On Structures of Inverted Index for Query Processing Efficiency

Xingshen Song[1(✉)], Xueping Zhang[3], Yuexiang Yang[1], Jicheng Quan[2], and Kun Jiang[1]

[1] College of Computer, National University of Defense Technology, Changsha, China
{songxingshen, yyx, jiangkun}@nudt.edu.cn
[2] Aviation University of Air Force, Changsha, China
jicheng_quan@126.com
[3] Information Center, PLA University of Science and Technology, Nanjing, China
university1128@sina.cn

Abstract. Inverted index has been widely adopted by modern search engines to effectively manage billions of documents and respond to users' queries. Recently, many auxiliary index variants are brought up to enhance the engine's compression ratio or query processing efficiency. The most successful auxiliary index structures are Block-Max Index and Dual-Sorted Index, both used for quickening the query processing. More precisely, Block-Max Index is designed for efficient top-k query processing while Dual-Sorted Index introduces pattern matching to solve complex query. There is little work thoroughly analyses and compares the performance of the two auxiliary structures. In this paper, an in-depth study on Block-Max Index and Dual-Sorted Index is presented, with a survey on related top-k query processing strategies. Finally, experimental results on TREC GOV2 dataset with detailed analysis show that Dual-Sorted Index achieves the best query processing performance at the price of huge space occupation, moreover, it sheds light upon the prospect of combining compact data structures with inverted index.

Keywords: Performance evaluation · Inverted index · Block-Max index · Dual-Sorted index · Query processing

1 Introduction

Inverted index is adopted as the core component to fast respond to enormous queries. Given a collection of D documents, an inverted index can be seen as a big table mapping each unique *term* to a *posting list* which contains all the document identifiers (called *docid*) and the number of occurrences in the document (called the *frequency*), and possibly other information like the positions of each occurrence within the documents. We postulate that postings have docids and frequencies, but do not consider other data such as positions or contexts, thus the postings fit in the format (d_i, f_i). The set of terms is called *lexicon*, which is relatively small compared to postings. Its advantages are clear: (i) it removes the redundancy in the documents where the same term occurs more than once, stores only one mapping from term to document instead; (ii) its primitives are composed of docids, document frequencies and positions, these elements can be stored separately or combined arbitrarily; (iii) the posting lists of query terms can be processed in parallel to accelerate the procedure, also, various list intersection and skipping algorithms have been implemented to reach early termination [1–3].

On the other hand, the inverted index may consist of many millions of postings and it can be hardly fit into the main memory, thus compression is needed to save space and reduce the number of disk access. Different ordering schemes and traversal strategies have a large impact on the performance of query response. Also, the inverted index is inferior in searching for substrings and in languages which the terms are not as discrete in English. All these issues have forced variant implementations of inverted index. Different orderings in the lists of documents associated with a term, and different auxiliary information, fit widely different IR tasks. Index designers have to choose the right order for one such task, rendering the index difficult to use for others.

Among various schemes of inverted index, Block-Max Index [4–6] and Dual-Sorted Index [7, 8] are two successful works that have been continuously studied by researchers. Block-Max Index first partitions the sequence of each term into blocks of fixed size(say, 64 or 128 elements), and compresses each block independently with faster list-oriented encoders like simple-X or PFD [9]; then stores the maximum impact value and the head docid for each block in uncompressed form, enabling to skip large parts of the lists. It is simple with little space occupation, but leads to considerable performance gains in conjunctive query and DAAT style pruning approaches. Dual-Sorted Index is a variant of inverted index using wavelet tree, a balanced binary tree-like compact data structure. The wavelet tree can store a sequence (e.g., the posting list) from a symbol universe (e.g., the docid) within asymptotically the same space required by a plain representation of the sequence [10–12]. Dual-Sorted Index allows combining an ordering by decreasing term frequency with an ordering by increasing docid, more importantly, it supports not only typical query scheme, but also sophisticated operations in pattern matching. While researchers keep improving both techniques continuously, missing from the literature is a study that thoroughly measures properties and performances of these two indexes. In this paper, we provide a comprehensive comparison and analysis of the space occupation and response efficiency for different query schemes of these two indexes, using an open source search engine platform-Terrier [13].

This paper is structured as follows: Section 2 provides a background on query processing strategies and wavelet trees; Sect. 3 summarizes both Block-Max Index and Dual-Sorted Index; Sect. 4 describes our experimental setup and comparison results; Conclusions and future work follow in Sect. 5.

2 Background

2.1 Query Processing Strategies

Given a query, the most basic processing form is called *Boolean query processing*, which intersects or merges posting lists of query terms according to their logical relation (AND or OR) [14, 15]. Search engines usually use *ranked query processing*, where a ranking function is used to compute a score for each document passing a simple Boolean filter, and then the k top-scoring documents are returned. To traverse the index structure, there are two basic techniques, DAAT and TAAT. The former searches the posting list of each query term one after another, keeping a temporary accumulator to build the result set; while the latter traverses in an interleaved fashion, keeping aligned by docid. To facilitate different strategies, posting lists are always reordered, in TAAT, lists are sorted by term impact in non-increasing order. Meanwhile, DAAT uses docid-sorted lists instead. DAAT performs well on AND query and supports dynamic pruning algorithms like WAND [16] and Maxscore [17], hence is widely adopted in current search engines.

In WAND, an ingenious pointer movement strategy based on *pivoting* is used, which allows it to skip many documents that would be evaluated by an exhaustive algorithm. It stores for each posting list the highest impact score of any posting in the list, called *maxscore*. There are three steps when the algorithm processes the lists: pivot selection, alignment check and evaluation. Left hand side of Fig. 1 gives a glance into the procedure of WAND, terms are sorted in increasing order by their current docids, maxscores are precalculated and the current minimum score of top-k results is known and used as a threshold, maxscores are accumulated from top to bottom, docid whose sum excesses threshold is chosen as pivot. Thus, we can align the lists above to the position which is no less than pivot docid, if the pivot docid appears in these lists then we evaluate this document, otherwise we sort the lists according to the current docids and pivot again.

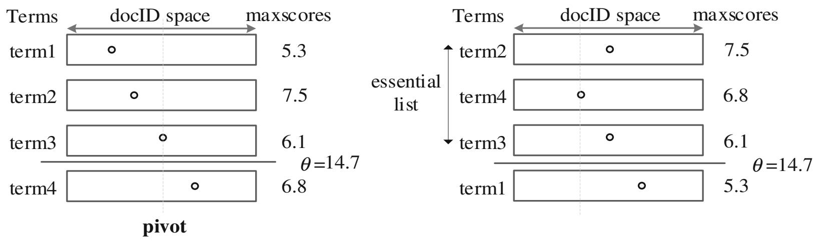

Fig. 1. On the left is the pivot selection phase in WAND. Posting lists are sorted by their current docid and term3 is selected as pivot term. On the right is the selection of essential lists in Maxscore. Lists are sorted by their maximum score and essential lists are those whose sum overpasses current threshold.

Maxscore also precalculates the maximum score of each list, before query processing lists are sorted by their maxscores in decreasing order, after scoring some documents, we again have a threshold that a document must meet to be ranked into current top-k results. According to the threshold lists are divided as essential and non-essential lists. Note that no document can make it into the top-k results just using postings in the non-essential lists, and at least one of the essential terms has to occur in any top-k documents. Each time we pick out the least docid from essential lists, and pointers of other lists are aligned to it, scoring is executed incrementally from top to bottom, once we find current document fail to pass the threshold, scoring is aborted and another docid is picked until essential lists reach their ends.

2.2 Wavelet Tree

Recent years, several researchers have been making efforts to bring the compact data structure to bear on the problems in IR, in particular ranked document retrieval. Brisaboa et al. [18] present an encoding scheme called Directly Addressable Codes (DACs), which enables direct access to any element of the encoded sequences without the need of sampling method, but the symbols must be stored in complete form, making it hard to compress; Culpepper et al. [19, 20] adopt the HSV data structure in combination with a wavelet tree-base representation to retrieve top-k results, but it is not safe; Petri et al. [21] describe a hybrid index consisting of a pruned suffix of document-level posting lists to efficiently handle large intervals, and fast sequential exhaustive processing of smaller sections to create document-level lists on-the-fly. Although quite different in their details, the common vision of these work is to use breakthroughs in compressed pattern matching as an efficient algorithmic base on which the more sophisticated operations required by IR systems can be built.

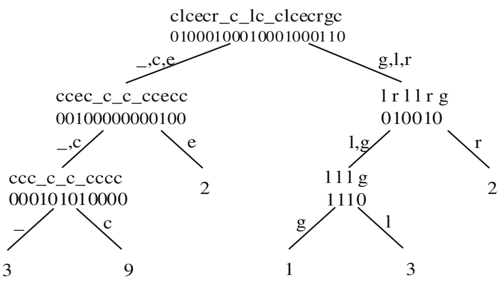

Fig. 2. A wavelet tree over a sequence "clcecr c lc clcecrgc". In each node, the top row shows subsequence $S_v[1, n_v]$ and the second row shows bitmap B_v. Spaces in the sequence is represented by underscores. Since leaf nodes store only one symbol, a sum is saved instead. Here is an example of letters, however, wavelet tree can extend to docid or term frequency sequences easily.

Wavelet tree is a versatile data structure which stores a sequence $S[1, n]$ over an alphabet $\Sigma[0 \cdots \sigma]$, within a space requirement of $nlog\sigma(1 + o(1))$ bits. Figure 2 gives an example of the structure of wavelet tree. The tree is a complete balanced binary tree, where each node handles a range of symbols. The root handles $[0 \cdots \sigma)$, its children nodes bisect the range recursively until reach the leaves which only hold a single symbol. The detail procedure is as follows: each node v in the tree handling the range $[\alpha_v., .\omega_v)$ represents the subsequence $S_v[1, n_v]$ of S formed by the symbols in $[\alpha_v., .\omega_v)$, instead of storing S_v, a bitmap B_v is stored, so that $B_v[i] = 0$ if $S_v[i] < a_v + 2^{[log(\omega_v - \alpha_v)]-1}$ and $B_v[i] = 1$ otherwise. Then the alphabet interval breaks into two roughly equal parts:$[\alpha_\upsilon, \alpha_\upsilon + 2^{\lceil log(\omega_\upsilon - \alpha_\upsilon)\rceil - 1})$ and $[\alpha_\upsilon + 2^{\lceil log(\omega_\upsilon - \alpha_\upsilon)\rceil - 1}, \omega_\upsilon)$, ditto for S_v and B_v, generating their left and right children nodes. It is easy to figure out the tree has a height of $log\sigma$, and it has exactly σ leaves and σ - 1 internal nodes. The B_v in each level exactly occupies n bits, for a total of at most $n\lceil log\sigma\rceil$. Storing the tree pointers, and the pointers to the bitmaps, requires $O(\sigma logn)$ further bits, if we use the minimum $logn$ bits for the pointers.

Within that space, the wavelet tree is able to return any sequence element $S[i]$, and also to answer another two queries that are fundamental in succinct data structure for text retrieval. Note that searching in bitmap B_v to obtain $B_v[i]$ can be solved in constant time, which enables the following three operations return in $O(log\sigma)$ time.

access(S, i): returns the symbol at the position i in the sequence S.
rank$_c(S, i)$: returns the number of times symbol c appears in the prefix $S[1, i]$.
select$_c(S, i)$: returns the position of the i-th occurrence of symbol c in sequence S.

3 Auxiliary Index Structures

3.1 Block-Max Index

A Block-Max Index (BMI), in a nutshell, augments the commonly used inverted index structure, where for each distinct term t we store a sorted list of the docids of those documents where t occurs, with upper-bound values for blocks of these docids. That is, for every say 64 docids of documents containing a term t, we store the maximum term-wise score of any of these documents with respect to t. This then allows algorithms to quickly skip over blocks of documents whose scores are too low to make it into the top results, as shown in Fig. 3.

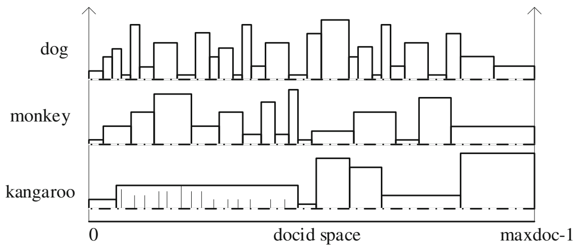

Fig. 3. Three inverted lists are piecewise upper-bounded by the maximum scores in each block. Inside each block are various values in compressed form as shown in the bottom list, one block needs to be decompressed when its maximum score has the potential to be ranked in top-k results, or the whole block will be skipped.

BMI is designed particularly to speed up early termination algorithms like Maxscore and WAND. We refer to the approach in [4] as Block-Max WAND (BMW), and the approach in [22] as Block-Max Maxscore (BMM). BMW defines two functions used in list traversal: *deep pointer movement* refers to receiving a docid from the current list that is equal to or greater than a given one and it usually involves a block decompression; *shallow pointer movement* only moves the current pointer to the corresponding block without decompression. BMW picks a candidate pivot using the list maxscores as in WAND, but then it uses shallow movement to check if it is necessary to decompress the block and evaluate the pivot based on the maxscores of the block, which helps filtering out most of the candidates and achieving much faster query response. Also, if a pivot d fails to make it into the top results, instead of picking the docid next to it, the pointer is moved to the header of the next block, since d is ruled out based on the maxscore of current block. Different from BMW, BMM uses a preprocessing step rather than an online one to detect and align block boundaries. The blocks are repartitioned into intervals with interval boundary, then BMM runs Maxscore within each interval, selecting non-essential lists and doing partial scoring using block maxscores instead of list maxscores.

Shan et al. [5] propose an optimization on BMI which combines the static score such as PageRank and the IR score into one score to give a correct and better estimation of document's upper bound score, they also recalibrate candidate documents using local BMW and BMM to omit invalid scoring. Dimopoulos et al. [6] compare performances of WAND- and Maxscore-based algorithms with and without BMI, then build on their observations by designing and implementing new techniques for exploiting BMI, in particular docid-oriented block selection schemes, on-the-fly generation of BMI, and a new recursive query processing algorithm that uses a hierarchical partitioning of inverted lists into blocks. The core idea of their improvement is to decouple the choice of blocks for storing block maxscores from the choice of blocks for inverted index compression, making the BMI as a structure separate from the inverted lists. In some cases, much smaller blocks are chosen to get better pruning power, however, this also results in a much larger space occupation. To solve the problem, they have defined block boundaries based not on the number of postings in a block, but based on docid space and carefully tuned the block size to achieve good space-time tradeoffs, moreover, on-the-fly Block-Max generation is proposed to further shrink the size of BMI with little time overhead.

3.2 Dual-Sorted Index

The main data structure used in Dual-Sorted Index (DSI) is wavelet tree. To make it suitable for document retrieval, symbols of letters are substituted by docids in the range $[1, D]$, where D denotes the total number of documents in the collection. Let $L_t[1, df_t]$ be the list of docids in which term t appears, in decreasing tf order. Let $N = \sum_t df_t$ be the total number of occurrences of distinct terms in the documents. All the posting lists L_t are concatenated into a unique list $L[1, N]$, and the starting position s_t of list L_t within L is also stored. The sequence L of docids is then represented with a wavelet tree. Note that the structure is a complete balanced binary tree with D leaves. The leaves are labeled left-to-right with the symbols $[1, D]$ in increasing order. For any internal node v of the wavelet tree, let L_v be a subsequence of L containing only the docids on the leaves in the subtree with root v. At each node v of depth ℓ, the docids are split by their most significant bit (*msb*) in the position of $logD$ - ℓ, for each docid, if $msb_{logD\text{-}\ell} = 0$ it will appear below the left child of v, or the right child otherwise. With this property, the docid can be restored by recording its path (e.g. consider alphabet 0, 1, 2, 3, 4 = 000, 001… 100; after 3 times turning left, we descend down to the leaf 0).

Note that L_t of each term is sorted in decreasing *tf* order, symbols in wavelet tree are sorted in increasing order inherently, thus making the index dual-sorted for different type of retrieval. The *tf* values are sorted in differential and run-length compressed form in a separate sequence. Finally, the s_t sequence is represented using a bitmap $S[1, N]$, preprocessed for *rank* and *select* queries. Thus $s_t = select_1(S, t)$, and also $Rank_1(S, i)$ tells where $L[i]$ belongs to. Analysis of wavelet tree shows that space occupied by L is $NH_0(L) + o(NlogD)$ bits, here $NH_0(L) = \sum_d dt_d log \frac{N}{dt_d} \leq NlogD$ and dt_d denotes the number of distinct terms in document d. The *tf* values are stored in a sequence $W[1, N]$ aligned to L, which is also asymptotically similar to the space needed by L. The s_t values are represented to support constant time *rank* and *select* queries, requiring $Vlog\frac{N}{V} + O(V) + o(N)$ bits, which is less than the usual pointers from the vocabulary to the list of each term and here V denotes the number of distinct terms in collection. Roberto improves DSI's time efficiency using a fast implementation from [23] that uses 37.5 % extra space on top of bitmap since L is not expected to be compressible, also s_t is replaced by V pointers from term t to the starting positions of the list L_t in L.

The three fundamental queries are detailed as follows. For $access(L, i)$, a look-up is started from the root node v, if $B_v[i] = 0$ we descend to the left child and i is updated using $i = rank_0(B_v, i)$, or right child with $i = rank_1(B_v, i)$ otherwise. This process is continued recursively until a leaf is reached and $L[i]$ is the concatenated path in binary. For $rank_c(L,i)$, bits in c tell the path to descend, the only thing needs to do is updating i to be the number of times of the current bit in depth ℓ appears up to position i in the node until a leaf is reached. For $select_c(L,i)$, the look-up is processed upwards to the root, as the path is already clear, each level ℓ_i. i is updated by $i = select_0(B_\ell, i)$ if $c[\ell] = 0$, or $i = select_1(B_\ell, i)$ if $c[\ell] = 1$. More complex operations can be implemented by combining the above three queries like retrieving all the values in a range $L[i, j]$ and retrieving the k-th value in a range $L[i,j]$. These algorithms can be easily adopted in Boolean conjunctive and disjunctive queries, when we find the $|q|$ intervals $[s_t, e_t]$ of the query words using pointers from the vocabulary to the inverted lists L_t, we can track all the $|q|$ ranges simultaneously and recursively merge or intersect them to obtain a result. Note that the docids in each list are sorted by term frequencies, so we can immediately compute the documents score and retain the k highest scoring documents. See more details of the algorithms in [7, 8].

4 Experiments

4.1 Experimental Setup

We implement both BMI and DSI for comparison, as an external implementation to compare we choose Terrier as a baseline, which is a highly flexible and effective open source search engine. Terrier supports both disjunctive and conjunctive queries in

DAAT and TAAT traversals with different weighting models, we will show the result that although inferior to the two state-of-the-art indexes, it does gain performance achievement after carefully tuned.

In our experiments, we use the TREC GOV2 collection, which consists of 25.2 million web pages and about 32.8 million terms in the vocabulary crawled from the gov Internet domain. The uncompressed size of these web pages is 426 GB. The collection is indexed using Terrier IR platform, all terms have the Porter stemmer applied, and stopwords have been removed. The docids are assigned by the sequence of their occurrence.

For retrieval, a total of 1000 queries are selected from the TREC2005 Efficiency track queries and classified into different categories according to the number of distinct terms in the query. Unless stated otherwise, the default number of documents retrieved for each query equals to 20 that a result page will contain and BM25 is used as ranking function in order to keep consistent with work in [4, 6].

All the implementations are carried out on an Intel(r) Xeon(r) E5620 processor running at 2.40 GHz with 128 GB of RAM and 12,288 KB of cache. The default physical block size is 16 KB, unless stated otherwise algorithms are implemented using C ++ and compiled with GCC 4.8.1 with –O3 optimizations. In all our runs, the whole inverted index is completely loaded into main memory, in order to warm up the execution environment, each query set is run 4 times for each experiment, and the response times only measured for the last run. Our implementations are available at https://github.com/Sparklexs/Dualsorted-master.

4.2 Index Size Comparison

First we compare the index size of each structure. The baseline is compressed using both Gamma and OptPFD. As depicted in [4], BMI compresses docid-gaps and frequencies using OptPFD, compared with baseline OptPFD, BMI barely expands its size, as it only augments the index with local maxscores and pointers to the header of blocks. For DSI, its docids and frequencies are stored separately, the frequency lists are compressed using Gamma codec, however, the docid lists are hard to compress since they are stored in wavelet trees in primitive form. Moreover, in order to achieve direct access to any symbol in the tree, some additional structures are also stored, all these result in the size of DSI grows nearly 4 times larger than the rest indexes. Table 1 gives the detail of each index. We also show the results after docid reassignment for the collection. The idea of docid reassignment is to reorder the documents in the collection so that similar documents are clustered together, thus shrinking the gap between docids and the index size, it also improves the speed of dynamic pruning for that both potential and invalid documents are batched up to reach an early termination, here we choose to reorder the documents based on an alphabetic sorting of their URLs. As the right hand side of Table 1 shows, baseline and BMI reduce nearly 20 % size after docid reassignment, however, DSI seems insensitive to reassignment as it hardly changes size in both situations, this is mainly due to the fact that DSI separately stores docids in their primitive form and frequencies in decreasing order without relying on docid order. Also note that BMI cuts out the same size as baseline OptPFD, which can be explained by the fact that they adopt the same structure except BMI stores its additional information in its raw format.

Table 1. Size in GB of different inverted index for GOV2

	before reordering	after reordering	Δ
Baseline Gamma	9.86	7.11	2.75
Baseline OptPFD	9.45	7.33	2.12
BMI OptPFD	10.86	8.75	2.12
DSI Gamma	39.2	39.2	0

4.3 Query Processing Efficiency

Next, we compare the timing results of each index for different length of queries. First we use the index without reordering and the processing strategy chosen for baseline index is DAAT, to distinguish different compressions we name the result of baseline Gamma as GDAAT and ODAAT for baseline OptPFD, BMM is short for BMI using Maxscore and BMW for BMI using WAND. For now there isn't any dynamic pruning techniques used in DSI in the literature, as arbitrary symbols in the index are directly accessible, so we simply use DAAT-like strategy for DSI (which is quite different from ordinary DAAT since the documents are not fetched in ascending order). The results are shown in Fig. 4, we omit the result of a single term query, since all the processing strategies degenerate into DAAT with only one posting list and performances of these methods become the same. It is worth mentioning that our implementations achieve a close performance to what has been reported in [21].

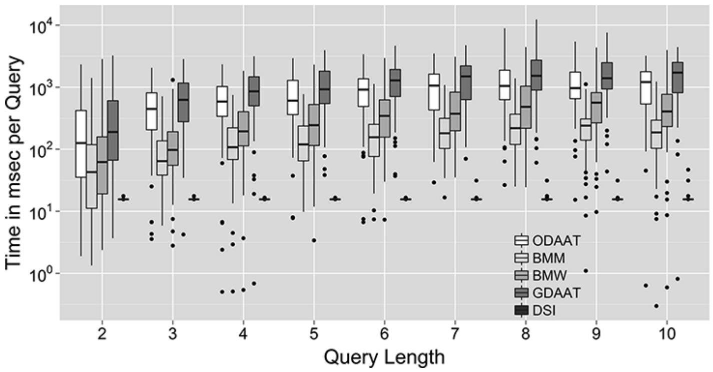

Fig. 4. Efficiency for queries of different length using different indexes. The interquartile range is shown in the box part and solid line in the box represents the median, points represent the outliers. One thing to take notice of is that the box part of DSI shrinks into a solid line.

It is clear that the same structure using different compressions gains performance gap between ODAAT and GDAAT, which can be explained by the fact that shift and concatenation operations are inevitable to decode an integer using bit-oriented codecs, however, list-oriented codecs can decode a batch of integers with a single operation. BMI with dynamic pruning methods further improves processing efficiency, nonetheless, BMM achieves better performance than BMW, which is different from the result in [21], after an inspection into the procedure of both methods, we conclude that the list sorting consumes much time in pivot selection phase of BMW, also BMM selects candidate documents in the essential lists which are likely to belong to more important terms, while BMW selects candidate documents based on their docids without considering the terms' importance, with more query terms, it causes more mis-scoring candidates that slows down the performance of BMW. Here we only implement the basic BMI, some other methods like on-the-fly BMI generation and hierarchical-layered blocks adopted in [6] remain future investigation. When it comes to DSI, we get an interesting result that its time consume is rarely low compared with other strategies, which are an order of magnitude higher than DSI; another surprising phenomenon is that it stays considerably stable when query length grows while other strategies raise their time consume to different extent, the box part is contracted into a solid line which can be hardly noticed. It can be concluded without doubt that DSI outperforms all the others. However, this performance is achieved at the cost that the huge size structure of DSI is fully loaded into the main memory and occupies nearly 50 GB space, while others keep low memory occupation within 6 GB. This result is also consistent with the inference before, we postulate that scoring a document and maintaining a priority queue for the result cost constant time, then the only factor that influences time efficiency is the size of the collection, as the three basic operations can be implemented in $O(logD)$ time, so is the intersection and merge of the posting lists. Also the documents in DSI are sorted by term frequency in decreasing order, the potential candidates to be ranked in top-k lie in the front of each posting list. With this in mind, we can just evaluate few documents rather than the whole list and the query time is significantly reduced. Another thing to be noticed is that the outliers always appear below the box, which is caused by the fact that posting lists for uncommon terms are missing or pruned by dynamic pruning strategies.

Second we reproduce this experiment with reordered index. Figure 5 shows the results for the case of reordered indexes, as expected, performance of structures which use dynamic pruning techniques shows promising benefits in term of query response time compared to the prior. However, DAAT and DSI remain unaffected, as DAAT processes the query in an exhaustive way and documents in DSI are ordered by term frequency actually. Reordering works extremely well for BMW, performance gap between it and BMM is sharply reduced. Indeed, reordering clusters the related documents and shorten the time of pivot alignment. For query length no longer than 5, BMM and BMW are about the same efficiency, their average response times fall below 10^2 ms, as query length grows, the performance gap rises again.

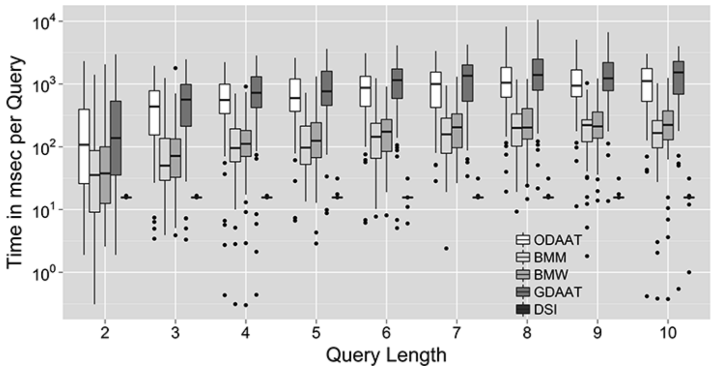

Fig. 5. Efficiency for queries of different length using reordered indexes.

5 Conclusions

In this paper we have presented an explicit study of two state-of-the-art inverted index structures used in safe retrieval in disjunctive queries. Our main contribution is an experimental comparison of these two indexes combined with different compression techniques and traversal strategies, we also discuss the effect on query processing efficiency brought by docid reassignment. As shown in the result, DSI reveals its superiority over other techniques in query response, however, a serious disadvantage is obvious that it occupies too much storage and memory space. Overall, DSI is a promising structure taking its feature of bridging the gap between IR problems and pattern matching data structures into account.

There are still many open problems and opportunities for future research, including mitigating the space issue using on-the-fly index generation and extending DSI with dynamic pruning techniques and list-oriented compression codecs.

References

1. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. (CSUR) **38** (2), 6 (2006)
2. Lemire, D., Boytsov, L., Kurz, N.: SIMD Compression and the Intersection of Sorted Integers (2014). arXiv:1401.6399
3. Barbay, J., López-Ortiz, A., Lu, T., Salinger, A.: An experimental investigation of set intersection algorithms for text searching. J. Exp. Algorithmics (JEA) **14**, 7 (2009)
4. Ding, S., Suel, T.: Faster top-k document retrieval using block-max indexes. In: Proceedings of the 34th international ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 993–1002. ACM (2011)
5. Shan, D., Ding, S., He, J., Yan, H., Li, X.: Optimized top-k processing with global page scores on block-max indexes. In: Proceedings of the fifth ACM International Conference on Web Search and Data Mining, pp. 423–432. ACM (2012)

6. Dimopoulos, C., Nepomnyachiy, S., Suel, T.: Optimizing top-k document retrieval strategies for block-max indexes. In: Proceedings of the sixth ACM International Conference on Web Search and Data Mining, pp. 113–122. ACM (2013)
7. Navarro, G., Puglisi, S.J.: Dual-sorted inverted lists. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 309–321. Springer, Heidelberg (2010)
8. Konow, R., Navarro, G.: Dual-Sorted Inverted Lists in Practice. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 295–306. Springer, Heidelberg (2012)
9. Catena, M., Macdonald, C., Ounis, I.: On inverted index compression for search engine efficiency. In: de Rijke, M., Kenter, T., de Vries, A.P., Zhai, C., de Jong, F., Radinsky, K., Hofmann, K. (eds.) ECIR 2014. LNCS, vol. 8416, pp. 359–371. Springer, Heidelberg (2014)
10. Navarro, G.: Wavelet trees for all. J. Discrete Algorithms **25**, 2–20 (2014)
11. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theoret. Comput. Sci. **426**, 25–41 (2012)
12. Claude, F., Navarro, G.: The wavelet matrix. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 167–179. Springer, Heidelberg (2012)
13. Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., Lioma, C.: Terrier: A high performance and scalable information retrieval platform. In: Proceedings of the OSIR Workshop, pp. 18–25 (2006)
14. Li, X., Wang, Y., Li, X., et al.: Parallelizing skyline queries over uncertain data streams with sliding window partitioning and grid index. Knowl. Inf. Syst. **41**(2), 277–309 (2014)
15. Ma, D., Rao, L., Wang, T.: An empirical study of SLDA for information retrieval. In: Salem, M.V.M., Shaalan, K., Oroumchian, F., Shakery, A., Khelalfa, H. (eds.) AIRS 2011. LNCS, vol. 7097, pp. 84–92. Springer, Heidelberg (2011)
16. Petri, M., Culpepper, J.S., Moffat, A.: Exploring the magic of WAND. In: Proceedings of the 18th Australasian Document Computing Symposium, pp. 58–65. ACM (2013)
17. Turtle, H., Flood, J.: Query evaluation: strategies and optimizations. Inf. Process. Manage. **31**(6), 831–850 (1995)
18. Brisaboa, N.R., Ladra, S., Navarro, G.: DACs: bringing direct access to variable-length codes. Inf. Process. Manage. **49**(1), 392–404 (2013)
19. Culpepper, J., Navarro, G., Puglisi, S.J., Turpin, A.: Top-*k* Ranked document search in general text databases. In: Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
20. Culpepper, J.S., Petri, M., Scholer, F.: Efficient in-memory top-k document retrieval. In: Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 225–234. ACM (2012)
21. Petri, M., Moffat, A., Culpepper, J.S.: Score-safe term-dependency processing with hybrid indexes. In: Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval, pp. 899–902. ACM (2014)
22. Chakrabarti, K., Chaudhuri, S., Ganti, V.: Interval-based pruning for top-k processing over compressed lists. In: IEEE 27th International Conference on Data Engineering (ICDE), 2011, pp. 709–720. IEEE (2011)
23. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proceeding of WEA, pp. 27–38 (2005)