To Index or not to Index: Time-Space Trade-Offs in Search Engines with Positional Ranking Functions

Diego Arroyuelo*
Dept. of Informatics,
Univ. Técnica F. Santa María
Yahoo! Labs Santiago, Chile
darroyue@inf.utfsm.cl

Senén González†
University of Chile.
Yahoo! Labs Santiago, Chile
sgonzale@dcc.uchile.cl

Mauricio Marin‡
University of Santiago, Chile
Yahoo! Labs Santiago, Chile
mmarin@yahoo-inc.com

Mauricio Oyarzún
University of Santiago, Chile
Yahoo! Labs Santiago, Chile
mauricio.silvaoy@usach.cl

Torsten Suel§
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
suel@poly.edu

ABSTRACT

Positional ranking functions, widely used in web search engines, 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 one should index positional data or not.

We show that there is a wide range of practical time-space trade-offs. Moreover, we show that both position and textual data can be stored using about 71% of the space used by traditional positional indexes, with a minor increase in query time. This yields considerable space savings and outperforms, both in space and time, recent alternatives from the 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.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2.4 [**Systems**]: Textual databases

General Terms

Algorithms, Experimentation, Performance

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 even the way in which people think and remember things [35]. Indeed, *web search engines* are one of the most important tools that give access to the huge amount of information stored in the web. The success of a web 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 [6, 10, 25]. State-of-the-art ranking functions, on the other hand, combine simple term-based ranking schemes such as BM25 [10], link-based methods such as Pagerank [7] or Hits [24], and up to several hundred other features derived from documents and search query logs.

Recent work has focused on *positional ranking functions* [32, 27, 36, 28, 33, 40, 10] that improve result quality by considering the positions of the query terms in the documents. Thus, documents where the query terms occur close to each other might be ranked higher, as this could indicate that the document is highly relevant for the query. To support such positional ranking, the search engine must have access to the position data. This is commonly done by building an index for all term positions within documents, called a *positional index*. The goal is to obtain an index that is efficient in terms of both index size and access time.

As shown in [32], positional ranking can be carried out in two phases. First, 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. In the second phase, the term positions are used to rerank these documents by refining their score values. (Additional third or fourth phases may be used to do further reranking according to hundreds of additional features [38], but

*Partially funded by Fondecyt Grant 1-110066.
†CONICYT Thesis Work Support Code 78100003.
‡Partially supported by FONDEF D09I1185.
§Supported by NFS Grants IIS-0803605 and IIS-1117829.

SIGIR'12, August 12–16, 2012, Portland, Oregon, USA.

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 return as part of the result page. This requires access to the actual text in the indexed web pages. It is well known [40, 21] that storing position data requires a considerable amount of space, typically about 3 to 5 times the space of an inverted index storing only document identifiers and term frequencies. Furthermore, storing the documents for snippet generation requires significant additional space.

This paper focuses on alternative approaches to performing the above two-step document ranking process and the query snippet-generation phase. The aim is to optimize both space and query processing time. One important feature of position data is that it only needs to be accessed for a limited number of promising documents, say a few dozens or hundreds of documents. This access pattern differs from that for document identifiers and term frequencies, which are accessed more frequently, making access speed much more important. For position data, on the other hand, we could consider somewhat slower but smarter alternative representations without losing too much efficiency at query time [40].

In this paper, we push this idea further and 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 to compressing the text collection: (1) *wavelet trees* [23], which are succinct data structures from the combinatorial pattern matching community, and (2) 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 [21]. More importantly, these representations can be used for both positional reranking and snippet generation. One concern is how these alternatives impact query processing speed, and thus we will study the resulting trade-offs between running time and space requirement.

Thus, *to index or not to index position data*, that is the research question that we hope to answer in this paper. To our knowledge, such alternative approaches for implementing positional ranking functions have not been rigorously compared before. Our main result 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. Thus, we provide new alternatives for practical compression of position and text data, outperforming recent approaches in [34].

Following current practice in search engines [21, 14], 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 [21, 14].

2. BACKGROUND AND RELATED WORK

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 from a vocabulary $\Sigma = \{1, \ldots, V\}$. Notice that every term is represented by an integer, hence the documents are just arrays of integers. We also identify each document D_i with a unique document identifier (docID) i. 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 $\{j | D_i[j] = t\}$.

Throughout this paper, we assume that all term separators (like spaces, '**,**', '**;**', '**.**', etc.) have been removed from the text. Also, we assume that all terms in the vocabulary have been represented in a case-insensitive way. This is in order to facilitate the search operations that we need to carry out over the documents in order to compute (on the fly) the positions of a given query term. To be able to retrieve the original text (with separators and the original case) one can use the *presentation layer* introduced by Fariña et al. [17, 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 to obtain the original text. However, this scheme must be used on all the alternatives that we consider in this paper, and thus we disregard the overhead introduced by the presentation layer and focus only on the low-level details of compression (but keeping in mind that the original text can still be retrieved).

2.1 Inverted Index Compression

The efficiency of query processing in search engines relies on *inverted indexes*. These data structures store a set of *inverted lists* $I_1, \ldots, I_V$, which are accessed through a *vocabulary table*. The list I_t maintains a *posting* for each document containing the term $t \in \Sigma$. Usually, a posting in an inverted list consists of a docID, a term frequency, and the in-document positions of the term. (In real systems, the docIDs, term frequencies and in-document positions are often stored separately.) Indexes whose postings store in-document positions are called *positional inverted indexes*.

We assume that an inverted list I_t is divided into blocks of 128 documents each — the particular choice of 128 documents per block is an implementation issue. Given a block of I_t, the term-position data for all the documents in that block are stored in a *separate* block of variable size. The inverted lists of the query terms are used to produce the result for the query. Since the query results are usually large, the result set must be ranked by relevance.

For large document collections, the data stored in inverted indexes requires considerable amounts of space. Hence, the indexes must be compressed. To support efficient query processing (such as DAAT [10], WAND [9] or BMW OR [15]) and effective compression in the inverted lists, we sort them by increasing docID. Let $d_t[1..|I_t|]$ denote the sorted list of docIDs 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, every f_i is replaced with $f_i - 1$, since $f_i > 0$ always holds. For the positions, each $p_{i,j}$ is replaced with $p_{i,j} - p_{i,j-1} - 1$. 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 docIDs and frequencies, with many compression methods available [41, 10]. Some of them achieve, in general, a very good compression ratio, but at the expense of a lower decompression speed [10], for example Elias γ and δ encodings

[16], or Golomb/Rice parametric encodings [22], interpolative encoding [30]. Other methods achieve a (slightly) lower compression ratio, though with much higher decompression speed, for example VByte [39], S9 [3], and PForDelta [42], among others [10]. The best compression method depends on the scenario at hand.

2.2 Positional Indexes

Unfortunately, the scenario is not the same for compressing term positions, which is a problem where it has been difficult to make much progress. For instance, previous work [40] concludes that term positions in the documents do not follow simple distributions that could be used to improve compression (as is the case of, for instance, docIDs 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 [40] 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.

Positions in inverted indexes are used mainly in two applications, phrase searching and positional ranking schemes. In this paper we study positional ranking, 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 close together 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 allow for phrase searching as well. This scenario is left for future work.

A recent work on positional indexing is that of Shan et al [34]. They propose to use the *flat position indexes* [11, 14] as an alternative of positional inverted indexes. The result is that docIDs, term frequencies and position data can be stored in space close to that of positional inverted lists, yielding a reduction of space usage. However, this index does not store the text, which makes it less suitable in scenarios where text snippets must be generated.

2.3 Snippet Generation

Besides providing a ranking of the most relevant documents for a query, search engines must show query snippets and support accessing the "in-cache" version of the documents. Each snippet shows a portion of the result document, in order to help the user judge its likely relevance before accessing it. Turpin et al. [37] introduce a method to compress the text collection and support fast text extraction to generate snippets. However, to achieve fast extraction, they must use a compression scheme that uses more space than usual compressors. In a more recent work, Ferragina and Manzini [21] 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. One of their main concerns was how compressors can capture redundancies that arise very far apart in very long texts. Their results show that such large texts can often be compressed to just 5% of their original size.

2.4 Compressed Text Self-Indexes

Succinct or *compressed* data structures use as little space as possible to support operations as efficiently as possible. 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. A *compressed self-index* is a data structure that represents a text in compressed space, supports indexed search capabilities on the text, and is able to obtain any text substring efficiently [31]. It can be seen as compression tools with indexed search capabilities.

Given a sequence $T[1..n]$ 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. A *wavelet tree* [23] (`WT` for short) is a succinct data structure that supports rank and select operations, among many virtues [19].

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^v is divided at node v 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 $B^v[l] = \mathtt{1}$ if $T^v[l] \in \Sigma^{v_r}$, and $B^v[l] = \mathtt{0}$ otherwise. Given a `WT` node v of depth i, $B^v[j] = \mathtt{1}$ 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 [31]. 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 [23, 31].

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 $B^{root}[i] = \mathtt{0}$. 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 $\mathsf{rank}_\mathtt{0}(B^{root}, i)$. Otherwise, we move to v_r looking for the symbol at position $\mathsf{rank}_\mathtt{1}(B^{root}, i)$. 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 the upward path from the leaf corresponding to term c. Both operations can be implemented in $O(\log V)$ time; see [31] for details.

The space required by a `WT` is, in practice, about 1.1–1.2 times the space of the text [12]. In our application to IR, 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* [29]) and use these codes to determine the corresponding tree leaf for each term. Hence, the tree is not balanced anymore, but has a Huffman tree shape [12] such that frequent terms will be closer to the tree root than less frequent ones. This achieves a total space of $n(H_0(T)+1) + o(n(H_0(T)+1))$ bits, where $H_0(T) \leq \log V$ is the zero-order

empirical entropy of T [26]. In practice, the space is about 0.6 to 0.7 times the text size [12]. 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 $O(V)$.

2.5 Self-Indexes for IR Applications

There have been some recent attempts to apply alternative indexing techniques, such as self-indexes, in large-scale IR systems. In particular, we mention the work by Brisaboa et al. [8] and Arroyuelo et al. [5, 4]. The former [8] 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 [5, 4] extends [8] by supporting more IR-like operations on a `WT`. 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. The experimental results in [5] compare with an inverted index storing just docIDs, which of course yields a smaller index. However, `WT`s also store extra information, such as the term frequencies and, most important for us here, the compressed text and thus the term-position data.

Recent work [21] also tried to use (among other alternatives) a compressed self-index to compress web-scale texts, in order to allow decompression of arbitrary documents. Their conclusion is that compressed self-indexes still need a lot of progress in order to be competitive with standard compression tools, both in compression ratio and decompression speed. A contribution of our present work is a compressed self-index that is able to store web-scale texts and is competitive with the best state-of-the-art compressors. We think that this is a step forward in closing the gap between theory and practice in this area [20].

3. CONTRIBUTIONS

In this paper we study what are the best ways to organize in-document positions and textual data, in order to efficiently support positional ranking and snippet generation in text search engines. One of our main conclusions is that some compressed representations of the textual data — which are needed to support snippet generation — can also be used to efficiently obtain the term positions needed by positional ranking methods. Therefore, *no positional index is needed in many cases*, thus saving considerable space at little cost in terms of running time.

Our main contributions can be summarized as follows:

1. A study of several trade-offs for compressing position data. Rather than storing a positional index, we propose to compute the term positions from a compressed representation of the text. We explore and propose several compression alternatives. Our results significantly enhance current trade-offs between running time and memory space usage, enabling in this way more design choices for web search engines. One of our most interesting results is that both position and textual data can be stored in about 71% the space of current positional inverted indexes.

2. A study of several alternatives for compressing textual data, extending the alternatives studied in previous work [21]. In particular, we show that using the scheme in [37] (to compress text and support efficient snippet generation) before using a standard compressor yields good time-space trade-offs, extending the alternatives introduced in [18]. It is important to note that variants of the scheme in [37] have been adopted by some commercial search engines, which makes our results of practical interest.

3. We propose several improvements over wavelet trees [23], in order to make them competitive for representing document collections. We show how to improve the compression ratio by compressing the sequence associated to every node of the tree with standard compressors. The result is a practical web-scale compressed self-index that is competitive with the best state-of-the-art compressors.

4. COMPRESSING TERM-POSITION AND TEXTUAL DATA

Compressing in-document positions, i.e., the positions where each term occur within a document, has been recognized as a difficult task [40, 21]. Indeed, positions have become a bottleneck for compression compared to docIDs and frequencies. Moreover, recent work shows that the textual data can be compressed better than positions [21]. There are two main reasons for this. First, positions have a different distribution than docIDs and frequencies [40]. Second, since positions are stored separately for each term (recall Section 2.1), the local context for terms that is exploited by text compression schemes is not available in the positional inverted lists. Usually, positions require about 3 to 5 times the space of an inverted index storing docIDs and frequencies. Thus, efficient compression of in-document positions is an important challenge.

Positional inverted indexes are the standard solution to this problem [25, 10, 6]. In particular, [40] shows a detailed study of alternative ways to compress positional inverted indexes. However, it is not clear that using the methods in [40] is the best one can do. Notice that the in-document position data can be obtained (at query time) by searching for the query terms in the documents — a simple scan of the document suffices. Since textual data can be compressed much better than positions, this could decrease the space usage of positions. However, the question is how this impacts query performance. We investigate this issue in this paper. We assume that positions are used to support *positional ranking* as described in [10, 40].

Another important issue in web search engines is the ability to generate snippets for the query results that allow users to decide which of the result documents they should visit. In this context, snippets have been shown to improve the effectiveness of search engines. To provide snippets, a search engine must store a (simplified) version of the documents in the collection. In the case of web search engines, this means the entire textual web, which requires a considerable amount of resources. Thus, the textual data must be compressed [21].

4.1 Basic Query Processing Steps for Positional Ranking and Snippet Extraction

From now on we assume a search engine where positional ranking is used to score documents, and where snippets must

be generated for the top-scoring documents. Thus, solving a query involves the following steps:

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

2. **Positional Ranking Step:** Given the top-k_1 documents from the previous step, get the positions of the query terms within these documents. Then rerank the results using a positional ranking functions [10, 40].

3. **Snippet Generation Step:** After the re-ranking of previous step, get snippets of length s for the top-k_2 documents, for a given $k_2 \leq k_1$.

For instance, $k_1 = 200$ (as in [40]) and $k_2 = 10$ (as in most commercial search engines) are typical values for them. 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. Step 2 is usually supported by a positional inverted index [25, 10, 40]. Step 3 is supported by compressing the document collection and supporting the extraction of arbitrary documents. Our focus here is on alternative ways to implement the last two steps.

4.2 The Baseline: Positional Inverted Lists and Compressed Textual Data

This section describes and evaluates baseline approaches to support term positions indexing and snippet extraction.

4.2.1 Positional Inverted Lists

Positional inverted lists (`PILs`, for short) are the standard approach for indexing in-document position data in search engines [25, 10, 6]. In particular, we assume the representation explained in Section 2.1. To obtain position data at query time, given the docIDs of the top-k_1 results for a given query, we identify the PIL blocks containing the desired positional index entries. Then these blocks are *fully decompressed*, and the corresponding positions are obtained. A drawback here is that we need to decompress the entire `PIL` block, even if we only need a single entry in it. Thus, we might end up decompressing, in the worst case, k_1 blocks in each of the inverted lists involved in the query. Afterwards, these positions are used to rerank the top-k_1 documents, as in [40].

The access pattern for position data is much sparser than that for docIDs and frequencies, since positions must be obtained only for the top-k_1 documents. Thus, just a few positions are decompressed from the `PIL` in each query. Given this sparse access pattern and the high space requirement of positions (as discussed above), it is better to use compression methods with a good compression ratio, like Golomb/Rice compression. These are slower to decompress, yet the fact that only a few positions are decompressed should not impact in the overall query running time.

4.2.2 Compressed Textual Data

To compress the text collection and support decompressing arbitrary documents, a simple alternative that is used by several state-of-the-art search engines — for instance Lucene [13] — is to divide the whole collection into smaller text blocks, which are then compressed separately. The block size offers a time-space trade-off: larger blocks yield better compression, although decompression time is increased. Given the popularity [13, 21] and simplicity of this approach, we use it as the baseline for the compressed text.

4.2.3 Baseline Experiments

Experimental Setup.

We show now experiments for the baseline approaches. For this we use 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.

We use the TREC GOV2 collection, with about 25.2 million documents and about 32.86 million terms in the vocabulary. We work just with the text content of the collection (that is, we ignore the html code from the documents). This requires about 127GB 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 query log provided with the TREC GOV2 collection. All methods were implemented using C++, and compiled with `g++ 4.4.5`, with the full optimization flag `-O5`.

Experiments for Step 1.

In Table 1 we show the average query time (in milliseconds per query) for the initial query processing step (Step 1 of Section 4.1). We show results for two types of queries: traditional AND queries (using DAAT query processing and BM25 ranking) and the BMW OR approach from [15], which is one of the most efficient current solutions for disjunctive queries (using a two-layer approach, which yields slightly faster query times [15]). The index for docIDs and frequencies required 9,739 MB of space, using PForDelta compression for docIDs and S16 for frequencies. Notice that the

Table 1: Experimental results for the initial query processing step (Step 1) for AND and OR queries.

top-k_1	DAAT AND (ms/q)	BMW OR [15] (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

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 docIDs and the DAAT process. BMW OR, on the other hand, is an early-termination technique, and thus k_1 impacts the query time.

Experiments for Step 2.

In Table 2 (first two rows) we show experimental results for obtaining positions with the baseline `PILs`, using two compression schemes: Rice and S16, which offer the most interesting trade-offs [40]. We also show query times for different values of k_1, namely 50, 100, 150, 200 and 300 (the experiments in [40] only use $k_1 = 200$). As we can see, Rice

requires only about 90% the space of S16, but takes twice as much time. Comparing the query times of Step 2 for Rice and S16 with the query times of Step 1 in Table 1, we can see that position extraction is a small fraction of the overall time. Hence, we can use Rice to compress PILs and obtain a better space usage with only a minor increase in query time. For Rice, PILs use 2.91 times the space of the inverted index that stores docIDs and frequencies. For S16, this number is 3.22.

Experiments for Step 3.

Table 3 shows experimental results for the baseline for compressed textual data. Just as in [21], we divide the text into blocks of 0.2MB, 0.5MB or 1.0MB, and compress each block using different standard text compression tools. In particular, we show results for `lzma` (which gives very good results in [21]) and Google's `snappy` compressor [1], which is an LZ77 compressor that is optimized for speed rather than compression ratio. These two compressors offer the most interesting trade-offs among the alternatives we tried. As it can be seen, `lzma` achieves much better compression ratios than `snappy`. The compressed space achieved for the whole text is 8,133 MB for `lzma` and 27,388 MB for `snappy`.

The differences in extraction time are also considerable, with `snappy` being much faster. Note that [21] reports a decompression speed of about 35MB/sec for `lzma`. However, to obtain a given document we must first decompress the entire block that contains it. Hence, most of the 35MB per second do not correspond to any useful data. In other words, this does not measure effective decompression speed for our scenario, and thus we report per-query times rather than MB/s for both methods.

4.3 Computing Positions and Snippets from the Compressed Document Collection

We explore next the alternative of obtaining position data directly from the compressed text. This implies that in Step 2 of the query process, k_1 documents must be decompressed, rather than only k_2 in Step 3, as in the baseline.

Using Standard Text Compressors.

Our first approach is to obtain positions using the baseline for generating snippets from Section 4.2.2. In rows 3 and 4 of Table 2 we show the time-space trade-offs for this approach, using `lzma` and `snappy` compressors, and blocks of size 0.2 MB. We conclude that using `lzma` we can store positions and text in about half the space of `PIL`(the latter just storing positions). However, this approach is two orders of magnitude slower than the positional index. If we use `snappy` instead, we obtain an index that is 21.86% larger than `PIL` (Rice), and running times to obtain positions that are about an order of magnitude slower (this might be acceptable in some cases). In the following, we try to improve on both of these techniques.

Zero-order Compressors with Fast Text Extraction.

An alternative to compressing the text that could support faster position lookups is the approach from Turpin et al. [37]. The idea is to first 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 [2]. Note that the 128 most frequent terms in the collection are thus encoded in a single byte. Actually, [37] 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.

By using an integer compression scheme (such as VByte) for the text, we are able to decompress *any text portion* very efficiently (no text blocks are needed this time). Table 2 shows the resulting trade-offs for this alternative (see the row for "VByte"). Notice that we improve the query time significantly, making it competitive with PILs. The higher space usage is a concern, but note 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 3 bits of each nibble are used to represent the number. The results of Table 2 show space savings of about 10% over VByte. Also, notice that we are now able to use space close to that of `snappy` (with blocks of 0.2 MB), yet with a better query time.

The fast query time is due to two facts. First, methods like VByte and VNibble are able to decompress hundred of million integers (which in our case correspond to terms) per second [41]. Second, VByte and VNibble are able to decompress just the desired documents, without negative impact on compressed size. However, this is basically zero-order compression, and hence we are still far from the space usage of, for instance, `lzma`. We address this next.

Natural-Language Compression Boosters.

To obtain higher-order compression, we propose to use a so-called natural-language compression booster [18]. The idea is to use first a zero-order compressor on the text, then this compressed text is further compressed using some standard compression scheme. It has been shown that this can yield better compression ratios than just just using a standard compression scheme [18] (especially for smaller block sizes). In our case, we propose using Turpin et al's approach [37] as booster (using VByte and VNibble as we explained above) on the sequence of term identifiers, rather than Word Huffman or End-Tagged as in [18]. Our experiments indicate that the former are faster and use only slightly more space than the latter.

In Table 2 we show the trade-offs for this approach (see the rows for approach "Compression Boosters"). We show results for blocks of size 0.001, 0.01, 0.05, and 0.2 MB of VByte and VNibble compressed text. Overall, the reduction in space usage (compared to the original VByte approach) is considerable. Compared to `lzma` (0.2 MB), the result is a reduction in space usage of 16.68% (12,486 MB vs 14,987 MB), but at the cost of twice the running time as the original `lzma`. When using smaller blocks, however, the time to obtain positions rapidly improves, while the size does not increase too much. For `snappy`, on the other hand, we obtain a

Table 2: Experimental results for extracting term-position data (Step 2).

Approach	Compression Scheme	Space Usage	Position extraction time (ms/q)				
		(MB)	$k_1 = 50$	$k_1 = 100$	$k_1 = 150$	$k_1 = 200$	$k_1 = 300$
Positional indexes	PIL(Rice)	28,373	1.28	2.22	3.05	3.27	5.57
	PIL(S16)	31,338	0.74	1.12	1.43	1.75	2.51
Text compressors	lzma (0.2 MB)	14,987	137.60	260.36	375.21	482.09	684.94
	snappy (0.2 MB)	34,576	9.47	18.00	25.95	33.49	47.74
Zero-order Compressors	VByte	38,339	0.95	1.91	2.86	3.81	5.72
	VNibble	34,570	1.86	3.71	5.57	6.75	8.10
Compression Boosters	VByte + lzma (0.2 MB)	12,486	256.16	484.35	716.41	906.54	1,284.87
	VByte + lzma (0.05 MB)	13,981	70.32	133.18	192.09	246.94	351.71
	VByte + lzma (0.01 MB)	16,762	19.26	36.51	52.67	68.00	97.04
	VByte + lzma (0.001 MB)	22,340	6.11	11.60	16.80	21.72	31.10
	VByte + snappy (0.2 MB)	20,158	9.71	18.86	26.41	34.01	48.69
	VByte + snappy (0.05 MB)	20,366	2.36	4.48	6.47	8.36	11.95
	VByte + snappy (0.01 MB)	22,086	0.82	1.56	2.25	2.91	4.17
	VByte + snappy (0.001 MB)	27,919	0.45	0.86	1.24	1.60	2.30
Compressed self-indexes	WT(7 KB)	40,534	1.94	3.68	5.30	6.85	9.80
	WT(1 KB)	56,917	0.33	0.62	1.04	1.15	1.75
	WT(7 KB) + lzma	19,628	19.25	36.59	52.83	68.36	97.71
	WT(1 KB) + lzma	42,359	7.22	13.54	19.44	24.97	35.57
	WT(7 KB) + snappy	25,122	14.35	23.76	39.38	51.02	74.56
	WT(1 KB) + snappy	46,778	2.07	3.61	5.88	7.32	10.47

Table 3: Experimental results for compressing the document collection (Step 3).

Compressor	Block size	Space usage	Compression Ratio	Snippet extraction time (ms/q)		
	(MB)	(MB)		$k_2 = 10$	$k_2 = 30$	$k_2 = 50$
lzma	0.2	14,987	16.35	29	84	136
	0.5	13,489	14.72	63	181	292
	1.0	12,682	13.84	117	335	540
snappy	0.2	34,576	37.73	2	6	9
	0.5	34,426	37.57	5	14	23
	1.0	34,390	37.53	10	28	46

reduction of 41.69% in space for blocks of size 0.2 MB, with a very minor increase in query time. When we reduce the block size to 0.05 MB, the query time improves even more, and becomes competitive with the time to obtain positions from PIL (Rice). We note that using more advanced techniques from [40] we could obtain about 21 to 22 GB of space for PIL, making both techniques competitive in both space and time. However, VByte + snappy also contains the text within this space, allowing for use during snippet generation. Thus, we are able to store both text and positions in a representation that uses less space than PIL, which stores only positions.

4.4 A Compressed Self-Index for Positions and Snippets

Let T be the text obtained from the concatenation (in arbitrary order) of the documents in the collection. We represent T with a WT to obtain term positions and text snippets. Given a position i in T, one can easily obtain both the docID of the document that contains $T[i]$ and the starting position of a given document j by means of operations rank and select [5], assuming a table of document lengths.

Byte-Oriented Huffman WT.

Instead of a bit-oriented WT (as the ones explained in Section 2.4), we use the byte-oriented representation from [8], using the Plain Huffman encoder, which is the most efficient alternative reported in there. The idea is to first assign a Huffman code to each vocabulary term [29]. 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, instead of bit arrays as in Section 2.4. 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. See [8] for details.

To support rank and select, we use the simple approach from [8]. 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 [8].

To obtain position data assume that, given docID i for a top-k_1 document and a query term t, we want to obtain the 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 (as in Section 4.3). However, the decompression speed of a WT is much slower than that of the schemes used in Section 4.3, so we must use a more efficient way. An idea 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. 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 o is the number of occurrences of t within D_i, then we need 1 rank and $o+1$ selects to find them.

In Table 2 we show the experimental trade-offs for WT, for the different block sizes tested. As it can be seen, WT (7 KB) requires space close to (though slightly larger than) that of the VByte approach. WT (3 KB) and WT (1 KB) obtain better times, but requiring even more space. Moreover, WT (7 KB) is slower than PIL (Rice) and uses more space. The WT, on the other hand, includes the textual data, but still this space usage could leave it out of competition. Next, we introduce extra improvements to make them more competitive.

Achieving Higher-Order Compression with the WT.

Basically, WTs are zero-order compressors, which explains their high space usage. To achieve higher-order compression, notice that B^{root} contains the most significant byte of the Huffman encodings of the original terms. 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. Thus, we propose to compress the blocks of B^v in each WT node v by using some standard compressor.

Table 2 shows results for lzma and snappy, the best compressors we tried. Notice that WT (7 KB) + lzma achieves 19,628 MB, almost half the space used by WT (7 KB). The time to obtain positions becomes, on the other hand, an order of magnitude larger. WT (7 KB) + snappy achieves slightly better times, but using more space. Also, WT (7 KB) + lzma uses slightly less space than VByte + snappy (0.2 MB), but is somewhat slower. Overall, this significant reduction in space could make WTcompetitive.

5. EXPERIMENTAL RESULTS

We now show the time-space trade-offs for the overall solution space we explored. We use here the same basic setup as in Section 4.2.3, with the same parameters (block sizes) for each alternative. We consider the most competitive indexing alternatives from previous sections for positions and snippet generation, described in Table 4. All results include the time and space of the inverted index to carry out Step 1 of the query process, as well as of all structures used in Steps 2 and 3.

Table 4: Glossary of the indexing schemes used in Figure 1. All schemes include the inverted index.

Indexing Scheme	Description
Scheme 1	WT for positions and text.
Scheme 2	WT compressed with lzma for positions and text.
Scheme 3	WT compressed with snappy for positions and text.
Scheme 4	Text compressed with VByte/ VNibble for positions and text.
Scheme 5	VByte compression booster on snappy for positions and text.
Scheme 6	PIL (Rice) for positions and VByte/ VNibble for text.

Note that only Scheme 6 stores an index for position data. Figure 1 shows the different trade-offs for DAAT AND queries with BM25 ranking. Conclusions for OR queries are similar to that for AND. We only show results for $k_1 = 200$ and $k_2 \in \{10, 50\}$, which are representative of other values we tested.

As can be seen, Scheme 6, which uses PIL for positions and Turpin et al [37] for snippets, has one of the fastest query times among all alternatives, but space usage is high compared to other methods. This is because this scheme needs to store positions and text separately. The two points for Scheme 6 that are plotted correspond to using VByte (higher space usage) and VNibble.

Scheme 1 also offers a competitive query time (among the fastest alternatives), but still uses a considerable amount of space. The time-space trade-offs for the schemes that use WT are obtained for different block sizes within the WT nodes (1 to 7 KB). Scheme 2 and Scheme 3 compress the byte sequences of each WT node (as proposed in Section 4.4). As can be seen, the space usage is improved significantly, in some cases by a factor of two. However, query time degrades, making these alternatives less compelling.

Scheme 4 is very competitive in query time, but again its space usage is high. Scheme 5 corresponds to the compression boosters proposed in Section 4.3, and it obtains a very impressive trade-off. One of the most interesting settings is for blocks of size 0.05 MB. In this case, the overall space usage is 1.06 times the space of PIL (Rice), with a query time 1.21 times higher than Scheme 4 and 1.23 times higher than Scheme 6 (which uses PIL). For blocks of size 0.01 MB, Scheme 5 requires 1.12 times the space of PIL (Rice), with a query time that is 0.96 times the one of Scheme 4, and 0.98 the one of Scheme 6. Thus, using only slightly more space than PIL (Rice) (recall the results in Table 2), Scheme 5 includes everything needed for query processing: docIDs, frequencies, term positions, and the text needed to gener-

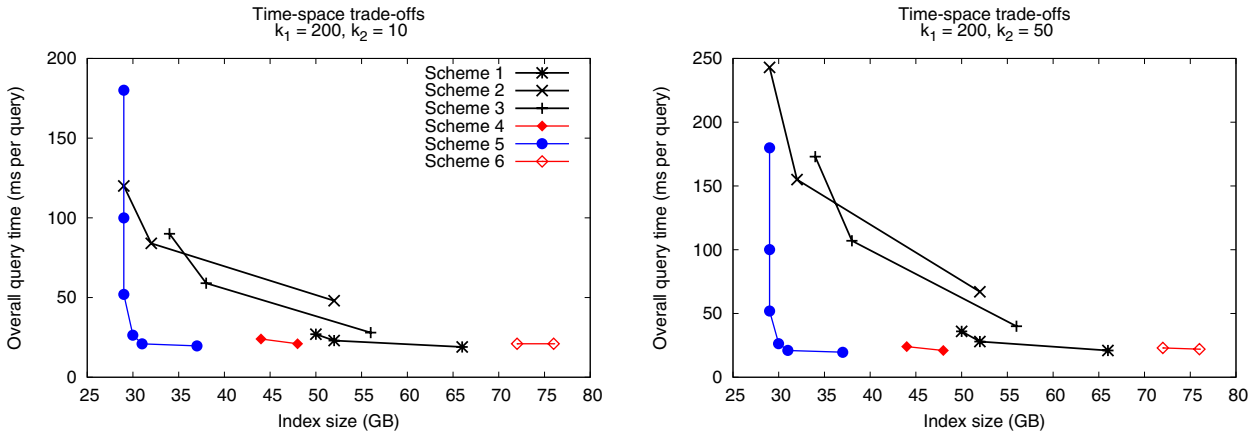

Figure 1: Time-space trade-offs for the overall query process for the GOV2 collection, including positional ranking and snippet generation. The size of each alternative includes the size of inverted index with docIDs and frequencies, which for the TREC GOV2 collection requires 9,739MB.

ate snippets. This is one of the most important conclusions in this paper, that "not to index" can be a real alternative for positional data in practical scenarios. As stated in Section 4.3, the space usage of `PIL`s can be reduced to about 21 GB–22 GB for the TREC GOV2 collection [40]. However, we would still need to add the inverted index and the compressed text to that space in order to support all query processing steps.

Finally, the smallest space alternatives we tested (which are not shown in Figure 1) are the ones that use the inverted index for query processing and `lzma` compression for positions and snippets. This achieves about 22,225 MB of space. This scheme includes everything needed for query processing, and uses only 78% the space of `PIL`. However, query processing time increases significantly, to more than 400 ms per query. This scheme could be useful in some cases where the available memory space is very restricted, such that a larger index would mean going to disk.

A recent alternative [34] proposes to use *flat positional indexes* [11, 14] to support phrase querying; this index could also be used for positional ranking. This is basically a positional index from which docID and frequency information can also be obtained. The results reported for the GOV2 collection in [34] give an index of size 30,310 MB that includes docIDs and frequencies, but not the text needed for snippet generation, making this approach uncompetitive for our scenario.

6. CONCLUSIONS

From our study we can conclude that there exists a wide range of practical time-space trade-offs, other than just the classical positional inverted indexes. 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 relevant points in the trade-off, we propose a compressed document representation based on the approach in [37] combined with Google's `snappy` compression [1]. This allows us to compute position and snippet data using less space than a standard positional inverted index that only stores position data. Even if we include the space used for document identifiers and term frequencies, this approach uses just 1.12 times the space of a positional inverted index, with the same or slightly better query time.

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 address in previous work [40, 21]. Finally, we also showed that compressed self-indexes such as wavelet trees [23] can be competitive with the best solutions in some scenarios.

7. REFERENCES

[1] `http://code.google.com/p/snappy/`.

[2] V. N. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. of 21st Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 290–297, 1998.

[3] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

[4] D. Arroyuelo, V. Gil-Costa, S. González, M. Marin, and M. Oyarzún. Distributed search based on self-indexed compressed text. *Information Processing and Management*, 2012. To appear.

[5] D. Arroyuelo, S. González, and M. Oyarzún. Compressed self-indices supporting conjunctive queries on document collections. In *SPIRE*, LNCS 6393, pages 43–54, 2010.

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

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

[8] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, 2012. To appear.

[9] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level

retrieval process. In *Proc. of 12th International Conference on Information and Knowledge Management*, pages 426–434. ACM, 2003.

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

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

[12] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *SPIRE*, LNCS 5280, pages 176–187. Springer, 2008.

[13] D. Cutting. Apache Lucene. `http://lucene.apache.org/`.

[14] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, page 1, 2009.

[15] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. of 34th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 993–1002, 2011.

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

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

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

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

[20] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[21] P. Ferragina and G. Manzini. On compressing the textual web. In *WSDM*, pages 391–400, 2010.

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

[23] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

[24] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. of ACM*, 46(5):604–632, 1999.

[25] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[26] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[27] D. Metzler and W. B. Croft. A markov random field model for term dependencies. In *Proc. of 28th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2005.

[28] G. Mishne and M. Rijke. Boosting web retrieval through query operations. In *Proc. of 27th European Conference on IR Research*, 2005.

[29] A. Moffat. Word-based text compression. *Software, Practice, and Experience*, 19(2):185–198, 1989.

[30] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[31] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[32] Y. Rasolofo and J. Savoy. Term proximity scoring for keyword-based retrieval systems. In *Proc. of 25th European Conference on IR Research*, 2003.

[33] R. Schenkel, A. Broschart, S. Hwang, M. Theobald, and G. Weikum. Efficient text proximity search. In *14th String Processing and Information Retrieval Symposium*, 2007.

[34] D. Shan, W. X. Zhao, J. He, R. Yan, H. Yan, and X. Li. Efficient phrase querying with flat position index. In *CIKM*, pages 2001–2004, 2011.

[35] B. Sparrow, J. Liu, and M. Wegner. Google effects on memory: Cognitive consequences of having information at our fingerprints. *Science*, 333(6043):776–778, 2011.

[36] T. Tao and C. Zhai. An exploration of proximity measures in information retrieval. In *Proc. of 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2007.

[37] A. Turpin, Y. Tsegay, D. Hawking, and H. Williams. Fast generation of result snippets in web search. In *Proc. of 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 127–134, 2007.

[38] L. Wang, J. J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. of 34th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 105–114, 2011.

[39] H. Williams and J. Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.

[40] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *Proc. of 32nd Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 147–154, 2009.

[41] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[42] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, page 59, 2006.