The Quantile Index – Succinct Self-Index for Top-k Document Retrieval

Niklas Baumstark[1], Simon Gog[2], Tobias Heuer[3], and Julian Labeit[4]

1 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
niklas.baumstark@student.kit.edu
2 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
simon.gog@kit.edu
3 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
tobias.heuer@student.kit.edu
4 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
julian.labeit@student.kit.edu

Abstract

One of the central problems in information retrieval is that of finding the k documents in a large text collection that best match a query given by a user. A recent result of Navarro & Nekrich (SODA 2012) showed that single term and phrase queries of length m can be solved in optimal $O(m + k)$ time using a linear word sized index. While a verbatim implementation of the index would be at least an order of magnitude larger than the original collection, various authors incrementally improved the index to a point where the space requirement is currently within a factor of 1.5 to 2.0 of the text size for standard collections.

In this paper, we propose a new time/space trade-off for different top-k indexes. This is achieved by sampling only a quantile of the postings in the original inverted file or suffix array-based index. For those queries that cannot be answered using the sampled version of the index we show how to compute the query results on the fly efficiently. As an example, we apply our method to the top-k framework by Navarro & Nekrich. Under probabilistic assumptions that hold for most standard texts, and for a standard scoring function called term frequency, our index can be represented with only sublinearly many bits plus the space needed for a compressed suffix array of the text, while maintaining poly-logarithmic query times. We evaluate our solution on real-world datasets and compare its practical space usage and performance against state-of-the-art implementations. Our experiments show that our index compresses below the size of the original text. To our knowledge it is the first suffix array-based text index that is able to break this bound in practice even for non-repetitive collections, while still maintaining reasonable query times of under half a millisecond on average for top-10 queries.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Text Indexing, Succinct Data Structures, Top-k Document Retrieval

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.15

1 Introduction

Given a large collection of text documents, it remains an important problem in computer science to pre-process them in a way that afterwards user queries about the documents in the collection can be answered efficiently. The general setting for *top-k retrieval* that we concern ourselves with is the following: Given a *collection* of documents $\mathcal{D} = \{d_1, d_2, \dots, d_N\}$ where each document d_i is a string over some alphabet Σ, a *pattern* string $p \in \Sigma^*$, a *scoring function* $\omega : \mathcal{D} \times \Sigma^* \to \mathbb{R}$ and a parameter $k \in \{1, 2, \dots N\}$, find the k documents d_i that

© Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 15; pp. 15:1–15:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

contain p and maximize $\omega(d_i, p)$. If $\mathcal{D}$ and ω are given upfront, we might want to build an *index* data structure that is independent of the pattern p and allows us to process a series of patterns much more efficiently, without having to scan the entire document collection each time. This is surely not feasible for every possible scoring function ω, hence we restrict ourselves to an interesting subset of scoring functions. An example of such a function is *term frequency*, which simply counts how often the pattern appears in the document.

More complex scoring functions, such as Okapi BM25, incorporate more features – namely document frequency and document length – and often lead to better results in practice. In standard information retrieval nomenclature, the type of query described above is called a *phrase query*. Depending on the data structures used, phrase queries can be much harder to solve than *single term queries*, which consist of a single token (such as a word in a natural text). If suffix arrays are used as the basic building block of an index, these two types of queries can be treated the same because no parsing of the text into tokens is required.

There are two main approaches for solving the top-k retrieval problem. *Inverted indexes* are used almost exclusively in practice by real-world search engines such as Apache Lucene [20]. Inverted indexes store all occurrences (or *postings*) of a term in a document in *posting lists*. Decades of research have yielded very compact and fast indexes using this technique. Nevertheless, there are some inherent limitations of the inverted index approach: In order to support phrase queries efficiently, additional information needs to be stored alongside the postings, and heuristics are used to ensure that tokens appear together and in the correct order in the resulting documents.

In 2010, Culpepper et al. [3] presented the first implementation of Hon et al.'s top-k index for phrase queries that is based on *suffix arrays* [9]. Since then, various authors have improved the space efficiency of suffix array-based indexes significantly. The current state of the art can produce indexes that are within a factor of 1.5 to 2 of the text size, while supporting arbitrary phrase queries efficiently [11]. However, space usage is still a clear advantage of inverted indexes, which are typically much smaller than the original text.

We contribute a new technique that can be applied to both inverted indexes and suffix array-based indexes to decrease space usage by trading it for query time. We achieve this by storing only a fraction of the posting lists, and provide an algorithm that can reconstruct the rest of the result list on the fly, in the case where this is necessary because many results are requested – i.e. k is sufficiently large. We implemented our idea on top of an existing top-k index that is based on the framework by Navarro & Nekrich [16]. In particular, we took the implementation from [11] and improved its space usage by a factor of 2 to 2.5 using our quantile sampling method. Our experimental evaluation shows that query times are dependent on the specific query, but are within a factor of 10-20 of the unmodified index in the most unfavourable cases and comparable on average.

2 Related Work

One of the basic tasks closely related to top-k retrieval is *document listing* – enumerating all documents in which a pattern occurs. In 1998, Matias et al. augmented suffix trees in order to solve the document listing problem efficiently [12]. Muthukrishnan introduced an optimal time index for document listing in 2002 [13]. The solution uses a range minimum query data structure to enumerate all distinct document IDs in a lexicographic range. Sadakane proposed a succinct version of the index [18]. As the underlying data structure Sadakane uses a *compressed suffix array* (CSA). For a comprehensive survey on CSAs and basic succinct data structures used in text indexes we refer to [15]. In total Sadakane's solution needs

$|CSA| + 4n + o(n)$ bits to perform document listing, where $|CSA|$ is the size of a compressed suffix array. Since then numerous indexes based on CSAs have been proposed supporting all kinds of different operations. In the following we will focus on the top-k retrieval problem.

In 2009, Hon et al. introduced a general top-k retrieval framework and showed how to solve the problem using an index taking up only a linear number of words [9]. In their work they introduce the notion of weighted arrows in the generalized suffix tree. Queries are answered by enumerating the heaviest arrows pointing out of subtrees of the tree. Subsequently, Navarro & Nekrich showed how to further improve this framework to achieve optimal $O(m + k)$ query time, where m is the pattern length, and reduced its space consumption [16]. This is done by representing arrows as 2-d weighted grid points and using a three-sided prioritized orthogonal range search data structure to answer top-k queries. In 2013, Tsur published the first top-k document framework using optimal $|CSA| + o(n)$ bits of space with poly-logarithmic query times [21]. The query time was reduced by Thankachan & Navarro to $O(k \log^2 k \log^\epsilon n)$ suffix array accesses for any constant $\epsilon > 0$ [17]. For a comprehensive overview of the different problems and solutions concerning document retrieval please refer to the survey put together by Navarro [14].

In recent years there have been numerous practical implementations of suffix array-based document retrieval frameworks. In 2013, Konow & Navarro showed that with a frequency sampling of the suffix tree nodes they can actually implement the Navarro–Nekrich scheme using 3 to 4 times the original text size for non-repetitive collections while achieving top-k query times well below a millisecond [10]. In 2014, Brisaboa et al. introduced the K^2-treap to solve weighted top-k range queries faster and more space-efficiently in practice [2]. In the following year, Gog & Navarro simplified the implementation by Konow & Navarro by introducing a new mapping of suffix array ranges to coordinate ranges of the grid [8]. Their implementation uses 2.5 to 3 times the input size while maintaining comparable query times. Recently, Labeit & Gog proposed a technique to encode the document IDs of the grid points more compactly [11]. With this technique they achieve index sizes of 1.5 to 2 times the input size at the cost of higher query times. [7] gives a comprehensive overview of the techniques developed in this line of research.

Additionally, some of the prior art specializes on repetitive string collections. Gagie et al. show how document listing, top-k retrieval and document counting can be solved while exploiting the properties of repetitive collections [4]. They introduce the *interleaved LCP* array and *precomputed document lists*. Both concepts might also be applicable to top-k retrieval on non-repetitive collections.

3 Quantile Filtering

3.1 Preliminaries

For a collection $\mathcal{D}$ and scoring function ω we define $\mathcal{R}_p = (d_1, d_2, ...)$ to be the result list containing all documents in which p occurs in descending score order. In the literature $\mathcal{R}_p$ are often referred to as impact-ordered posting lists of term p. Consider the example collection $\mathcal{D} = \{\mathtt{d_1 : ATATT, d_2 : TTATA, d_3 : AATT, d_4 : TTA}\}$, pattern $p = \mathtt{TA}$ and term frequency as the scoring function. Then $\mathcal{R}_{\mathtt{TA}} = (d_2, d_1, d_4)$ as TA appears twice in d_2, once in d_1 and once in d_4. The top-k retrieval problem can be defined as computing the first k entries of $\mathcal{R}_p$, which we denote by the function $\texttt{top-}k(p, \mathcal{R})$.

Additionally we define $\mathcal{O}cc_p = ((d_1, pos_1), (d_2, pos_2), ...)$ to be the list of all occurrences of the pattern p. For our example collection $\mathcal{O}cc_{\mathtt{TA}} = ((d_1, 1), (d_2, 1), (d_2, 3), (d_4, 1))$. If we concatenate all documents to one string with a separator character we obtain $\mathcal{D}^* =$

Algorithm 1: The generic query algorithm.

```
Input: Search pattern p, integer k > 0
Output: Top-k documents containing pattern p, sorted by scoring function
1 if k · q ≤ count(p, D*) then
2     return top-k(p, R^q)
3 else
4     Occ_p ← locate(p, D*)
5     return top-k-on-the-fly(Occ_p)
```

ATATT#TTATA#AATT#TTA and can then compute $|\mathcal{O}cc_p|$ by counting the occurrences of p in $\mathcal{D}^*$. We can generate $\mathcal{O}cc_p$ by first computing all locations of p in $\mathcal{D}^*$ and then mapping the locations to pairs of document ID and location within the document. This mapping can be performed using a compressed bit vector. We call a pattern p *right-maximal* if it has two occurrences in $\mathcal{D}^*$ succeeded by different characters. The inner nodes of the suffix tree of $\mathcal{D}^*$ corresponds to all the right-maximal patterns. Finally, we denote the set of all $\mathcal{R}_p$ for all right-maximal patterns as $\mathcal{R}$.

3.2 Basic Framework

In the following we propose a sampling technique of the sorted posting lists $\mathcal{R}_p$. The sampling allows us to use a standard pattern matching index, such as a compressed suffix array (CSA), to reduce the number of postings which need to be represented by our top-k retrieval index. The pattern matching index only needs to support the basic operations `locate`(p) and `count`(p), which compute the set $\mathcal{O}cc_p$ and its cardinality, respectively.

Let $q \in \mathbb{N}$ be the *quantile parameter*. Then $\mathcal{R}_p^q$ is defined as the list containing the upper q-th quantile of $\mathcal{R}_p$, i.e. the $\lfloor \frac{|occ_p|}{q} \rfloor$ highest ranked elements of $\mathcal{R}_p$. We denote the set of all $\mathcal{R}_p^q$ lists for all right-maximal substrings p of $\mathcal{D}^*$ as $\mathcal{R}^q$. Our top-k retrieval index represents only the $\mathcal{R}^q$ lists, instead of the $\mathcal{R}$ lists. When solving top-k queries we first compute $|\mathcal{O}cc_p|$ using a pattern matching index. If $k \cdot q \leq |\mathcal{O}cc_p|$ the query can be answered using the top-k index representing $\mathcal{R}^q$. Otherwise the query is answered by computing the result list from $\mathcal{O}cc_p$ on the fly. Note that in the latter case the cardinality of $\mathcal{O}cc_p$ is bounded by $q \cdot k$, which ensures the efficiency of the framework. Algorithm 1 gives the pseudocode for the query algorithm.

With the proposed framework we can solve regular top-k queries on $\mathcal{R}$ using a top-k query on $\mathcal{R}^q$ plus additional operations on a pattern matching index and some on-the-fly computations. We will analyze the asymptotic behaviour of our index under the assumption that the input texts are randomly drawn from a source satisfying the Szpankowski A2 model [19]. This is reasonable to assume for most texts occurring in practice. The A2 model was first used in the context of document listing by Gagie et al. [5]. We use it to bound the height of the suffix tree with high probability by $\Theta(\log n)$.

By choosing different values for q we get different time/space trade-offs between the size of $\mathcal{R}^q$ and the time needed for `locate` and the on-the-fly computation. The following theorem characterizes the relation between q and the size of $\mathcal{R}^q$.

▶ **Theorem 1.** *For a document collection generated from a source satisfying the Szpankowski A2 model, $|\mathcal{R}^q| = O(\frac{n \log n}{q})$ holds with high probability, where $|\mathcal{R}^q| = \sum_p(|\mathcal{R}_p^q|)$ for all right maximal substrings p of $\mathcal{D}^*$.*

Proof. Each right-maximal substring p of $\mathcal{D}^*$ can be identified by a unique node v in the suffix tree of $\mathcal{D}^*$. We can bound the number of distinct documents in which p occurs by the number of nodes in the subtree rooted at v. Thus the total result list size of p can be bounded by $|\mathcal{R}_p| \leq |subtree(v)|$. Under Szpankowski's A2 model the generalized suffix tree has height $h = \Theta(\log n)$ with high probability (w.h.p). Hence the sum over all such subtrees is bounded by $\sum_v |subtree(v)| \in O(n \log n)$ and consequently $|\mathcal{R}| \in O(n \log n)$ w.h.p. By construction $\mathcal{R}^q$ has a factor of q less elements than the lists in $\mathcal{R}$ so in total we get $|\mathcal{R}^q| = O(\frac{n \log n}{q})$ w.h.p. ◀

3.3 Succinct Self-Index

We apply our method to a state-of-the-art implementation of the Navarro–Nekrich scheme for top-k retrieval. Our choice is based on the fact that their framework already includes a CSA for pattern matching and that recent implementations only need $O(\log \log n)$ index bits per input character[1] to represent $\mathcal{R}^q$. We first give a short overview of the Navarro–Nekrich scheme and then we apply Theorem 1 to show that our index indeed is succinct. For a more in-depth discussion of the ideas underlying the Navarro–Nekrich scheme and the various improvements since its inception please refer to [7].

The foundation of the top-k retrieval framework was laid out by Hon et al. [9]. The key idea is to answer top-k queries by inserting the edges of the suffix trees of the individual input documents into the suffix tree of $\mathcal{D}^*$. The edges are associated with the corresponding document ID and are directed towards the root. Additionally the score $\omega(d, p)$ is computed, where d is the corresponding document ID and p is the pattern represented by the node that the arrow points to. This score is used as the weight of the arrow. Hon et al. observed that top-k queries for a pattern p can be solved by enumerating the k heaviest arrows originating inside and pointing outside of the subtree representing p. Navarro & Nekrich showed that each arrow can be represented as a weighted 2-d grid point [16]. The x coordinates of a grid point is its position in a certain in-order traversal of the arrows by their origin. The y coordinate of a grid point is the tree depth of the tree node that it points to.

The set of all such grid points is called G. Top-k queries are solved by answering three-sided prioritized orthogonal range queries on G. The x range of the query is chosen so that it contains all arrows starting in the subtree corresponding to the query pattern p. The y range is chosen so that only arrows are reported that point out of the subtree corresponding to the query pattern p. Additionally, to increase the practical performance of the framework, *singletons* are handled in a separate document listing data structure. Singletons are arrows that start at suffix tree leaves and thus account for at least half of all the arrows.

► **Theorem 2.** *For a document collection generated from a source satisfying the Szpankowski A2 model, applying quantile filtering to the Navarro–Nekrich scheme yields a succinct top-k self-index with high probability. More precisely we obtain an index using $|CSA| + o(n)$ bits of space and $O(k \cdot polylog(n, m))$ time to answer a top-k query for a pattern of length m. Here $|CSA|$ is the space used by the compressed suffix array, which is assumed to support standard operations in time $O(polylog(n, m))$.*

Proof. We choose $q \in \Theta(\log n (\log \log n)^2)$ so that using Theorem 1 we get $|\mathcal{R}^q| \in o(n/\log \log n)$. We only keep those grid points from G that represent results in $\mathcal{R}^q$. We call

[1] To achieve $O(\log \log n)$ index bits per character the input collection needs to satisfy the Szpankowski A2 model and the values of the scoring function have to be encodable in $O(n \log \log n)$ bits.

this the quantile-filtered grid G_q which consequently contains at most $o(n/\log\log n)$ grid points. To store G_q in an augmented wavelet tree we need $O(\log\log n)$ bits per grid point. Thus the total space to store G_q is $o(n)$ bits. We apply coordinate compression on the x range of the grid G_q. Then the mapping of suffix tree leaves to x coordinates can be stored explicitly using $o(n)$ bits. Finally, we use the data structure from [11] to store the document IDs. Using Elias-Fano encoding and a similar analysis as in [11] the IDs can be represented indirectly in $O(\log\log n)$ bits per grid point. So all the document IDs can be represented in $o(n)$ bits. The overall index needs $|CSA| + o(n)$ bits of space with high probability.

The query time is dominated in the worst case by the $k \cdot q$ accesses to the CSA. They are needed to compute $\mathcal{O}cc_p$ in line 4 of Algorith 1. Depending on T_{CSA}, the access time of the CSA, we get an overall query complexity of $O(T_{CSA}(n,m) \cdot k \cdot \log n (\log\log n)^2)$. ◀

This succinct self-index for top-k document retrieval index is called the *quantile index*. For other grid representations (or scoring functions) with higher space usage we can simply adapt q accordingly. It can always be chosen such that the index space is bounded by $|CSA| + o(n)$ bits, potentially at the cost of a higher worst-case query time. In practice q can be chosen empirically such that the grid component of the index requires little space compared to the CSA component.

4 Index Details

Our practical implementation of the quantile index consist of five basic components, listed below. The list does not include the suffix tree of the collection $\mathcal{D}^*$ because it is needed only temporarily and can be discarded after the construction is finished.

- G_q, the 2-d weighted range search data structure over the filtered Navarro–Nekrich grid points. Each grid point corresponds to exactly one arrow and has an additional coordinate x representing a specific order of the arrows which is described in Section 4.1.
- a compressed suffix array (CSA) of the concatenation of all documents in the input collection. It is used to determine the lexicographical suffix array interval $[l, r]$ for a given query, and to implement the fallback case where $q \cdot k > r - l + 1$ and thus we cannot consult G_q. Different sampling rates s lead to different time/space trade-offs for this component.
- a compressed bit vector B marking the positions in the concatenated input $\mathcal{D}^*$ where a new document begins.
- a data structure DOC which stores the document IDs for each point in G_q. In Section 4.3 we describe how to incorporate the ideas from [11] to decrease the space usage of this component. We have implemented variants of our index with and without this optimization.
- a mapping bit vector H_q, which is used to map a query suffix array interval $[l, r]$ to a range of grid coordinates $[x_l, x_r]$.

To clarify how the quantile index works, Algorithm 2 shows how a query pattern p is processed, using term frequency as an example scoring function. In the fallback case, a linear-time selection algorithm such as the one of Blum et al. [1] is used to select the top-k results, after computing the full list of candidate results by scanning the compressed suffix array. Note that our index can support all the scoring functions that the original Navarro–Nekrich framework supports.

Algorithm 2: The query algorithm for the quantile index.

```
   Input: Search pattern p, integer k > 0
   Output: Top-k documents containing pattern p, sorted by term frequency
 1 [l, r] ← search(CSA, p)
 2 if k · q ≤ r − l + 1 then
 3     [x_l, x_r] ← mapSAInterval(H, l, r)
 4     points ← topKRangeSearch(G_q, k, [x_l, x_r] × [0, |p|))
 5     result ← empty list
 6     for p = (x_p, y_p) ∈ points do
 7         push(result, DOC [x_p])
 8 else
 9     freq ← new hash map with default value 0
10     for i = s to e do
11         docid ← rank(B, CSA[i])
12         freq[docid]++
13     result ← partialSort(freq, k)  // select and sort only the k elements with biggest weight
14 output result
```

4.1 Singletons and the Vector H

Each arrow a in the original Navarro–Nekrich framework is associated with a node v_a in the suffix tree of $\mathcal{D}^*$. Exactly one arrow is associated with each leaf of the suffix tree. Such arrows are called singletons because they correspond to postings with frequency one. The top-k index by Konow & Navarro handles singletons using a separate 1-dimensional range minimum data structure for space reasons [10]. One can view our approach as a generalization of this idea: Since for a subtree S of the suffix we only store the most heavy $\lfloor |S|/q \rfloor$ arrows pointing out of it, we only keep arrows that point out of some subtree of size $|S| \geq q$. This is unlikely to be the case for singleton arrows, hence most of them will be eliminated in practice. This allows us to handle singletons the same way as other arrows, without implementing a special case.

To transform the arrows into grid points and apply the Navarro–Nekrich technique, we order them using a specific depth-first traversal of the suffix tree and use the index of an arrow in that sequence as an additional coordinate for the arrow. Algorithm 3 illustrates the traversal. We obtain a sequence $(a_0, a_1, \ldots a_{M-1})$ of arrows, each represented by a tuple $a_i = (y_i, d_i, w_i)$ where y_i is the target depth of the arrow, d_i is the document associated with it and w_i is its weight. We define $x_i = i$ as the x coordinate of the arrow. The tuple (x_i, y_i, w_i, d_i) is a grid point.

Now we compute a bit vector H (ordered by x coordinate), which marks singleton grid points with a one and all other grid points with a zero. We observe that the one bits in this vector correspond exactly to the leaves in the suffix tree and hence to the entries of the suffix array. We can use a rank data structure on top of H (which computes the number of ones in any given prefix of H) to implement the `mapSAInterval` function used in Algorithm 2.

4.2 Construction

Our implementation of the quantile filtering works as follows: The suffix tree is traversed in depth-first order. B-trees are used to represent for each node v the set of grid points that correspond to arrows pointing out of the subtree rooted at v, ordered by decreasing weight. We can compute this set for a node v by merging the sets of its children. If we reuse the

Algorithm 3: Depth-first in-order traversal of the Navarro–Nekrich arrows.

```
1 Function OrderArrows(v)
    Input: Suffix tree rooted at v.
    Output: An ordered list of arrows originating in the subtree v
2   if v is not a leaf then
3       l ← leftmost child of v
4       OrderArrows(l)
5   output arrows originating at v
6   if v is not a leaf then
7       for children r of v except l do
8           OrderArrows(r)
```

Table 1 Collection statistics: number of characters n, number of documents N, average document length, alphabet size σ, and total size in MiB assuming one byte per character.

Collection	n	N	n/N	σ	$\lvert\mathcal{D}^*\rvert$ in MiB
ENWIKIBIG	8,945,231,276	3,903,703	2,291	211	8,535
ENWIKISML	68,210,334	4,390	15,538	206	65
REVISIONS	419,437,305	20,433	20,527	240	400
SOURCES	244,587,464	29,993	8,154	232	233

B-tree of the largest child subtree and merge the others into it, we only need to perform $O(n \log n)$ B-tree insertions in total. For each node v, we enumerate and mark the top q^{-1} fraction of grid points while eliminating points with $y \geq depth(v)$. The total runtime of this traversal is $O(n \log^2 n)$, under the assumption from above that the suffix tree height is bounded by $O(\log n)$ with high probability.

The result of this construction step is a bit vector QFILTER of size M that has a one bit set for all the grid points that appear in at least one top quantile. We can now build the 2-d weighted range search data structure G_q over only the marked grid points. K^2-treaps form a good compromise between space usage and practical performance, even though they do not provide meaningful worst-case guarantees [2]. We store QFILTER and a compressed version of H. Both are used in conjunction to map a suffix array interval to an interval of x coordinates during query time.

4.3 Using Offset Encoding to Compress DOC

The DOC component, which stores the document IDs for all grid points in G_q can be further compressed using the offset encoding technique from [11]. The resulting data structure is a bit vector which can be used in conjunction with the suffix array CSA and bit vector B to decode the document ID for a given grid point. This query time penalty, since suffix array accesses are slower than most other basic operations used in our query algorithm. Section 5 contains a detailed analysis of the compromise between space usage and query time using this approach.

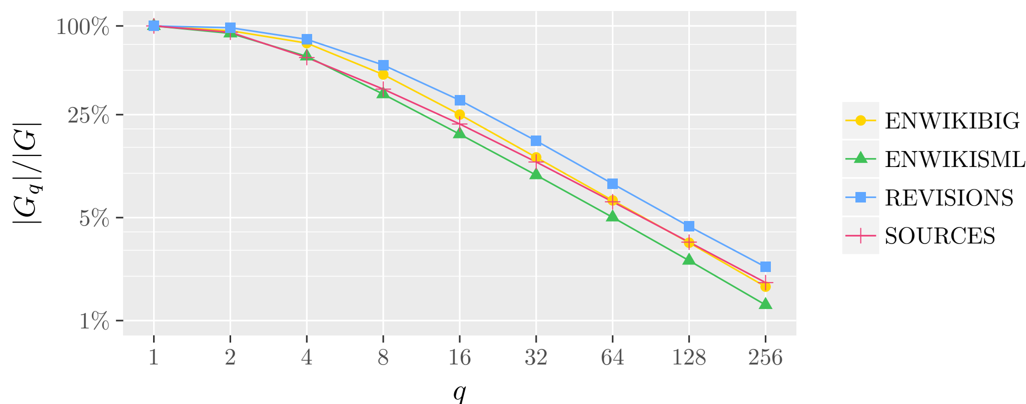

Figure 1 The number of grid points after quantile filtering with varying parameter q, relative to the total number of grid points in the Navarro–Nekrich implementation.

5 Experiments

To thoroughly evaluate the performance of our new top-k index and to compare it against the state of the art, we ran experiments using different input collections. The experiments were executed on a single machine equipped with four Intel Xeon E5-4640 processors, with a combined number of 32 cores and 64 hyper-threads. All experiments were executed in a single thread. The main memory is organized in four banks of 128 GiB each. All programs were compiled using GCC 5.2.0 with optimizations turned on. We used a variety of text collections for our experiments, including two dumps of the English Wikipedia of different sizes (ENWIKISML and ENWIKIBIG), all the revisions of 100 Finnish Wikipedia articles (one revision per document, REVISIONS) and a concatenation of files from the Linux and GCC source tree (SOURCES). Table 1 gives an overview of the basic statistics for each collection. We are comparing four different implementations: NN, the implementation of the Navarro–Nekrich framework by Gog & Navarro [8]; NNL, which adds the offset encoding technique from [11] to NN; Q, our implementation of the quantile index; QL, our quantile index with offset encoding.

For all implementations, the 2-d grid is represented by a K^2-treap as previous work has shown that K^2-treaps use less space than wavelet trees and provide similar query times in practice [7]. We were not able to include an implementation of [4] for time reasons. Their index is designed to perform very well on repetitive collections like REVISIONS, but for ENWIKIBIG its index size and query times are comparable to the NN implementation which is called SURF in their experiments.

The source code used for our experiments can be found at `https://github.com/kit-susi/quantile-index` and includes a script to download the input collections. The basic data structures used by our code are included from in the Succinct Data Structure Library [6].

5.1 Choosing the Index Parameters

Figure 1 shows how the number of stored grid points changes for different q. The experiment indicates that by using $q = 64$ less than ten percent of the grid points need to be stored for all the tested collections. Additionally we can observe that on these test collections the number of grid points $|G_q|$ is actually smaller than suggested by the bound $O(n \cdot \frac{\log n}{q})$. Even for $q \leq \log n$ we see substantial improvements.

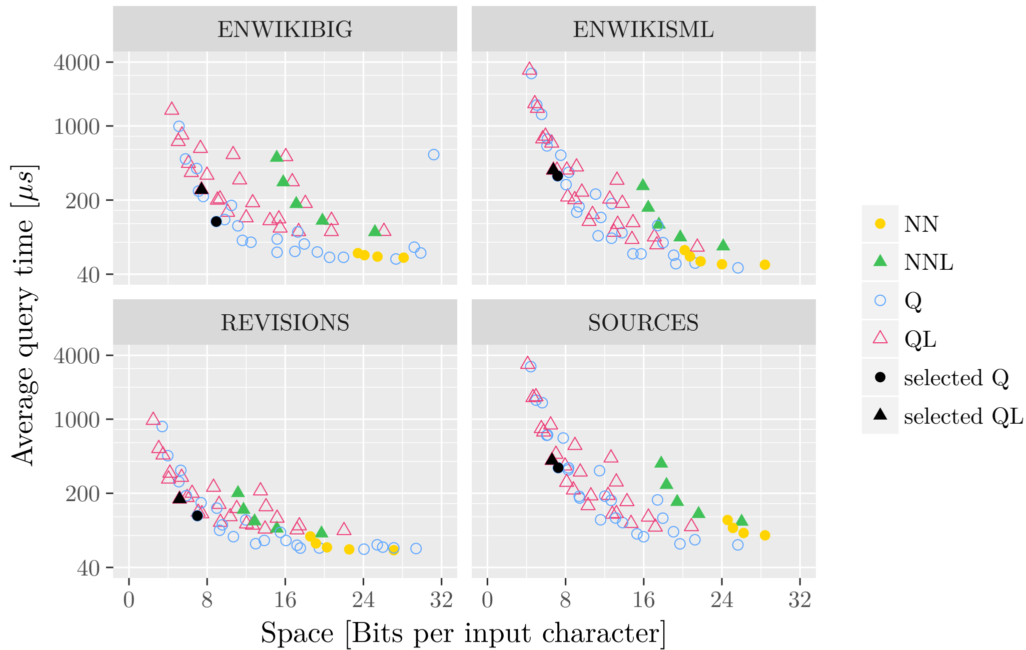

Figure 2 Different time/space tradeoffs resulting from varying quantile parameter $q \in \{8, 16, 32, 64, 128\}$ and CSA sampling parameter $s \in \{4, 8, 16, 32, 64\}$. The x-axis shows the total index size as ratio to the original collection size and the y-axis shows the average top-10 query times for random sampled queries of length $m = 5$ in microseconds. We additional mark the indexes with $(s, q) = (16, 64)$ as they were selected for the other experiments.

Figure 2 shows the different time/space trade-offs in our implementation of the index, resulting from varying CSA sampling parameter $s \in \{4, 8, 16, 32, 64\}$ and quantile parameter $q \in \{8, 16, 32, 64, 128\}$. For further experiments we use $s = 16$ and $q = 64$ as it presents an appealing compromise between query time and index size. Figure 3 shows the total size of the different index data structures with these parameters and highlights how much space is needed by the individual components. As an example, for ENWIKIBIG, our largest collection, the size of the quantile indexes Q and QL is about 65 and 50 percent smaller than the size of NN and NNL indexes, respectively. We note that the offset encoding technique from [11] is not as effective for the quantile index as it is for the NN implementation because the number of grid points is much smaller to begin with in relation to the size of the whole index.

5.2 Query Times

To get a first impression of the query performance of our index, we generated 100,000 random queries by uniformly drawing 5-grams from our biggest input collection. We ran the query set against our implementations, using different values of k, while measuring the individual query times. The results are shown in Figure 4: The average query times per reported document range from 3 to 50 microseconds across all the different implementations and for the document collection ENWIKIBIG. As expected from the results of [11], the variants using offset encoding (NNL and QL) have about a factor of 2 to 3 slower median query times than their counterparts. The plot also clearly shows a large variance in query time for the quantile indexes Q and QL. This is due to the two completely different code paths taken by Algorithm 2 depending on the size of the suffix array (SA) interval corresponding to the query pattern.

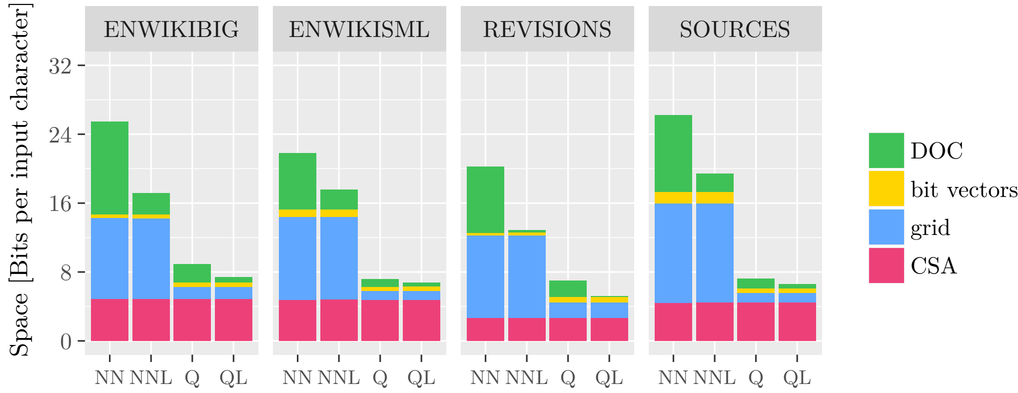

Figure 3 Total index size and size of individual components for $q = 64$ and $s = 16$ in bits per input character. *grid* includes the grid data structure and the RMQ data structure used by NN/NNL to handle singletons postings. *bit vectors* include the H vector and the QFILTER vector in the case of Q/QL.

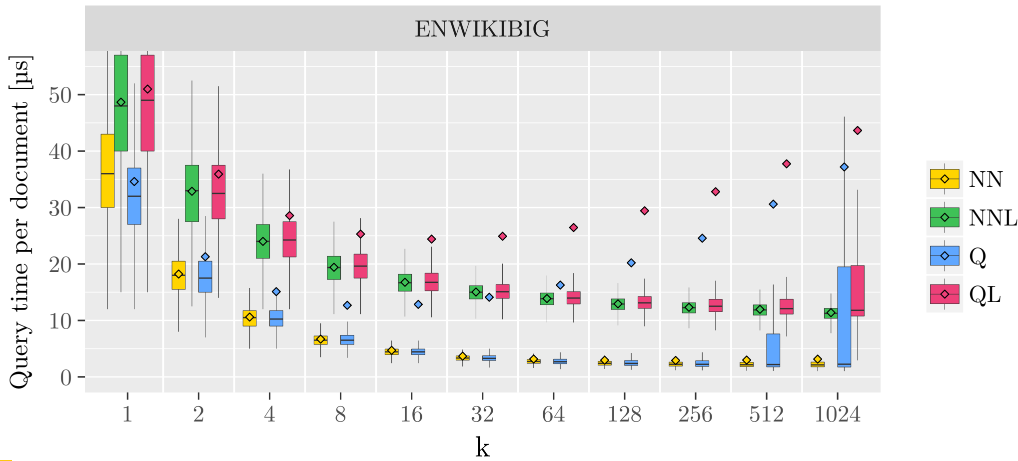

Figure 4 Timings to compute top-k results per reported document, for randomly drawn five-character queries. Both median and mean query times are highlighted, outliers are omitted for readability reasons.

Figure 5 captures the relation between SA interval size and query performance more explicitly. The selection of queries was done in a similar manner to the simple experiment described above, but this time the pattern length was chosen randomly between 3 and 10 for each query so as to vary the sizes of the SA intervals corresponding to the patterns. We requested $k = 10$ results for each invocation of the query algorithm. The plot shows the query times grouped by SA interval size. The separation between the two cases of Algorithm 2 is clearly visible: For the Q and QL implementations, we expect queries with an SA interval of size smaller than $q \cdot k = 640$ to trigger the fallback case where the suffix array range is scanned, which has a running time that is quasi-linear in the SA interval size. Other queries can make use of the more efficient grid search algorithm based on 2-d range queries, which is independent of the SA interval size. This behaviour is clearly exhibited in all four different collections. For ENWIKIBIG, the median query time for the most unfavourable SA interval sizes is still within a factor of 20 of the grid search algorithm in the case of Q. The plot also shows that the offset encoding only affects the grid search case of the query algorithm,

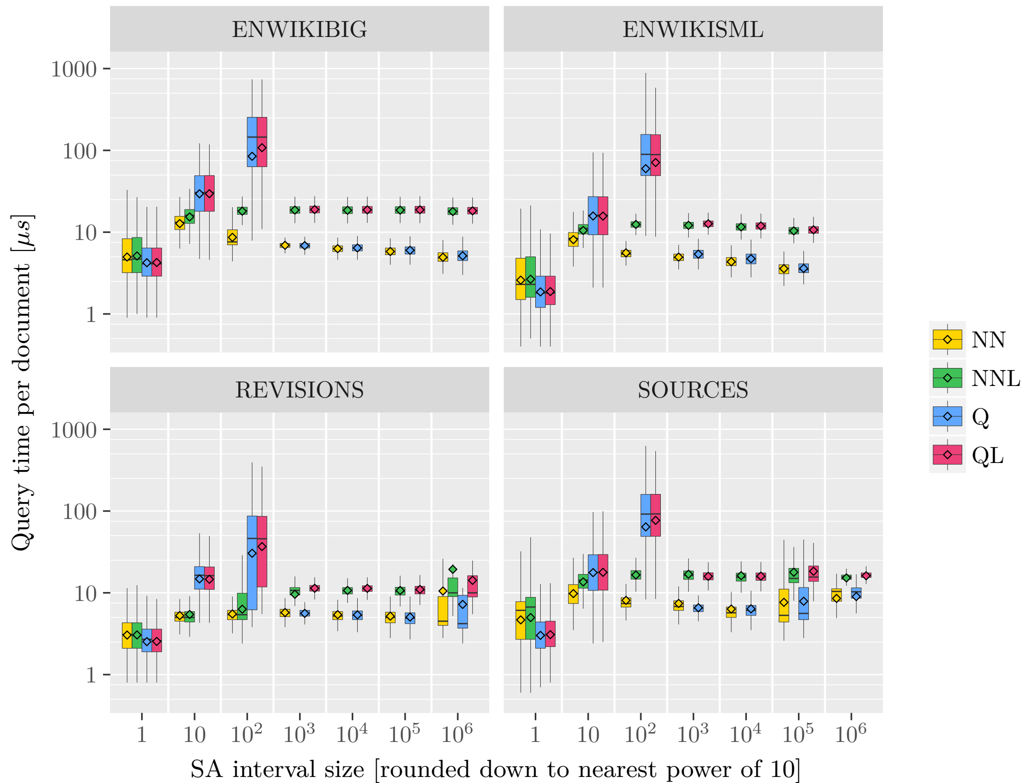

Figure 5 Timings per reported document for $k = 10$, using queries with different suffix array interval sizes. Both median and mean query times are highlighted, outliers are omitted for readability reasons.

not the fallback case. This is because in the fallback case, the suffix array entries for all occurrences of the pattern are known and the B bit vector can be consulted directly to decode the document IDs.

6 Conclusion and Future Work

We have introduced a general technique to reduce the space usage of existing top-k retrieval frameworks. This is achieved by storing only a fraction of the postings in the original index. The sampling is constructed in such a way that the total number of occurrences for query patterns which cannot be answered using the reduced index is bounded by a multiple of k. Hence it is feasible in this case to scan all occurrences and compute the top k results on the fly.

We show an exemplary application of this technique to a state-of-the-art suffix array-based self-index and evaluate its practical performance. Our experiments show that we obtain new time/space trade-offs for the problem at hand. In particular for real-world inputs the compressed suffix array is now the largest component of our proposed index. For non-repetitive texts this is to our knowledge the first index of this kind that is smaller than the original text but can still be used to reconstruct the text fully and support top-k queries with query times typically in the sub-millisecond range. An additional advantage of our framework is that singletons do not need to be handled in a separate data structure and thus it is easier to support more complex scoring functions. In the future we plan to implement

different scoring function used in real-world applications such as Okapi BM25. Currently the implementation of our construction algorithm is suboptimal, making it impractical to evaluate the algorithms on even larger collections. This is partly due to inefficiencies in the implementation and also an incomplete theoretical foundation. For example our construction algorithm works by filtering a pre-computed set of grid points. It should be possible to directly construct the quantile grid from the generalized suffix tree and achieve shorter construction times. Furthermore, the on-the-fly computation of query results for large enough k is currently implemented naively by scanning the suffix array. More elaborate techniques such as the sampled document array could be used to speed up this path of the query algorithm [17].

In this work, we only apply our general framework to one specific implementation. In future work we plan to evaluate whether the framework is also useful when applied to other existing data structures. For example, we are looking to combine a standard inverted index, where postings are ordered by document ID, with multiple small quantile inverted indexes, where postings are ordered by different custom scoring functions. With this construction it may be possible to increase the generality of inverted indexes – which are widely deployed in practice – even further.

References

1. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, pages 448–461, 1973.
2. N. Brisaboa, G. de Bernardo, R. Konow, and G. Navarro. K^2-Treaps: Range Top-k Queries in Compact Space. In *Proc. SPIRE*, pages 215–226, 2014.
3. J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. ESA*, pages 194–205, 2010.
4. T. Gagie, A. Hartikainen, K. Karhu, J. Kärkkäinen, G. Navarro, S. J. Puglisi, and J. Sirén. Document retrieval on repetitive collections. *Information Retrieval*, 2017. To appear.
5. T. Gagie, K. Karhu, G. Navarro, S. J. Puglisi, and J. Sirén. Document listing on repetitive collections. In *Proc. CPM*, pages 107–119, 2013.
6. S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337, 2014.
7. S. Gog, R. Konow, and G. Navarro. Practical compact indexes for top-k document retrieval. *J. Experimental Alg.*, 22(1):article 1.2, 2017.
8. S. Gog and G. Navarro. Improved single-term top-k document retrieval. In *Proc. ALENEX*, pages 24–32, 2015.
9. W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722, 2009.
10. R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. DCC*, pages 5–17, 2013.
11. J. Labeit and S. Gog. Elias-fano meets single-term top-k document retrieval. In *Proc. ALENEX*, 2017.
12. Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv. Augmenting suffix trees, with applications. In *Proc. ESA*, pages 67–78. Springer, 1998.
13. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666, 2002.
14. G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comp. Surv.*, 46(4):article 52, 2014.
15. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1), 2007. Article 2.

16 G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. SODA*, pages 1066–1078, 2012.

17 G. Navarro and S. Thankachan. Faster top-k document retrieval in optimal space. In *Proc. SPIRE*, pages 255–262, 2013.

18 K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Alg.*, 5(1):12–22, 2007.

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

20 The Apache Software Foundation. Apache Lucene. `http://lucene.apache.org/`.

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