Index Compression for BitFunnel Query Processing

Xinyu Liu, Zhaohua Zhang, Boran Hou, Rebecca J. Stones, Gang Wang*, Xiaoguang Liu*
College of Computer and Control Engineering, Nankai University, China
{liuxy,zhangzhaohua,houbr,becky,wgzwp,liuxg}@nbjl.nankai.edu.cn

ABSTRACT

Large-scale search engines utilize inverted indexes which store ordered lists of document identifies (docIDs) relevant to query terms, which can be queried thousands of times per second. In order to reduce storage requirements, we propose a dictionary-based compression approach for the recently proposed bitwise data-structure BitFunnel, which makes use of a Bloom filter. Compression is achieved through storing frequently occurring blocks in a dictionary. Infrequently occurring blocks (those which are not represented in the dictionary) are instead referenced using similar blocks that are in the dictionary, introducing additional false positive errors. We further introduce a docID reordering strategy to improve compression.

Experimental results indicate an improvement in compression by 27% to 30%, at the expense of increasing the query processing time by 16% to 48% and increasing the false positive rate by around 7.6 to 10.7 percentage points.

CCS CONCEPTS

• **Information systems → Information retrieval query processing**; **Search engine indexing**; **Search index compression**;

KEYWORDS

BitFunnel; Bloom filter; compression; query processing

1 INTRODUCTION

The main index structure in many current search engines [1] is the inverted index. However, motivated by the high efficiency of bitwise operations, predecessors [3, 5] combined bitmaps and inverted indexes to speed up query processing. Recently, Goodwin et al. [2] introduced BitFunnel, a bitmap-like data structure based on a Bloom filter [7].

The underlying data structure is split into *shards* according to document length, and each shard comprises of a collection of *mapping matrices*. Each term maps to a row or a few rows in the mapping matrices in each shard (as determined by Bloom filter multiplexing). Ordinarily, for each $i \in \{0, 1, \ldots, 6\}$, there is a mapping matrix of *rank i*. A column of the rank-i mapping matrix corresponds to a set of 2^i documents, and the columns of the rank-0 mapping matrix corresponds to individual documents. This is illustrated in a toy example in Figure 1.

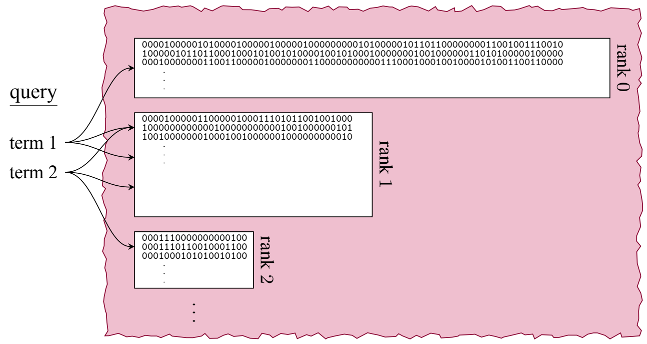


Figure 1: A toy figure of the mapping matrices in a shard, and which rows need to be intersected to process a 2-term query.

In each shard, we independently compute the query intersection results. Roughly speaking, we intersect the relevant rows of equal rank, and we concatenate each rank's result with itself (thereby doubling its length), which is intersected with the next-lower rank intersection results. We do this until we reach rank 0, which determines the final intersection results.

To reduce the impact of false positives, we ensure all the mapping matrices are sparse; this is achieved by varying the number of rows (which affects the size of the mapping matrices, and thus the overall storage requirements).

There is some work on speeding up bitwise operations for BitFunnel [6], however BitFunnel's collection of mapping matrices is costly in terms of space, and is typically larger than the corresponding inverted index. Motivated by these observations, we propose a method for compressing BitFunnel mapping matrices. More specifically:

(1) We propose a dictionary-based compression method for BitFunnel mapping matrices, whereby frequently occurring blocks are replaced by indices to their corresponding blocks in a dictionary.
(2) We also design a document reordering strategy to increase the redundancy in the bitmap structure, reduce the additional false positives to facilitate compression.

2 MAPPING MATRIX COMPRESSION

We propose a dictionary-based compression method, where each mapping matrix of each shard is compressed independently (with each mapping matrix having its own dictionary). We use b-bit

* Corresponding authors. Supported by NSF China grant 61602266, the Science and Technology Development Plan of Tianjin grants 17JCYBJC15300 and 16JCYBJC41900 and the Thousand Youth Talents Plan in Tianjin.

SIGIR '18, July 8–12, 2018, Ann Arbor, MI, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5657-2/18/07…$15.00
https://doi.org/10.1145/3209978.3210086

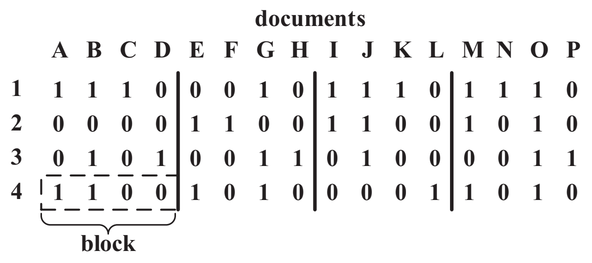
indices to represent k-bit *blocks* as *dictionary references*. We illustrate this method using a toy example in Figure 2. Specifically:

- We add a reference for the k-bit all-1 block and $2^b - 1$ most frequently occurring k-bit blocks to the dictionary. We say a k-bit block is *represented* if it has a dictionary reference; otherwise we say it is *unrepresented*.
- In the mapping matrix, every k-bit block is replaced by a dictionary index, where:
  (1) represented blocks are replaced by their corresponding dictionary index, while
  (2) unrepresented blocks $\mathfrak{b}$ are replaced by a dictionary index which references a sparsest k-bit block $\mathfrak{b}'$ which has a 1 wherever $\mathfrak{b}$ has a 1.

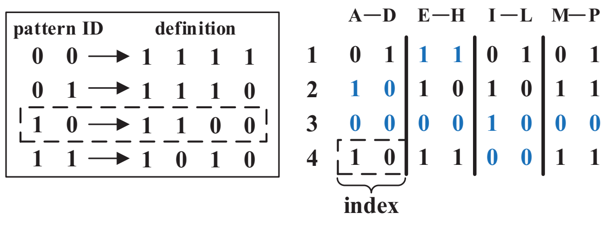
For example, in Figure 2, the highlighted dictionary reference is

$$10 \longrightarrow 1100,$$

and we have $b = 2$ and $k = 4$. The 4-bit block 1100 is represented, so in the mapping matrix, we replace it by its dictionary index (namely 10). However, the block 0010 is not represented, and in the mapping matrix it is replaced by the dictionary index for 1010 (namely 11).



(a) An uncompressed rank-0 mapping matrix, partitioned into k-bit blocks.



(b) The dictionary (left) and the compressed mapping matrix (right) partitioned into b-bit indices.

Figure 2: A toy example of the proposed compression method where k-bit blocks are compressed into b-bit indices (where $k = 4$ and $b = 2$). Unrepresented k-bit blocks have blue-colored b-bit indices.

When performing query processing, we have an additional decompression step whenever a compressed block is encountered. Aside from this, intersection proceeds as in BitFunnel.

In order to reduce false positives, we ensure the matrices are sparse. In a sparse matrix, dense blocks occur less frequently, so we expect a non-uniform distribution of k-bit blocks, and we expect this imbalance assists the proposed compression method.

2.1 Selective compression

After compressing all the matrices, we find that the false positive rate can exceed 60% while the storage cost is only halved (when $k = 32$ and $b = 16$). Therefore, we study selectively compressing rows of the mapping matrices, which is further motivated by Figure 3. To generate Figure 3, we only use rank-0 matrices (Pri in Section 4); we inspect shard 3, which has 4,477 rows and 7,373,033 documents; we experiment with the MillionSet query set (described in Section 4), which has 60,000 queries; and we reorder the rows in descending order of access frequency. We give two plots in Figure 3:

(1) The blue plot is the row access frequency $f(n)$, i.e., how frequently the n-th row is accessed with MillionSet.
(2) For $n \in \{0, 200, \ldots, 4200\}$, we compute the intersection results for the MillionSet query set when the 200 rows $\{n, n+1, \ldots, n+199\}$ are compressed. We define the *false positive rate* as the proportion of incorrectly included documents in the intersection results, which we plot as the red line in Figure 3.

A false 1 may result from either BitFunnel's Bloom filter method or from compression. The original BitFunnel mapping matrix has an overall false positive rate of 8.21% on shard 3, which is a lower bound on the false positive rate for the proposed compression method.

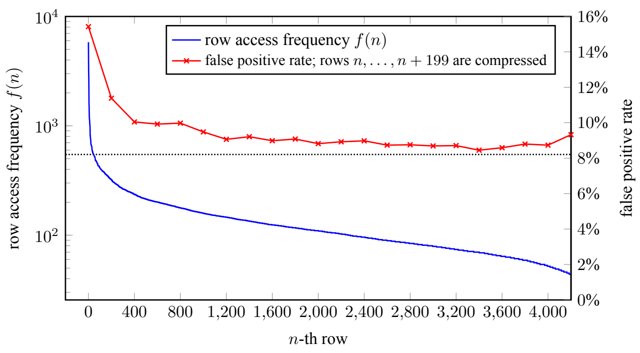


Figure 3: Access frequency (left axis; logarithmic) and the proposed method's false positive error rate (right axis; linear) for each row in the rank-0 mapping matrix for shard 3. The horizontal dashed line marks the lower bound: 8.21% false positive rate.

The main observation we make from Figure 3 is that a small number of rows of the mapping matrix are both accessed frequently and result in a greater proportion of errors—a double impact on the final false positive rate. Thus, we are motivated to store these rows uncompressed, in order to reduce the final false positive rate.

For a parameter $\alpha \in [0, 1]$, we leave uncompressed the first q rows, where q is the minimum value for which the first q rows are accessed at least αN times in total, where N is the total number of row accesses. The decision process is described in Section 4.1.

To implement this method, we randomly split the query set into two equal parts, with the first part used for determining access frequencies (and the other part as a sample of test queries). Thus, the choice of query set affects which rows are compressed, and the overall compression ratio.

3 DOCUMENT REORDERING

We also explore improving the compression through reordering the documents, i.e., permuting the document identifiers. Each shard is reordered independently, and reordering the documents in one shard will affect all the mapping matrices in that shard. So we only use this reordering method for Pri, aimed at decreasing unrepresented blocks in the rank-0 mapping matrix.

We illustrate the process in a toy example in Figure 4. When we compare Figure 2 (before reordering) to Figure 4 (after reordering) we see that the number of unrepresented blocks in the mapping matrix drops from 7 to 1.

documents

	O	B	N	K	J	I	H	L	M	E	C	P	G	A	D	F
1	1	1	1	1	1	1	0	0	1	0	1	0	1	1	0	0
2	1	0	0	0	1	1	0	0	1	1	0	0	0	0	0	1
3	1	1	0	0	1	0	1	0	0	0	0	1	1	0	1	0
4	1	1	0	0	0	0	0	1	1	1	0	0	1	1	0	0

(a) The original mapping matrix.

pattern ID		definition
0 0	$\longrightarrow$	1 1 1 1
0 1	$\longrightarrow$	1 1 0 0
1 0	$\longrightarrow$	1 0 1 0
1 1	$\longrightarrow$	0 0 0 1

	O—K	J—L	M—P	G—F
1	0 0	0 1	1 0	0 1
2	0 1	0 1	0 1	1 1
3	0 1	1 0	1 1	1 0
4	0 1	1 1	0 1	0 1

(b) The dictionary (left) and the compressed mapping matrix (right).

Figure 4: By reordering the documents in the example in Figure 2, we reduce the number of unrepresented blocks.

Given an d-column mapping matrix (where k divides d), we initially reorder the d documents in decreasing order of density, which, for the example in Figure 4, is given by

$$O, J, M, G, B, I, E, A, N, H, C, D, K, L, P, F,$$

which we color d/k documents at a time to illustrate the next step. We then interleave them into d/k groups of k documents: the most-dense d/k columns are moved to the 1st position in each group, the next-most-dense d/k columns are moved to the 2nd position in each group, and so on. In the example in Figure 4, this gives

$$\overbrace{O, B, N, K}^{\text{1st group}}, J, I, H, L, M, E, C, P, \overbrace{G, A, D, F}^{(d/k)\text{th group}}.$$

Motivated by heuristic ideas, we reorder the documents in order to increase block repetition and thereby decrease the number of unrepresented k-bit blocks for reducing false positives. We also try some other reordering strategies, for example, all documents in one mapping matrix or k documents in each block are ordered by density in descending order. However their performances are not as good as the reordering method proposed in this section.

4 EXPERIMENTAL RESULTS

We perform all experiments on the GOV2 collection, consisting of 25,205,183 documents. The content text for each document, including the body and title section in HTML are extracted and stop words are filtered out.

All experiments are carried out on a PC server with two Intel Xeon E5-2650 v4 CPUs and 512GB of memory. The number of physical cores on each CPU is 12 (with 24 threads), and clocked at 2.20GHz. The L1 instruction cache and data cache is 32KB, L2 cache is 256KB, and L3 cache is 30,720KB. The operating system is Linux CentOS 6.5, with kernel version 2.6.32. All programs are implemented in C++11 and are compiled with g++ version 5.4.0, with optimization flag `-O4`.

We use two query sets: (a) *MillionSet*, consisting of queries from the 2007, 2008, and 2009 TREC Million Query Track, containing 60 thousand queries in total; and (b) *TerabyteSet*, consisting of 100 thousand queries from the 2006 TREC Terabyte Track.

Using BitFunnel's original code, we generate mapping matrices using 9 shards and density approximately 0.1. BitFunnel provides the mapping matrices with extra information, but we recover the original mapping matrices through the mapping information (which terms map to which rows). We consider two versions of BitFunnel: (a) *Pri*, where there is only one rank, namely rank 0, and (b) *Opt*, where there are 7 ranks.

Among the various k and b values we test, the best compression rate, false positive rate, and intersection time is observed when $k = 32$ and $b = 16$, so we consistently use these values in the experiments.

4.1 Threshold selection

Different shards have various initial false-positive rates (high α suits those with high initial false-positive rates) and document content, and thus there are different distributions of 1's in the mapping matrices, so we determine α (which determines how many rows are compressed) by some simple experiments. By varying the threshold α we choose how to trade off compression for false positive errors. Figure 5 plots the false positive rate vs. the compression rate as α varies, for shards 0, 3, and 7 (representing the short, middle-sized and long documents). The *false positive rate* is defined as the proportion of false positive documents in the intersection results caused by compression (using TerabyteSet). The *compression rate* is defined as $\text{size}_{\text{comp}}/\text{size}_{\text{orig}}$. For short documents (shard 0), we observe the lowest false positive rate and highest compression ratio.

As a result of this experiment, throughout the paper, we use the α values in Table 1 to ensure the false positive rate caused by compression is around 10%.

Table 1: The threshold α we choose for the experiments. We also list the number of documents ($\times 10^6$).

	shard								
	0	1	2	3	4	5	6	7	8
Pri	0	0.5	0.6	0.7	0.8	0.8	0.8	0.8	0.8
Opt	0	0	0.6	0.8	0.9	0.8	0.8	0.8	0.8
no. docs	2.65	2.61	5.52	7.37	4.50	1.87	0.50	0.16	0.02

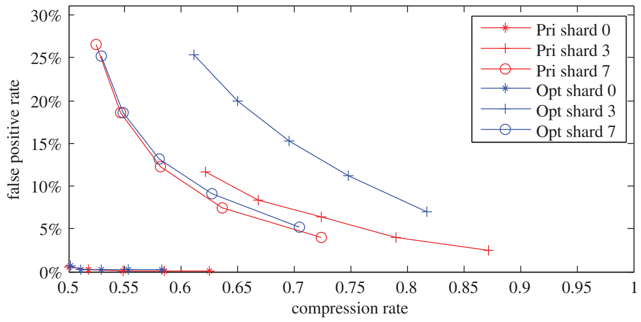


Figure 5: For BitFunnel Pri (red) and Opt (blue), the observed false positive rate and compression rate using the proposed dictionary-based compression method. From left to right, shard 0 has $\alpha \in \{0, 0.1, 0.2, 0.3, 0.4\}$, and shards 3 and 7 have $\alpha \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$. The x-axis starts at 0.5.

4.2 Compression

We first compare the false positive results before and after the reordering process; we use the selective compression method described in Section 2.1. For Pri, after reordering, the false positive rates of MillionSet and TerabyteSet are reduced up to 6.90 and 6.35 percentage points depending on the shard, respectively. Table 2 tabulates the total size of the mapping matrices we encounter. We observe around a 27% and 30% reduction in size for Pri and Opt, respectively.

Table 2: The total size of the mapping matrices, over all shards and ranks.

	size (GB)	bits per posting
Pri	14.22	19.64
Pri (comp.), MillionSet	10.45	14.44
Pri (comp.), TerabyteSet	10.20	14.09
Opt	15.22	21.01
Opt (comp.), MillionSet	10.77	14.87
Opt (comp.), TerabyteSet	10.72	14.80

We also try testing a traditional Bitmap compression method EWAH [4] (run-length), and the mathematical encoding methods Pfor [9] and Vbyte [8]. We see a decrease in size of 4.35% (EWAH), -1.0% (Pfor) and 11.3% (VByte), which is unsurprisingly poor. Due to the uneven distribution of 1's, the 32-bit subsequences in BitFunnel tend to be larger than differences in inverted indices, and all-0 and all-1 subsequences tend to be short, leading to poor compression with these methods.

4.3 Intersection

To give a meaningful comparison, we rewrite the query part of the BitFunnel code, keeping its algorithmic structure, while using the same optimization level and compile options for different methods.[1].

[1]Source code available from https://github.com/BitFunnelComp/dicComp.

In Table 3, we compare the per-query intersection time with and without the proposed dictionary-based compression. We see that the intersection time increases by around 16% to 48% when using dictionary-based compression. While the time for intersection increases due to compression, a 5ms per-query intersection time accounts for a small proportion of the entire query time (including top-k ranking, snippet generation, etc.), which will not have an significant impact on the end user.

In Table 3, we also see that the false positive rate increases by around 7.6 to 10.7 percentage points. Random false positives that arise are likely poorly related to the input query, in which case they would be excluded during, say, the top-k ranking process. As such, we feel this increase in false positives is not as major of a consideration as intersection time.

Table 3: Intersection time per query and the false positive rate.

	MillionSet		TerabyteSet	
	time (ms)	false pos. rate	time (ms)	false pos. rate
Pri	5.34	11.33%	6.18	7.56%
Pri (comp.)	6.44	19.09%	7.15	15.13%
Opt	3.41	6.79%	3.98	4.41%
Opt (comp.)	5.04	17.53%	5.47	13.38%

5 CONCLUSION

In this paper, we propose a dictionary-based method to compress BitFunnel's underlying index structure.

An avenue for future work is to adapt the compression method for use with a GPU-based or multi-threaded intersection method. When limited to a small memory size (e.g., the GPU memory), the proposed compression method would have a more beneficial trade-off. The proposed method could also be adapted to bitmap data structures, where each term corresponds to a unique row. Using bitmaps results in faster intersection, avoids the problem with false positives as a result of the Bloom filter, and may admit better compression, but the initial bitmap is much larger.

REFERENCES

[1] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw.* 30, 1-7 (1998), 107–117.
[2] Bob Goodwin, Michael Hopcroft, Dan Luu, et al. 2017. BitFunnel: Revisiting Signatures for Search. In *Proc. SIGIR.* 605–614.
[3] Andrew Kane and Frank Wm. Tompa. 2014. Skewed Partial Bitvectors for List Intersection. In *Proc. SIGIR.* 263–272.
[4] Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.* 69, 1 (2010), 3–28.
[5] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano indexes. In *Proc. SIGIR.* 273–282.
[6] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, et al. 2017. Ambit: In-memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology. In *Proc. MICRO.* 273–287.
[7] Xiujun Wang, Yusheng Ji, Zhe Dang, Xiao Zheng, and Baohua Zhao. 2015. Improved Weighted Bloom Filter and Space Lower Bound Analysis of Algorithms for Approximated Membership Querying. In *Proc. DASFAA.* 346–362.
[8] Hugh E. Williams and Justin Zobel. 1999. Compressing Integers for Fast File Access. *Comput. J.* 42, 3 (1999), 193–201.
[9] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE.* 59.