To Index or not to Index: Time-Space Trade-offs for Positional Ranking Functions in Search Engines[☆]

Diego Arroyuelo[a,b,*], Senén González[d], Mauricio Marin[e], Mauricio Oyarzún[c], Torsten Suel[f], Luis Valenzuela[a]

[a] *Dept. of Informatics, Universidad Técnica Federico Santa María, Chile*
[b] *Instituto Milenio Fundamento de los Datos, Chile.*
[c] *Universidad Arturo Prat, Iquique, Chile*
[d] *Software Competence Center Hagenberg GmbH, Austria.*
[e] *CeBiB, Dept. Ing. Informática, Universidad de Santiago, Santiago, Chile*
[f] *CSE Department, Polytechnic Institute of NYU. Brooklyn, NY, 11201*

Abstract

Positional ranking functions, widely used in web search engines and related search systems, improve result quality by exploiting the positions of the query terms within documents. However, it is well known that positional indexes demand large amounts of extra space, typically about three times the space of a basic nonpositional index. Textual data, on the other hand, is needed to produce text snippets. In this paper, we study time-space trade-offs for search engines with positional ranking functions and text snippet generation. We consider both index-based and non-index based alternatives for positional data. We aim to answer the question of whether positional data should be indexed, and how.

We show that there is a wide range of practical time-space trade-offs. Moreover, we show that using about 1.30 times the space of positional data, we can store everything needed for efficient query processing, with a minor increase in query time. This yields considerable space savings and outperforms, both in space and time, recent alternatives from literature. We also propose several efficient compressed text representations for snippet generation, which are able to use about half of the space of current state-of-the-art alternatives with little impact in query processing time.

Keywords: Positional indexing, text compression, index compression, wavelet trees, snippet generation.

1. Introduction

Web search has become an important part of day-to-day life, affecting the way in which people think and recall things [50]. *Search engines* are one of the most important tools that give access to the huge amount of information stored in text format. Roughly speaking, the success of a search engine mostly depends on its efficiency and the quality of its ranking function. To achieve efficient processing of queries, search engines use highly optimized data structures, including inverted indexes [8, 15, 33, 58]. State-of-the-art ranking functions, on the other hand, combine simple term-based ranking schemes such as BM25 [15], link-based methods such as Pagerank [11] or Hits [31], and up to several hundred other features derived from documents and search query logs.

[☆] A preliminary version of this paper appeared in Proc. of the 35th International ACM SIGIR conference on research and development in Information Retrieval (SIGIR'12).

[*] Corresponding author. Address: Av. Vicuña Mackenna 3939, Santiago, Chile. Phone: +56 2 2303 7215.
Email addresses: darroyue@inf.utfsm.cl (Diego Arroyuelo), sgonzale@dcc.uchile.cl (Senén González), mauricio.marin@usach.cl (Mauricio Marin), moyarzunsil@unap.cl (Mauricio Oyarzún), torsten.suel@nyu.edu (Torsten Suel), li.valenzuelaa@gmail.com (Luis Valenzuela)

A lot of recent work has focused on *positional ranking functions* [46, 35, 51, 36, 47], also known as *proximity ranking functions* [8, 15, 33], which aims to improve the result quality by considering the positions of the query terms in the documents: documents where query terms are closer to each other within a document might be ranked higher, as term proximity could indicate that the document is highly relevant for the query. To support positional ranking, a search engine must be able to access term position data. This is commonly supported by building an index for all term positions within the documents, called a *positional index*. The goal is to obtain an index that is efficient in terms of index size and access time.

As shown by Rasolofo and Savoy [46] positional ranking can be carried out in two phases:

1. A simple term-based ranking scheme (such as BM25) defined over a Boolean filter is used to determine a set of high-scoring documents, say, the top-200 documents.
2. In the second phase, the term positions are used to rerank these documents by refining their score values. (Additional phases may be used to do further reranking according to hundreds of additional features [56], yet this is orthogonal to our work.)

Once the final set of top-scoring documents has been determined (say, the top 10), it is necessary to generate appropriate *text snippets*, typically text surrounding the term occurrences, to be shown in the result page (i.e., *query-dependent* snippets [10]). This requires access to the text of the documents, which in modern information retrieval systems is given by *direct indexes*. A direct index [11, 43] is a mapping from document identifiers to the corresponding (usually compressed) content of the documents.

It is well known [59, 25] that storing position data requires a considerable amount of space, compared to the space used by a basic non-positional inverted index. The most space-efficient approaches for storing position data appear to be the ones in Yan et al. [59]. They propose alternative ways to compress positional inverted indexes, which typically achieve about 3 to 5 times the space of a non-positional inverted index. Furthermore, storing the document collection (using a direct index) for snippet generation requires additional space [25].

However, it is not clear whether using positional inverted indexes [59] is the best one could do for storing position data. Notice that position data can be obtained (at query time) by looking for the query terms within the documents — a simple linear-time scan of the document suffices. Since textual data can be compressed better than positions [25], this could decrease the space usage to support positional ranking functions. However, the question is how this impacts query performance, as positions must now be computed at query time. We investigate this issue in this paper.

This paper focuses on alternative approaches to performing the above two-step document ranking process [46], as well as the query snippet-generation phase. The aim is to optimize both space usage and query processing time. An important observation is that, at search time, we only need to access the position data of a limited number of promising documents, say up to a few hundred documents. This is because of the BM25 filter applied at the first step of the ranking process. This limited access pattern differs from that of document identifiers and term frequencies, which are accessed more frequently, making access speed much more important in these cases. For position data, on the other hand, we could consider somewhat slower but smarter alternative representations without affecting the efficiency at query time.

In this paper, we consider not storing the position data (i.e, the positional index) at all. Instead, we compute positions *on the fly* from a compressed representation of the text collection. We will study two alternative approaches for compressing the text collection:

- Compressed data structures, e.g. *wavelet trees* [28, 24, 39], which come from the combinatorial pattern matching community.
- Compressed document representations that support fast extraction of arbitrary documents.

It has been shown that, compared to positional indexes, web-scale texts can often be compressed in much less space [25]. The rationale is that positional indexes can only be compressed to the zero-order empirical entropy of the text [40], whereas the text can be compressed to higher orders using standard text compressors. More importantly, as it was mentioned before, these representations can be used for both positional reranking and snippet generation. One concern is how these alternatives impact query processing speed, and we will

thus study the resulting trade-offs between running time and space requirement. Although this is a simple scheme that is likely used in practice by some search engines and related applications, the research literature focus mostly on positional indexes (e.g., [30]). Thus, our study focus on comparing alternatives to compress the document collection, while supporting efficient access to the text content of the documents.

Therefore, *to index or not to index position data*, that is the research question that we hope to answer in this paper. To our knowledge, the different approaches for implementing positional ranking functions have not been rigorously compared so far. Previous work has considered using direct indexes for phrase queries [44] and for feature extraction [7]. However, these works can be considered complementary to our work, as we study the most efficient ways to implement a direct index in order to compute positional information. There is another very recent approach by Procházka and Holub [45], where traditional positional inverted indexes are improved by introducing a so-called *Positional Inverted Self-index*. Extrapolating their experimental results, several of our proposals should obtain better space/time performance. Finally, He and Suel [29] study positional inverted indexes for versioned document collections. In such a collection, each document can have several versions, each introducing slight changes on the previous one. We do not consider versioned document collections in this paper. However, we think that the approaches studied in this paper can be successfully applied to versioned document collections.

Our main conclusion is that we can store all the information needed for query processing (i.e., document identifiers, term frequencies, position data, and text) using space close to that of state-of-the-art positional indexes (which only store position data and thus cannot be used for snippet creation), with only a minor increase in query processing time. In this way, we provide new alternatives for practical compression of position and text data, outperforming recent approaches of Shan et al. [48].

Following current practice in search engines [25, 19], we assume a scenario where there is enough space to maintain index data structures completely in main memory, in compressed form. In this scenario, large text collections are usually partitioned over a number of nodes in a cluster, such that each partition fits into the memory of its node. This paper focuses on how to organize data within each partition, as also assumed in previous work such as [25, 19].

2. Background and Related Work

2.1. Document Collection and Text Representation

We define a document collection as follows:

Definition 2.1. Let $\mathcal{D} = \{D_1, \ldots, D_N\}$ be a document collection of size N, where each document is represented as a sequence $D_i[1..n_i]$ of n_i *terms* (or *words*) from a vocabulary $\Sigma = \{1, \ldots, V\}$, of V terms.

Thus, every term in Σ is represented by an integer from $\{1, \ldots, V\}$, and the documents are just sequences of integers.

Definition 2.2. Given a document D_i, we identify it with the unique document identifier (`docID`) i.

Definition 2.3. Given a term $t \in \Sigma$ and a document $D_i \in \mathcal{D}$, the *in-document positions* of t in D_i is the set $\{j, \quad D_i[j] = t\}$.

Throughout this paper, we assume that all term separators (like spaces, '`,`', '`;`', '`.`', etc.) have been removed from the text. Also, we map all terms to lower case. This allows us to search for specific terms more easily (as all variants of a word are transformed to lower case) and efficiently (as eliminating separators shortens the text, and we will carry out sequential scans on the compressed text when searching for a specific term). However, most search engines show snippets including the original separators, punctuation, and case. To be able to retrieve the original text (with separators and the original case) one could use the *presentation layer* introduced by Fariña et al. [22, Section 4]. This also supports removing stopwords and the use of stemming, among other vocabulary techniques. This extra layer requires extra space on top of that of the compressed text, as well as extra time in order to obtain the original text. However, this scheme could be used with all the alternatives that we consider in this paper. Thus, we disregard the possible overhead introduced by a presentation layer and focus only on the low-level details of compression (though keeping in mind that the original text can still be retrieved).

2.2. Inverted Indexes

The efficiency of query processing in search engines relies on *inverted indexes* [8, 15, 58, 33]. An inverted index is a data structure that stores a set of *inverted lists* $I_1, \ldots, I_V$, each I_t corresponding to a vocabulary term $t \in \Sigma$. These inverted lists are accessed through a *vocabulary table*. Each list I_t stores a *posting* for each document containing the term $t \in \Sigma$. We use $|I_t|$ to denote the number of postings in inverted list I_t. Given a user query, the inverted lists of the query terms are used to answer it.

Usually, a posting in an inverted list consists of the `docID` (the resulting index is called `docID` index), the term frequency (`docID`/frequency index), and the in-document positions of the term (*positional inverted index*). In real systems, the `docID`s, term frequencies, and in-document positions are often stored separately. We use this approach in this paper, as this allows us to use different compression techniques in each case, as well as more efficient processing at query time.

2.3. Inverted Index Compression

Due to the large volume of data on the web and its rapid and continuous growth, inverted indexes usually have high space requirements. Hence, index compression is fundamental for the efficiency of web search engines [15, 58, 19, 33, 9]. In this paper we will focus on gap-encoded inverted indexes, which is one of the most used approaches for compressing inverted indexes [8, 15]. However, there exist alternatives using different compression approaches, as for instance indexes based on Elias-Fano compression [55, 42], and indexes based on signatures like BitFunnel [27] (which is currently in production for Bing). However, although the space usage and query time achieved by these compression schemes differ from gap-encoded inverted indexes, we hope that our results will give insights to deal with these cases.

To support efficient query processing (such as DAAT [15], WAND [14], or BMW OR [20]), as well as effective compression of inverted lists, we sort them by increasing `docID`. Also, to avoid decompressing whole lists at query time, we assume that an inverted list I_t is divided into chunks of, say, 128 postings each.

Let $d_t[1..|I_t|]$ denote the sorted list of `docID`s for the inverted list I_t. Then, we replace $d_t[1]$ with $d_t[1]-1$, and $d_t[i]$ with $d_t[i] - d_t[i-1] - 1$, for $i = 2, \ldots, |I_t|$. In the case of frequencies, since it holds that $f_i > 0$, every f_i is replaced with $f_i - 1$, . Then these values are encoded with integer compression schemes that take advantage of the resulting smaller integers.

There has been a lot of progress on compressing `docID`s and frequencies, with many compression methods available [15, 58]. Some of them achieve a high compression ratio, but at the expense of a lower decompression speed [15], for example Elias γ and δ encodings [21], Golomb/Rice parametric encodings [26], or interpolative coding [38]. Other approaches may achieve a (usually slightly) lower compression ratio, though with faster decompression speeds; examples are VByte [57], S9 [4] and its variants, `PForDelta` [62], Quasi-Succinct Coding [55], or SIMD-BP128 [32]. The best compression method depends on the scenario at hand. We review next the most important compression schemes used along this paper.

2.3.1. Golomb/Rice Encoding

Given a sequence of integer numbers, Golomb compression [26] consists of defining a parameter B, which is used to compute the encoding of each number in the sequence. Let v be the average value from the input sequence. Then, a good choice is $B = 0.69v$ [58]. Given an integer i, it is encoded in two parts: a quotient $q = \lfloor i/B \rfloor$ (encoded in unary) followed by a remainder $r = i \bmod B$ (encoded in binary). If B is a power of two, this is known as *Rice encoding*, which allows for a more efficient implementation as multiplications (needed to decode a number) can be computed via bitwise shifts.

2.3.2. VByte Encoding

This method [57] encodes an integer number using a variable number of bytes. To do this, VByte uses the most significant bit in a byte as a flag. This flag indicates whether the current byte is the latest in the representation of the current integer or not. (Thus, for every byte, just 7 bits are used to represent the number.) The flag is called a continuator. If the continuator is "`0`", it indicates that the current byte is not enough to encode the current integer and hence the encoding must use the next byte. If the continuator is "`1`", the current byte is the last one of the current integer.

To encode a number in VByte, we try to accommodate its binary encoding in 7 bits. If so, we store the encoding in 7 bits and mark the continuator with a "1". Otherwise the least significant 7 bits are stored in a byte with continuator "0, and we proceed with the remaining bits, until no extra bytes are needed to store the encoding. The decoding process can be carried out very efficiently using bit-wise and arithmetic operations [15]. The fact that this encoding is byte-aligned makes the decoding process faster.

2.4. State of the Art for Positional Indexes

Compressing in-document positions is a problem where progress has been difficult to achieve [59]. In-document positions are mainly used in two applications: phrase searching and positional ranking schemes. In this paper we focus on positional ranking functions (also known as proximity ranking functions), where the positions of the query terms within the documents are used to improve the performance of a standard ranking such as BM25. The rationale is that documents where the query terms appear closer to each other could be more relevant for the query, so they should get a better score. Although we focus only on positional ranking functions, the compression schemes used in this paper should allow for phrase searching as well. This scenario is left for future work.

We next review the existing ways to index in-document positions.

2.4.1. Positional Inverted indexes

The standard solution for indexing in-document positions are *positional inverted indexes* [15, 33], also known as *full inversions* [33]. Given a term t, for each document d that contains t (and assuming that there are $f_{t,d}$ occurrences of t in d) there is a posting in the inverted list I_t containing $(d, f_{t,d}, \langle p_1, p_2, \ldots, p_{f_{t,d}}\rangle)$, such that $D_d[p_1] = D_d[p_2] = \cdots = D_d[p_{f_{t,d}}] = t$. Assuming that $p_1 < p_2 < \cdots < p_{f_{t,d}}$, in order to compress the in-document positions we store them differentially as $\langle p_1 - 1, p_2 - p_1 - 1, \ldots, p_{f_{t,d}} - p_{f_{t,d}-1} - 1\rangle$. We then use an integer compression scheme on these offsets.

Storing the in-document positions requires considerable space. A previous work [59] concludes that the offsets obtained from the in-document positions do not follow simple distributions that could be used to improve compression (as it is the case of, for instance, `docID`s and frequencies). As a result, a positional index is about 3 to 5 times larger than a `docID`/frequency index, and becomes a bottleneck in index compression. Another important conclusion from Yan et al. [59] is that we may only have to access a limited amount of position data per query, and thus it might be preferable to use a method that compresses well even if its speed is slightly lower. They propose to use compression schemes based on Rice compression. These methods use extra information (such as the document length and the number of occurrences of the term in the document) in order to parametrize Rice in a more efficient and adaptive way. Among the most efficient methods proposed by Yan et al. [59], we have Page-Adaptive Rice Coding (PA Rice for short) and Remaining-Page-Adaptive Rice Coding (RPA Rice for short). We will use these methods as baselines in our experiments.

2.4.2. Flat Position Indexes

Shan et al. [48] propose to use *flat position indexes* [17, 19] (also known as *schema-independent inverted indexes* [15], or simply *positional inverted indexes* [40]) as an alternative to positional inverted indexes. To build these indexes, let $\mathcal{T}[1..n]$ be the text obtained after the concatenation of the N documents in the collection, where $\sum_{i=1}^{N} n_i = n$. Given a term $t \in \Sigma$ (with a total of f_t occurrences in the whole collection), we define the corresponding inverted list I_t as containing positions $p_1 < p_2 < \cdots < p_{f_t}$ such that $\mathcal{T}[p_1] = \mathcal{T}[p_2] = \cdots = \mathcal{T}[p_{f_t}] = t$. In other words, flat position indexes regard term positions as global within $\mathcal{T}$, rather than local to a document as in positional inverted indexes. This has some advantages, as we will see next. In theory, the space usage is close to $nH_0(\mathcal{T})$ bits [40].

We also need to store the starting positions of each document within $\mathcal{T}$. This allows us to transform global positions into corresponding `docID`s. This setup is able to support the whole query process, saving considerable space as `docID`s and frequencies do not need to be stored (they are computed from the global positions). The result is that `docID`s, term frequencies, and position data can be stored in space close to that of positional inverted lists, yielding a reduction in space usage. However, the query process is somewhat slower on these indexes, due to the transformations between positions and `docID`s.

2.5. Snippet Generation

Besides providing a ranking of the most relevant documents for a query, another important issue for web search engines is the ability to generate query-dependent snippets for the results. Each snippet shows a portion of the result document, in order to help the user judge its likely relevance before accessing it. In this context, snippets have been shown to improve the effectiveness of search engines [52].

To support snippet generation, web search engines must store a (in some cases summarized [53]) copy of the documents. This usually requires considerable space and resources. Thus, the textual data must be compressed [25]. We now summarize compression approaches used in the literature to support snippet generation. There are two main approaches for snippet generation: *document-based* and *index-based* approaches [10]. In the former, snippets are computed by searching for the query terms in the result documents. In the latter, snippets are computed by means of a positional index. We focus on document-based snippet generation in this paper. This is the method of choice by open-source tools like Lucene, and appears to be more commonly used. Note that the index-based approach of course requires a positional inverted index.

2.5.1. Turpin et al.'s Approach

Turpin et al. [54] introduce a method to compress the text collection and support fast text extraction to generate snippets. The idea is to sort the vocabulary according to the term frequencies, and then assign term identifiers according to this order. In this way, the term identifier `0` corresponds to the most frequent term in the collection, `1` to the second most frequent term, and so on. The document collection is then represented as a single sequence of identifiers, where each term identifier is encoded using VByte [3]. Note that the 128 most frequent terms in the collection are thus encoded in a single byte. Actually, [54] uses a move-to-front strategy to store the encodings: the first time a term appears in a document, it is encoded with the original code assigned as before; the remaining appearances are represented as an offset to the previous occurrence of the term. We also use this approach in our experiments.

Thus, by using an integer compression scheme (such as VByte) for the text, it is possible to decompress *any text portion* very efficiently: just one small additional table is needed, indicating the starting position of each document.

2.5.2. Partition into Document Blocks

A simple alternative to compress the document collection and support decompressing arbitrary documents is to partition the whole collection into smaller document blocks, which are then compressed separately. This scheme is used by several state-of the art search engines, including Lucene [1]. Here, compressors based on LZ77 compression [61] are preferred, since they offer very efficient decompression performance (which is key for obtaining the documents to generate snippets). The document-block size offers a time/space trade-off: larger blocks yield better compression, although decompression time is increased.

Ferragina and Manzini [25] study how to store very large text collections in compressed form, such that the documents can be accessed when needed, and show how different compressors behave in such a scenario. Their approach for compressing textual data is to partition into document blocks: each block stores a subset of the documents. One of their main concerns was how compressors can capture redundancies that arise very far apart in very long texts. This is because LZ77 compression uses a (usually small) sliding window in which the text regularities are detected and compressed. Their results show that such large texts can often be compressed to just 5% of their original size. This was surprising, since usual compression ratios for generic text data are of about 20%, for the most effective compressors.

2.6. Succinct and Compressed Data Structures

Succinct or *compressed* data structures use as little space as possible to support operations as efficiently as possible [41]. Thus, large data sets (like graphs, trees, and text collections) can be manipulated in main memory, avoiding the secondary storage. In particular, we are interested in compressed data structures for text sequences. The following operations are the building block of many solutions in succinct data structures, and will be useful for our proposals.

[1] `http://lucene.apache.org/`.

Definition 2.4. Given a sequence (or *text*) $T[1..n]$ over an alphabet $\Sigma = \{1, \ldots, V\}$, we define operation $\mathsf{rank}_c(T, i)$, for $c \in \Sigma$, as the number of occurrences of c in $T[1..i]$. Operation $\mathsf{select}_c(T, j)$ is defined as the position of the j-th occurrence of c in T.

Two popular compression models [34] are the so-called *zero-order* and *k-order empirical entropy* of a sequence T, denoted $H_0(T)$ and $H_k(T)$, respectively. The value $H_0(T)$ is the average number of bits needed to encode each text symbol, provided we use $\log \frac{n}{n_c}$ bits to encode a symbol c that occurs n_c times in T. Notice how in this way the most frequent symbols obtain a shorter encoding. However, this model has a weakness: it does not account for the contexts that surround each symbol. This is particularly important in natural-language texts, where there are well-known dependencies between words in the text. In this case, the model aiming for $H_k(T)$ (the average number of bits needed to encode a symbol using a context of length $k > 0$) will provide better compression. An important property is that $H_k(T) \leq H_{k-1} \leq \cdots \leq H_1 \leq H_0 \leq \log V$, for any $k \geq 0$. Basically, this means that by considering longer contexts we can obtain improved compression.

2.6.1. Wavelet Trees

A *wavelet tree* [28] (`WT` for short) is a succinct data structure that supports rank and select operations, among many virtues [24, 39]. A `WT` representing a text T is a balanced binary search tree where each node v represents a contiguous interval $\Sigma^v = [i..j]$ of the sorted set Σ. The tree root represents the whole vocabulary Σ. At each node v, Σ^v is divided into two subsets, such that the left child v_l of v represents $\Sigma^{v_l} = [i..\frac{i+j}{2}]$, and the right child v_r represents $\Sigma^{v_r} = [\frac{i+j}{2} + 1..j]$. Each tree leaf represents a single vocabulary term. Hence, there are V leaves and the tree has height $O(\log V)$. For simplicity, in the following we assume that V is a power of two.

Let T^v be the subsequence of T formed by the symbols in Σ^v. Hence, $T^{root} = T$. Node v stores a bit sequence B^v such that $B^v[l] = \texttt{1}$ if $T^v[l] \in \Sigma^{v_r}$, and $B^v[l] = \texttt{0}$ otherwise. From another point of view, given a `WT` node v at level i, $B^v[j] = \texttt{1}$ iff the i-th most-significant bit in the encoding of $T^v[j]$ is `1`. In this way, given a term $c \in \Sigma$, the corresponding leaf in the tree can be found by using the binary encoding of c. Every node v stores B^v augmented with a data structure for $\mathsf{rank}/\mathsf{select}$ over bit sequences [41]. In Figure 1 we show an example `WT` for the text "CDEBFEGABBFCHHCDEABG".

The number of bits of the vectors B^v stored at each tree level sum up to n, and including the data structure every level requires $n + o(n)$ bits. Thus, the overall space is $n \log V + o(n \log V)$ bits [28, 41].

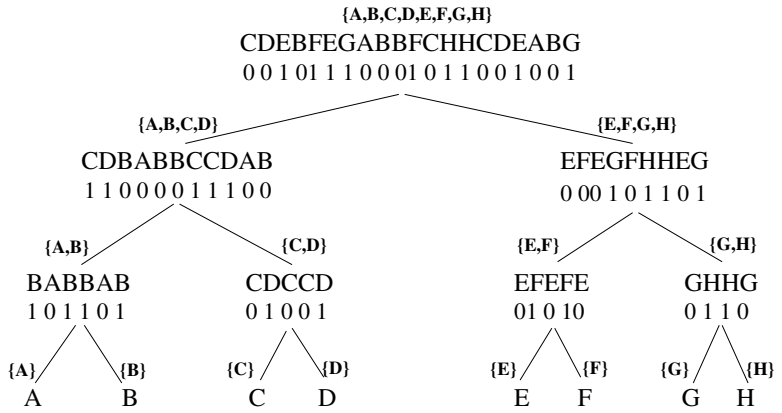

Figure 1: `WT` for the text "CDEBFEGABBFCHHCDEABG". Every `WT` node only stores the bit vectors.

Supporting Operations. Since a `WT` replaces the text it represents, we must be able to retrieve $T[i]$, for $1 \leq i \leq n$. The idea is to navigate the tree from the root to the leaf corresponding to the unknown $T[i]$. To do so, we start from the root, and check if $B^{root}[i] = \texttt{0}$. If so, the leaf of $T[i]$ is contained in the left

subtree v_l of the root. Hence, we move to v_l looking for the symbol at position $\mathsf{rank}_0(B^{root}, i)$. Otherwise, we move to v_r looking for the symbol at position $\mathsf{rank}_1(B^{root}, i)$. This process is repeated recursively, until we reach the leaf of $T[i]$, and runs in $O(\log V)$ time as we can implement the rank operation on bit vectors in constant time.

To compute $\mathsf{rank}_c(T, i)$, for any $c \in \Sigma$, we proceed mostly as before, using the binary encoding of c to find the corresponding tree leaf. On the other hand, to support $\mathsf{select}_c(T, j)$, for any $c \in \Sigma$, we must navigate upwards from the leaf corresponding to term c. Both operations can be implemented in $O(\log V)$ time; see [41] for details.

Space Usage in Practice. The space required by a `WT` is, in practice, about 1.1–1.2 times the space of the text [18]. In our application to represent document collections, this would produce an index larger than the text itself, which is excessive. To achieve compression, we can generate the Huffman codes for the terms in Σ (this is a *word-oriented Huffman coding* [37]) and use these codes to determine the corresponding tree leaf for each term. Hence, the tree is not balanced but has a Huffman-tree shape [18] such that frequent terms are closer to the root than less frequent ones. This achieves a total space of $n(H_0(T)+1)+o(n(H_0(T)+1))$ bits. In practice, the space is about 0.6 to 0.7 times the text size [18]. However, notice that we have no good worst-case bounds for the operations, as the length of the longest Huffman code assigned to a symbol could be $O(V)$.

2.6.2. Compressed Data Structures for IR Applications

There have been some recent attempts to apply alternative indexing techniques, such as compressed data structures, in large-scale information retrieval systems. In particular, we mention the work by Brisaboa et al. [13] and Arroyuelo et al. [6, 5]. The former [13] concludes that `WT` are competitive when compared with an inverted index for finding all the occurrences of a given query term within a single text. The latter [6, 5] extends [13] by supporting conjunctive queries on a `WT`. Also the work by Brisaboa et al. [12] shows how to use `WT` on Bytecodes [13] for solving ranked document retrieval. The result is that a `WT` can represent a document collection using $n(H_0(T)+1)+o(n(H_0(T)+1))$ bits while supporting all the functionality of an inverted index.

3. Positional Ranking and Snippet Generation: Baseline Approaches

Ranking is a fundamental part of the search process, and allows identification of documents that are most appropriate for the information need represented by a query. Typically, search engines first apply a simple ranking model to quickly select a small set of candidate documents from a potentially huge set of documents containing the query terms. While a lot of literature is focused on simple tf-idf ranking models, modern search systems use much more sophisticated ranking functions for selecting the final results from this initial set of candidates. One such example are positional ranking functions [15, 58], where the in-document positions of the query terms are used. The rationale behind positional ranking is that documents where the query terms occur close to each other might be ranked higher, since the term proximity could indicate that the query terms are highly correlated in the document.

Result snippets are another important tool used by current search engines to help users. These are typically carefully chosen fragments of the selected result documents that can help the user decide about the relevance of a result.

In this section we review the state of the art for position indexing and snippet generation, and provide experimental results that will be used as a baseline to determine the improvements introduced in this paper.

3.1. Basic Query Processing Steps for Positional Ranking and Snippet Extraction

From now on we assume a search engine that processes queries using the following steps:

- **Step 1: Basic Query Processing.** Given a user query, use an inverted index to obtain the top-k_1 documents according to some standard query processing approach (e.g., DAAT) and ranking function (e.g., BM25).

- **Step 2: Positional Reranking.** Given the top-k_1 documents obtained in the previous step, obtain the in-document positions of the query terms. Then rerank the results using a positional ranking functions [15, 46, 59].
- **Step 3: Snippet Generation.** After the re-ranking in the previous step, obtain snippets of length s for the top-k_2 documents, for a given $k_2 \leq k_1$.

For Step 2, in-document positions are obtained from a positional index, whereas for Step 3, text snippets are obtained from the compressed document collection. Typical values for the search parameters are $k_1 = 200$ and $k_2 = 10$. We assume $s = 10$ in this paper. The different values for these parameters should be chosen according to the trade-off between query time and search effectiveness that we want to achieve. We focus on alternative ways to implement Step 2 and Step 3.

3.2. Experimental Setup and Data

Throughout this paper, we use the following experimental setup. We ran our experiments in an HP ProLiant DL380 G7 (589152-001) server, with a Quadcore Intel(R) Xeon(R) CPU E5620 @ 2.40GHz processor, with 128KB of L1 cache, 1MB of L2 cache, 2MB of L3 cache, and 96GB of RAM, running version 2.6.34.8-68.fc13.i686.PAE of Linux kernel. All runs are performed single-threaded.

We use the TREC GOV2 document collection, with about 25.2 million documents and about 32.86 million distinct terms in the vocabulary. We enumerate the documents according to their original order in the collection. We work just with the text content of the collection (that is, we ignore the html code from the documents). This requires about 130,048 MB in ASCII format. When we represent the terms as integers, the resulting text uses 91,634 MB. We use a subset of 10,000 random queries from the TREC 2006 query log. All methods were implemented using C++, and compiled with `g++ 4.4.5`, with optimization flag `-O5`.

3.3. The Baseline: Positional Inverted Lists and Compressed Textual Data

Next, we describe and evaluate baseline approaches to support in-document position indexing and snippet extraction. These will be the starting points for the improvements proposed in this paper.

3.3.1. Supporting Query Processing (Step 1)

As we have already said, Step 1 is supported by inverted indexes. In our experiments, inverted lists have been divided into chunks of 128 `docID`s each, and implemented using layers: the first layer stores the `docID`s, whereas the second one stores the frequencies. An advantage of this approach is that every layer can be compressed differently with state-of-the-art methods. We used `PForDelta` for `docID`s and S16 for frequencies, which performed well. Note however that this choice will not impact our result much, as our focus is on the other steps. Also, a chunk in the frequency layer is decompressed only when the corresponding `docID` in the first layer needs to be fully scored. This process results in a fairly competitive query time to obtain the top-k_1 documents.

In our experiments, the index for `docID`s + frequencies for the GOV2 collection requires 9,739 MB, using the above-mentioned schemes for each layer.

For query processing, we assume the well-known Document-at-a-Time (DAAT) approach with BM25 ranking [15] to obtain the top-k_1 most relevant documents in Step 1. Table 1 shows the experimental average query time (in milliseconds per query) for Step 1. We show results for two types of queries: standard ranked AND queries and the BMW OR approach from [20], which is one of the most efficient current solutions for disjunctive (OR) queries.

Notice that the query time for AND is almost constant (within two decimal digits) with respect to k_1. The process to obtain the top-k_1 documents uses a heap of size k_1. However, operating the heap takes negligible time, compared to the decompression of `docID`s and the DAAT process. BMW OR, on the other hand, is an early-termination technique, and thus the value of k_1 impacts the query time significantly.

3.3.2. Supporting Positional Ranking (Step 2)

We implement Step 2 using two baseline approaches that we found to be the best performers: positional inverted indexes and flat position indexes.

Table 1: Experimental results for the initial query processing step (Step 1) for AND and OR queries. In both cases, BM25 ranking is used.

top-k_1	DAAT AND (ms/q)	BMW OR [20] (ms/q)
50	14.75	35.70
100	14.77	43.39
150	14.80	47.90
200	14.81	51.74
300	14.81	58.19

Positional Inverted Indexes. Positional inverted indexes add an extra layer to our inverted index representation. We refer to inverted lists in the positional layer as positional inverted lists (`PIL`, for short). Just as for the layers storing `docID`s and frequencies, we divide `PIL`s into chunks, each chunk storing all the in-document positions of the corresponding 128 documents in the `docID` layer. Given the `docID`s of the top-k_1 results for a given query, we obtain the corresponding in-document positions by identifying the `PIL` chunks containing the desired entries. Next, these chunks are *fully decompressed*, and the corresponding positions are obtained. This process is carried out for every query term. A drawback here is that we need to decompress the entire `PIL` chunk, even if we only need a few entries in it. Hence, if the query has q terms, in the worst case we would decompress qk_1 `PIL` chunks, assuming all position entries for a single term-document pair fit into one chunk. An improvement to this approach is the one proposed by Yan et al. [59], where the positional chunks are further divided into smaller sub-chunks. This adds extra space, but slightly reduces the position extraction time. We did not implement this approach here, but preferred the simpler one of just dividing the in-document position into chunks. This will not change the main conclusions of this paper, as the time to obtain positions from `PIL`s is just a small fraction of the overall query time.

Afterwards, these positions are used to re-rank the top-k_1 documents, as in Yan et al. [59]. Particular positional ranking functions are out of the scope of this paper: any function that uses term positions could be used. In our experiments we use the scoring model proposed by Büttcher et al. [16].

Since positions must be obtained only for the top-k_1 documents, it is better in general to use compression methods with a good compression ratio, like Golomb/Rice compression. These are slower to decompress, however since only a few positions are decompressed, this should not impact the overall query time much. According to [59], the most effective compression techniques are variants of Rice and S16. Table 2 shows experimental results for position extraction with `PIL`s (see the upper three rows), using the compression schemes Rice, Page-Adaptive Rice (PA Rice) [59], and S16. We show query times for different values of k_1, namely 50, 200 and 300. We also tested with Remaining-Page-Adaptive Rice compression [59]. However, the space is only slightly smaller than PA Rice, while being considerably slower at obtaining in-document positions.

As it can be seen, Rice requires only about 90% of the space of S16, but takes twice as much time. On the other hand, PA Rice obtains a much better compression ratio. Notice that using Rice, `PIL`s are 2.91 times bigger than the inverted index that stores `docID`s and frequencies, whereas for PA Rice and S16 these numbers are 2.25 and 3.22, respectively. Comparing the query times of Step 2 for the various `PIL` alternatives against the query times of Step 1 in Table 1, we can see that position extraction is actually a small fraction of the overall time.

Flat Position Index. We implemented flat position indexes, using `PForDelta` compression [48]. We show experimental results in Table 2 (see the lower row). As it can be seen, position extraction times are much faster than those of `PIL`, yet using more space. However, recall that within the same space, flat position indexes are able to retrieve `docID`s and frequencies (i.e., they support both Step 1 and Step 2 within the reported space), though at reduced performance for Step 1. This is 3.26 times the space of an inverted index with `docID`s and frequencies.

Table 2: Experimental results for extracting in-document position data (Step 2) using the baseline approaches. The compression ratio is computed as s/u, where s is the space of the data structure and $u = 91,634$ is the size (in MB) of the uncompressed text.

Approach	Compression Scheme	Compression Ratio	Position extraction time (msec/query)		
			$k_1 = 50$	$k_1 = 200$	$k_1 = 300$
Positional inverted indexes	`PIL` Rice (no text)	30.96	1.28	3.27	5.57
	`PIL` PA Rice (no text)	23.93	1.87	5,78	8.13
	`PIL` S16 (no text)	34.20	0.74	1.75	2.51
Flat position index	`PForDelta` (no text)	31,738	34.64	0.09	0.33	0.49

3.3.3. Supporting Snippet Generation (Step 3)

Given the popularity and simplicity of the approach explained in Section 2.5.2, we use it as the baseline for compressing the text.

Table 3 shows the experimental results. Just as in [25], we divide the text into blocks of 200 KB, 500 KB and 1,000 KB, and compress each block using different standard text compression tools. In particular, we show results for `lzma` [2] (which gives very good results in [25], so it serves as a comparison with the results in that work), and `lz4` [3] (which is a variant of LZ77 compression, improved for speed rather than compression ratio). These two compressors offer the most interesting trade-offs among the alternatives we tried.

From Table 3 we can conclude:

- `lzma` achieves much better compression ratios than `lz4`. Also, in all cases the compression ratio improves as we increase the block size, as expected. This is because more text regularities can be detected by the LZ77 variants. For comparison, if we now compress the text as a whole, without the block structure —which would not be useful for snippet generation, but maybe for archival purposes—, the compressed space achieved is 8,133 MB for `lzma` (compression ratio of 8.87%), and 29,808 MB for `lz4` (compression ratio of 32.53%). Notice that the compression ratio for `lzma` is similar to that reported by Ferragina and Manzini [25]. Also, notice that the whole text compressed with `lzma` requires less space than `PIL`s (using PA Rice compression), which shows how powerful a higher-order compressor can be, particularly for large texts [25].

- Regarding extraction time, we can also observe important differences among alternatives, with `lz4` being the fastest and `lzma` being much slower.

It is important to mention that Ferragina and Manzini [25] report a decompression speed of about 35 MB/sec for `lzma`. However, to obtain a given document we must first decompress the entire block that contains it. Hence, most of the 35 MB/sec do not correspond to any useful data. In other words, this does not measure effective decompression speed, and thus we report per-query times rather than MB/sec.

Finally, notice that the space usage of `lzma` is much smaller than the space used to store in-document positions (using either `PIL`s or flat positional indexes). This is because standard text compressors can take advantage of text regularities, as they compress using higher-order models such as $H_k(T)$. Positional postings, on the other hand, are stored separately for each term, so inter-term regularities cannot be used to improve compression (they are basically compressed to zero-order entropy). A relevant question here is: How can one represent in-document positions in such a way that the text regularities are preserved and used to improve compression? Our answer to this question is one of the most important contributions of this paper.

[2] `http://code.google.com/p/lzma/`.

[3] `http://code.google.com/p/lz4/`.

Table 3: Experimental results for compressing the document collection (Step 3). The compression ratio is computed as s/u, where s is the space of the data structure and $u = 91,634$ is the size (in MB) of the uncompressed text.

Compressor	Block size (KB)	Compression Ratio	Snippet extraction time (ms/q)		
			$k_2 = 10$	$k_2 = 30$	$k_2 = 50$
`lzma`	200	22.12	58.02	165.06	266.44
	500	20.60	124.34	356.01	575.93
	1,000	19.64	231.14	662.22	1,071.51
`lz4`	200	46.90	3.03	8.43	13.40
	500	46.53	6.07	17.17	27.58
	1,000	46.40	11.20	31.90	51.43

4. Using the Textual Data to Compute In-Document Positions

Compressing in-document positions has been recognized as a difficult task [59, 25]. As it can be seen in the experiments of Section 3, the size of the two structures combined (positions and text) can be up to 7 times the space used by the `docID`s + frequencies inverted index. Moreover, Ferragina and Manzini [25] show that the textual data can be, indeed, compressed better than positions. The conclusion is that positions have become a bottleneck for compression compared to `docID`s and frequencies. There are two main reasons for this:

- In-document positions have a distribution that is different from that of `docID`s and frequencies [59].
- Since in-document positions are stored separately for each term (recall Section 2.3), the local context of the terms that is exploited by high-order text compression schemes is not available in positional inverted lists.

Thus, the efficient compression of in-document positions is a challenging problem. In this section we focus on alternative approaches to carry out Step 2 and Step 3 from Section 3.1. The aim is to optimize both space usage and query processing time.

4.1. The Proposed Approach

We propose not to store in-document position data at all, but to *compute* them *on the fly* from a compressed representation of the text collection. We will focus on two alternative approaches for achieving this:

- *Wavelet trees* [28, 24, 39], which are succinct data structures from the combinatorial pattern matching community (see Section 4.2).
- Compressed document representations that support fast extraction of arbitrary documents (see Section 4.3).

Our aim is to take advantage of the high compressibility of text in order to support fast and space-efficient positional ranking. Since in our proposal the compressed text is used for both positional reranking and snippet generation, we can save a lot of space overall.

One of the main concerns is how these alternatives might impact query processing speed, as we must decompress entire documents and then search for the query terms within them. We next study the resulting trade-offs between running time and space requirement.

4.2. A Succinct Data Structure for Computing In-Document Positions and Snippets

First, we explore the alternative of representing the text collection using a Wavelet Tree (`WT` for short) [28, 24, 39] data structure. We use the `WT` functionality to obtain in-document positions and snippets. This data structure replaces the positional inverted lists and the text collection, aiming at a better space usage.

4.2.1. Representing the Document Collection

We represent the document collection according to the following definition.

Definition 4.1. Let $T = \$D_1\$D_2\$\cdots\$D_N\$$ be the text obtained from the concatenation (in arbitrary order) of the documents in the collection, where the documents are separated by a special symbol $\$ \notin \Sigma$.

We represent the resulting sequence T with a `WT`. Given a position i in T, one can easily obtain the `docID` of the document that contains $T[i]$ as $\mathsf{rank}_{\$}(T, i)$, as well as the starting position of a given document D_j by means of $\mathsf{select}_{\$}(T, j) + 1$.

4.2.2. Obtaining In-Document Positions from the `WT`

Assume that, given a `docID` i and a query term t, we want to obtain the in-document positions of t within D_i. A simple solution could be to extract document D_i from the `WT`, and then search for t within it. The total time would be then proportional to the document length. A more efficient way is to use operation select to find every occurrence of t within D_i, hence working in time proportional to the number of occurrences of the term (which is always smaller than or equal to the document length, being much smaller in typical cases). See Figure 2 for an illustration of this process. Let d be the starting position for document D_i in T. Hence, there are $r \leftarrow \mathsf{rank}_t(T, d)$ occurrences of t before document D_i, and the first occurrence of t within D_i is at position $j \leftarrow \mathsf{select}_t(T, r + 1)$, the second occurrence at position $j' \leftarrow \mathsf{select}_t(T, r + 2)$, and so on. Overall, if R is the number of occurrences of t within D_i, then we need 1 rank and $R+1$ selects to find them.

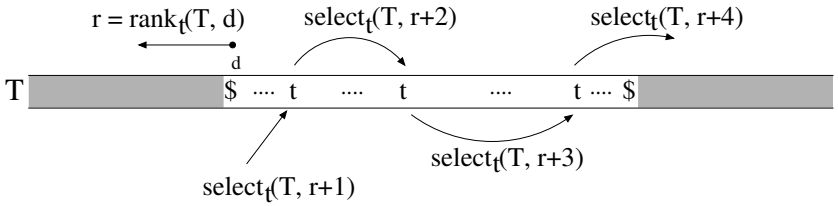

Figure 2: In-document position extraction from a `WT` , using operation select.

4.2.3. Byte-Oriented Huffman `WT`

Instead of a bit-oriented `WT` (as the one explained in Section 2.6), we use the byte-oriented representation from [13], using the Plain Huffman encoder. This is the most efficient alternative reported by Brisaboa et al. for representing natural-language texts. The idea is to assign a Huffman code to each vocabulary term [37]. Then, we store the most-significant *byte* of the encoding of each term in array B^{root}. That is, each `WT` node v stores an array of bytes B^v, rather than bit arrays as in Section 2.6. Next, each term in the text is assigned to one of the children of the root, depending on the first byte in the encodings. Notice that in this way the `WT` is 256-ary.

To support rank and select, we also use the simple approach from [13]. Given a `WT` node v, we divide the corresponding byte sequence B^v into superblocks of s_b bytes each. For each superblock we store 256 *superblock counters*, one for each possible byte. These counters tell us how many occurrences of a given byte there are in the text up to the last position of the previous superblock. Also, each superblock is divided into blocks of b bytes each. Every such block also stores 256 *block counters*, similarly as before. The difference is that the values of these counters are local to the superblock, hence less bits are used for them.

To compute $\mathsf{rank}_c(T, i)$, we first compute the superblock j that contains i, and use the superblock counter for c to count how many c there are in T up to superblock $j-1$. Then we compute the block i' that contains i and add (to the previous value) the block counter for c. Finally, we must count the number of c within block i'. This is done with a sequential scan over block i'. This block/superblock structure allows for time-space trade-offs. In our experiments we use $s_b = 2^{16}$. Hence, superblock counters can be stored in 16 bits each. We consider $b = 1$ KB, $b = 3$ KB and $b = 7$ KB. Operation select is implemented by binary searching the superblock/block counters; thus no extra information is stored for this [13].

Table 4: Experimental results for extracting in-document position data (Step 2) from the document collection. The compression ratio is computed as s/u, where s is the space of the data structure and $u = 91,634$ is the size (in MB) of the uncompressed text.

Approach	Compression Scheme	Compression Ratio	Position extraction time (msec/query)		
			$k_1 = 50$	$k_1 = 200$	$k_1 = 300$
Compressed data structures	`WT`(7 KB)	44.23	1.91	6.79	9.70
	`WT`(1 KB)	62.11	0.32	1.17	1.64
	`WT`(7 KB) + `lzma`	28.44	42.90	168.83	244.17
	`WT`(1 KB) + `lzma`	50.72	17.28	65.78	94.74
	`WT`(7 KB) + `lz4`	35.23	14.58	52.13	74.09
	`WT`(1 KB) + `lz4`	55.90	2.11	7.52	10.77
Text compressors	`lzma` (200 KB)	22.12	266.21	930.09	1,323.37
	`lz4` (200 KB)	46.90	13.41	44.53	62.28
Zero-order compressors	VByte	41.84	0.41	1.40	2.02
	VNibble	37.73	1.86	6.75	8.10
	Byte-Oriented Huffman	41.55	1.09	3.82	5.45
Compression boosters	VByte + `lzma` (200 KB)	20.51	519,42	1,826.67	2,601.60
	VByte + `lzma` (50 KB)	22.70	162.35	551.62	779.05
	VByte + `lzma` (10 KB)	24.78	75.14	220.73	296.83
	VByte + `lz4` (200 KB)	28.42	12.48	39.38	54.17
	VByte + `lz4` (50 KB)	29.30	6.30	18.20	24.16
	VByte + `lz4` (10 KB)	31.49	4.85	12.81	16.37

In Table 4 we show the experimental trade-offs for `WT`, for the different block sizes tested. Note that `WT` (1 KB) obtains better times than `PIL`s (recall Table 2), yet requiring considerably more space. An advantage of the `WT` structure is, on the other hand, that it contains the text, hence it can be used to search for positions and produce document snippets. Also, `WT` (7 KB) is a bit slower than `PIL` (PA Rice) and uses more space. Even though the `WT` includes the textual data, its high space usage could be a drawback, in particular in schemes where the text is not necessary. Next, we try extra improvements to make `WT`s competitive.

4.2.4. Approaching Higher-Order Compression with the `WT`

Basically, `WT`s are zero-order compressors: each term in the text is encoded according to its frequency, such that the most frequent terms are encoded using a short encoding. This explains the high space usage obtained in our experiments.

To achieve higher-order compression (i.e., having into account inter-term regularities to compress), notice that B^{root} contains the most-significant byte of the Huffman encodings of the original terms in the text. Thus, the original text structure is at least partially preserved in the structure of B^{root}, which might thus be as compressible as the original text. A similar behavior can be observed in internal nodes for the remaining bytes that form the encodings of the terms. Thus, we propose to compress the blocks of B^v in each `WT` node v by using some standard (LZ77-based) compressor. Since only the first byte of each term is kept in the first level of the `WT`, the text regularities will be closer now (in terms of bytes that separate each other). This can improve the compression ratio achieved by an LZ77 compressor, since more regularities will fit within the sliding window.

In particular, we use the `lzma` [4] and `lz4` [5] compressors. Table 4 shows the results. Notice that:

- `WT` (7 KB) + `lzma` achieves 26,065 MB of space, a decrease of 35.69% over `WT` (7 KB). This shows the effectiveness of this approach. The time to obtain positions becomes, on the other hand, significantly slower.
- `WT` (7 KB) + `lz4` is faster, using space that is closer to that of `PIL`s. However, query times are still slow in comparison (by about an order of magnitude).

Overall, one can see that using higher-order compression on `WT`s yields good space savings (although this is slower and uses more space than `PIL`s). This significant decrease in space could make `WT`s competitive in scenarios where storing the text is necessary. Moreover, `WT`s are interesting by themselves since they can support query processing without using extra space [6], saving the space of the inverted index. In other words, Steps 1, 2 and 3 could be supported using, say, 26,065 MB (the space of `WT` (7KB) + `lzma`).

4.3. Computing In-Document Positions from Compressed Text Representation

We consider next representing the text collection using different text compressors. Unlike in Section 4.2, no complex data structure is used on the text this time. Thus, the whole documents needed for a query must be decompressed, and the query terms must be sought in the text. We consider several ways to compress the text.

4.3.1. Using Standard Text Compressors

As a first approach, we obtain the in-document positions using the baseline for generating snippets from Section 3.3.3. At search time, we first sort the top-k_1 documents by increasing docID, and then they are obtained from their corresponding blocks. The sorting step is carried out in order to avoid decompressing a document block multiple times. In the worst case, we must decompress k_1 document blocks, compared with the (worst-case) k_1 positional chunks for each query term from `PIL`s. Next, the query terms are sought within these documents, and their positions obtained. We carry out a single scan on each document, checking whether each text position contains one of the query terms or not. For the snippet extraction step, no further decompression is needed, since the top-k_2 documents have been already decompressed.

In Table 4 (see the rows for the "Text compressors" approach) we show the experimental time-space trade-offs obtained, using `lzma` and `lz4` compressors, and blocks of size 200 KB. As it can be seen:

- By using `lzma` we are able to store in-document positions and text data in about 0.92 times the space of `PIL` (PA Rice) —the latter just storing positions. Although this seems promising from the space-usage perspective, this approach is about two orders of magnitude slower than `PIL`s. Moreover, the time is about 18 times slower than that of a DAAT AND in Step 1 of the query process (recall Table 1 on page 10), which limits its use in real systems.
- By using `lz4` we obtain a space that is 1.96 times larger than `PIL` (PA Rice), whereas the time to obtain in-document positions is about 7.17 to 7.68 times slower than using `PIL`s (depending on k_1). For $k_1 = 50$, this is still acceptable, as it corresponds to about 0.91 times the time of Step 1 for DAAT AND queries. For larger values of k_1, the total time makes this alternative less competitive.

Though the space usage achieved with `lzma` is competitive, the time needed to obtain in-document positions from it is still too high. On the other hand, `lz4` achieves better position extraction times, yet its space usage is still high compared with `PIL` (PA Rice). In what follows, we shall study several approaches to achieve (as much as possible) the best of both worlds: a space usage similar to that of `lzma`, as well as the competitive position extraction time of `lz4`.

[4] `http://www.7-zip.org/`
[5] `http://lz4.github.io/lz4/`

4.3.2. Zero-order Compressors with Fast Text Extraction

We consider now using the approach from Turpin et al. [54], already introduced in Section 2.5.1. Recall that no document blocks are needed, as this approach supports the efficient extraction of individual documents. This could be key to achieve faster position lookups. Table 4 shows the results for this alternative (see "Zero-order compressors"). We can conclude that:

- Using VByte compression on Turpin et al.'s approach, we improve the position extraction time significantly, making it faster than `PILs`. This indicates that being able to extract just the desired k_1 documents is a big advantage, compared to decompressing a whole block with standard compressors. The higher space usage is, however, a concern. Yet, recall that we also represent the text within this space, not just the position data as in `PILs`.
- We also tried other compression schemes, such as `PForDelta` and S9, obtaining poorer compression ratios and similar decompression speed. The only method that improved the compression ratio is VNibble, a variant of VByte that represents any integer with a variable number of nibbles (i.e., half bytes). As in VByte, one bit of each nibble is used as a continuation bit, so only the remaining 3 bits are used to represent the number. The results of Table 4 show space savings of about 10% over VByte.
- Finally, we tried Byte-Oriented Huffman compression, obtaining a space usage that is slightly better than VByte, while being slower for position lookups—the decoding process of Huffman uses the Huffman tree, which makes it slower than VByte.

The highly competitive position extraction time of these alternatives is due to two facts. First, methods like VByte and VNibble are able to decompress hundreds of millions of integers (corresponding to terms) per second [60]. Second, VByte and VNibble are able to decompress just the desired documents. However, this is basically zero-order compression, and thus we are still far from the space usage of, for instance, `lzma`. The goal of the next approach is to maintain the position-extraction times of VByte or VNibble while achieving higher-order compression.

4.3.3. Natural-Language Compression Boosters

To approach higher-order compression, we propose to use a so-called natural-language compression booster [23]. The compression process is carried out in two consecutive steps. In the first step we compress the text using a compression booster, which is a (byte-oriented) zero-order compressor—e.g., either Byte-Oriented Huffman, or VByte/VNibble based on Turpin et al.'s approach. In the second step, this zero-order compressed text is further compressed by using some LZ77-based compression scheme, using again document blocks as in Section 4.3.1.

Since blocks are defined after the text has been pre-compressed with the compression booster, we will be able to accommodate more documents within each block. In other words, the compression booster virtually enlarges the document blocks. But it also virtually enlarges the LZ77 window [23], and hence more regularities can be detected, improving compression.

It has been shown that this approach yields better compression ratios than using just a standard compression scheme [23], in particular for small document blocks. Rather than byte-oriented Huffman or End-Tagged coding as in [23], we propose using Turpin et al.'s [54] approach as the booster (using VByte and VNibble as we explained above). Our experiments indicate that Turpin et al.'s approach is faster and uses only slightly more space.

In Table 4 we show results for the compression boosting approaches. We use "VByte + `lzma`" to indicate that VByte is used as booster for the `lzma` compressor (the same notation is used for `lz4`). In our experiments, VByte outperformed VNibble, hence we only show results for the former. That is, even though VNibble on its own was able to achieve a better space usage than VByte, the combination of VByte as booster uses from 9% to 16% less space than VNibble as booster, depending on the higher-order compressor used. The rationale is that since VByte is byte aligned, higher-order compressors (which are also byte aligned) are able to detect the regularities of the text in a better way. VNibble, on the other hand, is not byte aligned, then many regularities are not detected, worsening the compression ratio.

Our experiments indicate that the decrease in space over the original VByte approach is considerable:

- Comparing VByte + `lzma` (200 KB) with `lzma` (200 KB), the result is a decrease in space usage of 7.27% (18,794 MB vs 20,268 MB), but at the cost of about twice the running time of the original `lzma`.
- For VByte + `lz4` (200 KB), the size is reduced by 39.39% compared with `lz4` (200 KB), whereas the average position extraction time is also improved. The fact that more documents fit within a block results in an improved compression ratio. Also, and for the same reason, less blocks are decompressed, thus improving query time.

Notice also that when using smaller blocks, the time to obtain positions rapidly improves, while the space usage is not increased considerably. We can conclude that VByte + `lz4` offers the most interesting trade-offs. For VByte + `lz4` (50 KB), the space is 1.22 times the space of `PIL` PA Rice. Also, VByte + `lz4` (50 KB) is about 3 times slower than `PIL` PA Rice. However, two things should be recalled. First, VByte + `lz4` (50 KB) contains positions and text using just 22% more space than `PIL` PA Rice. Second, the position extraction time of VByte + `lz4` (50 KB) ranges between 0.43 ($k_1 = 50$) and 1.63 ($k_1 = 1.63$) times the query time of a DAAT AND query (recall Table 1). Thus, the overall query time should not be affected too much, while reducing space usage considerably.

4.4. Reducing the Number of Accessed Document Blocks via Document Reorganization

The results in Table 4 indicate that our approaches are effective for in-document position lookup. In particular, the results for compression boosters indicate that reducing the number of document blocks accessed (and hence decompressed) at query time, yields significant improvements in query-processing time. We next study a way to further reduce the number of accessed document blocks.

Ideally, during the execution of Step 2 of the query process, the top-k_1 documents relevant to a query are stored within the same (or a few) document block. In this way, the document extraction task would take only a small fraction of query time. Unfortunately, this is not usually the case if the documents are stored arbitrarily (e.g., following a random order): Table 5 shows the number of document blocks decompressed at query time for different values of k_1 and block sizes. Column "Unsorted" in the table (which corresponds to the original order in which the documents are stored in the GOV2 collection) shows us that the number of document blocks that are accessed in Step 2 almost equals k_1. In other words, most of the top-k_1 documents that are relevant to a query are stored in different blocks.

Table 5: Number of document blocks decompressed at query time.

Document Block Size	k_1	Number of Accessed Blocks		
(KB)		Unsorted	URL	URL + Association Rules
200	50	49.7	32.4	28.1
	200	197.3	109.9	100.5
	300	295.0	156.7	155.9
500	50	47.3	30.4	25.7
	200	191.4	100.2	90.6
	300	288.7	141.6	140.5
1,000	50	49.1	28.9	24.1
	200	192.3	93.4	83.8
	300	286.0	131.0	129.8

Next, we study ways to arrange the documents within the document blocks, aiming to minimize the number of blocks decompressed at query time. This is not trivial, as we must decide at construction time which documents will be relevant for queries that are unknown at that point (because they will be issued at query time), and hence they must be stored in the same document block. Another important aspect to consider is space usage. One would want to store similar documents close together in the same block, in

order to improve LZ77 compression. Finally, this could have the additional benefit of reducing the number of secondary-memory accesses if the document collection is stored on secondary storage.

We arrange the documents using the following approaches:

4.4.1. URL Sorting

Documents are first sorted lexicographically according to their URLs [49]. Next, we store them using this order. For the GOV2 collection, this has several advantages, as documents that are relevant to a query tend to be stored contiguously [49, 60]. This is good for our purposes. In order to maintain the proper operation of the search system, we keep a map $M[1..N]$ such that the document with `docID` i has been stored at position $M[i]$ within the URL sorting. In this way we are able to find the desired documents within the document database.

4.4.2. Sorting using URLs + Association Rules

Next, we try to improve the above document storage, using the most-common user queries to determine which documents are relevant to each other. In this case, after sorting by URLs (and obtaining the corresponding map M), we apply a further step which is based on *association rule mining* [1]. The idea is to use a query log to learn query patterns (e.g., which terms are commonly used together in queries), to then store the relevant documents to these queries within the same blocks. We will regard user queries as transactions for the algorithm Apriori [2].

We first eliminate stopwords from the query log, and use only 50,000 queries from the TREC 2006 query log to run algorithm Apriori. We decided to use a minimum support of 50. The result is a set of association rules $r_1, \ldots, r_s$, each rule being a list of terms that can be proven statistically related in the query log. Let $R[1..N]$ be an array such that $R[i]$ stores two fields: $R[i].rule$ and $R[i].ranking$ (both initialized to 0).

We now consider each association rule r_i as a query, and process it using the inverted index to obtain its top-1024 documents (or less if there are not enough documents relevant to the query). For each document D_j with ranking r in the top-1024 of r_i, we check if $R[M[j]].ranking > r$. If so, we set $R[M[j]].rule \leftarrow i$, $R[M[j]].ranking \leftarrow r$. This is because each document D_j can be relevant to several association rules. Hence, $R[M[j]].ranking$ keeps the highest rank r for D_j (among all the top-1024 in which D_j appears), whereas $R[M[j]].rule$ is the association rule for which it holds.

Let $M'[1..N]$ be a mapping, similar to M. Next, we set a variable $\texttt{next} \leftarrow 1$, and traverse again the association rules r_i, for $i = 1, \ldots, s$. For every D_j in the top-1024 of r_i we check if $R[M[j]].rule = i$ holds. In such a case, we set $M'[i] \leftarrow \texttt{next}$, and $\texttt{next} \leftarrow \texttt{next} + 1$. Once every association rule has been processed in this way, we finally traverse mapping R again, and for every j such that $R[j].rule = 0$, we set $M'[j] \leftarrow \texttt{next}$, and $\texttt{next} \leftarrow \texttt{next} + 1$ Finally, we use the space of M to make a copy of M'. Next, to obtain the order in which the documents must be stored, we invert M' (notice that M' is actually a permutation of $1, \ldots, N$, and hence can be inverted in $O(N)$ time).

4.4.3. Experimental Results

Table 5 shows the number of document blocks decompressed at query time, for the different orderings we propose. As it can be seen, the number of decompressed blocks can be reduced from almost 100% (unsorted) to about 50% the value of k_1. This yields important improvements in query processing time, as we shall see next.

Table 6 replicates the experiments from Table 4, this time for a document collection that has been reorganized using URL + association rules to reduce the number of accessed blocks. As it can be seen, `WT`s and zero-order compressors are insensitive to the document reorganization, as they are zero-order compressors. For the remaining alternatives, however, there are important improvements, as we discuss next.

For `WT`(7KB) + `lzma`, the space usage is 19,628 MB, an improvement of 24.69% over the result in Table 4. Query time also improves significantly, making it a competitive scheme. `WT`(7KB) + `lz4`, on the other hand, uses more space: 24,911 MB (an improvement of 22.83%). Although query time is not improved as much as for `lzma`, this scheme is still competitive.

Another interesting approach is `lzma` (200 KB), which requires 14,987 MB. This is 26.05% less than the same scheme in Table 4. Query time is, however, still not competitive.

Table 6: Experimental results for extracting in-document position data (Step 2), in this case from a *reorganized* document collection. The compression ratio is computed as s/u, where s is the space of the data structure and $u = 91,634$ is the size (in MB) of the uncompressed text. Column "Space" shows the percentage in which the space usage is improved with respect to the corresponding scheme in Table 4. Columns corresponding to "Position extraction time" show the improvement of query time, also regarding the results from Table 4.

Approach	Compression Scheme	Space	Compression Ratio	Position extraction time (% improvement)		
				$k_1 = 50$	$k_1 = 200$	$k_1 = 300$
Compressed data structures	`WT`(7 KB)	0.00%	44.23	0.00%	0.00%	0.00%
	`WT`(1 KB)	0.00%	62.11	0.00%	0.00%	0.00%
	`WT`(7 KB) + `lzma`	24.69%	21.42	55.12%	59.50%	59.98%
	`WT`(1 KB) + `lzma`	8.87%	46.23	58.21%	62.04%	62.45%
	`WT`(7 KB) + `lz4`	22.83%	27.18	0.20%	2.07%	0.68%
	`WT`(1 KB) + `lz4`	9.02%	50.85	1.42%	2.79%	2.69%
Text compressors	`lzma` (200 KB)	26.05%	16.35	48.31%	48.16%	48.24%
	`lz4` (200 KB)	29.25%	33.18	50.78%	47.85%	46.86%
Zero-order compressors	VByte	0.00%	41.84	0.00%	0.00%	0.00%
	VNibble	0.00%	37.73	0.00%	0.00%	0.00%
	Byte-Oriented Huffman	0.00%	41.55	0.00%	0.00%	0.00%
Compression boosters	VByte + `lzma` (200 KB)	33.56%	13.63	50.68%	50.37%	51.99%
	VByte + `lzma` (50 KB)	32.78%	15.26	56.68%	55.23%	54.85%
	VByte + `lzma` (10 KB)	26.18%	18.29	74.36%	69.19%	67.30%
	VByte + `lz4` (200 KB)	29.50%	20.04	50.80%	45.04%	43.12%
	VByte + `lz4` (50 KB)	29.80%	20.56	71.11%	64.61%	61.92%
	VByte + `lz4` (10 KB)	24.93%	23.64	85.97%	81.10%	78.86%

Compression boosters seem to be the ones that benefit the most from reorganizing the document collection. VByte + `lzma` (200 KB) achieves impressive 12,486 MB, however query time is still too high to be competitive. VByte + `lzma` (10 KB) achieves 16,762 MB, with relatively competitive query time.

Finally, VByte + `lz4` (50 KB) and VByte + `lz4` (10 KB) are the most effective schemes, with a space usage of 18,845 MB and 21,665 MB, respectively. Query time is also pretty competitive, roughly matching the efficiency of `PILs`.

4.5. Further Experiments

We carry out now additional experiments in order to compare the alternatives from different points of views. We aim at understanding what affects the position-lookup time of each alternative.

4.5.1. Average Position Extraction Time versus Query Length

An interesting question is how the query length (i.e., the number of query terms that appear in the query) affects the position extraction time of the different alternatives. Figure 3 shows experimental results for $k_1 = 50$ (left) and $k_1 = 300$ (right). As it can be seen, the performance of schemes based on compression boosting (VByte + `lz4`) remains (almost) unaffected as query length increases. The rationale is that most of the time is spent decompressing the document blocks to obtain the top-k_1 documents. This task is independent of the query length. After that, the query terms are sought within these documents, which takes $O(q \cdot n_j)$ time, where q is the query length and n_j is the document length. That is, the time spent searching for the query terms depends on the query length. Fortunately, our experiments indicate that this time is negligible when compared with the time spent decompressing the documents.

For PILs, on the other hand, we can see that the overall time grows as the query length increases. The rationale is that as the query length increases, more inverted list are involved in the query process. Hence, more chunks in the positional lists must be decompressed.

As it can be seen, for $k_1 = 50$ scheme VByte + `lz4` (Block 10KB), for instance, outperforms PILs for queries of length ≥ 4. For $k_1 = 300$ scheme VByte + `lz4` (Block 10KB) outperforms PILs for queries of length ≥ 5. We can conclude that approaches based on compression boosters become a more attractive approach as query length increases.

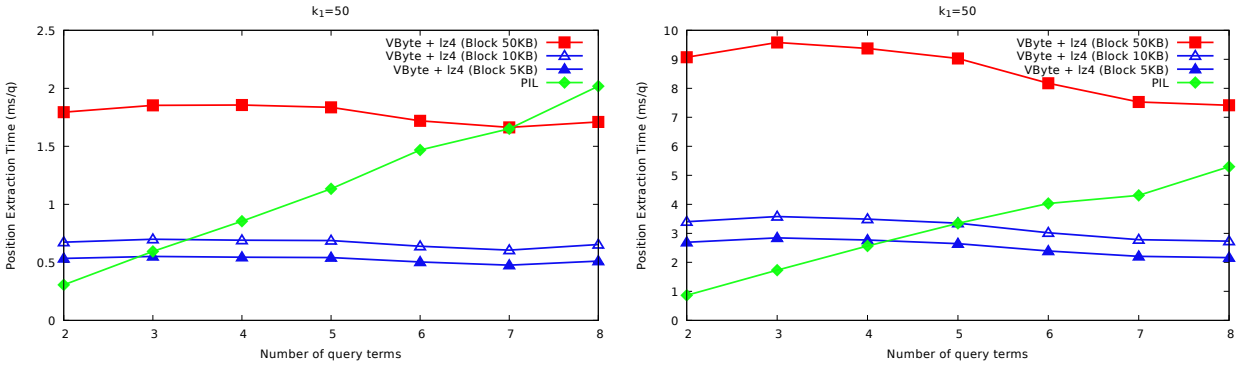

Figure 3: Average position extraction time for increasing query length.

4.5.2. Average Position Extraction Time versus k_1

We now test how the value of k_1 affects the position extraction time. Figure 4 shows the corresponding experimental results. Notice that the growth for PILs is the slowest as k_1 increases. This is because as k_1 grows, the number of PIL chunks that must be decompressed does not grow linearly with k_1 (recall that every PIL chunk stores 128 documents). Document blocks, on the other hand, store much less documents (only tens of them), hence there is a high probability of decompressing a number of document blocks that is roughly linear in k_1. We already saw this behavior in the results shown in Table 5. Moreover, as we increase the document-block size, more useless documents are decompressed. This explains why the time for blocks of size 50 KB grows faster.

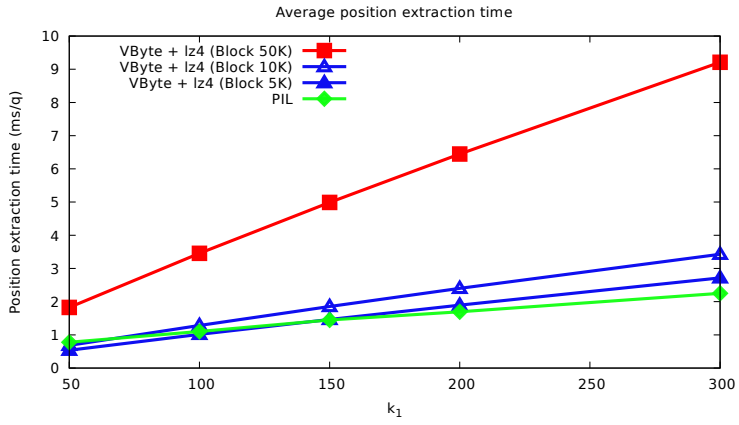

Figure 4: Average position extraction time for increasing k_1.

5. Experimental Results for the Whole Query Process

In this section we use the best alternatives from the previous section to build complete search schemes that allow us to carry out the whole query process, as described in Section 3.1. The goal is to show the available trade-offs for the complete search process. Also, looking at the big picture will allow us to understand how the different position-lookup strategies affect the overall query-processing time. We study two scenarios: one in which snippet generation is needed, and another where snippets are not needed. We assume that the documents have been stored as proposed in Section 4.4, using the URL + association rules approach. The results for the unsorted document collection can be obtained from the results in Tables 1, 2, 3, and 4.

5.1. Scenario 1: Query Processing with Snippet Generation

The first scenario tests the three steps from Section 3.1 (i.e., query processing, positional ranking step, and snippet generation). Recall that the total space usage of the GOV2 inverted index is 9,739 MB, storing `docID`s (compressed with `PForDelta`) and frequencies (compressed with S16). For the query process, we set $k_1 \in \{50, 300\}$ and $k_2 \in \{10, 50\}$. We use the TREC 2006 query log. We test the following schemes, all of them including the inverted index to support Step 1:

- **Text compressors:** we test `lzma` and `lz4` compressors on the text, and use it to obtain positions and snippets. Document blocks are of size 200 KB, 500 KB and 1,000 KB.
- **Compressed data structures:** we try `WT`, `WT` + `lzma`, and `WT` + `lz4`, using `WT` blocks of size 1 KB, 2 KB and 7 KB.
- **Zero-order compressor:** we compress the text using VByte/VNibble. The two points in the trade-off are obtained by using VNibble compression for the text (the least space) and VByte compression for the text.
- **Compression boosters:** We try the schemes VByte + `lzma` and VByte + `lz4`, using document blocks of size 5 KB, 10 KB, 50 KB and 200 KB.
- **Positional Inverted Indexes:** The following schemes are the baseline for positional inverted indexes. Each curve in the plots will have three points, corresponding to the different compression schemes used for `PIL`s, namely PA Rice (the leftmost point in the corresponding curves), Rice (the center point), and S16 compression (the rightmost point). We tried the 3 most efficient approaches to compress the text for snippet extraction, namely `PIL` & `lz4` (using document blocks of 200 KB for `lz4`), `PIL` & VNibble, and `PIL` & VByte + `lz4` (using document blocks of size 50 KB for the compression booster).

5.1.1. DAAT AND Queries

We first test DAAT AND queries for Step 1, which correspond to a small fraction of the overall query processing time. Hence, the process used to obtain in-document positions should be efficient, and small differences among alternatives could influence the total processing time.

For $k_1 = 50$ and $k_2 = 10$ (Figure 5, upper left), it can be seen that schemes that use `PIL`s have a higher space usage. Some of these schemes offer a competitive query time, yet their space usage makes them unaffordable. Scheme `PIL` & VByte + `lzma` is the most space-efficient alternative using `PIL`s, yet it is slower and cannot compete. This shows in practice what we have suggested all along in this paper: state-of-the-art solutions based on `PIL`s have a higher space usage, as in this scenario they need to store the text to produce snippets.

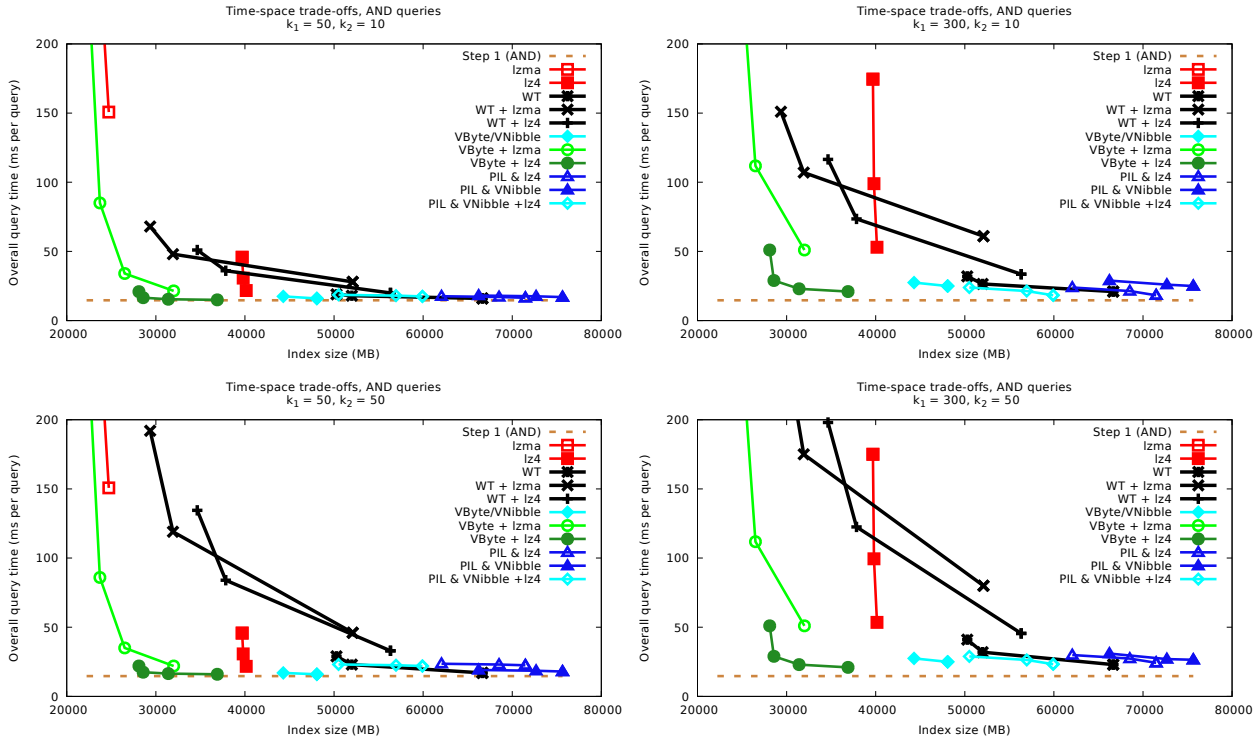

Figure 5: Time-space trade-offs for the overall query process for the GOV2 collection. With $k_1 = 50$ (left) and $k_1 = 300$ (right), and $k_2 \in \{10, 50\}$, including Step 1 (AND queries), Step 2, and Step 3.

Results for Compressed Data Structures. Scheme `WT` offers also a highly competitive query time, using less space than schemes based on `PIL`s. However, the space usage is still high. Alternatives `WT` + `lzma` and `WT` + `lz4` yield a significant decrease of about 42% in space usage. However, the resulting query time degrades quickly, and it becomes less competitive. It is important to recall that `WT`s by themselves can be used to support query processing [6]. Hence, we could get rid of the inverted index, saving considerable space: the total space usage would be about 19,628 MB (only `WT` (7 KB) + `lzma`), making it smaller than all the alternatives we tested. Query time, on the other hand, would increase considerably. Thus, this solution would only be attractive in cases where the available space is small.

Results for Zero-Order Compressors. Scheme VByte/VNibble yields an interesting trade-off, using less space than `WT` and achieving a highly competitive query time. Indeed, we have achieved a decrease of about 21% in space, compared to the fastest alternative based on `PIL`s (i.e., `PIL` & VByte + `lz4`). Besides being smaller, VByte/VNibble has a query time comparable to that of `PIL` & VByte + `lz4`. However, the space usage is still high compared to the alternatives we study next.

Results for Text Compressors. Scheme `lz4` improves the space usage, compared with VByte/VNibble, providing also a competitive query time. Scheme `lzma`, on the other hand, reduces the space usage significantly, yet at the price of a much slower (and impractical) query time. Actually, this is the smallest-space alternative we tested (although it is not fully shown in the figures). This achieves about 22,225 MB of space, including everything needed for query processing. This is only 1.01 times the space of `PIL` PA Rice (which uses 21,925 MB, see Table 2). However, query processing time increases significantly, to more than 400 msecs per query. This scheme could be useful in cases where the available memory is small, such that a larger index would mean accessing disk.

Results for Compression Boosters. When we add the compression boosters to `lzma` and `lz4` (i.e., schemes VByte + `lzma` and VByte + `lz4`), we obtain the most competitive trade-offs. For instance, compared to `PIL` + `lz4` (the fastest and among the most space-efficient alternatives based on `PIL`), VByte + `lz4` (50 KB block, the third point in the curve, from the right) achieves a reduction in space usage of about 49.81% (56,958 MB versus 28,584 MB), with a query time that is just 1.03 times slower (from 16.95 msecs per query to 17.49 msecs per query). This shows the effectiveness of this approach.

Moreover, recall that the positional index `PIL` PA Rice requires 21,925 MB of space (Table 2). Scheme VByte + `lz4`, on the other hand, requires 28,584 MB (including the basic `docID`/frequency non-positional inverted index). That is, in 1.30 times the space of only `PIL`s, we can store the inverted index (`docID`s and frequencies), the positional data, and the text collection. In other words, using only slightly more space than `PIL` PA Rice, scheme VByte + `lz4` includes everything needed for query processing. This is one of the most important results and conclusions in this paper: "not to index" positional data can be a highly competitive alternative in practical scenarios.

If we now compare `PIL` + `lz4` with VByte + `lzma` (for blocks of size 10 KB, the second point in the curve, from the right), we obtain a decrease in space usage of about 53.47% (56,958 MB versus 26,501 MB), while the query time is 1.99 times slower (from 17.47 msecs per query to 34.93 msecs per query). In other words, we are able to accommodate the complete index in 1.21 times the space of `PIL`s, and still are able to offer a (somewhat) competitive query time.

Results for $k_2 = 50$. In this case the conclusions are similar as before (see Figure 5). The only schemes that are affected (i.e., where query time is significantly increased) are those using `PIL`s and `WT`s. The former because more documents must be decompressed per query in order to obtain their snippets. The latter because we need to show snippets for more documents. This means that we need to obtain more words from the text, hence traversing the `WT` intensively. Schemes that use the compressed text for snippets and positions must decompress k_1 documents to obtain position data, to then extract snippets for the already decompressed top-k_2 documents (recall that $k_2 \leq k_1$).

5.1.2. BMW OR Queries

Figure 6 shows experimental results for BMW OR queries. The results obtained are slightly different to that for AND queries, because Step 1 of query processing is considerably slower. The aim is to illustrate that a slower Step 1 yields a smaller difference between schemes based on `PIL` and our approaches. For instance, for $k_1 = 50$ and $k_2 = 10$ we can conclude that scheme VByte + `lzma` with text blocks of size 10 KB uses 1.21 times the size of `PIL` PA Rice (which only stores positional data). The query time is 1.43 times slower than `PIL` & `lz4`. Scheme VByte + `lz4` uses just 1.30 times the space of `PIL`s, and achieves a query time that is 1.005 times slower than `PIL` & `lz4`. As in the previous section, we conclude that VByte + `lz4` and VByte + `lzma` (using text blocks of size 50 KB) offer very interesting trade-offs, allowing us to replace `PIL`s in many cases.

5.2. Scenario 2: Query Processing without Snippet Generation

Next, we test the performance in a scenario where text snippets are not needed. That is, Step 3 of Section 3.1 is not carried out. Thus, we now have just one scheme using `PIL`s. In our experiments we set $k_1 \in \{50, 300\}$. As in Scenario 1, all schemes include the inverted index.

Figure 7 (left side) shows results for AND queries. The most important result is that the scheme based on `PIL`s are not competitive in space usage, and provide no significant benefit in speed, over schemes such as VByte + `lz4` and VByte + `lzma`. That is, saving the space needed for the text collection is not enough to be competitive against our proposal. Thus, "not to index" positional data is even effective in scenarios where snippets are not needed. Figure 7 (right side) shows results for BMW OR queries. The conclusions are similar to the previous cases.

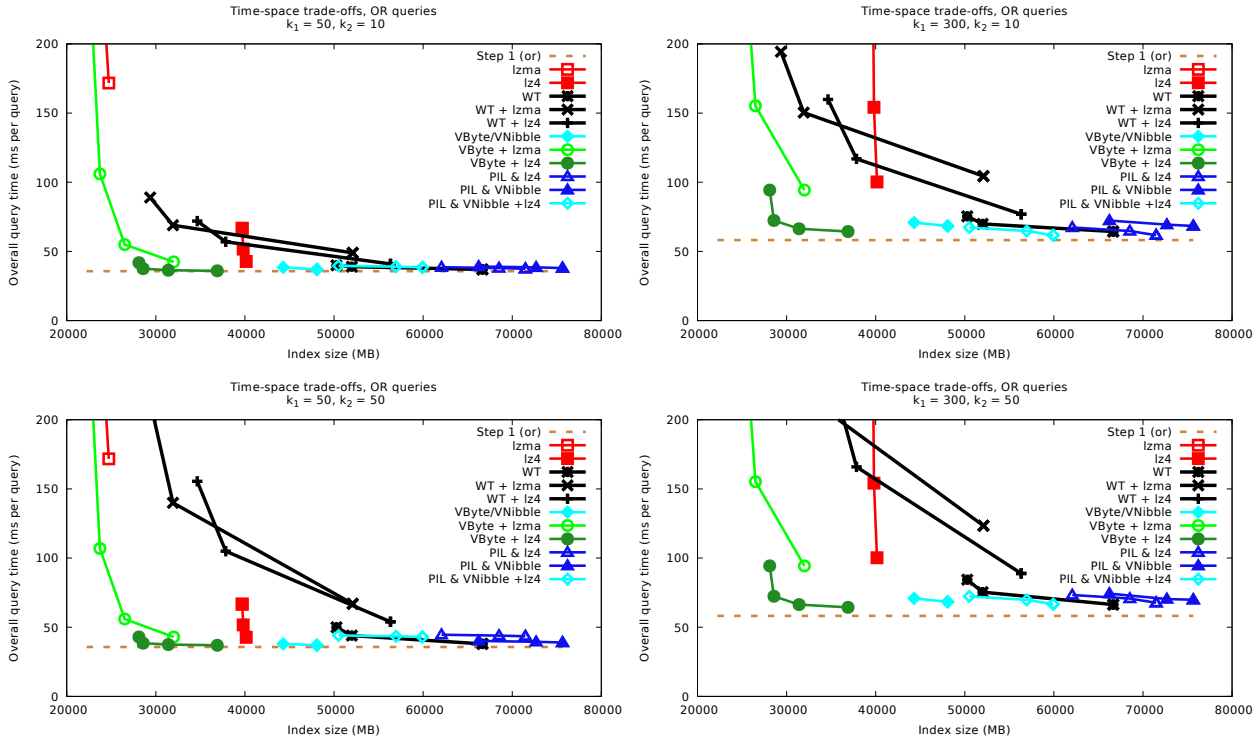

Figure 6: Time-space trade-offs for the overall query process on the GOV2 collection. With $k_1 = 50$ (left) and $k_2 = 300$ (right), and $k_2 \in \{10, 50\}$, including Step 1 (BMW OR queries), Step 2, and Step 3.

6. Conclusions

In this paper, we have demonstrated that, besides classical positional inverted indexes, there exists a wide range of practical time-space trade-offs for implementing positional ranking functions. We studied several alternatives, trying to answer the question of whether it is necessary to index position data or not. As one of the most interesting trade-offs, we propose a compressed document representation based on the approach from Turpin at al. [54], which we combined with `lz4` compression. This allows us to compute position and snippet data using less space than a standard positional inverted index that only stores position data. In particular, using about 1.30 times the size of a positional inverted index, which only stores position information, we can have everything needed for efficient query processing, namely inverted index (storing `docID`s and frequencies), position information, and document collection (for snippet generation).

This means that in many practical cases, "not to index" position data may be the most efficient approach. This provides new practical alternatives for positional index compression, a problem that has been considered difficult to further improve in previous work [59, 25]. Finally, we also showed that compressed data structures such as wavelet trees [28] can be competitive with the best solutions in some scenarios.

Another scenario where document-compression schemes can potentially outperform positional indexes is that of dynamic text collections. That is, documents can be added or deleted. A scheme that does not index positions can handle these operations more efficiently than traditional positional indexes, e.g., schemes based on `lz4` compression (with compression boosting), where new documents can be easily appended. Deletions should be handled more carefully, however they can be made efficient using standard techniques.

Our study was carried out assuming that there is enough main memory to store the inverted index, the positional inverted index, and the corresponding document collection (in compressed form) completely in main memory. In a scenario where there is no main-memory space to keep all the involved data structures, indexes must reside on secondary storage (e.g., on SSD memory or disk), and parts of them must be brought

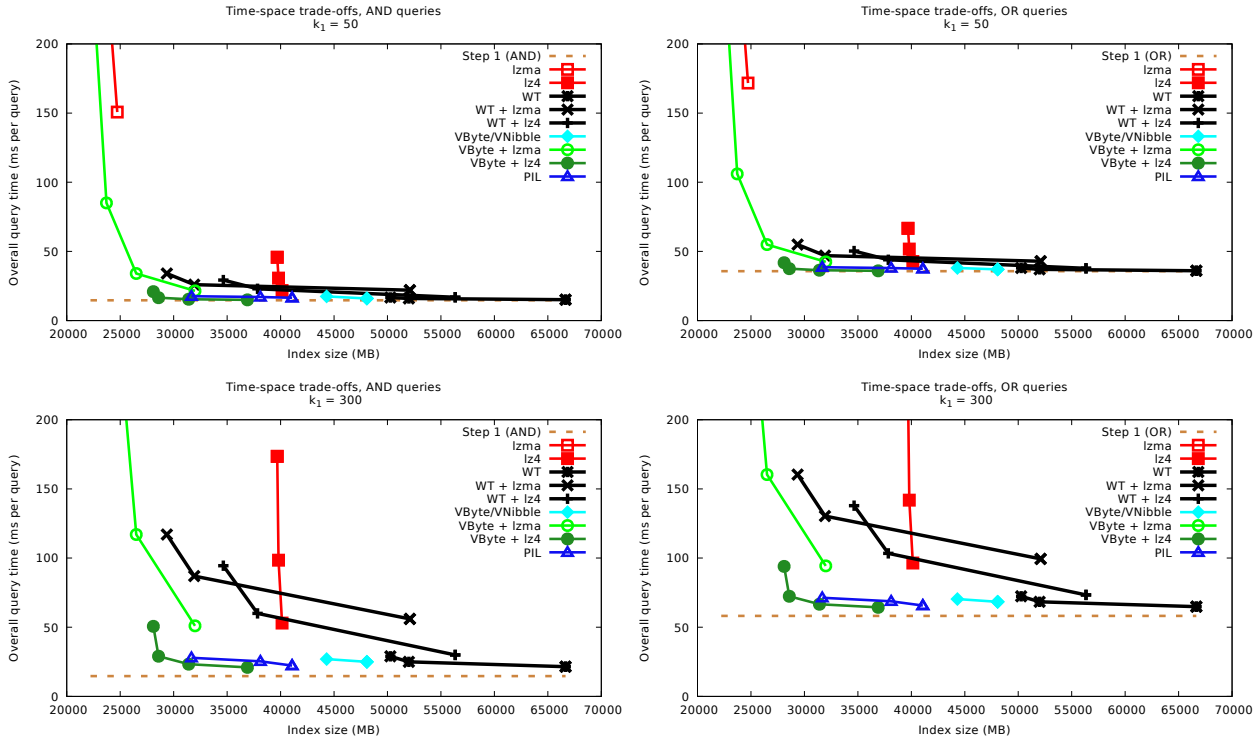

Figure 7: Time-space trade-offs for the overall query process on the GOV2 collection. With $k_1 \in \{50, 300\}$, including Step 1 (AND queries on the left, BMW OR queries on the right), and Step 2.

to main memory on demand. We think that the approach using VByte along with `lz4` compression has some advantages in this case. First, since positions are obtained directly from the text, we only need to perform secondary-memory accesses during Step 2 of the query process. Positional inverted indexes, on the other hand, must access secondary storage during Steps 2 and 3, which could yield a higher number of accesses. In particular, if we consider just Step 2, when fetching position information for k_1 candidate documents for a t-term query, a document-compression approach performs k_1 secondary-storage seeks. An approach based on positional indexes, on the other hand, will perform up to $t \cdot k_1$ seeks, assuming that all the occurrences of a query term in a document can be obtained with a single seek from a positional index. Second, the document reorganization approach that we propose in Section 4.4 helps in reducing the number of document blocks that must be accessed at query time, improving the locality of access.

Finally, we think that our approaches can be used to efficiently support phrase queries. Positional inverted indexes need to perform a number of operations that make phrase queries expensive, whereas using the text to search for the phrase terms could be more efficient. This topic needs further research, which we defer as future work.

Acknowledgments

Diego Arroyuelo partially supported by Fondecyt Grant 11121556 and Millennium Institute for Foundational Research on Data (IMFD). Senén González was partially supported by the **Austrian Science Fund (FWF):[I2420-N31]**, project: *Higher-Order Logics and Structures*, and by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH. Mauricio Marin partially supported by CeBiB FB0001 Conicyt. Torsten Suel supported by NFS Grants IIS-0803605 and IIS-1117829. Luis Valenzuela partially funded by Fondecyt Grant 11121556.

References

[1] Agrawal, R., Imielinski, T., Swami, A., 1993. Mining association rules between sets of items in large databases. In: Proc. of ACM SIGMOD International Conference on Management of Data. ACM Press, pp. 207–216.

[2] Agrawal, R., Srikant, R., 1994. Fast algorithms for mining association rules in large databases. In: Proc. of 20th International Conference on Very Large Data Bases (VLDB). Morgan Kaufmann, pp. 487–499.

[3] Anh, V. N., Moffat, A., 1998. Compressed inverted files with reduced decoding overheads. In: Proc. of 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 290–297.

[4] Anh, V. N., Moffat, A., 2005. Inverted index compression using word-aligned binary codes. Information Retrieval 8 (1), 151–166.

[5] Arroyuelo, D., Gil-Costa, V., González, S., Marin, M., Oyarzún, M., 2012. Distributed search based on self-indexed compressed text. Information Processing and Management 48 (5), 819–827.

[6] Arroyuelo, D., González, S., Oyarzún, M., 2010. Compressed self-indices supporting conjunctive queries on document collections. In: Proc. 17th International Symposium on String Processing and Information Retrieval. LNCS 6393. Springer, pp. 43–54.

[7] Asadi, N., Lin, J., 2013. Document vector representations for feature extraction in multi-stage document ranking. Information Retrieval 16 (6), 747–768.

[8] Baeza-Yates, R., Ribeiro-Neto, B., 2011. Modern Information Retrieval - the Concepts and Technology Behind Search, Second Edition. Pearson Education Ltd., Harlow, England.

[9] Barla Cambazoglu, B., Baeza-Yates, R., 2015. Scalability Challenges in Web Search Engines. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers.

[10] Bast, H., Celikik, M., 2014. Efficient index-based snippet generation. ACM Transactions on Information Systems 32 (2), 6:1–6:24.

[11] Brin, S., Page, L., 1998. The anatomy of a large-scale hypertextual web search engine. Journal of Computer Networks 30 (1–7), 107–117.

[12] Brisaboa, N., Cerdeira, A., Navarro, G., Pedreira, O., 2012. Ranked document retrieval in (almost) no space. In: Proc. 19th International Symposium on String Processing and Information Retrieval. LNCS 7608. Springer, pp. 155–160.

[13] Brisaboa, N., Fariña, A., Ladra, S., Navarro, G., 2012. Implicit indexing of natural language text by reorganizing bytecodes. Information Retrieval 15 (6), 527–557.

[14] Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J., 2003. Efficient query evaluation using a two-level retrieval process. In: Proc. 12th International Conference on Information and Knowledge Management (CIKM). ACM Press, pp. 426–434.

[15] Büttcher, S., Clarke, C., Cormack, G., 2010. Information Retrieval: Implementing and Evaluating Search Engines. MIT Press.

[16] Büttcher, S., Clarke, C., Lushman, B., 2006. Term proximity scoring for ad-hoc retrieval on very large text collections. In: Proc. of 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 621–622.

[17] Clarke, C., Cormack, G., Burkowski, F., 1995. An algebra for structured text search and a framework for its implementation. Computer Journal 38 (1), 43–56.

[18] Claude, F., Navarro, G., 2008. Practical rank/select queries over arbitrary sequences. In: Proc. 15th International Symposium on String Processing and Information Retrieval. LNCS 5280. Springer, pp. 176–187.

[19] Dean, J., 2009. Challenges in building large-scale information retrieval systems: invited talk. In: Proc. of 2nd International Conference on Web Search and Web Data Mining (WSDM). p. 1.

[20] Ding, S., Suel, T., 2011. Faster top-k document retrieval using block-max indexes. In: Proc. of 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 993–1002.

[21] Elias, P., 1975. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory 21 (2), 194–203.

[22] Fariña, A., Brisaboa, N., Navarro, G., Claude, F., Places, A., Rodríguez, E., 2012. Word-based self-indexes for natural language text. ACM Transactions on Information Systems 30 (1), article 1.

[23] Fariña, A., Navarro, G., Paramá, J., 2012. Boosting text compression with word-based statistical encoding. Computer Journal 55 (1), 111–131.

[24] Ferragina, P., Giancarlo, R., Manzini, G., 2009. The myriad virtues of wavelet trees. Information and Computation 207 (8), 849–866.

[25] Ferragina, P., Manzini, G., 2010. On compressing the textual web. In: Proc. of 3rd International Conference on Web Search and Web Data Mining (WSDM). ACM, pp. 391–400.

[26] Golomb, S., 1966. Run-length encoding. IEEE Transactions on Information Theory 12 (3), 399–401.

[27] Goodwin, B., Hopcroft, M., Luu, D., Clemmer, A., Curmei, M., Elnikety, S., He, Y., 2017. Bitfunnel: Revisiting signatures for search. In: Proc. of 40th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 605–614.

[28] Grossi, R., Gupta, A., Vitter, J. S., 2003. High-order entropy-compressed text indexes. In: Proc. of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). ACM/SIAM, pp. 841–850.

[29] He, J., Suel, T., 2012. Optimizing positional index structures for versioned document collections. In: Proc. of 35th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 245–254.

[30] Huston, S., Croft, W. B., 2014. A comparison of retrieval models using term dependencies. In: Proc. 23rd International Conference on Information and Knowledge Management (CIKM). pp. 111–120.

[31] Kleinberg, J. M., 1999. Authoritative sources in a hyperlinked environment. Journal of the ACM 46 (5), 604–632.
[32] Lemire, D., Boytsov, L., 2015. Decoding billions of integers per second through vectorization. Software, Practice and Experience 45 (1), 1–29.
[33] Manning, C., Raghavan, P., Schütze, H., 2008. Introduction to Information Retrieval. Cambridge University Press.
[34] Manzini, G., 2001. An analysis of the Burrows-Wheeler transform. Journal of the ACM 48 (3), 407–430.
[35] Metzler, D., Croft, W. B., 2005. A Markov random field model for term dependencies. In: Proc. of 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 472–479.
[36] Mishne, G., Rijke, M., 2005. Boosting web retrieval through query operations. In: Proc. of 27th European Conference on IR Research. LNCS 3408. Springer, pp. 502–516.
[37] Moffat, A., 1989. Word-based text compression. Software, Practice, and Experience 19 (2), 185–198.
[38] Moffat, A., Stuiver, L., 2000. Binary interpolative coding for effective index compression. Information Retrieval 3 (1), 25–47.
[39] Navarro, G., 2014. Wavelet trees for all. Journal of Discrete Algorithms 25, 2–20.
[40] Navarro, G., 2016. Compact Data Structures - A Practical Approach. Cambridge University Press.
[41] Navarro, G., Mäkinen, V., 2007. Compressed full-text indexes. ACM Computing Surveys 39 (1).
[42] Ottaviano, G., Venturini, R., 2014. Partitioned elias-fano indexes. In: Proc. of 37th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 273–282.
[43] Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., Johnson, D., 2005. Terrier information retrieval platform. In: Proc. of 27th European Conference on IR Research. LNCS 3408. Springer, pp. 517–519.
[44] Panev, K., Berberich, K., 2014. Phrase queries with inverted + direct indexes. In: Proc. 15th International Conference on Web Information Systems Engineering (WISE). LNCS 8786. pp. 156–169.
[45] Procházka, P., Holub, J., 2017. Towards efficient positional inverted index. Algorithms 10 (1), 30.
[46] Rasolofo, Y., Savoy, J., 2003. Term proximity scoring for keyword-based retrieval systems. In: Proc. of 25th European Conference on IR Research. LNCS 2633. Springer, pp. 207–218.
[47] Schenkel, R., Broschart, A., Hwang, S., Theobald, M., Weikum, G., 2007. Efficient text proximity search. In: Proc. of 14th String Processing and Information Retrieval. LNCS 4726. Springer, pp. 287–299.
[48] Shan, D., Zhao, W. X., He, J., Yan, R., Yan, H., Li, X., 2011. Efficient phrase querying with flat position index. In: Proc. of 20th ACM Conference on Information and Knowledge Management (CIKM). ACM, pp. 2001–2004.
[49] Silvestri, F., 2007. Sorting out the document identifier assignment problem. In: Proc. of 29th European Conference on IR Research. LNCS 4425. Springer, pp. 101–112.
[50] Sparrow, B., Liu, J., Wegner, M., 2011. Google effects on memory: Cognitive consequences of having information at our fingerprints. Science 333 (6043), 776–778.
[51] Tao, T., Zhai, C., 2007. An exploration of proximity measures in information retrieval. In: Proc. of 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 295–302.
[52] Tombros, A., Sanderson, M., 1998. Advantages of query biased summaries in information retrieval. In: Proc. of 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 2–10.
[53] Tsegay, Y., Puglisi, S., Turpin, A., Zobel, J., 2009. Document compaction for efficient query biased snippet generation. In: Proc. of 31th European Conference on IR Research. LNCS 5478. Springer, pp. 509–520.
[54] Turpin, A., Tsegay, Y., Hawking, D., Williams, H., 2007. Fast generation of result snippets in web search. In: Proc. of 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 127–134.
[55] Vigna, S., 2013. Quasi-succinct indices. In: Proc. 6th ACM International Conference on Web Search and Data Mining (WSDM). pp. 83–92.
[56] Wang, L., Lin, J. J., Metzler, D., 2011. A cascade ranking model for efficient ranked retrieval. In: Proc. of 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 105–114.
[57] Williams, H., Zobel, J., 1999. Compressing integers for fast file access. Computer Journal 42 (3), 193–201.
[58] Witten, I. H., Moffat, A., Bell, T. C., 1999. Managing Gigabytes: Compressing and Indexing Documents and Images (2nd ed.). Morgan Kaufmann Publishing.
[59] Yan, H., Ding, S., Suel, T., 2009. Compressing term positions in web indexes. In: Proc. of 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 147–154.
[60] Yan, H., Ding, S., Suel, T., 2009. Inverted index compression and query processing with optimized document ordering. In: Proc. of 18th International Conference on World Wide Web (WWW). ACM, pp. 401–410.
[61] Ziv, J., Lempel, A., 1977. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23 (3), 337–343.
[62] Zukowski, M., Héman, S., Nes, N., Boncz, P., 2006. Super-scalar RAM-CPU cache compression. In: Proc. of 22nd International Conference on Data Engineering (ICDE). IEEE Computer Society, p. 59.