(This is a sample cover image for this issue. The actual cover is not yet available at this time.)

Simulation Modelling Practice and Theory 22 (2012) 74–91

Contents lists available at SciVerse ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Improved retrieval effectiveness by efficient combination of term proximity and zone scoring: A simulation-based evaluation

Leonidas Akritidis, Dimitrios Katsaros *, Panayiotis Bozanis

Department of Computer and Communication Engineering, University of Thessaly, Greece

ARTICLE INFO

Article history:
Received 30 August 2011
Received in revised form 2 December 2011
Accepted 4 December 2011

Keywords:
Web
Search engines
Inverted index
Simulation
Evaluation

ABSTRACT

During the past few years, the commercial Web search engines have augmented their underlying index structures by significantly enriching the information which describes the appearance of a word within a document Dean (2009) [7]. This enriched information is now used in complex and effective functions which rank documents by taking into consideration hundreds of features, with respect to a user query. Despite the evolution of the search engines, the past research has mainly concentrated on improving plain Web indexes storing typical data only. In this work we study the problem of organizing an inverted index storing additional information. In particular, we examine how the physical locations of a document, called zones, can be efficiently integrated with such an index structure. We introduce TZP, an encoder which compresses these zones in combination to the positions of a word in a document, by employing a fixed number of bits for each portion of a word's inverted list. We demonstrate that our method allows direct access to the compressed zones and positions without expensive look-ups, avoids decoding any unnecessary information, while its overall index size is analogous or even better when compared against state-of-the art schemes. Moreover, we examine how the word positions can be combined to the zones to improve retrieval effectiveness. We introduce BM25TOPF, a scheme which incorporates term proximity and zone weighting into a single ranking formula. Unlike other term proximity approaches, BM25TOPF also takes into account the ordering of the query terms by rewarding the documents containing them in the correct order. Our experiments with the Web Adhoc Task of TREC 2009 and a set of own queries show that BM25TOPF outperforms the current state-of-the-art approaches by a margin between 6% and 11%.

1. Introduction

Nowadays, the repositories of the major search engines consist of tens of billions of documents Dean [7] and as the Web becomes larger and the crawling technology evolves, these repositories are expected to grow further. Furthermore, search engines accept and answer thousands of queries per second attempting to quickly retrieve the most suitable documents for each submitted query. In such a dynamic environment where the available information, the workload and the user expectations expand, search engines have to constantly scale up in terms of both efficiency (query throughput) and effectiveness (quality of query results).

The inverted index is the primary data structure used by search engines for storing document-related data and metadata. According to [27,31], an appropriately constructed inverted index can improve the performance of query processing

* Corresponding author. Tel.: +30 2421074975.
E-mail address: dkatsar@inf.uth.gr (D. Katsaros).

1569-190X/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.simpat.2011.12.002

dramatically. Due to the importance of the inverted index in the overall efficiency of a search system, there has been a lot of research conducted towards its optimization. Optimization primarily regards two critical issues: *compression* and *organization*. The former is a key issue for reducing the overall index size and minimizing the transfer costs from either disk or main memory. The latter enables partial access of the index structure, that is, a query can be answered without having to traverse all the available information stored in it.

Several engineers (see for instance Dean [7]) have revealed that the information stored in the inverted index search engines has tripled during the past few years. However, in the literature we mainly encounter strategies and algorithms concerning typical inverted indexes, which almost always store very limited data: document identifiers, word-document frequencies and word positions in a document. Compared to the hundreds of the parameters employed by the major search engines for ranking their documents [28,24], this data is apparently inadequate.

In this work we study the potential of including additional information within the inverted index. In particular, we adopt the idea of partitioning a Web document into locations of special interest, namely *zones*. The document zones were introduced in Manning et al. [12], but to the best of our knowledge, issues regarding the compression and organization of such indexes have never been studied before. In this paper we investigate the meaning of a word's appearance within a document; we replace the plain positional data by the occurrences, a piece of information which contains both the position and the zone of the document where this specific word appears.

In the sequel, we propose a method which allows compact storage of zones along with the corresponding word positions. Our approach, namely TZP, operates in combination with the *block-based* inverted list organization, a strategy introduced by Moffat and Zobel [14] which splits an inverted list into blocks. Block-based schemes allow us to skip large, unneeded portions of the index during query processing. TZP is designed to support all the partitioning strategies that have been proposed so far (refer to [15,3,1,30,6]), and operates in two steps: In the first step, the compressor packs each position-zone pair of a block into a 32-bit space and in the next phase, these packets are encoded together by employing a fixed number of bits. This scheme enables the direct access of the occurrence data for each posting, by using a limited number of pointers (one pointer per block).

Finally, one of our main motives was to examine whether the usage of the additional information can really lead to search results of higher quality. Discovering a ranking function which combines many different parameters (i.e. frequencies, term proximity, zone weighting, document lengths etc) is a challenging task. In this paper we initially examine some state-of-the-art probabilistic retrieval functions, such as BM25 (firstly introduced Robertson and Jones [18]), a variant which takes into consideration the zone of the document where a term appears (BM25F, Lu et al. [11]), and another variant which takes into consideration term proximity, namely BM25TP Buttcher et al. [5]. In the sequel, we propose an enhancement to the BM25TP, *BM25TOP*, which takes into consideration both term proximity and correct term ordering (that is, whether the terms in a document appear in the same order as in the query). Finally, we inject the concepts of BM25TOP to BM25F to produce *BM25TOPF*, a ranking function which is sensitive to term proximity, correct term ordering and zone weighting.

As a summary, the contributions of this paper are:

- We introduce a two-level encoder for positions and zones, namely TZP. TZP compresses the data by using a fixed number of bits for each position-zone pair of the same block of an inverted list.
- We show that the fixed-bit policy adopted by TZP allows very fast decompression, whereas it also enables us to access directly only the data actually required for processing a query. In other words, with TZP we are not obliged to look-up for the positional and zone data of a particular posting since we are allowed to compute their location, and moreover, we do not have to decode any unnecessary information.
- We investigate the usefulness of combining term proximity and zone weighting for document ranking. In particular, we first propose BM25TOP, an enhancement to the original BM25TP function which is sensitive to the correct term ordering. We also introduce BM25TOPF, a ranking method that allows term proximity, correct document ordering and zone weighting to be combined into a single scoring formula.
- All our contributions are experimentally evaluated by using the Clueweb09-T09B document collection consisting of roughly 50 million English documents.

The rest of the paper is organized as follows: In Section 2 we examine the state-of-the-art methodologies for organizing inverted indexes and we cite the relevant work. Section 3 contains the description of zones and consists of three Subsections: SubSection 3.1 discusses the new form of postings after the inclusion of zones, SubSection 3.2 introduces the TZP compression algorithm for zones and positions, and SubSection 3.3 demonstrates how the data encoded by TZP can be efficiently accessed and decompressed. In Section 4 we provide descriptions of some popular ranking functions and we propose our own scoring approaches. Finally, Section 5 contains the experimental evaluation of our methods and propositions, whereas in Section 6 we finalize this work by stating our conclusions.

2. Preliminaries and related work

The inverted index is the primary data structure constructed and maintained by the search engines to serve user queries. There is a significant amount of research regarding the efficient organization of these indexes and in this Section we briefly describe some basic elements deriving from the related theory.

A typical inverted index structure consists primarily of two components: (i) the *lexicon*, a list sorted in ascending lexicographical order containing all the distinct words appearing in the collection and (ii) the *inverted file*, that stores all the occurrences of each word in the collection. These occurrences are organized in *inverted lists*. In its simplest form, an inverted list I_t of a term t stores a list of *postings* which contain the integer identifiers of the documents (docIDs) where t appears into.

To support ranked query processing, we store additional information within an inverted list: (i) the term-document frequency (or just frequency) $f_{d,t}$, which reveals how many times a term t appears in document d i.e., each posting S_i is of the form (d_i, f_i) and (ii) some positional data, $p_{i,k}$ indicating the position of the term in the document. In this case the postings are of the form $(d_i, f_i, p_{i,0}, p_{i,1}, \ldots, p_{i,f_i-1})$. The inverted lists can be sorted either by docID, or by another attribute (frequency or another scoring value Brin and Page [4]). In this paper, we consider the situation where the inverted lists are sorted by increasing docID order; this setup allows more effective index compression (refer to [27,6]) and supports the parallel traversal of all of the query terms' inverted lists during query processing Manning et al. [12].

There are two main methods exploiting the inverted index to evaluate a query: *term-at-a-time* and *document-at-a-time* Turtle and Flood [26]. The first approach initially orders the query terms in increasing frequency order; in the sequel the inverted list of each term is repeatedly merged with the lists of the other terms, leading to the final result list (see Witten et al. [27] for a detailed description). On the other hand, in the latter method, the inverted lists are scanned in parallel sequentially retrieving the documents which are relevant to the query. Document-at-a-time evaluation is essential for large document collections where we can predict the number of documents that could be retrieved (by using statistical methods) and the operation can be terminated as soon as adequate qualitative documents have been retrieved Boldi and Vigna [3].

To perform efficient parallel scanning of several inverted lists, it is beneficial that we maintain a mechanism which allows us to skip large portions of the list by seeking the first docID larger than or equal to a given one. In this way we avoid decoding useless portions of the list and we are able to quickly access the desired information. For this reason, multiple works such as [14,15,3,1,30,6] propose partitioning the inverted list into a series of adjacent *blocks* and maintain one or more pointers pointing to the beginning of each block. During the evaluation of a query, the processor makes use of these pointers to locate the correct block and decode only the data actually needed.

A block-based inverted list organization is depicted in Fig. 1. The data in each block is stored in three *chunks*: the first chunk is used to store the docIDs, the second chunk stores the corresponding frequency values, whereas the third chunk contains the positional data. Since the number of the occurrences of a term within a document can be infinitely large, an important issue posed by the usage of a block-based scheme is to identify the location of the positional data for a particular posting. The problem becomes even more challenging in case the corresponding data is compressed, because we need to locate the desired information within the *compressed* sequence.

To address this issue, the researchers have proposed two basic ways to organize the data: (a) interleaving, i.e. the positional data of a particular block are stored after the docIDs and the frequencies of the same block, and (b) creating a completely separate structure for positions with its own lookup mechanism. For instance, Yan et al. [28] propose interleaving, and they introduce a fairly standard hierarchical look-up structure to access the positions. For each block of an inverted list, this structure stores one docID and one pointer to the beginning of the block. Furthermore, within each block the positional data is organized into sub-blocks of $\mathcal{B}$ postings. For each of these sub-blocks, a pointer is used to store its offset from the beginning of the block. To retrieve the positional data of a specific posting we first search for the correct sub-block and then we decode all the positions for the postings of this sub-block. Then the positions of the particular posting are retrieved by using the aforementioned offset value. Nevertheless, this look-up operation is quite expensive and can decelerate query processing especially in case $\mathcal{B}$ is small. Furthermore, this additional structure occupies extra space in memory.

On the other hand, Transier and Sanders [25] organize positions by employing a separate structure, namely indexed list. This list consists of two levels; the upper level contains pointers pointing to the data stored at the lower level. That is, for

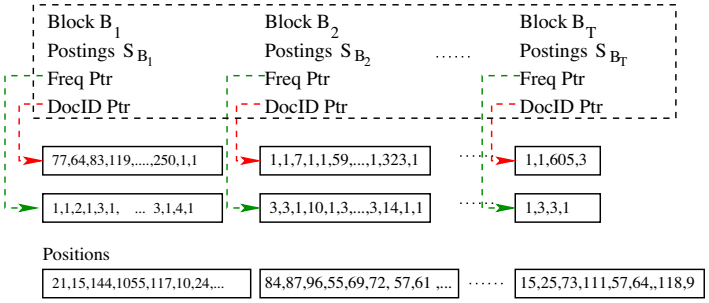


Fig. 1. Partitioning an inverted list into T blocks of postings (block-based organization). In the upper part we depict the skip table which stores two pointers per block: one pointing at the docIDs and one pointing at the frequency values.

each posting they store a pointer which shows the location of the corresponding positional values. Although this approach offers direct access to the positional data without look-ups, it requires even more space than the aforementioned look-up structure since one pointer per posting is very expensive.

In this paper we propose a hybrid between interleaving and data structures. In particular, we choose to store the positional data contiguously (i.e. not in blocks), but along with each block we also record a pointer which in combination with our encoding method, allows us to access *directly* the positions of a specific posting. This renders our approach much more economic than that of Transier and Sanders [25], since instead of requiring one pointer per posting, it requires one pointer per block.

On the other hand, the problem of selecting appropriate block sizes has been widely studied: For instance, the work of Moffat and Zobel [14] proposes setting skip pointers every $\sqrt{N_t}$ postings of the inverted list, where N_t is the number of documents containing t. A number of research articles [30,8,28] place skips each time a fixed number of postings (i.e. 128) has been encountered. Other papers study the issue of dynamically setting skip pointers in a fashion which maximizes query throughput. For instance, Boldi and Vigna [3] embed compressed perfect skip lists in an inverted list to increase the processing speed, whereas Silvestri and Venturini [22] introduced a novel class of encoders which partition the list in an optimal way that maximizes decompression rates, by using dynamic programming.

Here we do not examine in depth the issue of inverted list partitioning. We rather focus on the efficient representation of positions and zones which allows fast query evaluation. However, notice that all the aforementioned skipping/partitioning techniques can be used in combination with our proposals with no additional effort.

3. Document zones

There are several features that differentiate Web documents from plain textual documents: the former include hyperlinks allowing the reader to quickly navigate from one page to another, and they also possess a visual structure determined by the usage of HTML tags. A typical Web page usually includes a title, some anchor text associated with its outgoing links, headings and other locations of special interest such as meta-tags and URL. Apparently, the appearance of a word in different locations of such a document is of different importance. For instance, the words used in a document's title usually represent its content and an effective search engine should treat these words in a different way than the ones occurring in the normal text.

The structure of the Web documents have gained very little attention by the inverted index researchers and engineers. The vast majority of the relevant work takes into consideration only the position of a word to describe its occurrence within a document. One exception to this rule is the early work of Brin and Page [4] which introduces plain and fancy *hits* to identify words appearing in a document's text and title respectively. However, this work ignores the rest of the physical locations of a document whereas it limits the maximum position value to an upper bound of 12 bits.

To address this problem, Manning et al. [12] partition each document into several parts of special interest called *zones*. Zones are distinct, arbitrary locations of a Web document containing free text and delimited by page formatting tags. It is not mandatory to be contiguous (they can span across multiple locations of the document), but they cannot overlap. Unfortunately, the aforementioned work does not address the issue of the efficient representation of zones within the inverted index.

Table 1 records some the most typical zones that a Web page can be partitioned into. Each of these zones is assigned a unique integer value, the *ZoneID*. Notice that for the case we are studying (Web pages), we consider these eight zones only. Nevertheless, with a slight modification our approach can support more zones and larger ZoneIDs.

In this work we examine how zones can be used to improve the retrieval effectiveness of a search engine. Moreover, we study efficient methodologies of including them within the inverted index, the primary data structure that search engines employ to answer user queries.

Table 1
Zones of a typical web page.

Zone	ZoneID	HTML
Body (normal text)	0	<body>... </body>
Anchor text	1	<a>...
Title text	2	<title>... </tile>
Document's URL	3	–
Headings	4	... , ... ,...
Page description	5	<meta name= ''Description'' Content="...">
Image description	6	<img alt="..." ...
Label text	7	<label>... </label>

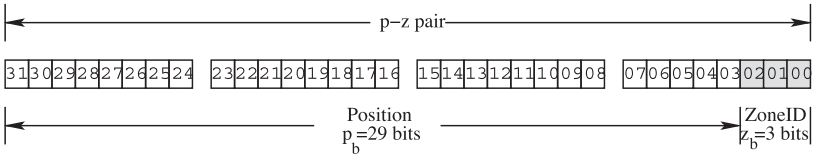


Fig. 2. First phase of TZP; Encoding a single term occurrence (position-zone pair) in a 32-bit space for $z_b = 3$ and $p_b = 29$.

3.1. Integrating zones within the inverted index

We now show how a typical inverted index can be enriched by including zones. Within each posting S_i of the inverted list I_t of a term t, we replace the positional data with a more general quantity $h_{i,j}$, describing the jth *occurrence* of t within a document d_i. Therefore, the new form of the posting S_i is:

$$S_i : (d_i, f_i, h_{i,0}, h_{i,1}, \ldots, h_{i,f_i-1}) \tag{1}$$

Since we desire to describe the occurrence of a term by using both positions and zones, $h_{i,j}$ is of the form

$$h_{i,j} : p_{i,j}, z_{i,j} \tag{2}$$

that is, each term occurrence is now described by a *position–zone (p–z)* pair and not by just using positional values. From now on, we shall use the word occurrence to refer to such position-zone pairs. Compared to a typical inverted index, this enriched form provides additional features and functionality, since:

- It is capable of answering a wider range of queries, for example "*Retrieve all the documents having the terms University AND Thessaly in title AND the term gr in their URL.*"
- We can exploit more sophisticated and effective ranking functions such as the BM25F Lu et al. [11], or fabricate other robust scoring approaches by taking the additional parameters into consideration.

The inclusion of additional information makes index organization and compression more challenging. The requirements of Web-scale search engines include compactness (i.e. the new data must be stored as efficiently as possible) and speed (i.e. we must be able to quickly access and decode the index data). In the following Subsections we discuss further these requirements.

3.2. Encoding ZoneIDs and positional data

Inevitably, the inclusion of zoneIDs in an inverted list will lead to an increase of the overall size of the index. In this Subsection we study how we can minimize this effect and preserve high decompression rates.

Let us return to the posting scheme described by Eqs. (1) and (2). A naive approach for encoding each term occurrence (Eq. (2)) would suggest using two separate integers, one for storing the positional value and one for storing the zoneID. Undoubtedly, this approach is prohibitively expensive since we cannot afford wasting 64 bits[1] for each term occurrence.

For this reason, we suggest a two-level compression scheme, namely TZP, which initially encodes one position-zone pair into a 32-bit space, and then, for each block of the inverted list, it employs a fixed number of bits to encode the occurrences of that block. Fixed-bit compression such as Packed Binary Anh and Moffat [2] has two major advantages; at first, it allows fast decompression since the decoding process is fairly simple and second, it enables access to the positional data of a particular posting without look-ups and without decoding useless data. We shall defend these claims shortly, especially the second one which is extremely important since look-ups could drastically decelerate query processing Yan et al. [28].

Our analysis begins by considering the block-based list organization of Fig. 1. Suppose that the inverted list I_t of a term t is partitioned into $\mathcal{B}_{l_t}$ blocks and each block $\mathcal{B}_i \in \mathcal{B}_{l_t}$ is comprised of $S_{\mathcal{B}_i}$ postings. Here we do not study the manner an inverted list is partitioned into blocks, however to render our proposals compatible with the existing partitioning approaches, we assume that $S_{\mathcal{B}_i}$ is *not* equal for all blocks (i.e. the blocks include variable numbers of postings).

Now we demonstrate how the occurrences $h_{\mathcal{B}_i}$ included in the postings $S_{\mathcal{B}_i}$ of a block $\mathcal{B}_i$ can be efficiently encoded by using our proposed TZP approach. During the first phase and for every term occurrence $h_j \in h_{\mathcal{B}_i}$, we reserve the z_b least significant bits of a 32-bit integer to encode the zoneID z_j, whereas the rest $p_b = 32 - z_b$ bits are used to represent the positional value p_j. In this work we mainly focus on standard Web pages with eight zones at most, thus a setting of $z_b = 3$ suffices. However, for different types of documents alternative setups could be selected. Finally, applying this method to all the occurrences of the block results in a sequence of new integers having values equal to $|h_j| = 2^{z_b} p_j + z_j$ (Fig. 2).

[1] In this paper we assume that each integer occupies 32 bits.

Algorithm 1. Encoding a bundle of K position-zone pairs with TZP. The function *Encode_Occurrence* encodes a single pair with respect to the number of bits z_b that we reserve for zones (in our case $z_b = 3$). *Encode_Occurrences* compresses an entire bundle of K position–zones pairs; after the maximum occurrence value h_{max} and the number of the required bits C have been determined (steps 9–10), we store the binary representation of each occurrence within the bit-vector $\mathcal{H}$ (steps 12-15). *WriteBits* is a typical bit-writer function which stores an integer into $\mathcal{H}$ by using C bits.

```
int Encode_Occurrence(p,z,z_b)
1.  int h ← 0
2.  h ← L_{z_b}(p)
3.  h ← h|z
4.  return h

byte Encode_Occurrences(p[K],z[K],K)
1.  int i ← 0, h_max ← 0
2.  while (i < K) {
3.    h[i] = Encode_Occurrence(p[i],z[i],3)
4.    if (h[i] > h_max) {
5.      h_max = h[i]
6.    }
7.    i++
8.  }
9.  int C ← ⌈log2(h_max − 1)⌉
10. byte H ← allocate ⌈KC/8⌉ bytes
11. i ← 0
12. while (i < K) {
13.   WriteBits(H, h[i], C)
14.   i++
15. }
16. return H
```

At the second phase, we first select the highest occurrence value $|h_{\mathcal{B}_i}|_{max}$ for each block $\mathcal{B}_i$ of the inverted list. In the sequel, we use $C_{\mathcal{B}_i} = \lceil log_2(|h_{\mathcal{B}_i}|_{max}) \rceil$ bits to produce a binary representation[2] of each occurrence in that block, and we store these representations into a bit vector. This operation is very similar to the Packed Binary approach.

A pseudocode demonstrating how TZP is used to encode position–zone pairs is presented in Algorithm 1. The operators that we use in our algorithm representation, include bitwise AND (&), bitwise OR (|), left-shift of a value v by b bits ($\mathcal{L}_b(v)$) and right-shift of value v by b bits ($\mathcal{R}_b(v)$).

TZP is not designed to operate on single integers; we can rather think of it as a method which effectively compresses two correlated integers from which one of them has an upper bound. In our examined occasion, we restrict zones to occupy only three bits. TZP can be generalized to encode more than two integers. Therefore, if we have N integers which all describe a single phenomenon (i.e. term occurrence in a document) and $N-1$ of them can be restricted to have an upper bound and relatively small values, TZP provides an efficient mechanism for their compression.

3.3. Accessing ZoneIDs and positional data

Now we show how our fixed-bit encoding allows access to the occurrences of a particular posting without look-ups. For each block $\mathcal{B}_i \in \mathcal{B}_{l_t}$ of an inverted list, we associate two values: (a) a pointer $R_{\mathcal{B}_i}$ which points at beginning of the occurrence data of $\mathcal{B}_i$, and (b) a value $C_{\mathcal{B}_i}$ which represents the fixed number of bits used to encode each occurrence of $\mathcal{B}_i$. With this information we can *compute* the location where the occurrences of a particular posting $S^j_{\mathcal{B}_i}$ start from, and avoid searching for it during query processing. The following equation provides the exact bit where the occurrences of $S^j_{\mathcal{B}_i}$ start from:

$$\mathcal{L}^j_{\mathcal{B}_i} = R_{\mathcal{B}_i} + C_{\mathcal{B}_i} \sum_{x=0}^{j-1} f_{x,\mathcal{B}_i} \tag{3}$$

where $f_{x,\mathcal{B}_i}$ is the xth frequency value stored within $\mathcal{B}_i$. Eq. (3) informs us that to locate the occurrence data for a particular posting, we first need to retrieve the associated $R_{\mathcal{B}_i}$ pointer value; then, we sum up all the frequency values of the previous postings of the current block $\mathcal{B}_i$. This sum of frequencies reveals the number of the occurrences between the beginning of the block and the location of the desired data. Since each of these occurrences is represented by a fixed number of bits, we just

[2] Recall that we are able to encode every positive integer ranging from $0 \ldots N-1$ by using $\lceil \log_2 N \rceil$ bits.

need to multiply the sum by $C_{\mathcal{B}_i}$ to locate the first *compressed* occurrence of the posting. The operation ends by decoding the next $f_{j,\mathcal{B}_i} C_{\mathcal{B}_i}$ bits and the occurrences are retrieved.

Algorithm 2. Decoding position-zone pairs for the jth posting of a block $\mathcal{B}_i$ according to TZP. Initially we sum up all the frequency values of the previous postings (steps 2–5) of the current block $\mathcal{B}_i$, and we locate the bit where the occurrences of the jth posting start from (step 6). Then, we employ the *ReadBits* function to read each encoded occurrence from the bit-vector $\mathcal{H}$, starting from different points each time (steps 8–12). In the second phase, we extract the positional and zone data out of each occurrence (steps 14–18). At step 15 we retrieve the three lest significant bits of $h[x]$ which represent the zoneID, whereas at step 16 we right shift $h[x]$ by 3 positions to retrieve the positional value.

int *Decode_Occurrences*$(j, \mathcal{B}_i, \mathcal{H})$
1. **int** $x \leftarrow 0$, $s \leftarrow 0$
2. **while** $(x < j)$ {
3. $s \leftarrow s + f_{x,\mathcal{B}_i}$
4. $x++$
5. }
6. **int** $start \leftarrow R_{\mathcal{B}_i} + sC_{\mathcal{B}_i}$
7. $x \leftarrow 0$
8. **while** $(x < f_{j,\mathcal{B}_i})$ {
9. $h[x] \leftarrow ReadBits(\mathcal{H}, C_{\mathcal{B}_i}, start)$
10. $start \leftarrow start + C_{\mathcal{B}_i}$
11. $x++$
12. }
13. $x \leftarrow 0$
14. **while** $(x < f_{j,\mathcal{B}_i})$ {
15. $z[x] \leftarrow h[x]$& `0x00000007`
16. $p[x] \leftarrow \mathcal{R}_3(h[x])$
17. $x++$
18. }
19. **return** p, z

Our proposed methodology has a series of advantages over other organization approaches:

- It allows us to directly access the desired data by using Eq. (3) without look-ups. Consequently, query processing is accelerated.
- It saves the space cost of maintaining a separate look-up structure Yan et al. [28].
- It uses much fewer pointers than the approach of Transier and Sanders [25] which employs indexed lists.
- It enables decoding of the information actually needed, without the need to decompress entire blocks or sub-blocks of integers (unlike other compression techniques such as PFOR-DELTA [9,32]).

Another important issue is to determine where we should store $R_{\mathcal{B}_i}$ and $C_{\mathcal{B}_i}$. A convenient location is within the skip structure (upper part of Fig. 3); for each entry of the table, we also record these two values and in case the processor decides that a block should be scanned, we are able to immediately retrieve them.

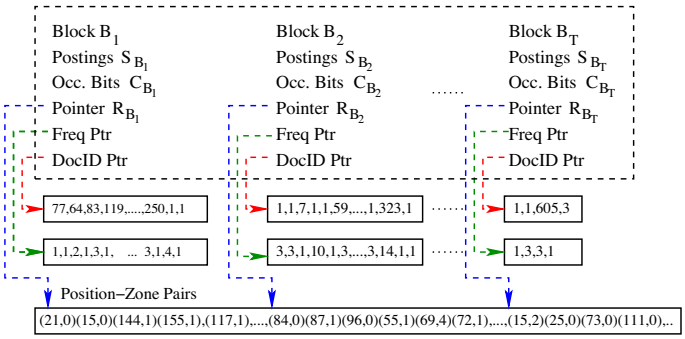


Fig. 3. Partitioning an inverted list into adjacent blocks of postings (block-based organization) according to TZP. The positional and zone data are packed separately at the end of the inverted list. For each block of the list, we store within the skip table (a) a pointer pointing at the starting bit of the corresponding occurrence data and (b) the number of bits used to encode each occurrence of the block.

Similarly to encoding, the decoding process of TZP also involves two phases. Algorithm 2 includes a pseudocode describing the entire operation. Suppose that we need to access the positional and zone data for the jth posting of block $\mathcal{B}_i$. Initially, we locate the data as indicated by Eq. (3). If $f_{j,\mathcal{B}_i}$ is the associated frequency value of $S^j_{\mathcal{B}_i}$, we sequentially read $f_{j,\mathcal{B}_i}$ groups of $C_{\mathcal{B}_i}$ bits from the compressed sequence; each group represents an encoded occurrence of this posting. In the next phase, we extract the desired zone and positional data out of each occurrence.

4. Probabilistic retrieval

The issue of returning results of high quality is of primary importance for every search engine. Its success is mainly based on the capability of the ranking function to locate results satisfying the information needs of the users.

In this Subsection we provide a brief overview of the current state-of-the-art ranking functions and in the sequel, we provide our own ranking method that exploits the additional information stored in our proposed index setup. In Table 2 we summarize the symbols used by the presented scoring functions and, also, we explain their meaning.

4.1. The BM25 function

The BM25 weighting scheme was developed by Robertson and Jones [18] as a way of building a probabilistic model sensitive to term frequency and document length. It is not a single function, but actually a whole family of scoring functions, with slightly different components and parameters. One of the most prominent instantiations of the function is as follows.

Given a query Q containing the terms $t_1, t_2, \ldots, t_n$, each document d is assigned a relevance score that is given by the following formula:

$$S_{BM25}(d) = \sum_{i=1}^{n} w_{t_i} \frac{f_{d,t_i}(k_1+1)}{f_{d,t_i}+K}, \tag{4}$$

$$K = k_2\left(1 - b_1 + \frac{b_1 l_d}{\bar{l}}\right) \tag{5}$$

where k_1, k_2 and b_1 are three predefined constants. Although there are some other variants of the BM25 weighting scheme, the one we provide here is the most popular among them.

4.2. Zone weighting

Zone weighting is related to the appropriate result scoring when ranking structured or semi-structured documents. BM25F is a model proposed by Lu et al. [11] and takes into consideration the physical position of a term within a document (i.e. in which field of an XML document a term appears). The introduction of zones in Section 3 and the usage of the enriched index structure, allows us to use BM25F in standard Web documents.

For a user query $Q = \{t_1, t_2, \ldots, t_n\}$, scoring according to BM25F is performed in two phases: Initially, we obtain the accumulated weight of a query term t_i over all fields as follows:

$$W_z(d,t_i) = \sum_{z_{j,d} \in d} \frac{S_{z_j} f_{z_{j,d},t_i}}{1 - b_2 + b_2 \frac{l_{z_{j,d}}}{\overline{l_{z_j}}}} \tag{6}$$

Table 2
Notation.

Symbol	Meaning
C	The entire document collection (corpus)
d	An arbitrary document in the collection
$z_{j,d}$	The jth zone of d
N	Number of documents in C
Q	A user query
t_i	The ith term in Q
N_{t_i}	Number of documents containing t_i
$w_{t_i} = \log(N/N_{t_i})$	Inverse Document Frequency (IDF) of t_i
$p_{t_i,Q}$	The position of t_i in query Q
$p_{t_i,d}$	The position of t_i in document d
p_{t_i,z_j}	The position of t_i in zone z_j
f_{d,t_i}	Number of occurrences of t_i within d
$f_{z_{j,d},t_i}$	Number of occurrences of t_i within $z_{j,\,d}$
l_d	The length of d (number of terms)
$\bar{l} = \left(\sum_{i=1}^{N} l_i\right)/N$	The average document length in C
$l_{z_{j,d}}$	The length of $z_{j,d}$ (number of terms)
$\overline{l_{z_j}} = \left(\sum_{j=1}^{N} l_{z_{j,d}}\right)/N$	The average length of z_j in C

where b_2 is a predefined constant usually set equal to $b_2 = 0.75$, and S_{z_j} is a static score assigned to each document zone. The existence of the S_{z_j} score allows us to modify the importance of a term appearing in special locations within a document. For example, we could assume that a document having the query terms in the title is more relevant to this specific query, than a document which contains these terms in its normal text.

On the second phase, the accumulated weights are being used to compute the BM25F scores according to the following formula:

$$S_{BM25F} = \sum_{i=1}^{n} w_{t_i} \frac{W_z(d,t_i)}{W_z(d,t_i) + k_3} \tag{7}$$

where, similarly, w_{t_i} represents the IDF of t_i and k_3 is a predefined constant.

4.3. Term proximity scoring

Document retrieval functions based on the vector space model (see for instance [19,16,10]) and the bag-of-words representation of documents, such as Okapi BM25 have been proved to be effective in ad-hoc information retrieval tasks. One of their drawbacks is that they do not take the proximity of query terms within a document into account. However, there are many queries where the best results contain the query terms in a single phrase, or at least in close proximity as indicated by [20,21,13].

The BM25TP scheme is an expansion to the BM25 weighting, attempting to integrate term proximity into the original scoring function. It was firstly presented by Buttcher et al. [5] and it is very similar to the one proposed by Rasolofo and Savoy [17]. We examine this model because in experiments with the TREC collection it was the only one exhibiting significantly better performance than the others, according to Schenkel et al. [21].

Suppose a user submits the query $Q = \{t_1, t_2, \ldots, t_n\}$. With every query term t_i we associate an accumulator $acc_d(t_i)$ that stores the term's proximity score within the current document d. Whenever the query processor encounters a posting belonging to a query term t_x, it computes the distance (number of postings) between this posting and the previous posting belonging to the term t_y. If $t_x \neq t_y$, then we increment the accumulators for both terms according to the following formulas:

$$acc_d(t_x) = acc(t_x) + w_{t_x} \frac{1}{(p_{t_x,d} - p_{t_y,d})^2}, \tag{8}$$

$$acc_d(t_y) = acc(t_y) + w_{t_y} \frac{1}{(p_{t_y,d} - p_{t_x,d})^2}, \tag{9}$$

If $t_x = t_y$ we leave the accumulators unchanged. By using these accumulators the BM25TP score is now defined as follows:

$$S_{BM25TP} = S_{BM25} + \sum_{i=1}^{n} min\{1, w_{t_i}\} \frac{acc_d(t_i)(k_1+1)}{acc_d(t_i) + K} \tag{10}$$

The BM25TP provides significant improvements in result quality over BM25 scoring. These gains are becoming even greater as the size of the document collection increases. This is due to the fact that for larger collections, the probability of finding non-relevant documents that contain the query terms by chance is greater than for smaller collections.

One drawback of this method is its insensitivity to the query terms ordering, since it holds that $(p_{t_x,d} - p_{t_y,d})^2 = (p_{t_y,d} - p_{t_x,d})^2$. Consequently, BM25TP assigns scores to the documents regardless of the query formulation and fails to distinguish the difference between queries such as *John is faster than Mary* and *Mary is faster than John* as Manning et al. [12] mention. To address this issue, we initially introduce a quantity $\xi(t_x, t_y)$ which receives the following values:

$$\xi(t_x,t_y) = \begin{cases} 1, & p_{t_x,Q} - p_{t_y,Q} > 0 \\ -1, & p_{t_x,Q} - p_{t_y,Q} < 0 \end{cases} \tag{11}$$

That is, in case we examine a document containing two terms which appear in reverse order than in the query, $\xi(t_x, t_y)$ receives a constant negative value (-1), whereas a constant positive value is assigned in the opposite case. The following fraction:

$$a_d(t_x,t_y) = (p_{t_x,d} - p_{t_y,d})/\xi(t_x,t_y) \tag{12}$$

is always positive in case the terms in the document are positioned in the same order as in the query, and negative in the opposite case. Now, to reward both term proximity and correct term ordering, we need to replace the square difference $(p_{t_x,d} - p_{t_y,d})^2$ in the denominators of Eqs. (8) and (9) by a function which is:

- Always positive, regardless of $p_{t_x,d} - p_{t_y,d} > 0$ or $p_{t_x,d} - p_{t_y,d} < 0$; i.e., we do not desire to assign negative scores.
- Becoming higher as $p_{t_x,d} - p_{t_y,d}$ increases and vice versa; i.e., it rewards term proximity (since the score is inversely proportional to this quantity).
- Becoming higher in case of $p_{t_x,d} - p_{t_y,d} < 0$ and vice versa; i.e., it rewards correct term ordering.

A function which satisfies these conditions is the polynomial

$$\varphi_d(t_x, t_y) = (a_d(t_x, t_y))^2 - a_d(t_x, t_y) + 1 \tag{13}$$

which is always positive for all $a_d(t_x) \geqslant 1$ and $a_d(t_x) \leqslant 0$, which in our case is always true, since $p_{t_x,d} - p_{t_y,d}$ receives values within $(-\infty, -1] \cup [1,\infty)$. Notice that there is an infinite number of polynomials satisfying the requirements that we have set, however, here we have chosen the simplest one which requires the minimum processing. By replacing the square difference $(p_{t_x,d} - p_{t_y,d})^2$ by the polynomial $\varphi_d(t_x)$, the accumulators of Eqs. (8) and (9) are transformed according to the following relationships:

$$acc'_d(t_x) = acc'_d(t_x) + w_{t_x} \frac{1}{\varphi_d(t_x, t_y)}, \tag{14}$$

$$acc'_d(t_y) = acc'_d(t_y) + w_{t_y} \frac{1}{\varphi_d(t_x, t_y)}, \tag{15}$$

This new form satisfies all the requirements that he have set: In case the query terms appear close in a document, the value of $\varphi_d(t_x)$ is small, hence the accumulator value increases. Furthermore, if these terms appear in the opposite order with respect to the given query, $\varphi_d(t_x)$ increases thus reducing the overall document's score.

Based on the accumulators of Eq. (14) we introduce a modified scoring formula, BM25TOP, which apart from term proximity, it also takes into account the query formulation and the query terms ordering. The scores of BM25TOP are provided by the following equation:

$$S_{BM25TOP} = S_{BM25} + \sum_{i=1}^{n} min\{1, w_{t_i}\} \frac{acc'_d(t_i)(k_1 + 1)}{acc'_d(t_i) + K} \tag{16}$$

where K is calculated as previously by using Eq. (5). Note that the proposed enhancement for term ordering applies not only to BM25TP, but also, to all term proximity scoring schemes which utilize position-dependent accumulators, such as the effective variant proposed by Schenkel et al. [21].

4.4. Combining term proximity and ordering with zone weighting

Until now, we have discussed several scoring models that take different parameters into consideration. BM25TP is specially designed towards evaluating term proximity in a query, whereas our BM25TOP enhancement also takes into consideration the way a query is formulated. On the other hand, BM25F emphasizes on the physical location of a term in a document. Nevertheless, to the best of our knowledge there is no publicly known model which combines term proximity, query term ordering and zone weighting into a single scoring formula.

The main idea is that term proximity is a feature which should be further rewarded when the query terms are positioned within the same zone. For instance, when two or more terms of a given query are encountered in close proximity within the title of a document, then the score of this document should increase.

To incorporate term proximity and zone scoring, we initially replace the document accumulators of Eq. (14) by the *zone accumulators*, determined by the following relationship:

$$acc'_{z_j}(t_x) = \begin{cases} acc'_{z_j}(t_x) + w_{t_x} \frac{1}{\varphi_d(t_x, t_y)}, & t_x, t_y \in z_j \\ acc'_{z_j}(t_x), & otherwise \end{cases} \tag{17}$$

Therefore, instead of assigning one accumulator per query term per document, we assign one accumulator per query term per *zone* and we compute each of them according to Eq. (17).

Now let us return to Eq. (6) which represents the accumulated weights of the query terms over all the document zones. As we have already mentioned, we desire to reward term proximity when the terms of a query appear into the same zone. For this reason, we integrate into these weights our modified accumulator value; the new weights are now evaluated by applying:

$$\mathcal{W}_z(d, t_i) = \sum_{z_{j,d} \in d} S_{z_j} \left(1 + \frac{1}{k_2} \frac{acc'_{z_j}(t_i)}{acc'_{z_j}(t_i) + k_1}\right) \frac{f_{z_{j,d}, t_i}}{1 - b_2 + b_2 \frac{l_{z_{j,d}}}{l_{z_j}}} \tag{18}$$

In this equation, the left quantity rewards the occurrence of a query term in a particular document zone, whereas the right one rewards both term proximity and correct ordering. Notice that the special weight S_{z_j} applies to both terms, since term proximity is not equally important for all zones. Consequently, the occurrence of two adjacent query terms in the title of a document is more significant than a similar occurrence in its plain text. On the second phase, the accumulated weights are being used to compute the BM25TOPF scores according to the following formula:

$$S_{BM25TOPF} = \sum_{i=1}^{n} w_{t_i} \frac{\mathcal{W}_z(d, t_i)}{\mathcal{W}_z(d, t_i) + k_2} \tag{19}$$

where similarly, w_{t_i} represents the IDF k_2 is a predefined constant. BM25TOPF awards documents that contain all, or some of the query terms multiple times in significant locations (i.e. title, headings etc) and moreover, the documents having the query terms close to each other. It also takes into consideration the query term ordering, the document length, the zones length and the inverse document frequencies of the query terms.

As we demonstrate in our experimental section, BM25TOPF improves retrieval effectiveness by a significant margin compared to the existing approaches.

5. Experiments

The evaluation of the proposed methods is divided in two main parts: The first part consists of the experiments we have conducted to attest the efficiency of TZP; to the best of our knowledge, there are no works studying in depth the issue of including zones within the inverted index. Consequently, there is a lack of similar approaches to compare our proposals with. For this reason, we implemented the existing strategies for encoding and organizing the positional data in typical indexes and we slightly modified them in order to support zones.

The experimentation of this part is mainly focused on the presentation of the space and time benefits deriving from the usage of TZP in terms of both compression efficiency and speed of occurrences access and decompression. The document collection we utilize here is the Clueweb09-T09B data set, which is a subset of a larger collection, Clueweb09.[3] This subset consists of 50, 220, 423 pages written in English and occupies approximately 1.42 TB in uncompressed form.

In the real-world Web search engines, each machine typically searches a subset of the collection, consisting of up to tens or hundreds of millions of Web pages Zhang et al. [30]. Each query server maintains its own part of the index, hence the entire index is partitioned into shards Dean [7]. Since in this work we desired to emulate a large-scale search system as precisely as possible, we follow a similar strategy for constructing the index and we provide separate measurements for each shard. In total, the generated index structures in this work consist of ten shards.

The second part of our experiments includes measurements of the quality of the results generated by the BM25TOP and BM25TOPF ranking functions. To ensure an unbiased evaluation, we compared our methods against their three opponents (BM25, BM25F and BM25TP) by employing the Web Adhoc (WA) Task of the TREC-2009 Web Track Soboroff et al. [23]. The query set of WA consists of 50 topics and for each query, a list of relevant documents is provided.

All the results we present in this work are achieved on a machine equipped with CoreI7 920 processor (having the additional processing cores and HyperThreading disabled) and 12 GB of RAM. The system was running the 64-bit distribution of Ubuntu Linux 10.04.

5.1. Experimental index setups

Here we describe the index setups that we shall utilize in order to compare our approach. The first approach is the one illustrated in Fig. 1, a scheme introduced by Ding et al. [8] where the inverted lists are partitioned into blocks of fixed sizes of 128 postings. The authors suggest PForDelta (P4D) for encoding docIDs, frequencies and positions. The positions are organized by using interleaving, that is, each block of the inverted list also accommodates the corresponding positional data. To locate of the positions for a particular posting, we construct the fairly standard hierarchical look-up structure introduced by Yan et al. [29] which stores the appropriate pointers and docID values.

P4D encodes batches of 128 integers (as indicated by Heman [9]); however, there are occasions where we have fewer than 128 integers to encode. In such occasions, we are obliged to construct a *defective* (refer to Boldi and Vigna [3]) block of integers where we use dummy entries to fill up the remaining space. Apparently, this leads to compression losses which become remarkable when encoding positions. For this reason, Yan et al. [29] introduce a more compact P4D variant, namely OptP4D which would automatically adapt to under full chunks by using another compression method such as Variable Byte or Simple16.

VSEncoding is another encoding scheme proposed recently Silvestri and Venturini [22]. In contrast to the other encoding schemes, this one automatically determines the size of each block by computing the optimal manner that a list of integers should be partitioned, with respect to the value of a cost function. In the sequel, VSEncoding encodes the integers of each block by utilizing a fixed number of bits (according to the largest value in the block). Although VSEncoding is found to perform well in decompressing docIDs, in this paper we use it to encode the positional data of an inverted list with the aim of comparing it against TZP.

In our experiments we expanded the three aforementioned index organization approaches with the aim of supporting zones. In particular, for each block of the inverted list we appended a fourth chunk as illustrated in Fig. 4. This fourth chunk is used to store the encoded sequence of the zoneIDs. To locate the positional and zone data we employ a look-up structure similar to the one proposed in Yan et al. [29]. The difference is that apart from the regular pointers showing the location of the positional data, it is also required to store an equal number of pointers pointing at the respective zone data.

Finally, to demonstrate the overall growth in the size of an index caused by the inclusion of zones, we also constructed a standard positional index for our experiments. The setup we selected is similar to the one of Fig. 1. DocIDs and frequencies

[3] http://boston.lti.cs.cmu.edu/Data/clueweb09/.

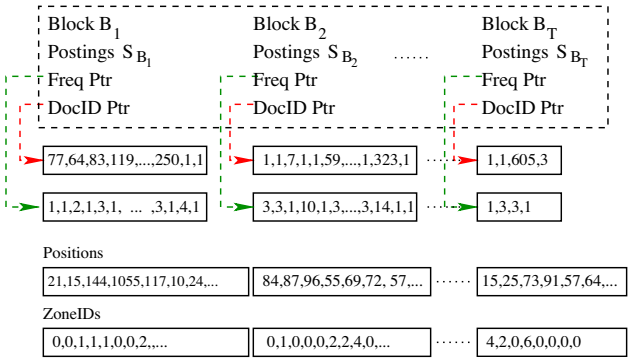


Fig. 4. Expanding the partitioned inverted list of Fig. 1 to store zones. Apart from the three standard chunks which store the docIDs, frequencies, and positions, we allocate one more to store the desired values.

are encoded by using the P4D method, whereas the positional data are organized by using interleaving and compressed by applying the OptP4D variant.

5.2. Evaluation of the space requirements

In this Subsection we examine the space occupied by the aforementioned index organizations. Initially we record the inverted file sizes for both positional and enriched indexes, to evaluate the benefits deriving from the usage of TZP. In the sequel, we measure the space occupied by the accompanying data structures, that is, the skip table and the positions (and zones) look-up structure.

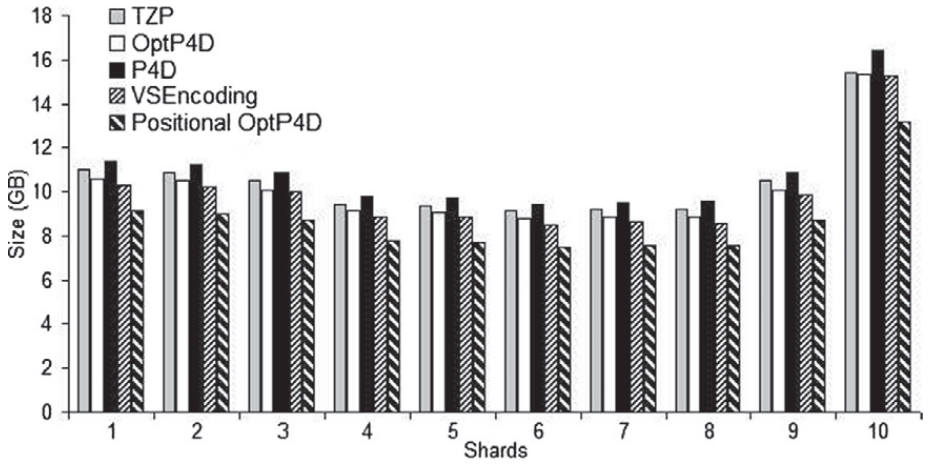
5.2.1. Compressed inverted file sizes

The left part of Fig. 5 illustrates the inverted file sizes expressed in GB for each of the ten constructed shards. Furthermore, in the right part of the same Figure, we present the overall index sizes for each organization approach.

The comparison of TZP against P4D and OptP4D shows that our method is more effective than the plain P4D approach. P4D encodes groups of 128 integers; for each group, we select a parameter b in a manner that the large majority of the data elements (i.e. 90%) can be coded by using b bits. The rest of the elements are called *exceptions* and are coded by employing 8, 16, or 32 bits. On the other hand, TZP encodes the word occurrences by using a fixed number of bits, regardless of their values. Therefore, one could expect that P4D would perform better than TZP.

Nevertheless, this is not valid, since P4D cannot deal effectively with the groups that have fewer than 128 elements each (defective groups). In these groups, the empty space is padded with dummy entries and this leads to significant wasted space. As we already mentioned, OptP4D addresses this problem by using P4D to encode the regular groups containing 128 values, but instead switches to a different code for the defective ones.

In comparison with P4D, TZP produced an index that is about 3.9% smaller. On the other hand, OptP4D and VSEncoding outperformed our method by a margin of 3.2% and 5.3% respectively. However, both of these methods require a look-up structure in order to locate and access the positional and zone data of a particular posting. As we will show shortly, this structure has remarkable space requirements, therefore the overall space occupied by these indexes is increased.



Index Type	Size(GB)
TZP	104.68
OptP4D	101.30
P4D	108.88
VSEncoding	99.16
Posit. OptP4D	92.91

Fig. 5. Compressed inverted file sizes per shard (Left), and total index sizes for all ten shards (Right) for our four experimental setups.

The standard positional index occupies in total approximately 92 GB. In comparison to the other three approaches, we conclude that the inclusion of zones within the inverted index leads to an increase to the occupied space by a margin that fluctuates between 8.3% (for OptP4D) and 14.7% (for P4D).

5.2.2. Sizes of the auxiliary data structures

In this Subsection we examine the space requirements of the auxiliary data structures. The auxiliary structures include the skip table which allows us to partially access and decode an inverted list data during query processing, and the positions look-up structure which enables us to access the positional and zone data for a particular posting. Recall that P4D and OptP4D require both of these structures, whereas TZP is accompanied only by the skip table of Fig. 3.

Our TZP approach suggests storing the occurrence data contiguously (i.e. not interleaving) and maintaining only one pointer per block to locate the desired data per posting. However, P4D and OptP4D encode groups of 128 elements, hence, in an interleaving scheme, a block can contain multiple positional and zone chunks. For each of these chunks, it is necessary to store a separate pointer to be able to access their data. In other words, we are usually obliged to store multiple positional and zone pointers for each block of the inverted list. VSEncoding also encodes groups of integers, but the size of each group is not fixed. However, the phenomenon of a block containing multiple positional and zone chunks still exists.

Additionally, we notice that in case VSEncoding is employed to encode the occurrence data, it constructs a large number of small blocks of integers. More specifically, we found that on average, for each block of 128 docIDs, VSEncoding organizes the respective occurrence data in approximately 41 blocks. If we choose to store two pointers per each of these blocks (one for positions and one for zoneIDs), we must waste about 40.6 GB of disk space for all 10 index shards. Furthermore, the large population of pointers causes a deceleration in query processing, since searching for the right pointer in the aforementioned look-up structure is now slower. For this reason, we store occurrence pointers in a way identical to the one we apply at the OptP4D case (that is, we set pointers per 128 compressed occurrences).

Fig. 6 depicts the sizes of the auxiliary data structures for all of the 10 shards of our examined indexes. Moreover, in the Table 3 we present the overall auxiliary data structure sizes for all ten shards. In the second column of this Table we record the type of the index which makes use of a specific data structure. For instance, the TZP approach exploits the skip table along with the pointers we described in SubSection 3.3, whereas OptP4D employs both the skip table and the occurrence look-up structure.

Notice the difference between the occurrence and position look-up structures. The former stores pointers pointing at the positional and zone data, whereas the latter maintains pointers pointing only at the positional data. The inclusion of these additional zone pointers causes an increase of about 30% to the look-up structure (3.21 GB vs 4.58 GB).

The skip table which includes pointers (recall the upper part of Fig. 3) is much more economic than the look-up structures themselves. As a matter of fact this data structure occupies approximately 53% of the space occupied by the data structures of the positional P4D approach (skip table plus the positions look-up structure, 4.9 GB) and only the 41.4% of the space occupied by the auxiliary structures of the P4D approach (skip table plus the occurrences look-up, 6.27 GB).

In Table 4 we present the overall index sizes (inverted file plus auxiliary data structures) for each of the examined schemes. In conclusion, we notice that the superiority of OptP4D and VSEncoding over TZP in terms of compressed sizes is compensated by the significantly smaller data structures that accompany our proposed index scheme.

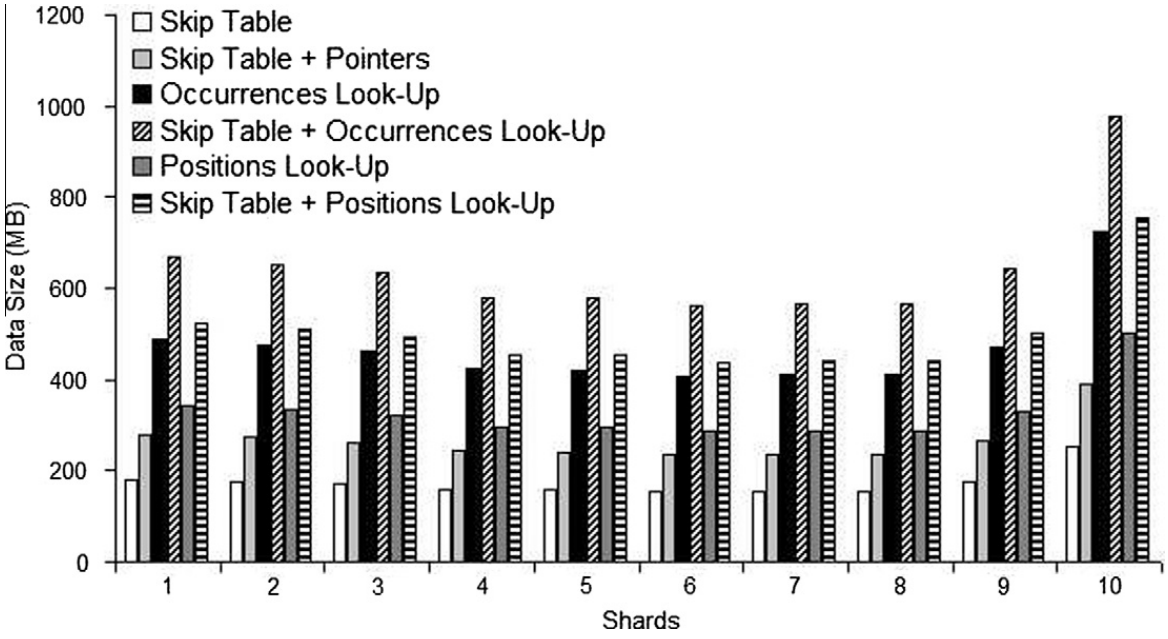


Fig. 6. Sizes of the auxiliary data structures used by our four examined indexes per shard.

Table 3
Total auxiliary data structure sizes for all 10 shards.

Data structure	Index	Size (GB)
Skip table	All except TZP	1.69
Skip table + pointers	TZP	2.60
Positions look-up	Positional OptP4D	3.21
Skip table + positions look-up	Positional OptP4D	4.90
Occurrences look-up	P4D/OptP4D/VSEncoding	4.58
Skip table + occurrences look-up	P4D/OptP4D/VSEncoding	6.27

Table 4
Total inverted index sizes expressed in GB, for all 10 shards.

Index type	Inverted files	Data structures	Total
TZP	104.68	2.60	107.28
OptP4D	101.30	6.27	107.57
P4D	108.88	6.27	115.15
VSEncoding	99.16	6.27	105.43
Positional OptP4D	92.91	4.90	97.81

5.3. Query throughput evaluation

Query throughput is a measure characterizing the ability of a search engine to quickly process the incoming queries. The speed at which a query is evaluated is of critical importance for achieving the highest possible throughput.

For efficiency, search engines process their queries in two phases Zhang et al. [30]: During the first phase, they traverse the inverted lists of the query terms and they identify the best results by using docIDs and frequency values only. In the sequel, they construct the final ranked list by applying more sophisticated techniques (such as the term proximity and/or zone weighting methods) for the K best results of the previous phase only.

In this experimental phase we adopt this strategy with the aim of examining the performance gains which derive from the usage of our model. For the needs of this experiment we employed the query set of the Web Adhoc Task of the TREC-2009 Web Track, comprised of 50 topics (we provide more details on this query set in SubSection 5.4). Initially we attest the occurrence decompression rates achieved by TZP in comparison with P4D and OptP4D. In the sequel, we present the time saved by the omission of the occurrence look-up structure and finally, we measure the average size of the data accessed per query, for each of the index setups of our analysis.

5.3.1. Occurrences access and decompression

In this subsection we discuss the performance gains deriving from the omission of the occurrence look-up structure. The positional and zone data are required during the second phase of query processing therefore, by avoiding to constantly seek for these values leads to significant benefits during this phase only. Recall that TZP allows direct access to the occurrence data without look-ups and moreover, it enables decompression only of the data actually needed during query processing.

To measure the imposed time penalty during query processing, we have implemented this look-up structure for positions and zoneIDs. In Fig. 7 we illustrate the average times consumed to search for the desired data per query, when K is assigned different values.

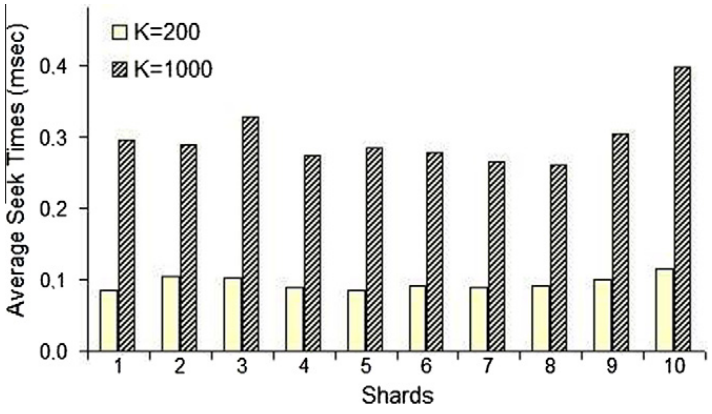


Fig. 7. Occurrence look-up structure: average seek times for positional and zone data per query and per shard.

Table 5
Number of postings involved in the second phase of query processing for $K = 200$ and $K = 1000$.

	$K = 200$	$K = 1000$
Queries	50	50
Query terms	102	102
Postings	19,568	89,969
Postings per query	391.36	1,799.38

Table 6
Occurrence decompression times per query and per posting, for $K = 200$ and $K = 1000$.

K		TZP	OptP4D	P4D	VSE
$K = 200$	Decompressed Occurrences	374,251	2,756,128	2,834,432	2,756,128
	Total Decompression Time (msec)	1.64	8.57	8.34	8.02
	Average Time per Query (msec)	0.0328	0.1714	0.1668	0.1604
	Average Time per Posting (sec)	0.0838	0.4379	0.4262	0.4099
$K = 1000$	Decompressed Occurrences	1,044,234	11,192,608	12,394,752	11,192,608
	Total Decompression Time (msec)	6.04	35.92	34.07	33.15
	Average Time per Query (msec)	0.1208	0.7184	0.6814	0.6630
	Average Time per Posting (sec)	0.0671	0.3992	0.3787	0.3684

The average seek times per query and per shard are 0.1 and 0.3 milliseconds for $K = 200$ and $K = 1000$, respectively. The values of K have not been selected arbitrarily; we assign $K = 200$ because in Yan et al. [29] it is mentioned that higher values of K do not lead to any further precision gains. We also choose to assign $K = 1000$, since the major Web search engines return at most 1000 results in response to the queries they receive. Notice that the average times do not scale proportionally to the values of K, since the value of K increases fivefold whereas the times tripled. This is explained by the fact that some of our test queries produce fewer than 1000 results.

The seek penalty illustrated in Fig. 7 is avoided by employing TZP and the methodology described in SubSections 3.2 and 3.3. Recall that our proposed techniques allow us to pre-calculate the location where the desired data is stored. Hence, no look-up operations are required at the second phase and the speed gains are significant, especially in the case where the system receives thousands of queries per second.

Until now we have shown that the avoidance of occurrences look-ups during query evaluation offers an acceleration to the entire operation. We now demonstrate how our model provides faster occurrence decompression and leads to even better performances. Initially, in Table 5 we present the number of postings processed in the second phase of query evaluation, for $K = 200$ and $K = 1000$, for all the 50 queries that we have submitted to the system.

In SubSection 3.3 we discussed that when encoding occurrences, TZP dominates over the group-based compression schemes such as P4D because we can access only the data actually required and we do not need to decode entire groups of possibly unnecessary values. On the other hand, in case P4D is selected, we need to decode an entire block of 128 elements, even if only one value is required to process the query.

In Table 6 we record the values of several interesting characteristics of the second phase of query processing for our three examined index setups. The first four rows of the table concern the setting $K = 200$, whereas the next four rows represent our measurements for $K = 1000$.

The first row shows the sum of the decompressed occurrences for all 50 queries of our test set. TZP accesses much fewer values than the other approaches since in general, it decodes 7–12 times less data that P4D, OptP4D and VSEncoding. Hence, we expect that our method would decode the desired information much faster than its opponent setups.

Indeed, in case we set $K = 200$, TZP consumes approximately 1.64 milliseconds to decompress the occurrence data for all the 50 submitted queries, a value which is translated to an average of 0.033 ms per query and 83.8 ns per posting. On the other hand OptP4D and P4D were roughly 5.2 and 5.1 times slower, respectively. Regarding VSEncoding, the method is faster than both P4D and OptP4D, but considerably slower than TZP. Similar speed differences are also observed in the case of $K = 1000$; TZP decodes the positional and zone data in times which are 5.9, 5.6, and 5.4 times faster than the respective ones of OptP4D, P4D and VSEncoding.

5.4. Retrieval effectiveness

To evaluate the performance of our proposed method in a fair and unbiased manner, it is required that we obtain a predefined set of queries. Furthermore, for each of these queries, we need to possess a list of humanly judged relevant documents. For this reason, we have employed the results of the Web Adhoc (WA) Task of TREC-2009 Web Track Soboroff et al. [23]. This task consists of 50 topics (test queries) all accompanied by a corresponding list of relevant documents from the Clueweb09-T09B data set.

For the needs of this experiment, we have developed a query serving system consisting of ten query servers and a broker. Each server was assigned a different index shard, whereas the broker was responsible for merging the results generated by

Table 7
Parameter setting for the various ranking methods (Top) and zone weighting scenario for the BM25F and BM25TOPF functions (Bottom).

Parameter	Value
k_1	1.2
k_2	2.0
k_3	2.0
b_1	0.9
b_2	0.75
Zone	S_{z_j}
Body (normal text)	1
Anchor text	1
Title text	6
Document's URL	2
Headings	4
Page description	3
Image description	1
Label text	1

Table 8
Performance of different retrieval methods for the 50 queries of the Adhoc task of TREC-2009 web track.

Retrieval method	MAP	*P*@10	*P*@20	*P*@30	*R*-Precision
BM25	0.0599	0.2820	0.2460	0.2047	0.1127
BM25TP	0.0634	0.2820	0.2530	0.2220	0.1216
BM25TOP	0.0658	0.2940	0.2780	0.2387	0.1238
BM25F	0.0730	0.3140	0.2690	0.2407	0.1318
BM25TOPF	0.0784	0.3360	0.2820	0.2627	0.1390

each server. For the evaluation, we used the 'trec_eval' standard program utilized by the TREC community in order to calculate several measures indicating the retrieval effectiveness of a system. These measures are Mean Average Precision (*MAP*), *R*-Precision and Precision@n(*P@n*) for $n = 10$, 20 and 30.

BM25TOPF is compared against BM25, BM25TP, BM25TOP and BM25F. The values of the different parameters of Section 4 that we used in our experiments are recorded on the left part of Table 7. In addition, on the right part of Table 7 we provide the weighting scenario that we have set in order to evaluate BM25F and BM25TOPF. According to this scheme, a term occurring in a document's title is considered six times more important than one appearing within the main text, whereas the ones appearing in the headings of a document are four times stronger than the normal words.

Table 8 shows the performance of the five examined retrieval methods in the 50 queries of the WA task. The first point that is highlighted by the presented results is that all methods performed better than the plain BM25 model. The improvements are somehow limited when term proximity scoring is considered (BM25TP and BM25TOP), and become significant for our zone weighting scheme (BM25F).

BM25TOPF outperformed all of its adversary approaches and the results indicate that term proximity in combination with zone weighting indeed leads to improved retrieval effectiveness. The Mean Average Precision we achieved by using BM25TOPF was 0.0784 in comparison to the MAP value of 0.073 performed by the second best method, BM25F. This is translated into a result quality improvement of about 6.8%. Furthermore, BM25TOPF performed much better than the proximity-only approaches; the MAP values for BM25TP and BM25TOP were 0.0634 and 0.0658, respectively.

The combination of term proximity with correct term ordering in BM25TP also leads to better performance; BM25TOP produced results which were more qualitative by approximately 3.8% (in terms of MAP) than those generated by BM25TP. However, both term proximity functions were outperformed by BM25F and BM25TOPF. This indicates that zone weighting is a more important feature than term proximity when ranking documents in Web search engines.

Regarding the average Precision values at cut off points 10, 20, and 30, BM25TOPF exhibited better performance that its adversary approaches. The P@10 value was 0.336, 6.5% higher than the corresponding P@10 value achieved by BM25F. Regarding the precision value at cut-off point 20, BM25TOP was slightly outperformed by BM25TOPF, but defeated both BM25TP and BM25F.

6. Conclusions

In this paper we have studied the possibility to integrate additional information within an inverted index. For this reason we attempted to identify several locations of special interest within a Web document, called zones. We assigned each of

these zones a unique identifier (zoneID) and we studied effective ways of enriching the information stored within an index with these identifiers.

In particular, we examined the problem of organizing and compressing such an index. We have adopted block-based index organizations which allows us to individually access and decompress the various inverted list data during query processing. Initially, we introduced occurrences, a piece of information describing both the position of a word within a document, and its corresponding zone. In the sequel, we introduced TZP, a compression method co-operating with all of the existing partitioning strategies encountered in the literature, and offers compact storage of the positional data along with the zoneIDs. TZP uses a 32-bit space to store a single word occurrence, and in the sequel, it allocates a fixed number of bits in order to encode the occurrences of an entire block of the inverted list.

We have also studied how the term occurrences of an inverted list can be efficiently accessed and decompressed. We have demonstrated that by exploiting a small number of pointers, the query processor can directly access the desired data by calculating its location without performing any look-ups in additional structures. This is an advantage that gives a boost to query evaluation. We have also discussed that TZP allows us to decode the information actually needed during query processing, without the need of touching any unnecessary information. Our experiments have shown that the TZP scheme outperforms other state-of the-art approaches by decoding the occurrence data 5–6 times faster.

Finally, we have proposed a ranking function, BM25TOPF which combines term proximity and zone weighting to compute the score of a document during query processing. The new ranking function works in combination with the enriched index only and attempts to prove its usefulness in returning more qualitative results to the user queries.

All these contributions were extensively evaluated through detailed experiments by using an indexing system and a query serving module that we have developed. The data set we employed is the Clueweb09-T09B, a large set of 50 million pages crawled recently. BM25TOPF was evaluated by using a the set of 50 queries of the Web Adhoc Task of TREC-09. Our experiments with this query set have revealed that, in general, BM25TOPF achieves higher precision than the existing state-of-the-art approaches, and provides more qualitative results.

References

[1] V. Anh, A. Moffat, Structured index organizations for high-throughput text querying, Lecture Notes in Computer Science 4209 (2006) 304.
[2] V. Anh, A. Moffat, Index compression using 64-bit words, Software: Practice and Experience 40 (2) (2010) 131–147.
[3] P. Boldi, S. Vigna, Compressed perfect embedded skip lists for quick inverted index lookups, Lecture Notes in Computer Science 3772 (2005) 25.
[4] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Computer Networks and ISDN Systems 30 (1–7) (1998) 107–117.
[5] S. Buttcher, C. Clarke, B. Lushman, Term proximity scoring for ad-hoc retrieval on very large text collections, Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 2006, p. 622.
[6] F. Chierichetti, R. Kumar, P. Raghavan, Compressed web indexes, in: Proceedings of the 18th International Conference on World Wide Web, 2009, pp. 451–460.
[7] J. Dean, Challenges in building large-scale information retrieval systems: invited talk, in: WSDM '09: Proceedings of the Second ACM International Conference on Web Search and Data Mining, 2009, pp. 1–1.
[8] S. Ding, J. He, H. Yan, T. Suel, Using graphics processors for high performance IR query processing, in: WWW '09: Proceedings of the 18th International Conference on World Wide Web, 2009, pp. 421–430.
[9] S. Heman, Super-Scalar Database Compression between RAM and CPU Cache. Master's Thesis, University of Amsterdam, Amsterdam, The Netherlands, 2005.
[10] D. Lee, H. Chuang, K. Seamons, Document ranking and the vector-space model, IEEE Software 14 (2) (1997) 67–75.
[11] W. Lu, S. Robertson, A. MacFarlane, Field-weighted XML retrieval based on BM25, Lecture Notes in Computer Science 3977 (2006) 161–171.
[12] C. Manning, P. Raghavan, H. Schtze, Introduction to Information Retrieval, Cambridge University Press, 2008.
[13] D. Metzler, W. Croft, A Markov random field model for term dependencies, Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, New York, NY, USA, 2005, pp. 472–479.
[14] A. Moffat, J. Zobel, Self-indexing inverted files for fast text retrieval, ACM Transactions on Information Systems (TOIS) 14 (4) (1996) 349–379.
[15] G. Navarro, E. De Moura, M. Neubert, N. Ziviani, R. Baeza-Yates, Adding compression to block addressing inverted indexes, Information Retrieval 3 (1) (2000) 49–77.
[16] V. Raghavan, S. Wong, A critical analysis of the vector space model for information retrieval, Journal of the American Society for Information Science and Technology 37 (5) (1986) 279–287.
[17] Y. Rasolofo, J. Savoy, Term proximity scoring for keyword-based retrieval systems, in: Advances in Information Retrieval: 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 14–16, 2003: proceedings. Springer Verlag, 2003, p. 207.
[18] S. Robertson, K. Jones, Relevance weighting of search terms, Journal of the American Society for Information Science and Technology 27 (3) (1976) 129–146.
[19] G. Salton, A. Wong, C. Yang, A vector space model for automatic indexing, Communications of the ACM 18 (11) (1975) 620.
[20] K. Sayood, Introduction to Data Compression, Morgan Kaufmann, 2000.
[21] R. Schenkel, A. Broschart, S. Hwang, M. Theobald, G. Weikum, Efficient text proximity search, Lecture Notes in Computer Science 4726 (2007) 287.
[22] F. Silvestri, R. Venturini, VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming, in: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, 2010, pp. 1219–1228.
[23] I. Soboroff, N. Craswell, C. Clarke, Overview of the trec 2009 web track, 2009.
[24] V. Tran, H. Tsuji, R. Masuda, A new QoS ontology and its QoS-based ranking algorithm for web services, Simulation Modelling Practice and Theory 17 (2009) 1378–1398.
[25] F. Transier, P. Sanders, Engineering basic algorithms of an in-memory text search engine, ACM Transactions on Information Systems (TOIS) 29 (1) (2010) 2:1–2:36.
[26] H. Turtle, J. Flood, Query evaluation: strategies and optimizations, Information Processing & Management 31 (6) (1995) 831–850.
[27] I. Witten, A. Moffat, T. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann, 1999.
[28] H. Yan, S. Ding, T. Suel, Compressing term positions in web indexes, in: SIGIR '09: Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval, 2009a, pp. 147–154.
[29] H. Yan, S. Ding, T. Suel, Inverted index compression and query processing with optimized document ordering, in: WWW '09: Proceedings of the 18th International Conference on World Wide Web, 2009b, pp. 401–410.

[30] J. Zhang, X. Long, T. Suel, Performance of compressed inverted list caching in search engines, in: WWW '08: Proceeding of the 17th International Conference on World Wide Web, 2008, pp. 387–396.
[31] J. Zobel, A. Moffat, Inverted files for text search engines, ACM Computing Surveys (CSUR) 38 (2) (2006).
[32] M. Zukowski, S. Heman, N. Nes, P. Boncz, Super-scalar RAM-CPU cache compression, in: ICDE '06: Proceedings of the 22nd International Conference on Data Engineering, 2006, p. 59.