}_1| \leq |\mathcal{L}_2|$) using the merging algorithm of Brown and Tarjan's [1979]. This follows from finger-searching for list $\mathcal{L}_1$'s elements in the binary search tree for list $\mathcal{L}_2$. The total time is $O(|d_r| \log |d_r|)$ time, which can be shown by induction as follows. Without loss of generality, assume that the root of the suffix tree for d_r has two children v_1 and v_2 (if there are more children then we can merge them two at a time). Let n_1 be number of leaves in the subtree of v_1 and n_2 similarly for v_2 with $n_1 \leq n_2$. Thus, $n_1 + n_2 = |d_r|$. By induction, we can assume that computing $mindist$ over all nodes in subtree of v_1 (resp. v_2) takes $O(n_1 \log n_1)$ time (resp. $O(n_2 \log n_2)$ time). Then, computing $mindist$ over the whole suffix tree of d_r involves merging these two lists and obtaining the $mindist$ value at the root, taking a total of (ignoring constant factors) $n_1 \log n_1 + n_2 \log n_2 + n_1 \log(n_2/n_1) \leq |d_r| \log n_2 \leq |d_r| \log |d_r|$ time; this completes the argument for the induction. (See a similar analysis in Shah and Farach-Colton [2002].)

As the time to calculate $mindist$ scores over the suffix tree of a document d_r is $O(|d_r| \log |d_r|)$, this implies an $O(n \log n)$-time algorithm for calculating $mindist$ scores of all the N-structure entries in the GST. In general, the construction algorithm takes linear time plus a linear number of score function calculations.

4. COMPACT SPACE STRUCTURES

We first give a brief overview of the techniques we used for obtaining our space efficient (i.e., compact and succinct) indexes. The component GST can be replaced by its compressed version. However, the challenge is to efficiently encode the augmented information within small space; in particular δ_f and δ_l fields in N-structures [8] and

[6] If the lca is the node itself, then we set δ_f to be ∞.

[7] $mindist(v_1)$ denotes the minimum distance between the positions appearing in the subtree of v_1. If we stick to the earlier definition, this is exactly $\mathrm{TP}(path(v_1), d_r)$.

[8] Notice that the other three fields within N structures are used only for construction.

I-structures. We first remove δ_f and δ_l fields by deriving an alternative linear space index in Section 4.1. Here we choose term frequency as the score function, which is succinctly calculable. Therefore, the $score$ field from I-structures can also be deleted. Next each document id can be encoded in $\log D$ bits instead of $O(\log n)$ bits. The crucial part is the encoding of *origin* values. We show that it is possible to encode them in $O(n)$ bits overall. This leads to a compact space index taking roughly $3n \log D$ bits of space (in addition to the space for GST). The space is further optimized based on a sequence of observations such as I-structure entries with $score$ equals 1 can be deleted. These techniques lead to our compact space indexes described in Theorem 4.5 and Theorem 4.6.

4.1. An Alternative Linear Space Index

In this section, we present an alternative linear-space index without δ_f and δ_l fields, and achieve an $O(p + \log^2 \log n)$ term in the query time, which is still better than the original $O(p \log \log n)$ term. For this purpose, we introduce a criterion that categorizes the I-structure entries as *near* and *far*. Each *far* entry will be associated with some node, and the entries associating with the same node will be maintained together as a combined I-structure; this in turn reduces the number of boundaries to be searched to $O(p/\pi + \pi)$, where π is a sampling factor. We shall use a predecessor search structure (instead of the δ fields) to compute the boundaries. The result is summarized in the following lemma.

LEMMA 4.1. *There exists an index of size $O(n)$ words for the top-k frequent document retrieval problem with query time $O(p + \log^2 \log n + k \log \log \log n + k \log k)$.*

First, we mark all those nodes in GST whose $depth(\cdot)$ is a multiple of π ($depth$ of root is 0). Thus, any unmarked node is at most π nodes away from its lowest marked ancestor. Also, the number of marked ancestors of any node w is equal to $\lceil depth(w)/\pi \rceil$. Next, we categorize the I-structure entries as *far* and *near* as follows:

> *An entry $\langle$document id r, score s, origin $v\rangle$ in an I-structure I_w associated with a node w in* GST *is* near*, if there exists no marked node on the path from its origin v (inclusive) to w (exclusive); else, the entry is* far.

We restructure the entries such that a *far* entry is maintained in a *combined I-structure* (I^{c}-structure) associated with some marked node, as follows: Let x be the first marked node on the path from node w to *root*, and if an entry $e = \langle r, s, v\rangle$ in I_w is *far*, then we remove e from I_w and maintain e as $e' = \langle r, s, v, \eta\rangle$ in the combined I-structure $\mathsf{I}^{\mathsf{c}}_x$ associated with x; the fourth component $\eta < \pi$ (which we call as target_depth) is given by $depth(w) - depth(x)$. The I-structure I_w with its *far* entries removed will be called the *residue I-structure* $\mathsf{I}^{\mathsf{r}}_w$ of w.

The combined I-structures are maintained as follows: First, we sort the entries, in ascending order, according to the preorder ranks of the origins. Then, each $\mathsf{I}^{\mathsf{c}}_x$ is maintained using four separate arrays $\mathsf{Doc}^{\mathsf{c}}_x$, $\mathsf{Sco}^{\mathsf{c}}_x$, $\mathsf{Ori}^{\mathsf{c}}_x$ and $\mathsf{Dep}^{\mathsf{c}}_x$ such that the jth entry of $\mathsf{I}^{\mathsf{c}}_x$ is denoted by $\mathsf{I}^{\mathsf{c}}_x[j]$, and has the value $\langle \mathsf{Doc}^{\mathsf{c}}_x[j], \mathsf{Sco}^{\mathsf{c}}_x[j], \mathsf{Ori}^{\mathsf{c}}_x[j], \mathsf{Dep}^{\mathsf{c}}_x[j]\rangle$. We maintain a predecessor search structure [Willard 1983] over the $\mathsf{Ori}^{\mathsf{c}}_x$ array, and an RMQ structure (Lemma 2.5) over the $\mathsf{Sco}^{\mathsf{c}}_x$ array. We also maintain the structure described in Lemma 2.6 over the $\mathsf{Dep}^{\mathsf{c}}_x$ array. Similarly, each residue I-structure $\mathsf{I}^{\mathsf{r}}_w$ is stored as three separate arrays $\mathsf{Doc}^{\mathsf{r}}_w$, $\mathsf{Sco}^{\mathsf{r}}_w$ and $\mathsf{Ori}^{\mathsf{r}}_w$ such that the jth entry of $\mathsf{I}^{\mathsf{r}}_w$ is denoted by $\mathsf{I}^{\mathsf{r}}_w[j]$, and has the value $\langle \mathsf{Doc}^{\mathsf{r}}_w[j], \mathsf{Sco}^{\mathsf{r}}_w[j], \mathsf{Ori}^{\mathsf{r}}_w[j]\rangle$. As before, we maintain a predecessor search structure [Willard 1983] over the $\mathsf{Ori}^{\mathsf{r}}_w$ array, and an RMQ structure (Lemma 2.5) over the $\mathsf{Sco}^{\mathsf{r}}_w$ array. The total number of entries in the combined I-structures and the residue I-structures is exactly same as the number of I-structure entries, which is at most $2n$ (Lemma 3.3). Therefore, the overall space can be bounded by $O(n)$ words.

4.1.1. Answering Queries. Recall the notation from previous sections, where v_P represents the locus node of P and v'_P represents the rightmost leaf in the subtree of v_P. Let $u_1, u_2, \ldots, u_{depth(v_P)}$ denote the (proper) ancestors, and $u^*_1, u^*_2, \ldots, u^*_{\lceil depth(v_P)/\pi \rceil}$ denote the marked (proper) ancestors of v_P, respectively, in the order in which they appear on the path from v_P to *root*.[9] Note that both $u_{depth(v_P)}$ and $u^*_{\lceil depth(v_P)/\pi \rceil}$ denote the *root* node.

Let λ be an integer such that u_λ is the child of u^*_1 on the path from u^*_1 to v_P (if the parent of v_P is marked, then we say $\lambda = 0$). Therefore, $u_{\lambda+1}$ and u^*_1 denote the same node. Now, we show that instead of looking for answers from all those $depth(v_P)$ I-structures $\mathsf{I}_{u_1}, \mathsf{I}_{u_2}, \ldots, \mathsf{I}_{root}$, it is sufficient to search for answers within a fewer number of carefully chosen I^{r}-structures and I^{c}-structures, as shown in Figure 4.

LEMMA 4.2. *Let $[L_i, R_i], [L^{\mathsf{r}}_i, R^{\mathsf{r}}_i]$ and $[L^{\mathsf{c}}_i, R^{\mathsf{c}}_i]$ be the maximal contiguous ranges in $\mathsf{I}_{u_i}, \mathsf{I}^{\mathsf{r}}_{u_i}$ and $\mathsf{I}^{\mathsf{c}}_{u^*_i}$, respectively, such that all those entries in $\mathsf{I}_{u_i}[L_i, R_i]$, $\mathsf{I}^{\mathsf{r}}_{u_i}[L^{\mathsf{r}}_i, R^{\mathsf{r}}_i]$, and $\mathsf{I}^{\mathsf{c}}_{u^*_i}[L^{\mathsf{c}}_i, R^{\mathsf{c}}_i]$ for $i \geq 1$ originate from the subtree of v_P. Then,*

(1) *entries in* $\bigcup_{i=1}^{\lambda+1} \mathsf{I}^{\mathsf{r}}_{u_i}[L^{\mathsf{r}}_i, R^{\mathsf{r}}_i]$ *are the same as the* near *entries in* $\bigcup_{i=1}^{depth(v_P)} \mathsf{I}_{u_i}[L_i, R_i]$.

(2) *entries in* $\mathsf{I}^{\mathsf{c}}_{u^*_1}[L^{\mathsf{c}}_1, R^{\mathsf{c}}_1]$ *with* $\mathsf{Dep}^{\mathsf{c}}_{u^*_1}[\cdot] < depth(v_P) - depth(u^*_1)$ *correspond to the* far *entries in* $\bigcup_{i=1}^{\lambda} \mathsf{I}_{u_i}[L_i, R_i]$.

(3) *entries in* $\bigcup_{i=2}^{\lceil depth(v_P)/\pi \rceil} \mathsf{I}^{\mathsf{c}}_{u^*_i}[L^{\mathsf{c}}_i, R^{\mathsf{c}}_i]$ *correspond to the* far *entries in* $\bigcup_{i=\lambda+1}^{depth(v_P)} \mathsf{I}_{u_i}[L_i, R_i]$.

PROOF. Any entry in I_{u_i} originating in the subtree of v_P is a *far* entry if $i > \lambda + 1$, because $u_{\lambda+1}$ is the first marked node from v_P to the root. Thus, all the *near* entries in all the I-structures I_{u_i} that originate in the subtree of v must be exactly those entries in $\mathsf{I}^r_{u_1}, \mathsf{I}^r_{u_2}, \ldots, \mathsf{I}^r_{\lambda+1}$. This gives the result in (1).

For each *far* entry e in $\bigcup_{i=1}^{\lambda} \mathsf{I}_{u_i}[L_i, R_i]$, there will be a corresponding entry e' in $\mathsf{I}^{\mathsf{c}}_{u^*_1}[L^{\mathsf{c}}_1, R^{\mathsf{c}}_1]$. However, the converse is not true; $\mathsf{I}^{\mathsf{c}}_{u^*_1}[L^{\mathsf{c}}_1, R^{\mathsf{c}}_1]$ may contain some entry b' whose corresponding *far* entry is not within $\bigcup_{i=1}^{\lambda} \mathsf{I}_{u_i}[L_i, R_i]$. This happens if and only if the target node of b is not an ancestor of v_P (See Figure 4 for an example). We can remove such entries with the constraint $\mathsf{Dep}^{\mathsf{c}}_{u^*_1}[\cdot] < depth(v_P) - depth(u^*_1)$; this gives the result in (2).

Finally, for entries in $\bigcup_{i=2}^{\lceil depth(v_P)/\pi \rceil} \mathsf{I}^{\mathsf{c}}_{u^*_i}[L^{\mathsf{c}}_i, R^{\mathsf{c}}_i]$, each of their target nodes must be an ancestor of v_P; thus, they are exactly the *far* entries in $\bigcup_{i=\lambda+1}^{depth(v_P)} \mathsf{I}_{u_i}[L_i, R_i]$. This gives the result in (3). $\square$

Based on the above lemma, after the initial pattern search in $O(p)$ time, we can compute k candidate entries from each category, and then compute the actual top k answers by comparing the scores of these $3k$ entries. In category (i), we have $\lambda + 1 \leq \pi = \log \log n$ boundaries to be searched, which takes $O(\pi \log \log n)$ time, and then we retrieve the k candidate answers in unsorted order in $O(\pi + k)$ time using RMQ structure (Lemma 2.5) over the $\mathsf{Sco}^{\mathsf{r}}_{\{\cdot\}}$-arrays. In category (iii) we search for at most $\lceil depth(v_P)/\pi \rceil$ boundaries, which takes $O((p/\pi + 1) \log \log n))$ time, and then we retrieve the k candidate answers, unsorted, in $O(p/\pi + k)$ time with the RMQ structure (Lemma 2.5) over

[9]Assume that P is not an empty string.

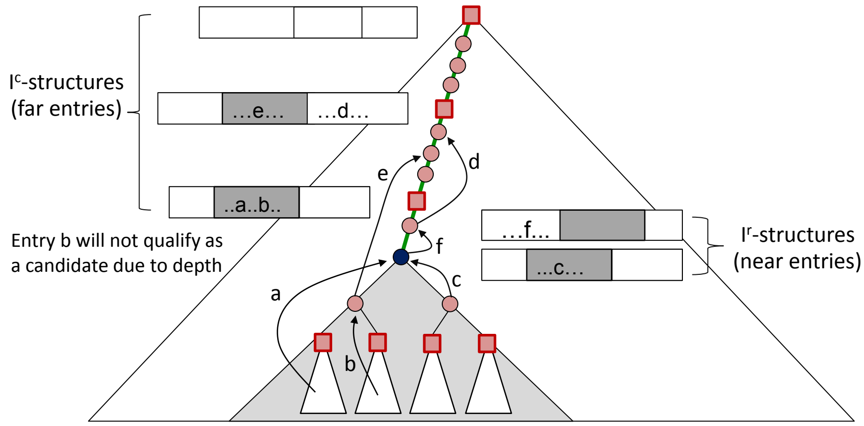

Fig. 4. Querying on I^c and I^r structures

the $\mathsf{Sco}^{\mathsf{c}}_{\{\cdot\}}$-arrays. For category (ii), we use the structure described in Lemma 2.6 over the array $\mathsf{Dep}^{\mathsf{c}}_{u_1^*}$. The time to get the candidates is $O((\log \pi + k)(\log \pi + t_{score}))$ (Note that $t_{score} = O(1)$). Finally, it takes $O(k \log k)$ time to sort these $O(k)$ candidate documents and report those k highest-scoring ones as the final output. Putting everything together and setting $\pi = \log \log n$, we obtain Lemma 4.1.

4.2. Achieving Compact Space

This section shows how to encode our alternative linear-space index in compact space. The major contribution is that, instead of using $O(\log n)$ bits for an entry, we design some novel encodings so that each entry requires only $\log D + \log \pi + O(1)$ bits. The GST will be replaced by a CSA (refer to Section 2.3) along with the tree encoding of GST in $4n + o(n)$ bits (refer to Section 2.5). Thus, the locus node v_P can be obtained by first computing the suffix range $[sp, ep]$ of P in $t_s(p)$ time using CSA and then by taking the lca of leaves ℓ_{sp} and ℓ_{ep} in GST using the tree encoding structure in $O(1)$ time (refer to Section 2.5). A core component of our index is the document array E (refer to Section 2.4), which can be used for efficient encoding and decoding of entries in I^{c}- and I^{r}-structures. We remark that the original I-structures are not stored anymore.

4.2.1. Document ID Encoding. Each document id can be encoded in $\log D$ bits. First we obtain an array $\mathsf{Doc}^{\mathsf{r}} = \mathsf{Doc}^{\mathsf{r}}_1 \mathsf{Doc}^{\mathsf{r}}_2 \mathsf{Doc}^{\mathsf{r}}_3 \cdots$ by concatenating $\mathsf{Doc}^{\mathsf{r}}_{\{\cdot\}}$-arrays in ascending order of the preorder rank of the node to which it is associated. Let m_i represent the number of elements in $\mathsf{Doc}^{\mathsf{r}}_i$. We maintain a bit vector $\mathsf{B}^{\mathsf{r}} = 10^{m_1}10^{m_2}10^{m_3}\cdots$, with a constant-time rank/select structure over it [Raman et al. 2007]. Note that $\mathsf{Doc}^{\mathsf{r}}$ can be represented in $n_{near} \log D$ bits and B^{r} in $2n + n_{near} + o(n)$ bits, where n_{near} (resp., n_{far}) represents the number of I-structure entries that are *near* (resp., *far*). Now given any i and j, the position of $\mathsf{Doc}^{\mathsf{r}}_i[j]$ within $\mathsf{Doc}^{\mathsf{r}}$ can be located in $O(1)$ time as follows: Find the ith occurrence of 1 in B^{r}, count the number of 0s till that position, and add j. After that the desired $\mathsf{Doc}^{\mathsf{r}}_i[j]$ value can be reported in $O(1)$ time. In a similar way, the arrays $\mathsf{Doc}^{\mathsf{c}}_{\{\cdot\}}$ can also be encoded and maintained in $n_{far} \log D + O(n)$ bits. The overall space can be bounded by $(n_{near} + n_{far}) \log D + O(n) \leq 2n \log D + O(n)$ bits. That is, $2 \log D + O(1)$ bits per entry.

4.2.2. Term Frequency Encoding. Given an entry (in an I^{r}- or an I^{c}-structure) with origin v and document id r, the corresponding score $\mathtt{freq}(v, r)$ is exactly the number of occurrences of r in $\mathsf{E}[i..j]$, where ℓ_i and ℓ_j are the leftmost leaf and the rightmost leaf of v, respectively. Thus, given the values v and r, we can find i and j in constant time based on the tree encoding of GST, and then compute $\texttt{freq}(v, r)$ in $O(\log\log D)$ time based on two rank queries on E. Therefore, we can safely discard the *score* field completely for all I^{c}- and I^{r}-structures, but instead keep the RMQ structures over them; this requires only $2 + o(1)$ bits per entry [Fischer and Heun 2011].

4.2.3. Origin Encoding. Encoding the origin arrays ($\mathsf{Ori}^{\mathsf{r}}_{\{\cdot\}}$ and $\mathsf{Ori}^{\mathsf{c}}_{\{\cdot\}}$) is the trickiest part and is based on the following lemma.

LEMMA 4.3. *For any given document d_r and any child node w_q of w (where w_q denotes the q^{th} leftmost child of w) , there cannot be more than one entry in I_w with document id r and origin from the subtree of w_q.*

PROOF. This can be proved via contradiction. Assume that there are two or more such entries. Then the *lca* of their origins must be a node in the subtree of w_q, and hence these entries will be associated with an I-structure of some node in the subtree of w_q instead of w. □

From the definition of N-structures, if there exists an entry in I_w with document id r and *origin* a node in the subtree of w_q for some $q \in [1, degree(w)]$, then this *origin* node is the *lca* of the leftmost leaf and the rightmost leaf with document id r in the subtree of w_q. To compute this *origin* node, we can use the document array E and the tree encoding of GST as follows: First find the leftmost leaf ℓ_a and the rightmost leaf ℓ_b in the subtree of w_q in $O(1)$ time (using $lmost\_leaf(w_q)/rmost\_leaf(w_q)$ operations on the tree encoding of GST, refer to Section 2.5), then find the first and last occurrences of r, say $\mathsf{E}[a']$ and $\mathsf{E}[b']$, among the entries in $E[a..b]$ in $O(\log\log D)$ time, and finally compute the *lca* of $\ell_{a'}$ and $\ell_{b'}$. In light of these findings, we show how to efficiently encode $\mathsf{Ori}^{\mathsf{r}}_{\{\cdot\}}$-arrays and $\mathsf{Ori}^{\mathsf{c}}_{\{\cdot\}}$-arrays.

Encoding $\mathsf{Ori}^{\mathsf{r}}_{\{\cdot\}}$*-arrays.* Instead of maintaining $\mathsf{Ori}^{\mathsf{r}}_w$-array, we maintain another array $\mathsf{Ori\_child}^{\mathsf{r}}_w$, such that $\mathsf{Ori\_child}^{\mathsf{r}}_w[j] = q$ if node $\mathsf{Ori}^{\mathsf{r}}_w[j]$ is from the subtree of w_q. As the elements in $\mathsf{Ori}^{\mathsf{r}}_{\{\cdot\}}$ are monotonically increasing, the elements in $\mathsf{Ori\_child}^{\mathsf{r}}_{\{\cdot\}}$ are also monotonically increasing. In addition, the value of each entry is between 1 and $degree(w)$. Therefore, we shall encode $\mathsf{Ori\_child}^{\mathsf{r}}_w$ in $(|\mathsf{I}^{\mathsf{r}}_w| + degree(w))(1 + o(1))$ bits (refer to Section 2.8), so that we can decode $\mathsf{Ori\_child}^{\mathsf{r}}_w[j]$ for any given j in constant time. Then, from $\mathsf{Ori\_child}^{\mathsf{r}}_w[j]$, we can decode $\mathsf{Ori}^{\mathsf{r}}_w[j]$ in $O(\log\log D)$ time as described earlier. The total space for encoding all $\mathsf{Ori}^{\mathsf{r}}_{\{\cdot\}}$-arrays can be bounded by $O(\sum_{w\in\mathsf{GST}}(|\mathsf{I}^{\mathsf{r}}_w| + degree(w))) = O(n)$ bits.

Encoding $\mathsf{Ori}^{\mathsf{c}}_{\{\cdot\}}$*-arrays.* First we introduce the following notions. Let w^* be a marked node in GST, then another node w^*_q is called its qth marked child, if w^*_q is the qth smallest (in terms of preorder rank) marked node with w^* as its lowest marked ancestor. Given the preorder rank of w^*, the preorder rank of w^*_q can be computed in constant time by maintaining an additional $O(n)$-bit structure as follows: Let GST^* be the tree induced by the marked nodes in GST, so that w^* is the lowest marked ancestor of w^*_q in GST if and only if the node corresponding to w^* in GST^* (say, w) is the parent of the node corresponding to w^*_q (say w_q) in GST^*. Moreover, w^*_q is said to be the qth marked child of w^* in GST, if w_q is the qth child of w in GST^*. Given the preorder rank of any marked node in GST, its preorder rank in GST^* (and vice versa) can be computed in constant time by maintaining an additional bit vector of size $2n + o(n)$ that maintains the information if a node is marked or not. We remark that this works only because the encoding is in preorder.

In the case of entries in a combined I-structure, Lemma 4.3 may not hold. However, the following holds: there cannot be two entries (that are *far*) in $\mathsf{I}^{\mathsf{c}}_{w^*}$ with the same document id and both their origins coming from the subtree of the same marked child w^*_q of w^*. Therefore, instead of array $\mathsf{Ori}^{\mathsf{c}}_{w^*}$, we shall maintain another array $\mathsf{Ori\_child}^{\mathsf{c}}_{w^*}$, such that $\mathsf{Ori\_child}^{\mathsf{c}}_{w^*}[j] = q$ if node $\mathsf{Ori}^{\mathsf{c}}_{w^*}[j]$ is from the subtree of w^*_q. Using a similar scheme as before, all $\mathsf{Ori}^{\mathsf{c}}_{\{\cdot\}}$-arrays can be encoded in $O(n)$ bits, and each entry can be decoded in $O(\log\log D)$ time using document array E and the tree encoding of GST.

4.2.4. Compressing Predecessor Search Structures. The predecessor search structure over $\mathsf{Ori}^{\mathsf{r}}_{\{\cdot\}}$-arrays and $\mathsf{Ori}^{\mathsf{c}}_{\{\cdot\}}$-arrays, which requires $O(\log n)$ bits per element, can be replaced by a sampled predecessor search structure as follows: If the length of an array is at most $\log^2 n$, we do not maintain any structure over such an array as we can answer a query by binary search in $O(\log\log n)$ time. Otherwise, we construct a new array by sampling every $(\log^2 n)$th element in $\mathsf{Ori}^{\mathsf{r}}_{\{\cdot\}}$, and maintain predecessor search structure over it. When answering a query, we can first search this sampled structure for an approximate answer, and then obtain the exact answer by a binary search on a smaller range of only $\log^2 n$ elements in the original array. The search time still remains $O(\log\log n)$. The overall space for these sampled structures can be bounded by $o(n)$ bits.

4.2.5. $\mathsf{Dep}^{\mathsf{c}}$*-arrays Encoding.* We use the result in Lemma 2.6, and the total space required can be bounded by $O(n\log\pi)$ bits. Note that $t_{score} = O(\log\log D)$ as $score$ values are no more stored explicitly.

4.2.6. Overall Performance. Finally, the RMQ structures are maintained as before, requiring $O(n)$ bits overall. Putting all together, the total space can be bounded by $|\mathsf{CSA}| + 3n\log D(1+o(1)) + O(n\log\pi)$ bits. The query answering algorithm remains the same as that in our linear index in Lemma 4.1, except that decoding *origin* and *term frequency* score takes $O(\log\log D)$ time. The initial time for pattern search and finding the locus node v_P is $t_s(p) + O(1)$. The time for predecessor search queries can be bounded by $O((p/\pi + \pi)\log\log n \log\log D)$. Note that this $\log\log D$ factor comes from the time for decoding $origin$ values. Then, the time to obtain the top k answers from Category (i) and Category (iii) in Lemma 4.2 will be $O((p/\pi + \pi + k)\log\log D)$ and that from Category (ii) will be $O((\log\pi + k)(\log\pi + \log\log D))$; finally it takes $O(k\log k)$ time for choosing the desired k answers from the above $3k$ answers. By setting $\pi = \log^2\log n$, we obtain the following lemma.

LEMMA 4.4. *There exists an index of size* $|\mathsf{CSA}| + 3n\log D(1+o(1)) + O(n\log\log\log n)$ *bits for the top-*k *frequent document retrieval problem with query time* $O(t_s(p) + p + \log^4\log n + k\log\log n + k\log k)$. □

The index space can be improved further. We first show how to remove the $O(n\log\log\log n)$ term. Note that when $D + \sigma > \log^{1/3} n$, the $O(n\log\log\log n)$ term can be absorbed in the $o(n\log D + n\log\sigma)$ term. Otherwise, we can construct a very simple index that consists of the following components: CSA, document array E, and a table that maintains the top k documents for all distinct patterns of length at most $\sqrt{\log n}$. Such a table can be maintained in $O(\sum_{i=1}^{\sqrt{\log n}} \sigma^i D\log D) = o(n)$ bits and can report the top k documents in optimal $O(p+k)$ time when $p < \sqrt{\log n}$. If $p \geq \sqrt{\log n}$, we shall simply compute the term frequency of all documents using E in $O(D\log\log D)$ time, and then report the top k highest-scoring ones in an extra $O(D\log D)$ time. As D is bounded by $O(\log^{1/3} n)$, the total time can be bounded by $O(t_s(p) + p)$. Therefore, the index space can be bounded by $|\mathsf{CSA}| + 3n\log D + o(n\log D + n\log\sigma)$ bits.

The space can be further reduced by $n \log D$ bits from the following observation: The term frequency is 1 for any entry whose origin is a leaf in GST, and there are n such entries (in I^c and I^r structures combined). We shall delete all such entries, only that a problem will arise when we query a pattern P for the top k answers, and $k' < k$ documents are reported. In this case, since only documents with term frequency of at least 2 are reported, we need check if there are documents with term frequency 1 to make up the top k answers.

To get documents with term frequency 1, we shall apply Muthukrishnan's chain array idea [Muthukrishnan 2002]. The chain array $\mathsf{C}[1..n]$ is defined with $\mathsf{C}[i] = j$, where $j < i$ is the largest number with $\mathsf{E}[i] = \mathsf{E}[j]$. As the chain array can be simulated using E as $j = \mathit{select}_{\mathsf{E}}(\mathsf{E}[i], \mathit{rank}_{\mathsf{E}}(\mathsf{E}[i], i) - 1)$ in $O(\log \log D)$ time, it will not be maintained explicitly. In addition, we will maintain an RMQ structure over C, taking $2n + o(n) = o(n \log D)$ bits. Let $[sp, ep]$ be the suffix range of P in the CSA. Then, we can obtain all those documents $d_{\mathsf{E}[i]}$ such that $sp \leq i \leq ep$ and $C[i] < sp$ using repeated RMQs; these documents are exactly those that contain P and are distinct [Muthukrishnan 2002]. To address our current problem, once we have obtained a document $d_{\mathsf{E}[i]}$ from the above procedure, we check if its term frequency is 1 in $O(\log \log D)$ time. Note that we only need to obtain and check up to k documents for our purpose, in which case, there will be $k - k'$ documents with term frequency 1 to make up the top k answers.[10] The overall time complexity is increased by $O(k \log \log D)$, and is thus unchanged.

THEOREM 4.5. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $|\mathsf{CSA}| + 2n \log D + o(n \log D + n \log \sigma)$ bits of space, such that whenever a pattern P (of p characters) and an integer k come as a query, the index returns those k documents with the highest $\mathtt{TF}(P, \cdot)$ values in decreasing order of $\mathtt{TF}(P, \cdot)$ in $O(t_s(p) + p + \log^4 \log n + k \log \log n + k \log k)$ time, where $\mathtt{TF}(P, d_r)$ of a document d_r counts the number of times P occurs in d_r.*

The index space can be further improved as summarized in the following theorem.

THEOREM 4.6. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $|\mathsf{CSA}| + n \log D + o(n \log D + n \log \sigma)$ bits of space, such that whenever a pattern P (of p characters) and an integer k come as a query, the index returns those k documents with the highest $\mathtt{TF}(P, \cdot)$ values in decreasing order of $\mathtt{TF}(P, \cdot)$ in $O(t_s(p) + p + \log^6 \log n + k((\log \sigma \log \log n)^{1+\epsilon} + \log^2 \log n + \log k))$ time, where $\mathtt{TF}(P, d_r)$ of a document d_r counts the number of times P occurs in d_r, and $\epsilon > 0$ is any constant.*

PROOF. If $\log D \leq (\log \sigma \log \log n)^{1+\epsilon}$, we shall use an alternative index as described in Lemma 5.5 in Section 5.1 (with constants adjusted properly) to achieve the result. Otherwise, $\log D > (\log \sigma \log \log n)^{1+\epsilon}$. Then, instead of using $n \log D(1 + o(1))$ bits to represent E, we choose the representation described in Corollary 2.3, whose space is $O(n \log \sigma \log \log n) = o(n \log D)$ bits. The resulting query time is $O(t_s(p) + (p/\pi + \pi) \log \log n \times \log^2 \log n + k \log^2 \log n + k \log k)$, as the time for $\mathit{rank}_{\mathsf{E}}$ operation is now $O(\log^2 \log n)$. By choosing $\pi = \log^3 \log n$, we achieve an index of total size $|\mathsf{CSA}| + n \log D(1 + o(1))$ bits with query time $O(t_s(p) + p + \log^6 \log n + k((\log \sigma \log \log n)^{1+\epsilon} + \log^2 \log n + \log k))$. The theorem thus follows. □

[10]In the boundary case where P occurs in fewer than k documents, we shall report all the documents obtained from querying the chain array.

5. SUCCINCT SPACE STRUCTURES

In Section 4, to save space, we used compression techniques and also explored the issue of maintaining fewer I-structure entries by eliminating those originating at leaves. Here, we further save space by sparsifying the linear-space index, and by keeping only selected top-scoring entries (pointers) which pass through some carefully chosen nodes. By doing this, we show that the augmenting information on the top of GST becomes $o(n)$ bits. Now, if the query requires us to report answers which are not stored as a part of these selected entries, we resort to on-the-fly computation (here also we choose term frequency as the score function). Notice that the more top-scoring entries we maintain, the less on-the-fly computation needs to be done while we perform query processing. In Section 5.1, we describe such a scheme, where we use $O(\log n)$ bits for maintaining an entry. Later in Section 5.2, we show how to encode an entry in $O(\log \log n)$ bits, which means we can now maintain even more top-scoring entries, thus allowing lesser number of on-the-fly computation. We start with the following notation:

— $\mathit{Leaf}(x)$ denotes the set of leaves in the subtree of node x in GST.
— $\mathit{Leaf}(x\backslash y)$ denotes the leaves in the subtree of x, but not in that of y. That is, $\mathit{Leaf}(x\backslash y) = \mathit{Leaf}(x) \setminus \mathit{Leaf}(y)$.

Let g be a parameter called the *grouping factor*. Using the following scheme, we identify a subset S_g of nodes, called *marked nodes*, in GST: First, we traverse the leaves of GST from left to right to form groups of g contiguous leaves. That is, the first group consists of leaves $\ell_1, \ell_2, \ldots, \ell_g$, the next group consists of $\ell_{g+1}, \ldots, \ell_{2g}$, and so on. In total, there will be $\lceil n/g \rceil$ groups. Next, for each group, we mark the lca in GST of its first and last leaves; the total number of marked nodes will be at most $\lceil n/g \rceil$. After that, we do further marking, such that if nodes u and v are marked, then $lca(u, v)$ will be marked. Finally, we mark the leftmost and the rightmost leaves within the subtree rooted at each marked node.

LEMMA 5.1. *The above marking scheme ensures the following properties:*

(1) *The number of marked nodes, $|S_g|$, is bounded by $O(n/g)$.*
(2) *If there is no marked node in the subtree of x, then $|\mathit{Leaf}(x)| < 2g$.*
(3) *The highest marked descendant node y of any unmarked node x, if it exists, is unique, and $|\mathit{Leaf}(x\backslash y)| < 2g$.*

PROOF. The number of groups at the end of first step is $\lceil n/g \rceil$, and at most one internal node corresponding to each group is marked. Thus, at the end of the first step, there are at most $\lceil n/g \rceil$ marked nodes. Next, we mark the *lca* of these marked nodes; the total number of marked nodes will at most be doubled (as the marked nodes now form an induced subtree, with marked nodes at the end of first step as leaves), so that it is bounded by $O(n/g)$. Finally, we mark the leftmost and the rightmost leaf nodes of every marked node. Thus, the the total number of marked nodes will at most be tripled, so that it is bounded by $O(n/g)$. This gives the result in (1).

Whenever $|\mathit{Leaf}(x)| \geq 2g$, there will be at least one group completely contained in the subtree of x. The *lca* of the first and the last leaves in such a group is within the subtree of x, and is marked. Thus, by contraposition, the result in (2) follows.

The last statement in the lemma can be proved as follows: Let ℓ_L and ℓ_R be the leftmost and the rightmost leaves in the subtree of x. Then, according to our marking scheme, y is the *lca* of leaves $\ell_{L'}$ and $\ell_{R'}$, where $L' = g\lceil L/g \rceil + 1$ and $R' = g\lceil R/g \rceil$. Let ℓ_{L^*} and ℓ_{R^*} be the leftmost and the rightmost leaves respectively, that are in the subtree of y. Then clearly $L \leq L^* \leq L' < L + g$ and $R \geq R^* \geq R' > R - g$. Therefore, $|\mathit{Leaf}(x\backslash y)| = (L^* - L) + (R - R^*) < 2g$. □

Let $\mathsf{top}(x, k)$ represent the list (or set) of top-k documents corresponding to a pattern with node x as the locus. Maintaining $\mathsf{top}(x, k)$ explicitly for all possible x values and k values is not possible in compressed space. Instead, we maintain $\mathsf{top}(x, k)$ only for marked nodes x (with respect to various carefully chosen g values) and for values of k that are powers of 2, such that $\mathsf{top}(x, k)$ for the general x and k can be efficiently computed on the fly. We next prove the following lemma.

LEMMA 5.2. *By maintaining an index called* GST_g *of size* $O((n/g)\log g) + O(n/\log^2 n)$ *bits, the following query can be answered in* $O(1)$ *time: Given a suffix range* $[sp, ep]$ *of a pattern* P *as an input, find the node* v_P^* *and the range* $[sp^*, ep^*]$*, where* (i) v_P^* *denotes the highest-marked descendent of the locus node* v_P *of* P*, and* (ii) ℓ_{sp^*} *and* ℓ_{ep^*} *denote, respectively, the leftmost leaf and the rightmost leaf in the subtree of* v_P^*.

PROOF. The index GST_g, requiring $O((n/g)\log g) + O(n/\log^2 n)$ bits of space, consists of the following components:

(1) A compact trie obtained by retaining only those nodes in GST that are marked. Then, corresponding to every marked node in GST, there will be a unique node in this trie and vice versa. As the number of marked nodes is $O(n/g)$, the topology of this trie can be maintained in $O(n/g)$ bits of space (refer to Section 2.5).
(2) A bit-vector $\mathsf{B_{no}}[1..2n]$, where $\mathsf{B_{no}}[i] = 1$ if the ith node in GST is marked, else 0. This can be maintained in $|S_g|\log(n/|S_g|) + O(|S_g|) + O(n/\log^{O(1)} n) = O((n/g)\log g) + O(n/\log^2 n)$ bits of space [Patrascu 2008],[11] so that the operations $select_{\mathsf{B_{no}}}(j)$ (the position of the jth 1 in $\mathsf{B_{no}}$) and $rank_{\mathsf{B_{no}}}(i)$ (the number of 1s in $\mathsf{B_{no}}[1..i]$) can be supported in $O(1)$ time.
(3) A bit-vector $\mathsf{B_{le}}[1..n]$, where $\mathsf{B_{le}}[i] = 1$ if the ith leftmost leaf in GST is marked, else 0. As in the case of $\mathsf{B_{no}}$, $\mathsf{B_{le}}$ will be maintained in $O((n/g)\log g) + O(n/\log^2 n)$ bits, so that it can support $select_{\mathsf{B_{le}}}(\cdot)$ and $rank_{\mathsf{B_{le}}}(\cdot)$ operations in $O(1)$ time.

Given an input suffix range $[sp, ep]$, the sp^*th leaf is the first marked leaf towards the right side of ℓ_{sp} (inclusive), and the ℓ_{ep}^*th leaf is the last marked leaf towards the left side of ℓ_{ep} (inclusive), in GST. These two leaves will correspond to the sp'th and the ep'th leaves in the compact trie, where

$$sp' = 1 + rank_{\mathsf{B_{le}}}(sp - 1) \text{ and } ep' = rank_{\mathsf{B_{le}}}(ep);$$

the desired values of sp^* and ep^* can thus be computed, in $O(1)$ time, by $sp^* = select_{\mathsf{B_{le}}}(sp')$ and $ep^* = select_{\mathsf{B_{le}}}(ep')$.

We now show how to find v_P^*, which is the lca of ℓ_{sp^*} and ℓ_{ep^*} in GST. As GST is not stored explicitly, we shall find v_P^* in an indirect way. First, we identify the leaf nodes corresponding to ℓ_{sp^*} and ℓ_{ep^*} in the compact trie, which is its sp'th and ep'th leaves. Next, we find their lca (say, with preorder rank x) in the compact trie; such a node will correspond to v_P^* in GST. It follows that v_P^* is the xth marked node in GST, so that we can finally find (the preorder rank of) v_P^* in GST by $select_{\mathsf{B_{no}}}(x)$. The procedure again takes $O(1)$ time in total, as it involves only a constant number of rank/select operations and an lca operation. □

[11] In the word-RAM model, we can represent a bit vector of length n with m 1s in $\log_2 \binom{n}{m} + O(n/\log^t n) + O(n^{3/4}\log^{O(1)} n)$ bits of space, so that each rank/select query can be supported in $O(t)$ time, where t is any positive integer constant (refer to Theorem 2 in [Patrascu 2008]). Moreover, $\log \binom{n}{m} \le m\log(ne/m) \approx m\log(n/m) + 1.44m$ [Pagh 2001].

5.1. The Compressed Index

Our compressed index will make use of both CSA of the concatenated text T of all the documents, and a compressed suffix array CSA_r of each individual document d_r. We prove the following in this section.

THEOREM 5.3. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $2|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits of space, such that whenever a pattern P (of p characters) and an integer k come as a query, the index returns those k documents with the highest $\mathtt{TF}(P, \cdot)$ values in decreasing order of $\mathtt{TF}(P, \cdot)$ in $O(t_s(p) + k \times t_{sa} \log^{2+\epsilon} n)$ time; here, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA, and $\epsilon >$ is any constant.*

A set $S_{\mathsf{cand}} \subseteq \mathcal{D}$ is called a *candidate set* of a query if it is a multiset that contains all those documents in the answers to the query. Therefore, once the candidate set is given, the top-k query can be answered by first finding the $\mathtt{TF}(P, d_r)$ score of each document $d_r \in S_{\mathsf{cand}}$, and then reporting the k highest-scoring ones.

LEMMA 5.4. *Once the candidate set S_{cand} is identified, a top-k query can be answered in $O(|S_{\mathsf{cand}}| \times t_{sa} \log \log n + k \log k)$ time using CSA and the structure described in Lemma 2.2.*

PROOF. First, we remove duplicates in S_{cand} if there are any. This can be easily done in $O(|S_{\mathsf{cand}}|)$ time by maintaining an extra bit vector $\mathsf{B}_{\mathsf{cand}}[1..D]$, where all its bits are initialized to 0. Note that this additional structure will not change the space bound in Theorem 5.3. Then, we scan all documents in S_{cand} one by one and do the following: If a document $d_r \in S_{\mathsf{cand}}$, then we check if $\mathsf{B}_{\mathsf{cand}}[r]$ is 0. If so, we set $\mathsf{B}_{\mathsf{cand}}[r] = 1$; otherwise, we delete such an occurrence of d_r (which is a duplicate) from S_{cand}. After scanning all the documents in S_{cand}, we can reset all bits in $\mathsf{B}_{\mathsf{cand}}$ back to 0 by rescanning S_{cand} once.

Next, we compute the $\mathtt{TF}(P, d_r)$ score for all those documents $d_r \in S_{\mathsf{cand}}$ in $O(t_{sa} \log \log n)$ time per document (refer to Lemma 2.4). To retrieve the top-k answers from this, we first find the score X of the kth highest-scoring document using the linear time selection algorithm [Blum et al. 1973]. Then, we get those documents whose scores are at least X; note that there may be more than k of them, because of ties. To get the desired answer, we shall remove the excess (whose scores are equal to X). Finally, we spend another $O(k \log k)$ time to obtain the answers in sorted order of their scores. □

The query time in the above lemma is dependent on the size $|S_{\mathsf{cand}}|$ of the candidate list. To speed up the whole process (so as to achieve the claimed result in Theorem 5.3), our objective is to find a candidate set whose size is as small as possible.

5.1.1. Index for Top-k Queries for a Fixed k. First, we define an index for answering top-k frequent queries, where k is fixed in advance. The index consists of (i) a compressed suffix array CSA of T; (ii) the document array E (represented in $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits, refer to Lemma 2.2); (iii) an auxiliary structure that includes (a) the GST_g index (refer to Lemma 5.1) with a grouping factor $g = k \log^{2+\epsilon} n$, and (b) for each marked node $x \in S_g$ in GST, we store $\mathsf{top}(x, k)$ explicitly in $k \log D$ bits. The total space of the auxiliary structures is $O((n/g) k \log D) + O(n/\log^2 n) = o(n/\log n)$ bits.

Query Answering. First, we find the suffix range $[sp, ep]$ of P in $t_s(p)$ time using CSA. Let v_P be the locus node of P. Then, we find v_P^* and $[sp^*, ep^*]$ in $O(1)$ time, where v_{P^*}

is the highest marked descendent of v_P (if it exists), and $[sp^*, ep^*]$ is the suffix range corresponding to v_P^* in GST (refer to Lemma 5.2). Then,

$$\mathsf{top}(v_P^*, k) \cup \left\{ d_{\mathsf{E}[j]} \mid j \in [sp, sp^* - 1] \cup [ep^* + 1, ep] \right\}$$

will be a candidate set.[12] The number of documents in $\mathsf{top}(v_P^*, k)$ is at most k, and the number of remaining documents in the candidate set is at most $2g$ (refer to Lemma 5.1). To construct the candidate set, we first retrieve all documents in $\mathsf{top}(v_P^*, k)$ in $O(k)$ time, as these documents are precomputed and explicitly stored at v_P^*; then, since each $\mathsf{E}[\cdot]$ value can be decoded in $O(t_{sa})$ time (refer to Lemma 2.2), we retrieve all the remaining documents in $O(g \times t_{sa})$ time. In summary, we obtain a candidate set of $O(g + k)$ documents in $O(g \times t_{sa} + k)$ time. Combining with Lemma 5.4, the top-k documents can be answered in another $O((g + k) \times t_{sa} \log\log n)$ time. By substituting $g = k \log^{2+\epsilon} n$ the resulting query time will be $O(t_s(p) + k \times t_{sa} \log^{2+\epsilon} n \log\log n) = O(t_s(p) + k \times t_{sa} \log^{2+\epsilon} n)$ (the $\log\log n$ term is absorbed in the $\log^\epsilon n$ term).

5.1.2. Index for Top-k Queries for General k. To support top-k queries for general k, we maintain CSA, E, and (at most) $\log D$ auxiliary structures of Section 5.1.1 for any fixed k that is a power of 2 (i.e., $k = 1, 2, 4, 8, \ldots, D$). Since an auxiliary structure for a specific k requires $o(n/\log n)$ bits, the overall increase in total space is bounded by $o(n)$ bits. Now, a top-k query for a general k can be answered by choosing $z = 2^{\lceil \log_2 k \rceil}$ and retrieving the top-z documents by querying on the auxiliary structure specific to z. Then, we select the k highest-scoring documents (using [Blum et al. 1973]) and report them in decreasing order of score. Since $k = \Theta(z)$, the resulting query time will be $O(t_s(p) + k \times t_{sa} \log^{2+\epsilon} n)$. This completes the proof of Theorem 5.3.

As a corollary, we can obtain a simple compact index by rederiving Theorem 5.3 with $g = z \log^{1+\epsilon} D$, and maintaining E explicitly as in Lemma 2.1. The resulting query time will be $O(t_s(p) + k \log^{1+\epsilon} D \log\log D + k \log k) = O(t_s(p) + k \log^{1+\epsilon} D)$ (the $\log\log D$ term is absorbed in the $\log^\epsilon D$ term).

LEMMA 5.5. *There exists an index of size $|\mathsf{CSA}| + n \log D(1 + o(1))$ bits for the top-k frequent document retrieval problem with $O(t_s(p) + k \log^{1+\epsilon} D)$ query time, where $\epsilon >$ is any constant.*

5.2. Faster Compressed Index

This section describes how to improve the index to speed up the query. The idea is to choose a smaller grouping factor, thereby reducing the size of the candidate set. However, this will result in more marked nodes, so that explicit storage of precomputed answers (with $\log D$ bits per entry) at these marked nodes will lead to a non-succinct solution. Our key contribution is to show how these precomputed lists can be encoded in $O(\log\log n)$ bits per entry. Our main result is summarized as follows.

THEOREM 5.6. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $2|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits of space, such that whenever a pattern P (of p characters) and an integer k come as a query, the index returns those k documents with the highest $\mathtt{TF}(P, \cdot)$ values in decreasing order of $\mathtt{TF}(P, \cdot)$ in $O(t_s(p) + k \times t_{sa} \log k \log^\epsilon n)$ time; here, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with*

[12] In the boundary case where v_P^* does not exist, the candidate set is simply $\{d_{\mathsf{E}[j]} \mid j \in [sp, ep]\}$, whose size is at most $2g$ (refer to Lemma 5.1).

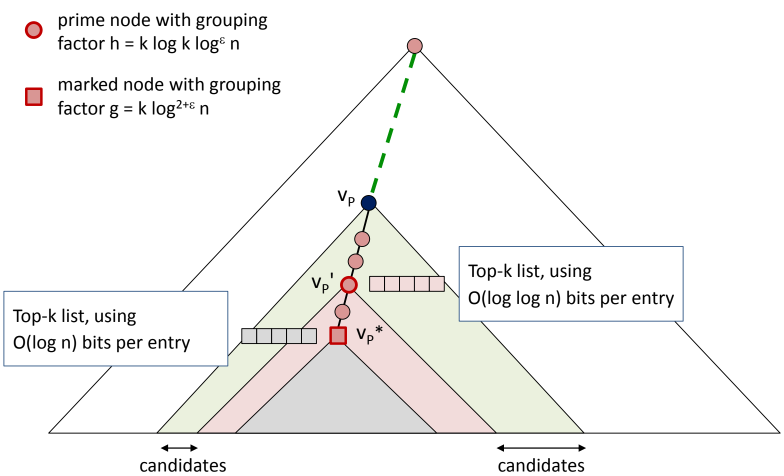

Fig. 5. Query answering with prime nodes and marked nodes

all the given documents in $\mathcal{D}$, or all the CSA*s of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using* CSA*, and $\epsilon >$ is any constant.*

5.2.1. Index for Top-k Queries for a Fixed k. Similar to the index in Section 5.1.1, we define an index for answering top-k frequent queries, where k is fixed in advance. The index consists of (i) a compressed suffix array CSA; (ii) the document array E (represented in $|\mathsf{CSA}^*| + D\log\frac{n}{D} + O(D) + o(n)$ bits, refer to Lemma 2.2) (iii) an auxiliary structure with respect to two grouping factors g and h, which is defined as follows. First, we mark the nodes in GST based on two grouping factors g and h, where $g = k\log^{2+\epsilon} n$ and $h = k\log k\log^{\epsilon} n$. Then, we maintain the corresponding GST_g and GST_h in a total of $O((n/g)\log g + (n/h)\log h) = o(n/k)$ bits (refer to Lemma 5.2).

In order to distinguish the marked nodes based of these two different grouping factors, we shall use the following terminology: If a node is marked as per the grouping factor g, we shall simply call it a marked node. Otherwise, if a node is marked as per the grouping factor h only, we shall call it as a *prime* node.

Query Answering. Let v_P be the locus node of the input pattern P in GST with v'_P and v^*_P, respectively, being its highest prime descendant and highest marked descendant (if they exist). Let $[sp, ep]$, $[sp', ep']$, and $[ep^*, ep^*]$, respectively, be the ranges of leaves within the subtree of v_P, v^*_P and v'_P. (See Figure 5 for an illustration.) Note that the following inequalities hold (refer to Lemma 5.1):

(1) $sp \le sp' \le sp^* \le ep^* \le ep' \le ep$;
(2) $sp' - sp < h$ and $ep - ep' < h$;
(3) $sp^* - sp' < g$ and $ep' - ep^* < g$.

Then,

$$\mathsf{top}(v'_P, k) \cup \left\{ d_{\mathsf{E}[j]} \mid j \in [sp, sp'-1] \cup [ep'+1, ep] \right\}$$

will be a candidate set, where we shall denote it by S^h_{cand}. The number of documents in $\mathsf{top}(v'_P, k)$ is at most k, and the number of the remaining documents in the candidate set is at most $2h$.

Once S^h_{cand} is given, it takes only an extra $O((h + k) \times t_{sa} \log\log n) = O(k \times t_{sa} \log k \log^\epsilon n)$ time for answering a top-k query (using Lemma 5.4). Note that the documents $d_{\mathsf{E}[j]}$ for $j \in [sp, sp'-1] \cup [ep'+1, ep]$ can be computed on the fly in $O(h \times t_{sa})$ time, which will not affect the overall time complexity. It remains to show how to obtain the list $\mathsf{top}(v'_P, k)$ efficiently. By the following lemma, the total query time can be bounded by $O(t_s(p) + k \times t_{sa} \log k \log^\epsilon n)$.

LEMMA 5.7. *We can encode* $\mathsf{top}(\cdot, k)$ *corresponding to every prime node in a total of* $O(n/(\log k \log^\epsilon n)) + o(n/\log n)$ *bits of space, such that* $\mathsf{top}(w', k)$ *of any prime node* w' *can be decoded in* $O(k \times t_{sa} \log\log n)$ *time.*

PROOF. We shall give an encoding of $\mathsf{top}(w', k)$ for each prime node w' that allows us to obtain a candidate set corresponding to w' as the locus. Then, by using Lemma 5.4, we can compute the desired $\mathsf{top}(w', k)$ based on the candidate set.

Let w^* be the highest marked descendent of w' (if it exists). Let $[L', R']$ and $[L^*, R^*]$, respectively, denote the range of leaves in the subtree of w' and w^*. A candidate set corresponding to w' as the locus (i.e., a superset of $\mathsf{top}(w', k)$) is given by

$$\mathsf{top}(w^*, k) \cup \left\{ d_{\mathsf{E}[j]} \mid j \in [L', L^* - 1] \cup [R^* + 1, R'] \right\}.$$

The set $\mathsf{top}(w^*, k)$ can be obtained in $O(k)$ time by maintaining $\mathsf{top}(\cdot, k)$ for each marked node explicitly, which requires a total of $O((n/g)k \log D) = o(n/\log n)$ bits. For the set $\{ d_{\mathsf{E}[j]} \mid j \in [L', L^* - 1] \cup [R^* + 1, R'] \}$ of the remaining documents, we select only the subset of its top k documents; then we see that this subset, when combined with $\mathsf{top}(w^*, k)$, still forms a candidate set corresponding to w' as the locus. In other words, even though we have $O(g)$ documents in this category, only at most k of them can be among $\mathsf{top}(w', k)$. Now, suppose that these k documents can be encoded in $O(k \log\log n)$ bits, while supporting decoding in $O(k \times t_{sa})$ time. Thus, the total space for all the encodings in all the prime nodes is $O(n/(\log k \log^\epsilon n))$ bits, and we can obtain the desired candidate set in a total of $O(k \times t_{sa})$ time. Consequently, $\mathsf{top}(w', k)$ can be computed in $O(k \times t_{sa} \log\log n)$ time using Lemma 5.4.

It remains to show how to encode the selected top k documents with the claimed performance. For each such document d_j, it can be associated with an integer $i \in [L', L^* - 1] \cup [R^* + 1, R']$ such that $\mathsf{E}[i] = j$. If we replace each such i by its relative position in $[L', L^* - 1] \cup [R^* + 1, R']$, this problem can be rephrased as the encoding of k distinct integers drawn from $[1, 2g]$. An encoding with $O(k \log\log n)$ bits of space and $O(k)$ decoding time can be achieved, by maintaining a bit vector $\mathsf{B}_{w',k}$ with constant-time *select* operations supported [Raman et al. 2007]; here, $\mathsf{B}_{w',k}[1..2g]$ is defined such that $\mathsf{B}_{w',k}[i] = 1$ if and only if i is an integer to be stored. Therefore $\mathsf{B}_{w',k}$ can be maintained in $k \log(2g/k) + O(k) = O(k \log\log n)$ bits of space, and the stored integers can be decoded by $select_{\mathsf{B}_{w',k}}(j)$ queries for $j = 1, 2, 3, \ldots, k$. Finally, given these integers (relative positions), the corresponding document can be retrieved in $O(t_{sa})$ time. This completes the proof. $\square$

Putting everything altogether, we have the following lemma.

LEMMA 5.8. *The auxiliary structure for a specific* k *takes* $O(n/(\log k \log^\epsilon n)) + o(n/\log n) + o(n/k)$ *bits of space. Given the suffix range* $[sp, ep]$ *of a pattern* P*, a top-*k *frequent document retrieval query can be answered in* $O(k \times t_{sa} \log k \log^\epsilon n)$ *time.*

5.2.2. Index for Top-k Queries for General k. To support top-k queries for general k, we maintain CSA, E, and (at most) $\log D$ auxiliary structures of Section 5.2.1 for $k = 1, 2, 4, 8, \ldots, D$, analogous to how we handle the general k case as in Section 5.1.2.

This requires a total of

$$\sum_{z=1,2,4,\ldots,D} \Big(O(n/(\log^{\epsilon} n \log z)) + o(n/\log n) + o(n/z) \Big) = o(n) \text{ bits}.$$

A top-k query can be answered by choosing $z = 2^{\lceil \log_2 k \rceil}$ and retrieving the top-z documents by querying on the auxiliary structure specific to z. Then, we select the k highest-scoring documents (using [Blum et al. 1973]) and report them in decreasing order of score. Combining with the fact that $k = \Theta(z)$, we obtain Theorem 5.6.

5.3. Extensions

Although we described our result in terms of term frequency as the scoring function, we can in fact extend it to some other scoring functions that are succinctly calculable. Unfortunately, we do not know if $\mathtt{TP}(\cdot,\cdot)$ is succinctly calculable. In contrast, $docrank(\cdot,\cdot)$ is not only succinctly calculable, but is trivial to compute. In fact, to support top-k queries with the $docrank$ metric, we do not even need the document array E using Lemma 2.2, but only the bit vector $\mathsf{B_E}$ and an array R of size $D \log D$ bits such that $\mathsf{R}[r]$ gives the relative $docrank$ of document d_r among the others; after the change, we can still compute $docrank$ of any document $d_{\mathsf{E}[i]}$ within the same time bound. This gives the following theorem.

THEOREM 5.9. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $|\mathsf{CSA}| + o(n) + D \log \frac{n}{D} + O(D) + D \log D$ bits of space, such that whenever a pattern P (of p characters) and an integer k come as a query, the index returns those k documents with the highest $docrank(\cdot)$ values in decreasing order of $docrank(\cdot)$ in $O(t_s(p) + k \times t_{sa} \log k \log^{\epsilon} n)$ time; here, $docrank(d_r)$ of a document d_r is a static importance score associated with d_r, $t_s(p)$ is the time to search for a pattern of length p with CSA, t_{sa} is the time to compute a suffix array entry with CSA, and $\epsilon >$ is any constant.*

See [Belazzougui and Navarro 2011] for a similar result, which appeared earlier but used different techniques.

6. MULTIPATTERN RETRIEVAL

In this section, we consider a generalization of the top-k document retrieval problem. Instead of a single pattern P, a query now consists of a set $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ of m patterns, and the relevance of a document d_r with respect to $\mathcal{P}$ depends only on the set of occurrences of all P_j in d_r. For simplicity, we first give an index for the simplest case, where $\mathcal{P}$ contains only two patterns P_1 and P_2 (of lengths p_1 and p_2, respectively). We choose $\mathtt{TF}(P_1, d_r)+\mathtt{TF}(P_2, d_r)$ as the score function $score(P_1, P_2, d_r)$ with an additional restriction that in order for a document d_r to be qualified as an answer, both P_1 and P_2 must occur in d_r. Therefore, $score(P_1, P_2, d_r)$ is given by $\mathtt{TF}(P_1, d_r)+\mathtt{TF}(P_2, d_r)$ if both $\mathtt{TF}(P_1, d_r)$,$\mathtt{TF}(P_2, d_r) > 0$, and is zero otherwise. We later show how our index can be modified to handle other score functions.

Our index is built from the succinct framework in Section 5. It consists of a suffix array SA (of size $O(n)$ words) in addition to GST (uncompressed, whose size is $O(n)$ words), a document array E, and auxiliary structures for answering for top-z queries for fixed $z = 1, 2, 4, \ldots, D$. The auxiliary structure for a specific z can be constructed with $g = \sqrt{nz \log D}$ as the grouping factor, where we identify the marked nodes in GST. Note that the marked node information can be maintained in $O(n/g)$ bits (refer to Lemma 5.2). Let $\mathsf{top}(u, v, k)$ denote the list of top-z documents with respect to the score function $score(path(u), path(v), \cdot)$. Then, corresponding to all pairs of marked nodes u^* and v^* in GST, we maintain the list $\mathsf{top}(u^*, v^*, z)$ explicitly. The space for each specific auxiliary structure is thus bounded by $O(n/g) + O((n/g) \times (n/g) \times z \log D) = O(n)$ bits, so that the total space for all the $O(\log D)$ auxiliary structures is bounded by $O(n \log D) = O(n \log n)$ bits, which is $O(n)$ words.

Query Answering. The algorithm to answer a query is analogous to that of our succinct index in Section 5. First, we find the locus nodes u_{P_1} and u_{P_2} of P_1 and P_2, respectively, in $O(p_1 + p_2)$ time using GST. Next, we set $z = 2^{\lceil \log k \rceil}$ (the minimum power of 2 greater than or equal to the input integer k). Then, using the auxiliary structure specific to this z (with grouping factor $g = \sqrt{nz \log D}$), we find the highest marked descendent of nodes, $u^*_{P_1}$ and $u^*_{P_2}$, of the locus nodes u_{P_1} and u_{P_2}, respectively. Afterwards, the set

$$\mathsf{top}(u^*_{P_1}, u^*_{P_2}, z) \cup \{d_{E[i]} \mid \ell_i \in \mathit{Leaf}(u_{P_1} \backslash u^*_{P_1}) \cup \mathit{Leaf}(u_{P_2} \backslash u^*_{P_2})\}$$

will be a candidate set S_{cand} that contains the desired top k answers.

Hence, by computing $\mathit{score}(P_1, P_2, d_r)$ of each document $d_r \in S_{\mathsf{cand}}$, and by choosing those k highest-scoring documents, we obtain the final output. Given the suffix ranges of P_1 and P_2, the score of any particular document can be computed in $O(\log \log n)$ time using E (refer to Lemma 2.2 and Lemma 2.4, as $\mathtt{TF}(P, d_r)$ can be evaluated by $\mathit{rank}_{\mathsf{E}}$, given the suffix range of P, and $t_{sa} = O(1)$ when SA is stored explicitly). As $|S_{\mathsf{cand}}| = O(g + z)$, the overall query time can be bounded by $O(p_1 + p_2 + \sqrt{nk \log D} \log \log n)$.

THEOREM 6.1. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$ words of space, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters, respectively) and an integer k come as a query, the index returns those k documents with the highest $\mathit{score}(P_1, P_2, \cdot)$ values in decreasing order of $\mathit{score}(P_1, P_2, \cdot)$ in $O(p_1 + p_2 + \sqrt{nk \log D} \log \log n)$ time; here, $\mathit{score}(P_1, P_2, d_r) = \mathtt{TF}(P_1, d_r) + \mathtt{TF}(P_2, d_r)$ if both $\mathtt{TF}(P_1, d_r)$ and $\mathtt{TF}(P_2, d_r)$ are greater than 0, and is zero otherwise.*

The space of the index described in the above theorem can be easily made succinct by the following modifications: (i) replace GST by its compressed version (space required is $|\mathsf{CSA}| + O(n)$ bits), (ii) replace E by its compressed version as described in Lemma 2.2, and (iii) build the auxiliary structure by choosing a higher grouping factor of $g = \sqrt{nz} \log D$. The overall space occupancy can thus be bounded by $2|\mathsf{CSA}^*| + O(n)$ bits, however the query time will be increased to $O(t_s(p_1) + t_s(p_2) + \sqrt{nk} \log D \times t_{sa} \log \log n)$. We summarize the result in the following theorem.

THEOREM 6.2. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $2|\mathsf{CSA}^*| + O(n)$ bits of space, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters, respectively) and an integer k come as a query, the index returns those k documents with the highest $\mathit{score}(P_1, P_2, \cdot)$ values in decreasing order of $\mathit{score}(P_1, P_2, \cdot)$ in $O(t_s(p_1) + t_s(p_2) + \sqrt{nk} \log D \times t_{sa} \log \log n)$ time; here, $\mathit{score}(P_1, P_2, d_r) = \mathtt{TF}(P_1, d_r) + \mathtt{TF}(P_2, d_r)$ if both $\mathtt{TF}(P_1, d_r)$ and $\mathtt{TF}(P_2, d_r)$ are greater than 0, and is zero otherwise, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA*

The above indexes can readily be adapted to handle the case with other score functions, with tradeoffs between the space for storing a data structure that can compute $\mathit{score}(\cdot, \cdot, \cdot)$ on the fly, and the per-document reporting time. In particular, the space remains $O(n)$ words for *linearly-calculable* score functions, where $\mathit{score}(\cdot, \cdot, d_r)$ can be computed on the fly by maintaining an $O(|d_r|)$-word index. For instance, when $docrank$ is the score function [13], the following result can be obtained.

THEOREM 6.3. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$ words of space or in $2|\mathsf{CSA}^*| + O(n)$ bits of space, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters, respectively) and an integer k come as a query, then among all those documents containing both P_1 and P_2, the index returns k documents with the highest $docrank(\cdot)$ values in decreasing order of $docrank(\cdot)$ in $O(p_1 + p_2 + \sqrt{nk \log D} \log\log D)$ time or in $O(t_s(p_1) + t_s(p_2) + \sqrt{nk} \log D \times t_{sa} \log\log n)$ time respectively; here, $docrank(d_r)$ of a document d_r is a static importance score associated with d_r, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA.*

Remark. The index of Theorem 6.3 can be used to solve the document listing problem for two patterns, where the task is to report all those documents containing both the input patterns P_1 and P_2. To do so, we simply set $k = D$ and then obtain the output of each query in $O(p_1 + p_2 + \sqrt{nD \log D} \log\log D)$ time (assuming our linear space structure). To reduce the last term in the query bound, we can issue the top-1 query, then top-2, then top-4, and so on until a top-q query returns the ndoc answers, where $\texttt{ndoc} < q$ denotes the number of documents in the desired output. Note that the patterns are searched only once here. Hence, the query time will be $O(p_1 + p_2 + \sqrt{n \log D} \log\log D + \sqrt{2n \log D} \log\log D + \sqrt{4n \log D} \log\log D + \cdots + \sqrt{n \times \texttt{ndoc} \log D} \log\log D) = O(p_1 + p_2 + \sqrt{(\texttt{ndoc}+1) \times n \log D} \log\log D)$. Using similar analysis the time for query on the succinct space structure can be bounded by $O(t_s(p_1) + t_s(p_2) + \sqrt{n(\texttt{ndoc}+1)} \log D \times t_{sa} \log\log n)$. Notice that our result clearly improves the earlier solution for this problem by Cohen and Porat [2010].

THEOREM 6.4. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$ words of space or in $2|\mathsf{CSA}^*| + O(n)$ bits of space, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters respectively) come as a query, the index returns all those* ndoc *documents containing both P_1 and P_2 in $O(p_1 + p_2 + \sqrt{(\texttt{ndoc}+1) \times n \log D} \log\log D)$ time or in $O(t_s(p_1) + t_s(p_2) + \sqrt{n(\texttt{ndoc}+1)} \log D \times t_{sa} \log\log n)$ time, here, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA.* □

In another problem introduced by Muthukrishnan [2002], the query asks for reporting all those documents with term proximity score at most an integer K, where P_1, P_2 and K and input parameters to the query, and term proximity score $\texttt{TP}_{two}(P_1, P_2, d_r)$ is defined as the distance between the closest occurrences of P_1 and P_2 within document d_r. If either P_1 or P_2, or both is absent in d_r, $\texttt{TP}_{two}(P_1, P_2, d_r)$ is infinity. An $O(n^{3/2} \log n)$-word index with $O(p_1 + p_2 + \sqrt{n} \log n + \texttt{output})$ query time was also proposed in [Muthukrishnan 2002], where output is the number of reported documents. Our framework can be used to derive the following linear space solution for the top-k version of this problem.

[13] For a document to be qualified as an output, both P_1 and P_2 must be present in it.

THEOREM 6.5. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$ words, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters, respectively) and an integer k come as a query, then among all those documents containing both P_1 and P_2, the index returns k documents with the* lowest *$\text{TP}_{two}(P_1, P_2, \cdot)$ values in increasing order of $\text{TP}_{two}(P_1, P_2, \cdot)$ in $O(p_1 + p_2 + \sqrt{nk \log D} \log^{\epsilon} n)$ time, where $\text{TP}_{two}(P_1, P_2, d_r)$ of a document d_r is the distance between the closest occurrences of P_1 and P_2 in d_r.*

PROOF. The index construction is exactly same as that of the result in Theorem 6.1 except that we use a different score function here. Additionally we maintain an orthogonal range successor/predecessor search structure over the suffix array SA. For this, we use the $O(n)$-word space structure by Navarro and Nekrich [2012a]. Therefore, given any suffix range $[L, R]$ and a position pos as input, the smallest (resp., the largest) $SA[i]$ value, where $i \in [L, R]$, succeeding (resp., preceding) pos can be computed in $O(\log^{\epsilon} n)$ time, where $\epsilon > 0$ is any small constant. Using similar analysis as that of Theorem 6.1, the total space occupancy can be bounded by $O(n)$ words.

We use the same terminologies as that of the proof of Theorem 6.1. The only difference in query algorithm, compared to that of Theorem 6.1 is the way we compute $score$ of documents corresponding to $\{d_{E[i]} \mid \ell_i \in \mathit{Leaf}(u_{P_1} \backslash u^*_{P_1}) \cup \mathit{Leaf}(u_{P_2} \backslash u^*_{P_2})\}$. Let $[sp^*_1, ep^*_1]$ and $[sp^*_2, ep^*_2]$ represents the range of leaves in the subtree of $u^*_{P_1}$ and $u^*_{P_2}$ respectively. Assume that a document $d_{E[i]}$ is a top-k candidate, and that its candidacy is due to an occurrence of P_1 (resp., P_2) at position $SA[i]$, then the corresponding $\text{TP}_{two}(\cdot, \cdot, \cdot)$ value can be computed in $O(\log^{\epsilon} n)$ time by retrieving the successor and predecessor of $SA[i]$ with $[ep^*_2, ep^*_2]$ (resp., $[sp^*_1, ep^*_1]$) as the input range, so that we can find the distance from the closest occurrence of P_2 (resp., P_1) from it. Therefore, using similar analysis the query time can be bounded by $O(p_1 + p_2 + \sqrt{nk \log D} \log^{\epsilon} n)$. □

6.1. Handling $m > 2$ Patterns

All the above results can be extended to handle the case where the query consists of a set of $m > 2$ patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$, with p_i denoting the length of P_i. Precisely, for a specific 2-power z, we choose a grouping factor $g = n^{1-1/m}(z \log D)^{1/m}$, identify the marked nodes in GST, and maintain top-z documents corresponding to each combination of $(u^*_1, u^*_2, \ldots, u^*_m)$, where u^*_i for any i denotes a marked node in GST. Over all $\log D$ choices of z, the total space can be bounded by $O((n/g)^m z \log D) \times \log D = O(n \log n)$ bits, or equivalently by $O(n)$ words. Note that m is fixed at index construction time. Then, whenever a query comes, we can quickly find a candidate set S_{cand} of $O(n^{1-1/m}(k \log D)^{1/m})$ documents, compute the score of a document (if needed) in S_{cand} in $O(m \log \log D)$ time, and finally output the k highest-scoring ones among them. Putting everything together, we can obtain an $O(n)$-word index with query time $O(\sum_{i=1}^{m} p_i + mn^{1-1/m}(k \log D)^{1/m} \log \log D)$.

7. CONCLUSION

In this paper, we presented space-efficient frameworks for desiging indexes for top-k string retrieval problems. Our frameworks are based on annotating suffix tree (or compressed suffix tree) with additional information. In particular, we maintain a suffix tree of the concatenated documents, superimpose the local suffix trees of the individual documents in terms of "pointers", and solve geometric range problems on these pointers. Our compact framework is based on encoding these pointers in smaller amount of bits, while the compressed framework further samples these pointers as they pass through some specially chosen nodes. These frameworks are fairly general and have also been shown to be practical [Patil et al. 2011; Culpepper et al. 2012; Navarro et al.

2011; Belazzougui et al. 2013]. Even though efficient solutions are already available for the central problem, there are still many interesting variations and open questions one could ask about. We conclude with some of them as listed below:

(1) The current I/O-optimal index requires $O(n \log^* n)$-word space [Shah et al. 2013]. It is interesting to see if we can bring down this space to linear (i.e., using $O(n)$ words) without sacrificing the optimality in the I/O bound. Designing indexes in the cache-oblivious model [Frigo et al. 1999] is another future research direction.
(2) The current space-optimal index for top-k frequent document retrieval is proposed by Navarro and Thankachan [2013]), whose per-document reporting time is $O(t_{sa} \log^2 k \log^\epsilon n)$. In contrast, the per-document reporting time of our compressed index (Theorem 5.6) is faster by a factor of $\log k$, but our index takes twice the size of text. An interesting problem is to design a space-optimal index, while keeping the query time the same as (or better than) that of ours (which is currently the fastest in compressed space).
(3) The *document selection* problem — where we want to obtain the kth highest-scoring document (or its score) corresponding to the query — may have useful IR applications in practice.
(4) Even though many succinct indexes have been proposed for top-k queries for frequency or PageRank-based score functions, it is still unknown if a succinct index with $O((p + k) \log^{O(1)} n)$ query time can be designed if the score function is term proximity (as it is not known to be succinctly calculable). Designing such an index even for special cases (say, with long query patterns only, or when we allow approximate score, etc.) or deriving lower bounds are interesting research directions. We remark that it is possible to design such an index for the special case where the input pattern is of length at least $\log^2 n$, by combining our succinct framework with known techniques [Hon et al. 2012; Chien et al. 2013].
(5) Approximate pattern matching (i.e., allowing bounded errors and don't cares) is another active research area [Cole et al. 2004]. Adding this aspect to document retrieval leads to many new problems. The following is one such problem: Report all those documents in which the edit (or Hamming) distance between one of its substrings and P is at most τ, where $\tau \geq 1$ is an input parameter.
(6) Indexing a highly repetitive or a highly similar document collection is an active line of research. In recent work, Gagie et al. [2013] propose an efficient document retrieval index suitable for a repetitive collection. An open problem is to extend the result for handling top-k queries.

REFERENCES

AGGARWAL, A. AND VITTER, J. S. 1988. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM 31,* 9, 1116–1127.

BELAZZOUGUI, D. AND NAVARRO, G. 2011. Improved Compressed Indexes for Full-Text Document Retrieval. In *Proceedings of International Symposium on String Processing and Information Retrieval*. 386–397.

BELAZZOUGUI, D., NAVARRO, G., AND VALENZUELA, D. 2013. Improved Compressed Indexes for Full-Text Document Retrieval. *Journal of Discrete Algorithms 18*, 3–13.

BENDER, M. A. AND FARACH-COLTON, M. 2000. The LCA Problem Revisited. In *Proceedings of Latin American Symposium on Theoretical Informatics*. 88–94.

BLUM, M., FLOYD, R. W., PRATT, V. R., RIVEST, R. L., AND TARJAN, R. E. 1973. Time Bounds for Selection. *Journal of Computer and System Sciences 7,* 4, 448–461.

BROWN, M. R. AND TARJAN, R. E. 1979. A Fast Merging Algorithms. *Journal of the ACM 26,* 2, 211–226.

CHAZELLE, B. 1988. A Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM Journal on Computing 17,* 3, 427–462.

CHIEN, Y. F., HON, W. K., SHAH, R., THANKACHAN, S. V., AND VITTER, J. S. 2013. Geometric BWT: Compressed Text Indexing via Sparse Suffixes and Range Searching. *Algorithmica*. To appear.

CLARK, D. R. 1996. Compact Pat Trees. Ph.D. thesis, University of Waterloo.

COHEN, H. AND PORAT, E. 2010. Fast Set Intersection and Two-Patterns Matching. *Theoretical Computer Science 411,* 40–42, 3795–3800.

COLE, R., GOTTLIEB, L.-A., AND LEWENSTEIN, M. 2004. Dictionary Matching and Indexing with Errors and Don't Cares. In *Proceedings of Symposium on Theory of Computing*. 91–100.

CULPEPPER, J. S., NAVARRO, G., PUGLISI, S. J., AND TURPIN, A. 2010. Top-k Ranked Document Search in General Text Databases. In *Proceedings of European Symposium on Algorithms*. 194–205.

CULPEPPER, J. S., PETRI, M., AND SCHOLER, F. 2012. Efficient in-memory top-k document retrieval. In *Proceedings of SIGIR Conference on Research and Development in Information Retrieval*. 225–234.

FARACH, M. 1997. Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of Symposium on Foundations of Computer Science*. 137–143.

FERRAGINA, P. AND MANZINI, G. 2005. Indexing Compressed Text. *Journal of the ACM 52,* 4, 552–581.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2007. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms 3,* 2.

FISCHER, J., GAGIE, T., KOPELOWITZ, T., LEWENSTEIN, M., MÄKINEN, V., SALMELA, L., AND VÄLIMÄKI, N. 2012. Forbidden Patterns. In *Proceedings of Latin American Symposium on Theoretical Informatics*. 327–337.

FISCHER, J. AND HEUN, V. 2007. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *Proceedings of Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. 459–470.

FISCHER, J. AND HEUN, V. 2011. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM Journal on Computing 40,* 2, 465–492.

FREDERICKSON, G. N. 1993. An Optimal Algorithm for Selection in a Min-Heap. *Information and Computation 104,* 2, 197–214.

FREDMAN, M. L., KOMLÓS, J., AND SZEMERÉDI, E. 1984. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM 31,* 3, 538–544.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-Oblivious Algorithms. In *Proceedings of Symposium on Foundations of Computer Science*. 285–298.

GAGIE, T., KARHU, K., NAVARRO, G., PUGLISI, S. J., AND SIRÉN, J. 2013. Document Listing on Repetitive Collections. In *Proceedings of Symposium on Combinatorial Pattern Matching*. 107–119.

GAGIE, T., NAVARRO, G., AND PUGLISI, S. J. 2010. Colored Range Queries and Document Retrieval. In *Proceedings of International Symposium on String Processing and Information Retrieval*. 67–81.

GAGIE, T., NAVARRO, G., AND PUGLISI, S. J. 2012. New Algorithms on Wavelet Trees and Applications to Information Retrieval. *Theoretical Computer Science 426*, 25–41.

GAGIE, T., PUGLISI, S. J., AND TURPIN, A. 2009. Range Quantile Queries: Another Virtue of Wavelet Trees. In *Proceedings of International Symposium on String Processing and Information Retrieval*. 1–6.

GOLYNSKI, A., MUNRO, J. I., AND RAO, S. S. 2006. Rank/Select Operations on Large Alphabets: A Tool for Text Indexing. In *Proceedings of Symposium on Discrete Algorithms*. 368–373.

GROSSI, R., GUPTA, A., AND VITTER, J. S. 2003. High-Order Entropy-Compressed Text Indexes. In *Proceedings of Symposium on Discrete Algorithms*. 841–850.

GROSSI, R. AND VITTER, J. S. 2005. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing 35,* 2, 378–407.

HON, W.-K., PATIL, M., SHAH, R., THANKACHAN, S. V., AND VITTER, J. S. 2013. Indexes for document retrieval with relevance. In *Space-Efficient Data Structures, Streams, and Algorithms*. 351–362.

HON, W. K., PATIL, M., SHAH, R., AND WU, S. B. 2010. Efficient Index for Retrieving Top-k Most Frequent Documents. *Journal of Discrete Algorithms 8,* 4, 402–417.

HON, W. K., SHAH, R., AND THANKACHAN, S. V. 2012. Towards an Optimal Space-and-Query-Time Index for Top-k Document Retrieval. In *Proceedings of Symposium on Combinatorial Pattern Matching*. 173–184.

HON, W. K., SHAH, R., THANKACHAN, S. V., AND VITTER, J. S. 2010. String Retrieval for Multi-pattern Queries. In *Proceedings of International Symposium on String Processing and Information Retrieval*. 55–66.

HON, W. K., SHAH, R., THANKACHAN, S. V., AND VITTER, J. S. 2012. On Position Restricted Substring Searching in Succinct Space. *Journal of Discrete Algorithms 17*, 109–114.

HON, W. K., SHAH, R., AND VITTER, J. S. 2009. Space-Efficient Framework for Top-k String Retrieval Problems. In *Proceedings of Symposium on Foundations of Computer Science*. 713–722.

HON, W. K., THANKACHAN, S. V., SHAH, R., AND VITTER, J. S. 2013. Faster Compressed Top-k Document Retrieval. In *Proceedings of Data Compression Conference*. 341–350.

HSU, B.-J. P. AND OTTAVIANO, G. 2013. Space-Efficient Data Structures for Top-k Completion. In *Proceedings of International Conference on World Wide Web*. 583–594.

KARPINSKI, M. AND NEKRICH, Y. 2011. Top-K Color Queries for Document Retrieval. In *Proceedings of Symposium on Discrete Algorithms*. 401–411.

KNUTH, D. E., MORRIS, J. H., AND PRATT, V. B. 1977. Fast Pattern Matching in Strings. *SIAM Journal on Computing 6,* 2, 323–350.

KONOW, R. AND NAVARRO, G. 2013. Faster Compact Top-k Document Retrieval. In *Proceedings of Data Compression Conference*. 351–360.

MANBER, U. AND MYERS, G. 1993. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing 22,* 5, 935–948.

MATIAS, Y., MUTHUKRISHNAN, S., SAHINALP, S. C., AND ZIV, J. 1998. Augmenting Suffix Trees, with Applications. In *Proceedings of European Symposium on Algorithms*. 67–78.

MCCREIGHT, E. M. 1976. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM 23,* 2, 262–272.

MUNRO, J. I., RAMAN, V., AND RAO, S. S. 2001. Space Efficient Suffix Trees. *Journal of Algorithms 39,* 2, 205–222.

MUTHUKRISHNAN, S. 2002. Efficient Algorithms for Document Retrieval Problems. In *Proceedings of Symposium on Discrete Algorithms*. 657–666.

NAVARRO, G. 2013. Spaces, Trees and Colors: The Algorithmic Landscape of Document Retrieval on Sequences. *CoRR abs/1304.6023*.

NAVARRO, G. AND MÄKINEN, V. 2007. Compressed Full-Text Indexes. *ACM Computing Surveys 39,* 1.

NAVARRO, G. AND NEKRICH, Y. 2012a. Sorted range reporting. In *SWAT*. 271–282.

NAVARRO, G. AND NEKRICH, Y. 2012b. Top-k Document Retrieval in Optimal Time and Linear Space. In *Proceedings of Symposium on Discrete Algorithms*. 1066–1077.

NAVARRO, G., PUGLISI, S. J., AND VALENZUELA, D. 2011. Practical Compressed Document Retrieval. In *Proceedings of Symposium on Experimental Algorithms*. 193–205.

NAVARRO, G. AND THANKACHAN, S. V. 2013. Faster Top-k Document Retrieval in Optimal Space. *Proceedings of International Symposium on String Processing and Information Retrieval*. To appear.

PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. 1999. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab. November.

PAGH, R. 2001. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing 31,* 2, 353–363.

PATIL, M., THANKACHAN, S. V., SHAH, R., HON, W. K., VITTER, J. S., AND CHANDRASEKARAN, S. 2011. Inverted Indexes for Phrases and Strings. In *Proceedings of SIGIR Conference on Research and Development in Information Retrieval*. 555–564.

PATRASCU, M. 2008. Succincter. In *Proceedings of Symposium on Foundations of Computer Science*. 305–313.

RAMAN, R., RAMAN, V., AND RAO, S. S. 2007. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. *ACM Transactions on Algorithms 3,* 4.

SADAKANE, K. 2007a. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 589–607.

SADAKANE, K. 2007b. Succinct Data Structures for Flexible Text Retrieval Systems. *Journal of Discrete Algorithms 5,* 1, 12–22.

SADAKANE, K. AND NAVARRO, G. 2010. Fully-Functional Succinct Trees. In *Proceedings of Symposium on Discrete Algorithms*. 134–149.

SHAH, R. AND FARACH-COLTON, M. 2002. Undiscretized Dynamic Programming: Faster Algorithms for Facility Location and Related Problems on Trees. In *Proceedings of Symposium on Discrete Algorithms*. 108–115.

SHAH, R., SHENG, C., THANKACHAN, S. V., AND VITTER, J. S. 2013. Top-k Document Retrieval in External Memory. In *Proceedings of European Symposium on Algorithms*. 803–814.

TSUR, D. 2013. Top-k Document Retrieval in Optimal Space. *Information Processing Letters 113,* 12, 440–443.

VÄLIMÄKI, N. AND MÄKINEN, V. 2007. Space-Efficient Algorithms for Document Retrieval. In *Proceedings of Symposium on Combinatorial Pattern Matching*. 205–215.

VITTER, J. S. 2008. Algorithms and Data Structures for External Memory. *Foundations and Trends® in Theoretical Computer Science 2,* 4, 305–474.

WEINER, P. 1973. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*. 1–11.

WILLARD, D. E. 1983. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Information Processing Letters 17,* 2, 81–84.

WITTEN, I., MOFFAT, A., AND BELL, T. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA, USA.