Faster BlockMax WAND with Longer Skipping

Antonio Mallia[1] and Elia Porciani[2]

[1] Computer Science and Engineering, New York University, New York, US
[2] Sease Ltd., London, UK

Abstract. One of the major problems for modern search engines is to keep up with the tremendous growth in the size of the web and the number of queries submitted by users. The amount of data being generated today can only be processed and managed with specialized technologies. BlockMax WAND and the more recent Variable BlockMax WAND represent the most advanced query processing algorithms that make use of dynamic pruning techniques, which allow them to retrieve the top k most relevant documents for a given query without any effectiveness degradation of its ranking. In this paper, we describe a new technique for the BlockMax WAND family of query processing algorithm, which improves block skipping in order to increase its efficiency. We show that our optimization is able to improve query processing speed on short queries by up to 37% with negligible additional space overhead.

Keywords: Top-k query processing · Inverted index · Early termination.

1 Introduction

In the past two decades, the amount of data being created has skyrocketed. The key to unlock the full potential of these huge datasets is to make the most of advances in algorithms and tools capable to handle it.

Many parts of the search engine architecture, including data acquisition, data analysis, and index maintenance, are facing critical challenges. Nevertheless, query processing is still the hardest to deal with, since workload grows with both data size and query load. Although hardware is getting less expensive and more powerful every day, the size of the data and the number of searches is growing at an even faster rate. Much of the research and development in information retrieval is, indeed, aimed at improving retrieval efficiency.

While, query processing in search engines is a fairly complex process, most systems appear to process a query by first evaluating a fairly simple ranking function over an inverted index. We focus on improving this initial step, which is responsible for a significant fraction of the overall work.

Traversing the index structures of all the query terms and computing the scores of all the postings is the way to evaluate exhaustively a user query. Unfortunately, the cost of each query increases linearly with the number of documents, making it very expensive for large collections. To overcome this problem, many researchers have proposed so-called early-termination techniques, for finding the top-k ranked results without computing or retrieving all posting scores.

In this work, we focus on such techniques for improving query processing efficiency without degrading effectiveness to rank K (known as *safe-to-rank*).

Our Contributions . We list here our main contributions.

1. We propose an optimization for the BlockMax WAND (BMW) family of algorithms, which exploits particular sequences of block max scores in order to perform longer skipping.
2. We embed an additional data structure that stores precomputed skips in order to overcome the run time search overhead introduced by compressing the block boundaries.

2 Background and Related Work

We first briefly explain the studied methods for both index compression and query processing. We refer to the referenced papers for more details that are omitted due to space restriction.

Index Organization. We consider a collection of documents indexed in an inverted index [14]. Each term occurring in the collection contributes a list of IDs of the documents containing it (usually along with respective document frequencies or other data used to rank documents), called a posting list. As a requirement for *Document-at-a-Time* (*DAAT*) query processing, which scans postings lists concurrently, the document IDs must be sorted in the ascending order. This allows us to compress it efficiently making it possible to keep the entire index in memory.

$\mathsf{NextGEQ_t(d)}$ is an operator which returns the smallest document ID in the inverted list of term t that is greater than or equal to d. A fast implementation of the function $\mathsf{NextGEQ_t(d)}$ is crucial for the efficiency of this process and it is strictly dependent on the compression algorithm used. One widely adopted solution to efficiently implement $\mathsf{NextGEQ_t(d)}$ operator is to divide each list into blocks that are individually encoded with the chosen encoding method.

Block-based encoding is not optimal when skipping is performed among the inverted lists, because it requires to decode an entire block to access a single element. This reflects the access pattern of early termination algorithms such as BMW, where entire segments of the posting lists are skipped. For this reason, we have chosen to use Partitioned Elias-Fano [9] as compression technique, which provides random access to compressed elements without decoding the whole sequence. Partitioned Elias-Fano has been recently proposed as an improvement of Elias-Fano, initially introduced by Vigna [12], in order to exploit the local clustering that inverted lists usually exhibit, resulting in reduced space usage.

Query Processing. Several algorithms have been proposed to accomplish exhaustive evaluation over an inverted index to find top-k documents. Broder et al. [1] introduced for the first time WAND, a solution which exploits an augmented index with maximum scores for each term of the posting lists. The algorithm maintains a top-k priority queue of the scores of the evaluated document, such

that a minimum threshold needs to be met by a document to enter the top-k. The idea behind WAND is to access the posting lists with an iterator keeping the postings ordered by ID. Each iteration of the algorithm sorts the terms by the current ID of the associated iterator and adds up the maximum scores of the terms until the threshold is reached. This allows to find the minimum document ID which has to be evaluated, allowing us to safely ignore all the preceding ones. BlockMax WAND [6] further improves WAND by better estimating the upper bounds by splitting a posting list into fixed-sized blocks and storing the maximum score per block. Additionally, BlockMax WAND refines the score upper bound of a candidate ID found by WAND by using these upper bounds. This operation is fast, as it involves no block decompression. Whenever the maximum score estimation would not be sufficient to enter the top-k, we can skip all document IDs belonging to the intersection of the current blocks involved, translating to a move to the minimum document of the current block boundaries.

Variable BlockMax WAND (VBMW) [8] generalizes BMW by allowing variable lengths of blocks. More precisely, it uses a block partitioning such that the sum of differences between maximum scores and individual scores is minimized. This results in better upper bound estimation and more frequent document skipping with the downside of a computational overhead at index building time in order to compute the optimal block partitioning.

3 Our Contribution

The efficiency of early termination algorithms is closely related to the number of documents skipped during index traversing. In the case of the BlockMax WAND family, the greatest contribution to skipping happens after the block upper bound is computed, specifically when the aggregated score does not reach the threshold and a move to the next block boundary can be safely performed. Advancing the term iterator to the following minimum block boundary is driven by the intuition that no documents in the current blocks can exceed the upper bound estimation. On the other hand, this choice is not guaranteed to be optimal.

We introduce a modification of the BMW algorithm which uses a new strategy to advance the term iterator farther than the current block boundaries, which results in longer but safe document skipping. The observation behind our strategy is that when an iterator is updated, if the max score of the block after the boundary does not increase, then we can state that the sum of the blocks upper bound will still not be greater than the threshold. The next iteration will then perform unnecessary computation until a new block skipping happens again. Our solution consists of identifying the next document ID to move at, by progressively skipping entire blocks until one with greater block max score is found. Algorithm 1 depicts how the new document ID is chosen.

Our first contribution is to implement this "longer skipping" strategy in both BMW and VBMW algorithms, where we named our variations BMW-LS and VBMW-LS respectively. To the best of our knowledge, there is no evidence in literature of BMW additional data being compressed, in the way it is done for

VBMW. Also, we experimented with both compressed version of the algorithms, named here C-BMW and C-VBMW (refer to [8] for the details) for the ones using the unmodified skipping strategy; C-BMW-LS and C-VBMW-LS for the ones using the new skipping strategy.

Query time search for a longer skip, although reduces the fully evaluated documents, is expensive from a computational point-of-view because of the compressed blocks information. For this reason, we introduced an alternative approach of the proposed solution which precomputes the skip size at index build time, storing the information interleaved with the blocks information. Considering that we need to store only one additional information – the distance in number of blocks to the last one that we can skip – we have chosen to encode it with a fixed number of bits. This represents a limitation for the maximum number of subsequent blocks that can be skipped, but we have experimentally observed that after a certain amount of bits per element there is no performance advantage. In our implementation and for the examined datasets, we used 3 bits per element, so that up to 7 blocks skips can be encoded. We named this implementation Precomputed Longer Skipping (PLS).

Algorithm 1 Find next doc ID

```
1:  next_docid ← MAX_DOCID
2:  for all t ∈ terms do
3:      block ← t.block
4:      s ← block.score
5:      docid ← block.boundary
6:      while block.score <= s do
7:          docid ← block.boundary
8:          block ← block.next
9:      end while
10:     if docid < next_docid then
11:         next_docid ← docid
12:     end if
13: end for
```

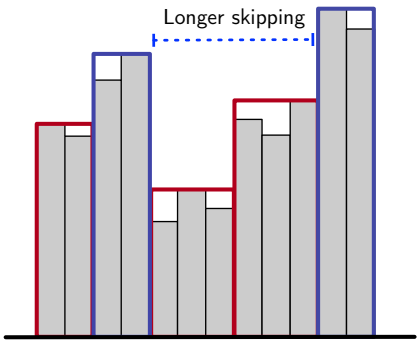

Fig. 1. Example of Longer Skipping

4 Experimental Results

Testing details All the algorithms are implemented in C++14 and compiled with GCC 7.3.0 with the highest optimization settings. The tests are performed on a machine with 8 Intel Core i7-4770 Haswell cores clocked at 3.40GHz with 32GiB RAM running Linux 4.15.0. The indexes are saved to disk after construction and memory-mapped to be queried so that there are no hidden space costs due to loading of additional data structures in memory. Before timing the queries we ensure that the required posting lists are fully loaded in memory. All timings are measured taking the results with minimum value of five independent runs. All times are reported in milliseconds.

The source code is available [3] for the reader interested in further implementation details or in replicating the experiments.

Datasets We performed our experiments on the following standard datasets.

- Gov2 is the TREC 2004 Terabyte Track test collection consisting of 25 million `.gov` sites crawled in early 2004; the documents are truncated to 256 kB.
- ClueWeb09 [2] is the ClueWeb 2009 TREC Category B collection consisting of 50 million English web pages crawled between January and February 2009.

For each document in the collection the body text was extracted using Apache Tika[4] and the words lowercased and stemmed using the Porter2 stemmer; no stopwords were removed. The doc IDs were assigned according to the lexicographic order of their URLs [11].

	Gov2	ClueWeb09
Documents	24622347	50131015
Terms	35636425	92094694
Postings	5742630292	15857983641

Table 1. Statistics for the test collections

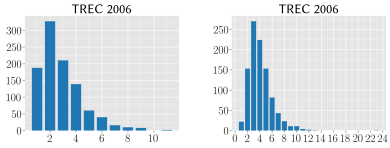

Fig. 2. Query length distribution

Queries To evaluate the speed of query processing we use the TREC 2005 and TREC 2006 Terabyte Track Efficiency Task, drawing only queries whose terms are all in the collection dictionary. From those sets of queries, we randomly select 1000 queries for each length. Figure 2 depicts the query distribution, which clearly shows that short queries dominate.

We have used BMW and VBMW as baseline with 40 elements per block in average, in both their uncompressed and compressed form. All the results, including the query times in milliseconds for the baselines and our proposed solutions, are presented in Table 2.

In our experiments, the proposed optimization improves short queries (from 2 to 4 terms) with negligible performance degradation for longer queries. This has shown to be true for both Gov2 and ClueWeb09 datasets and without noticeable differences for TREC 2005 and TREC 2006. Because of its pluggability, based on the query length this optimization can be enabled at query time with the intent of maximizing query performance. Although there is a noticeable improvement for all short queries, the maximum speedup is observable using ClueWeb09 on TREC 2005 queries where BMW-LS performs 37% faster than BMW and VBMW-LS reduces by 31% the time spent to process the query. We have chosen to show the results for BMW because it could be a better choice in the case where block-based compression algorithms [13] are used and because it has a simpler and faster offline build process where compared to VBMW; our optimization is orthogonal to any further improvements built on top of BMW [7, 4, 3, 5, 10].

[3] `https://github.com/pisa-engine/pisa/tree/ecir19-ls`
[4] `http://tika.apache.org`

Table 2. Query times (in ms) of different algorithms for several query lengths.

		Gov2					ClueWeb09				
		2	3	4	5	6+	2	3	4	5	6+
TREC 2005	BMW	1.22	3.07	4.68	7.43	16.73	4.63	11.37	16.68	25.72	55.99
	VBMW	0.99	1.91	2.69	4.21	**9.18**	3.17	6.39	8.92	14.46	32.04
	BMW-LS	0.93	2.88	4.61	7.41	17.40	2.92	10.20	16.76	26.84	60.42
	VBMW-LS	**0.78**	**1.77**	**2.63**	**4.20**	9.23	**2.18**	**5.66**	**8.57**	**14.44**	**31.95**
	C-BMW	1.33	3.39	5.13	8.27	18.26	5.19	12.78	19.09	29.19	63.32
	C-VBMW	1.10	2.08	2.93	**4.60**	**10.16**	3.53	6.97	9.86	**16.06**	36.26
	C-BMW-LS	1.38	3.42	5.32	8.26	18.74	5.36	13.11	19.42	29.93	65.08
	C-VBMW-LS	1.14	2.15	3.04	4.75	10.21	3.67	7.34	10.29	16.46	36.48
	C-BMW-PLS	1.12	3.10	5.00	8.00	18.77	3.89	11.19	18.41	29.58	65.80
	C-VBMW-PLS	**0.94**	**1.95**	**2.93**	4.71	10.17	**2.68**	**6.30**	**9.52**	16.07	**36.01**
TREC 2006	BMW	1.11	3.58	6.24	10.03	23.85	3.46	11.33	19.82	32.37	74.13
	VBMW	0.78	2.26	3.58	5.55	**12.88**	2.28	6.80	11.64	18.68	42.17
	BMW-LS	0.85	3.22	6.08	9.98	24.86	2.50	10.42	19.77	33.28	80.62
	VBMW-LS	**0.58**	**2.05**	**3.46**	**5.49**	12.92	**1.66**	**6.25**	**11.35**	**18.59**	**42.04**
	C-BMW	1.22	3.90	6.95	11.09	26.34	3.80	12.48	22.27	35.83	82.96
	C-VBMW	0.89	2.49	3.96	6.08	14.60	2.51	7.42	12.86	**20.40**	**46.87**
	C-BMW-LS	1.28	4.02	7.19	11.17	27.07	3.96	12.91	22.85	37.02	85.59
	C-VBMW-LS	0.91	2.57	4.06	6.23	14.88	2.61	7.68	13.21	20.99	47.48
	C-BMW-PLS	1.02	3.57	6.56	10.75	26.52	3.09	11.57	21.92	36.52	85.45
	C-VBMW-PLS	**0.72**	**2.34**	**3.86**	**6.07**	**14.54**	**1.98**	**6.88**	**12.51**	20.46	47.33

In contrast, it is interesting to notice that for the compressed version of the algorithms the run time optimization does not lead to any improvements, but actually results in a slower execution. The precomputed version of our optimization overcomes this issue and obtains almost the same gain of the run time version for the uncompressed BMW with a negligible overhead in index size (less than 1% of the total index size). PLS optimization is omitted for the uncompressed variants, considering that linear scan does not suffer the decompression overhead.

5 Conclusions

In this paper, we demonstrated the applicability of a longer skipping strategy to both BMW and VBMW, which results in marked benefits of processing time for short queries. We proposed two different variations. The former evaluates at query time the size of the possible skips and finds its best applicability with uncompressed blocks score information, while the latter precomputes and stores into the index this information which is ideal when blocks scores are compressed. Our extensive experiment analysis shows that both strategies improve on their direct competitors by up to 37%, with a negligible additional space usage in case of precomputed skips.

Finally, in the future, we also want to study the combination of our algorithm to existing and new threshold estimation techniques to study how those can be beneficial when combined with our longer skipping strategy.

Acknowledgments. Antonio Mallia's research was partially supported by NSF Grant IIS-1718680 "Index Sharding and Query Routing in Distributed Search Engines".

References

[1] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th Intl. Conf. on Information and Knowledge Management*, pages 426–434, 2003.

[2] J. Callan, M. Hoy, C. Yoo, and L. Zhao. Clueweb09 data set, 2009. URL `http://lemurproject.org/clueweb09/`.

[3] Caio Moura Daoud, Edleno Silva de Moura, André Luiz da Costa Carvalho, Altigran Soares da Silva, David Fernandes de Oliveira, and Cristian Rossi. Fast top-k preserving query processing using two-tier indexes. *Inf. Process. Manage.*, 52:855–872, 2016.

[4] Caio Moura Daoud, Edleno Silva de Moura, David Fernandes de Oliveira, Altigran Soares da Silva, Cristian Rossi, and André Luiz da Costa Carvalho. Waves: a fast multi-tier top-k query processing algorithm. *Inf. Retr. Journal*, 20:292–316, 2017.

[5] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proc. of the 6th ACM Intl. Conf. on Web Search and Data Mining*, pages 113–122, 2013.

[6] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proc. of the 34th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 993–1002, 2011.

[7] Andrew Kane and Frank Wm. Tompa. Split-lists and initial thresholds for wand-based search. In *Proc. of the 41st Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 877–880, 2018.

[8] Antonio Mallia, Giuseppe Ottaviano, Elia Porciani, Nicola Tonellotto, and Rossano Venturini. Faster blockmax WAND with variable-sized blocks. In *Proc. of the 40th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 625–634, 2017.

[9] Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano indexes. In *Proc. of the 37th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 273–282, 2014.

[10] Oscar Rojas, Veronica Gil-Costa, and Mauricio Marin. Efficient parallel block-max wand algorithm. In *Proceedings of the 19th International Conference on Parallel Processing*, pages 394–405, 2013.

[11] Fabrizio Silvestri. Sorting out the document identifier assignment problem. In *Proceedings of the 29th European Conference on IR Research*, pages 101–112, 2007.

[12] Sebastiano Vigna. Quasi-succinct indices. In *Proc. of the 6th ACM Intl. Conf. on Web Search and Data Mining*, pages 83–92, 2013.

[13] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. of the 18th Intl. Conf. on World Wide Web*, pages 401–410, 2009.

[14] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.