Positional Data Organization and Compression in Web Inverted Indexes

Leonidas Akritidis and Panayiotis Bozanis

University of Thessaly, Glavani 37, Volos, Greece

Abstract. To sustain the tremendous workloads they suffer on a daily basis, Web search engines employ highly compressed data structures known as inverted indexes. Previous works demonstrated that organizing the inverted lists of the index in individual blocks of postings leads to significant efficiency improvements. Moreover, the recent literature has shown that the current state-of-the-art compression strategies such as PForDelta and VSEncoding perform well when used to encode the lists docIDs. In this paper we examine their performance when used to compress the positional values. We expose their drawbacks and we introduce PFBC, a simple yet efficient encoding scheme, which encodes the positional data of an inverted list block by using a fixed number of bits. PFBC allows direct access to the required data by avoiding costly look-ups and unnecessary information decoding, achieving several times faster positions decompression than the state-of-the-art approaches.

1 Introduction

Due to the critical importance of the inverted index organization in the overall efficiency of a search engine, a significant part of IR research is conducted towards the determination of an effective index setup strategy. In particular, several works proposed methodologies for storing the index data in a special manner which allows us to skip large portions of the lists during query processing. These approaches suggest partitioning the inverted lists of the index in a number of adjacent blocks which can be individually accessed and decompressed. Undoubtedly, the omission of the unnecessary information stored within the index significantly accelerates the evaluation of a query, since the lists traversal is faster and we also decompress less data.

The benefits of these methods are magnified in the case where we store positional data within the index. This is due to the fact that the size of the positions is several times larger than that of docIDs and frequencies and the indexes containing positional values are about 3 to 5 times larger than the non-positional ones. Therefore, it is extremely important to devise an effective mechanism to organize and compress the positional data, since a naive solution could lead to prohibitively large indexes and reduced query throughput.

In this work we demonstrate that although the current block compression methods are both effective and efficient when applied at docIDs and frequencies, they do not perform equally well when they operate upon the positional data of

an inverted list. We introduce PFBC, a scheme which encodes the positions of an inverted list block by using a fixed number of bits allowing us to a) access the required data almost instantly and b) decode only the data actually needed, without touching any unnecessary information. We demonstrate that with a small cost in space, PFBC outperforms all the adversary compression methods in terms of speed, when applied on the positional data of the index.

2 Background and Related Work

In this Section we provide some elementary information about the inverted index organization and compression and we present the most significant related work.

Due to the huge volumes of text and the great length of the inverted lists, Web search engines store their inverted indexes in highly compressed forms either in main memory or disk. There is a multitudinous family of compression algorithms which can be used to encode the index data. The interested reader can refer to [2], [9] and [11] for overviews and performance benchmarks. Some of the recently proposed state-of-the-art schemes such as PForDelta [7] and VSEncoding [9] are capable of encoding entire bundles of integers achieving both satisfying compression effectiveness and very high decompression speeds.

One of the most important goals when designing query processing algorithms is to *skip* any unnecessary information stored within the inverted lists. For this reason, [8] introduced the block-based index organization, which suggests partitioning each list in blocks of fixed or variable sizes. Within each block, the data is organized in chunks which respectively accommodate the docIDs, the frequency values, and the positional data. Other optimized block-based organizations were introduced in [1], [3], [5].

Although the issue of inverted list partitioning is well-studied, the matter of the organization of the positional data is still open. The research that has been conducted towards this problem resulted into two basic approaches: (a) interleaving, i.e. the positional data belonging to a particular block is stored sequentially after the docIDs and the frequency values, and (b) creating a completely separate structure for positions with its own lookup mechanism. For instance, [12] describe a tree-like look-up structure which operates on interleaved positional data. On the other hand, [10] organize the positions by employing a separate structure, namely indexed list. The problems of these methods are (a) they require increased storage, (b) they decelerate query processing due to the look-up operations and (c) they decode redundant information.

3 The Positions Fixed-Bit Compression (PFBC)

In this Section we describe PFBC, a simple, yet efficient approach for encoding and organizing the positional data of an inverted list.

Our analysis begins by considering the block-based list organization of Figure 1. Suppose that the inverted list I_t of a term t is partitioned into B_{I_t} blocks and each block $B_i \in B_{I_t}$ is comprised of S_{B_i} postings. In the sequel, we identify

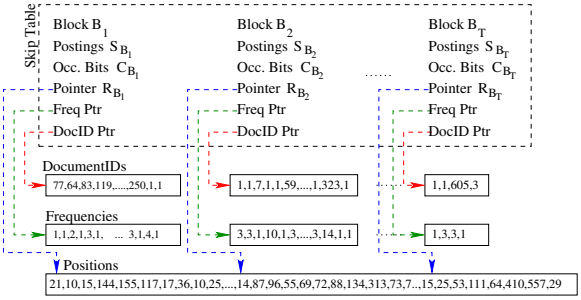

Fig. 1. Organizing an inverted list into blocks according to PFBC.

the highest positional value $|p_{B_i}|_{max}$ for each block B_i of the inverted list and we allocate a number of $C_{B_i} = \lceil log_2(|p_{B_i}|_{max} - 1) \rceil$ bits to produce a binary representation of each occurrence in that block. A pseudocode demonstrating how PFBC encodes a bundle of K positional values is presented in Algorithm 1.

The fixed bit compression methodology of PFBC is expected to introduce some compression loss in comparison to PForDelta. Actually, the latter encodes the largest integers of a list as exceptions and the rest of them by using a fixed-bit scheme, similar to the one we described. This operation is proved to be very effective in the case of docIDs, because in the docIDs blocks the number of large integers is small. However, when P4D is applied at blocks of positional data, the benefits are diminished because in such blocks the number of large integers cannot be predicted. Indeed, as we demonstrate by our experiments, PFBC is outperformed by P4D in terms of compressed sizes by only a small margin.

3.1 Accessing and decompressing the positional data with PFBC

To achieve direct access to the compressed positional data, it is required that we store two values for each block B_i of the list: (a) the aforementioned C_{B_i} value which denotes the number of bits we used to encode the positions of the block B_i, and (b) a pointer R_{B_i} pointing at the beginning of the positional data of B_i. Exploiting this limited amount of information, we are able to calculate the location of the positional data for any posting belonging to B_i. The following equation provides the exact bit $\mathcal{S}_j$ where the positions of a posting j start from:

$$\mathcal{S}_j = R_{B_i} + C_{B_i} \sum_{x=0}^{j-1} f_{x,B_i} \tag{1}$$

where f_{x,B_i} is the x^{th} frequency value stored within B_i. Consequently, to locate the positional data for an arbitrary posting j we first need to dereference the corresponding R_{B_i} pointer value. Then, we need to sum up all the frequency values of the previous postings of the block; this sum reveals the number of the positional values stored between the beginning of the block and the location

Algorithm 1 Encoding a bundle of K positional values with PFBC. After the identification of the highest positional value (steps 3-8), we calculate C, which is the number of bits required to encode all K integers (step 9). The function $write()$ in step 13 is used to store each p_i value into a storage $\mathcal{P}$ by using C bits.

byte $PFBC - Encode(K, p[K])$

1. **int** $i \leftarrow 0, p_{max} \leftarrow 0, C \leftarrow 0$
2. **byte** $\mathcal{P}$
3. **while** $(i < K)$ {
4. **if** $(p_i > P_{max})$ {
5. $p_{max} = p_i$
6. }
7. $i++$
8. }
9. $C \leftarrow \lceil \log_2(p_{max} - 1) \rceil$
10. $\mathcal{P} \leftarrow$ **allocate** $\lceil KC/8 \rceil$ bytes
11. $i \leftarrow 0$
12. **while** $(i < K)$ {
13. $write(p_i, \mathcal{P}, C)$
14. $i++$
15. }
16. **return** $\mathcal{P}$

of the desired data. Since the compressed positions are stored by using a fixed number of bits, we just need to multiply the sum by C_{B_i} to locate the first compressed position of the posting. The operation ends by decoding the next $f_{j,B_i}C_{B_i}$ bits and the positions are retrieved.

PFBC exhibits a wide range of advantages over the adversary approaches:

- It facilitates direct access to the positional data by using equation 1. No expensive look-ups for positions in tree-like structures are required. Consequently, query processing is accelerated;
- It saves the space cost of maintaining a separate look-up structure [12], since the involved pointers can be stored within the skip table;
- It uses fewer pointers than the indexed lists of Transier and Sanders [10];
- It enables decoding of the information actually needed, without the need to decompress entire blocks or sub-blocks of integers.

The R_{B_i} and C_{B_i} values are stored within the the skip structure (upper part of Figure 1); for each entry of the skip table, we also record these two values. This strategy is both effective and efficient; no extra space is required, but only the room occupied by the values themselves. Furthermore, in case the query processor decides that a posting belonging to a particular block should be exhaustively evaluated by decoding its corresponding positional data, we are able to immediately access R_{B_i} and C_{B_i}.

Algorithm 2 includes a pseudocode which demonstrates how PFBC is used to decompress the positional data for a specific posting. To access and decode

Algorithm 2 Decoding the positional data of the j^{th} posting of the block B_i.

int $PFBC - Decode(j, B_i, \mathcal{P})$

1. **int** $x \leftarrow 0, s \leftarrow 0$
2. **while** $(x < j)$ {
3. $\quad s \leftarrow s + f_{x,B_i}$
4. $\quad x++$
5. }
6. **int** $start \leftarrow R_{B_i} + sC_{B_i}$
7. $x \leftarrow 0$
8. **while** $(x < f_{j,B_i})$ {
9. $\quad p[x] \leftarrow read(\mathcal{P}, C_{B_i}, start)$
10. $\quad start \leftarrow start + C_{B_i}$
11. $\quad x++$
12. }
13. **return** p

the positions for the j^{th} posting of the block B_i, we initially accumulate all the frequency values of the previous $j-1$ postings of the block (steps 2–5). In the sequel, we read the R_{B_i} and C_{B_i} values from the skip table and we locate the required data as indicated by Equation 1. If f_{j,B_i} is the associated frequency value of the j^{th} posting, we sequentially read f_{j,B_i} groups of C_{B_i} bits from the compressed sequence; each group represents a positional value of this posting.

4 Experiments

In this Section we compare PFBC against the state-of-the-art compression methods. More specifically, we created three inverted indexes having their docIDs and frequencies organized in the same manner. Each inverted list was split into blocks of 128 postings and P4D was used to encode the DocIDs and the frequencies. In the first index we applied the strategy proposed by [12], i.e. the positions are compressed with OptP4D and accessed by using a separate look-up structure. In the second case, the positions are encoded according to VSEncoding [9] and accessed in a way identical to the one we apply at the OptP4D case (that is, we set pointers every 128 compressed positions).

The sample document collection we employed in our experiments is the Clueweb09-T09B data set, which consists of about 50 million pages. To simulate a real-world search engine environment, the document collection was split up into ten separate segments (called *shards*) consisting of about 5 million documents and each segment was indexed separately [6].

4.1 Compressed index sizes

Now let us evaluate the performance of PFBC against the adversary state-of-the-art approaches in terms of compression effectiveness. Apart from the size of

Data Structure	OptP4D	VSEncoding	PFBC
Inverted Index	90.8	90.2	92.0
Skip Table	1.7	1.7	1.7
Pointers to positions	-	-	2.1
Positions look-up	4.1	4.1	-
Total	96.6	96.0	95.8

Table 1. Overall space requirements (in GB) of our experimental index setups

the inverted file, we also measure the space occupied by the accompanying data structures (i.e. skip table, pointers to positions, and position look-up structure).

In Table 1 we record the overall space requirements of each index setup that we examine. Notice that each organization approach does not make use of all data structures. For instance, our PFBC approach does not require the existence of a positions look-up structure, whereas all strategies employ a skip table. The absence of a data structure is denoted by using a dash symbol.

Among our examined encoding algorithms, VSEncoding achieved the best compression performance; the ten inverted files of all shards occupy in total roughly 90.2 GB. On the other hand, OptP4D performed imperceptibly worse resulting in an inverted file which occupied less than 1% more space. As we anticipated, the usage of PFBC introduced some slight losses; Compared to VSEncoding, the inverted files of PFBC occupied in total about 2% more space.

In Table 1 we report the sizes of the auxiliary data structures. The skip table which includes the positional pointers (recall the upper part of Figure 1) is much more economic than the look-up structures themselves. As a matter of fact this data structure occupies approximately 65% of the space occupied by the data structures of the OptP4D and VSEncoding approaches (skip table plus the positions look-up structure, 5.8 GB). In the last row of Table 3 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 PFBC in terms of compressed sizes is compensated by the significantly smaller data structures that accompany our proposed index scheme. As a result, PFBC presents marginal savings of 0.02-0.08%.

4.2 Query Throughput

In this Subsection we examine the performance of PFBC against the adversary approaches in terms of speed during query processing. To perform this experiment, we submitted a set of 50 conjunctive queries drawn from the Web Adhoc Task of the TREC-2009 Web Track. For each query we measure several statistics such as the decompression times and the size of the accessed data.

The submitted queries were answered by employing a two-stage processing method: During the first phase we traverse the inverted lists of the query terms by employing DAAT, and we quickly identify the most relevant results by accessing docIDs and frequency values only. In the second phase we apply more complex

K		**OptP4D**	**VSEncoding**	**PFBC**
$K = 200$	Decompressed positions	2,756,128	2,756,128	374,251
	Decompressed positions/query	55,123	55,123	7,485
	Total access time (msec)	0.11	0.11	0
	Total decompression time (msec)	5.56	5.14	1.01
	Average time per query (msec)	0.11	0.10	0.02
$K = 1000$	Decompressed positions	11,192,608	11,192,608	1,044,234
	Decompressed positions/query	223,852	223,852	20,885
	Total access time (msec)	0.31	0.31	0
	Total decompression time (msec)	23.94	21.10	4.02
	Average time per query (msec)	0.48	0.42	0.08

Table 2. Access and decode times per query and per posting for different values of K.

ranking schemes such as BM25TP [4] to the K best results determined in the previous stage by retrieving the positional values. We experimented with two values of K; the first one is $K = 200$ and was selected because in [12] the authors prove that higher values do not lead to any further precision gains. Furthermore, since the major Web search engines return at most 1000 results, we also choose to set $K = 1000$. Since the inverted indexes we constructed were comprised of ten shards, we repeated our experiments ten times; each time the query processor was assigned a different index shard. The results of our experiments are illustrated in Table 2.

Table 2 is divided in two parts; the upper part contains the results we recorded for $K = 200$, whereas the lower one includes the results for $K = 1000$. The first line represents the total number of positional values accessed by each method for all the ten index shards, whereas the second line shows the number of the decompressed positions per query. PFBC outperforms the adversary approaches by a significant margin, since the fixed-bit compression scheme allows us to locate exactly the data we need to access and we do not have to decode entire blocks of integers. In total, the organization method with the look-up structure employed by OptP4D and VSEncoding touched 7.4 times more data than the one applied by PFBC for $K = 200$. In case we set $K = 1000$, PFBC is even more efficient, since the other methods decode about 10.7 times more data.

The next line reveals the average position look-up times consumed by each method per query. The algorithm of Table 2 allows PFBC to calculate the location of the positional data without searching for it, consequently, the latency is nullified in this case. Regarding the other two approaches which employ the aforementioned look-up structure, they introduce a latency of about 0.11 msec per query for $K = 200$ and 0.31 msec for $K = 1000$.

Now let us examine the decompression rates achieved by each method. The lines 3 and 7 of Table 2 include the total amount of time required to decode the positional values for all the 50 queries of our experiment. Furthermore, lines 4 and 8 reveal the average decompression time per query. On average, VSEncoding outperformed the OptP4D approach by a margin ranging between 1% and 2% for different values of K. PFBC was the fastest among the evaluated schemes,

since it achieved about 5 times faster decompression compared to VSEncoding for both settings of K.

5 Conclusion

In this paper we introduced PFBC, a method especially designed for organizing and compressing the positional data in Web inverted indexes. PFBC operates by employing a fixed number of bits to encode the positions of each inverted list block, and stores a limited number of pointers which enable the direct retrieval of the positional data of a particular posting. Compared to the current state-of-the-art compression techniques, PFBC offers improved efficiency allowing direct access without look-ups, and very fast decompression. The experiments we have performed on a 50 million document collection demonstrated that in contrast to OptP4D and VSEncoding, the proposed approach touches much fewer data and allows about 5 times faster positions decompression.

References

1. Anh, V., Moffat, A.: Structured index organizations for high-throughput text querying. In: String Processing and Information Retrieval, pp. 304–315 (2006)
2. Anh, V., Moffat, A.: Index compression using 64-bit words. Software: Practice and Experience **40**(2), 131–147 (2010)
3. Boldi, P., Vigna, S.: Compressed perfect embedded skip lists for quick inverted-index lookups. In: String Processing and Information Retrieval, pp. 25–28 (2005)
4. Buttcher, S., Clarke, C., Lushman, B.: Term proximity scoring for ad-hoc retrieval on very large text collections. In: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 621–622 (2006)
5. Chierichetti, F., Kumar, R., Raghavan, P.: Compressed web indexes. In: Proceedings of the 18th international conference on World wide web, pp. 451–460 (2009)
6. Dean, J.: Challenges in building large-scale information retrieval systems: invited talk. In: Proceedings of the Second ACM International Conference on Web Search and Data Mining, pp. 1–1 (2009)
7. Heman, S.: Super-Scalar Database Compression between RAM and CPU Cache. Master's Thesis. University of Amsterdam. Amsterdam, The Netherlands (2005)
8. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Transactions on Information Systems (TOIS) **14**(4), 349–379 (1996)
9. Silvestri, F., Venturini, R.: 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, pp. 1219–1228 (2010)
10. Transier, F., Sanders, P.: Engineering basic algorithms of an in-memory text search engine. ACM Transactions on Information Systems (TOIS) **29**(1), 2 (2010)
11. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes: Compressing and Indexing Documents and Images (1999)
12. Yan, H., Ding, S., Suel, T.: Compressing term positions in web indexes. In: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, pp. 147–154 (2009)