Efficient top-k document retrieval

Antonio Mallia

Computer Science and Engineering, New York University, NY, US
antonio.mallia@nyu.edu

Abstract. Over the past few decades, the IR community has been making a continuous effort to improve the efficiency of search in large collections of documents. Query processing is still one of the main bottlenecks in large-scale search systems. The top-k document retrieval problem, which can be defined as reporting the k most relevant documents from a collection for a given query, can be extremely expensive, as it involves scoring large amounts of documents. In this work, we investigate the top-k document retrieval problem from several angles with the aim of improving the efficiency of this task in large-scale search systems. Finally, we briefly describe our initial findings and conclude by proposing future directions to follow.

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.

The most common structure used for text retrieval is an *inverted index*. For each term in a parsed collection, it stores a list of numerical IDs of documents containing this term, typically along with additional data, such as term frequencies or precomputed quantized impact scores. We call all values associated with a (term, document)-pair a *posting*.

The first problem we encounter is efficient index representation. Given the extremely large collections indexed by current search engines, even a single node of a large search cluster typically contains many billions of integers. In particular, compression of posting lists is of utmost importance, since they account for much of the data size and access costs.

Likewise, the choice of a retrieval algorithm is crucial to the efficiency of query processing. While query processing in search engines is a complex process, most systems appear to process a query by first evaluating a fairly simple ranking function over an inverted index. Due to the large sizes of most search collections, retrieving all potential matches is infeasible and undesirable. Most advanced query processing algorithms make use of dynamic pruning techniques, which allow them to skip document evaluation while being able to retrieve the top-k most relevant documents for a given query without any effectiveness degradation of its ranking.

In this paper, we pose the following research questions:

RQ1 How does the choice of query processing algorithm depend on the compression method used?

RQ2 How is performance impacted by the number of results returned by the query processing algorithm?

2 Related Work

2.1 Index compression

Compression is an important technique employed in indexing systems to reduce the size of an inverted index. A large amount of research has as primary objective the inverted index space occupancy minimization, while maintaining fast query processing speed.

Binary packing. *Binary Packing* [1] groups numbers into fix-sized blocks. For each block, its selector b is the smallest number of bits required to binary-encode the largest element of the group. The selector is binary-encoded in one byte, followed by the values belonging to the group, each encoded in b bits. For better performance, we can store four selectors at a time in one 32-bit word, followed by their respective groups. Lemire and Boytsov [5] proposed a Binary Packing method that exploits SIMD instructions. This method, called SIMD-BP, packs 128 consecutive integers into as few 128-bit words as possible. Selectors are stored in groups of 16, to fully utilize 128-bit SIMD operations.

Variable Byte. The encodings in the *Variable Byte* family are known for their high decoding speed. Arguably the simplest and best known is VByte [12], which uses 7 bits per byte to store the binary representation of a number and one remaining bit to indicate whether the same binary code continues in the next byte. These continuation bits, when put together, form unary-encoded byte-lengths of encoded numbers. To improve decoding speed, Dean [3] proposed VarintGB, which groups these lengths together and encodes them in binary instead: one byte is used to store four 2-bit sizes of the next four integers, followed by their binary representations.

Recently, several SIMD-based implementations of variable-byte encodings have been shown to be extremely efficient [3, 10, 11]. Stepanov et al. [11] analyzed a family of SIMD-based algorithms, including a SIMD version of Group Varint, and found the fastest to be Varint-G8IU: Consecutive numbers are grouped in 8-byte blocks, preceded by a 1-byte descriptor containing unary-encoded lengths (in bytes) of the integers in the block. If the next integer cannot fit in a block, the remaining bytes are unused.

PForDelta. PForDelta [16] encodes a large number of integers (say, 64 or 128) at a time by choosing a k such that most (say, 90%) of the integers can be encoded in k bits. The remaining values, called exceptions, are encoded separately using another method. More precisely, we select b and k such that most values are in the range $[b, b + 2^k - 1]$, and thus can be encoded in k bits by shifting them to the range $[0, 2^k - 1]$. One variant, OptPFD[15], selects b and k to optimize for space or decoding cost, with most implementations focusing on space.

Elias-Fano. Given a monotonically increasing integer sequence S of size n, such that $S_{n-1} < u$, we can encode it in binary using $\lceil \log u \rceil$ bits. Instead of writing them directly, Elias-Fano coding [14] splits each number into two parts, a low part consisting of $l = \lceil \log \frac{u}{n} \rceil$ right-most bits, and a high part consisting of the remaining $\lceil \log u \rceil - l$ left-most bits. The low parts are explicitly written in binary for all numbers, in a single stream of bits. The high parts are compressed by writing, in negative-unary form (i.e., with the roles of 0 and 1 reversed), the gaps between the high parts of consecutive numbers. Sequential decoding is done by simultaneously retrieving low and high parts, and concatenating them. Random access requires finding the locations of the i-th 0- or 1-bit within the unary part of the data using an auxiliary succinct data structure. Furthermore, a NextGEQ(x) operation, which returns the smallest element that is greater than or equal to x, can be implemented efficiently. Observe that h_x, the higher bits of x, is used to find the number of elements having higher bits smaller than h_x, denoted as p. Then, a linear scan of l_x, the lower bits of x, can be employed starting from posting p of the lower bits array of the encoded list.

The above version of Elias-Fano coding cannot exploit clustered data distributions for better compression. This is achieved by a modification called *Partitioned Elias-Fano* (PEF) [9] that splits the sequence into b blocks, and then uses an optimal choice of the number of bits in the high and low parts for each block.

2.2 Query processing

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 processing 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 all posting scores.

MaxScore. In 1995 Turtle and Flood [13] firstly proposed an algorithm named MaxScore. MaxScore is an optimization technique which relies on the maximum impact scores of each term. It has been applied to both term-at-a-time (TAAT) and document-at-a-time (DAAT) evaluation strategies, but the latter is definitely the most effective optimization.

Given a list of query terms $q = \{t_1, t_2, \ldots, t_m\}$ such that $\max_{t_i} \geq \max_{t_{i+1}}$, at any point of the algorithm, query terms (and associated posting lists) are partitioned into *essential* $q_+ = \{t_1, t_2, \ldots, t_p\}$ and *nonessential* $q_- = \{t_{p+1}, t_{p+2}, \ldots, t_m\}$. This partition depends on the current threshold T for a document to enter the top-k results, and is defined by the smallest p such that $\sum_{t \in q_-} \max_t < T$. Thus,

no document containing only nonessential terms can make it into the top-k results. We can now perform disjunctive processing over only the essential terms, with lookups into the nonessential lists.

Variable Block-Max WAND. Similar to MaxScore, WAND [2] (Weighted or Weak AND) also utilizes the $\max_t$ values. One shortcoming of these techniques is that they use maximum impact scores over the entire lists. Thus, if $\max_t$ is much larger than the other scores in the list, then the impact score upper bound will usually be a significant overestimate. BlockMax WAND (BMW) [4] addresses this by introducing block-wise maximum impact scores. Variable BlockMax WAND [7, 6], or just VBMW, 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.

3 Results and Discussion

Testing details. All methods are implemented in C++17 and compiled with GCC 7.3.0 using the highest optimization settings[1][8]. The tests are performed on a machine with an Intel Core i7-4770 quad-core 3.40GHz CPU, with 32GiB RAM, running Linux 4.15.0.

Data Sets. We performed experiments on two standard datasets: Gov2 and ClueWeb09. For each document in the collection, the body text was extracted, the words lowercased and stemmed using the Porter2 stemmer; no stopwords were removed.

Queries. To evaluate query processing speed, we use TREC 2005 and TREC 2006 Terabyte Track Efficiency Task. We sample 500 queries from each set.

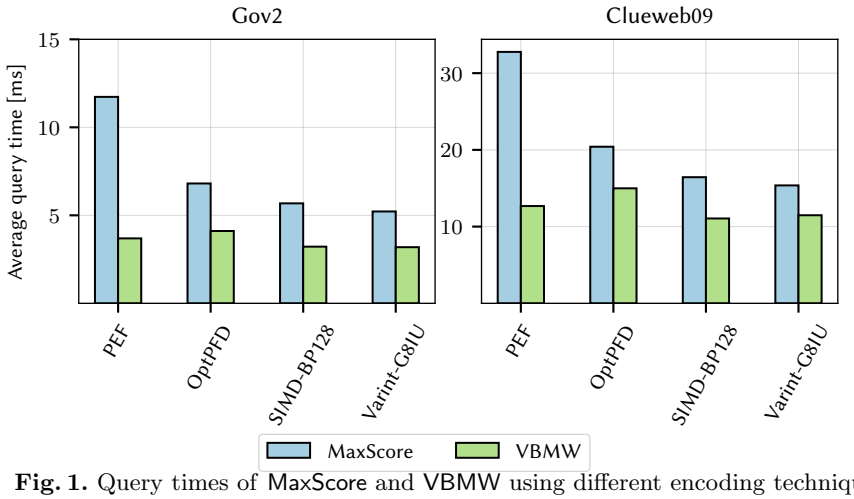

Fig. 1. Query times of MaxScore and VBMW using different encoding techniques.

[1] https://github.com/pisa-engine/pisa/

3.1 Improving Partitioned Elias-Fano

Figure 1 shows the average query time of MaxScore and VBMW with indexes encoded using different encoding techniques. It is interesting to notice that although PEF is time-efficient with VBMW, when it comes to the MaxScore query processing algorithm PEF performs poorly. The reason can be identified in PEF's ability to efficiently perform skipping within a posting list, while being suboptimal in sequential decoding. In fact, the latter still represents a considerable fraction of the time spent on processing a query using MaxScore. Future research could focus on redesigning PEF to have a twofold decoding ability. While we desire to maintain its fast skipping capabilities, PEF needs to match the sequential decoding speed of the modern SIMD-enabled competitors. As a result, a query algorithm could exploit this duality by switching to the most appropriate access policy while traversing the index.

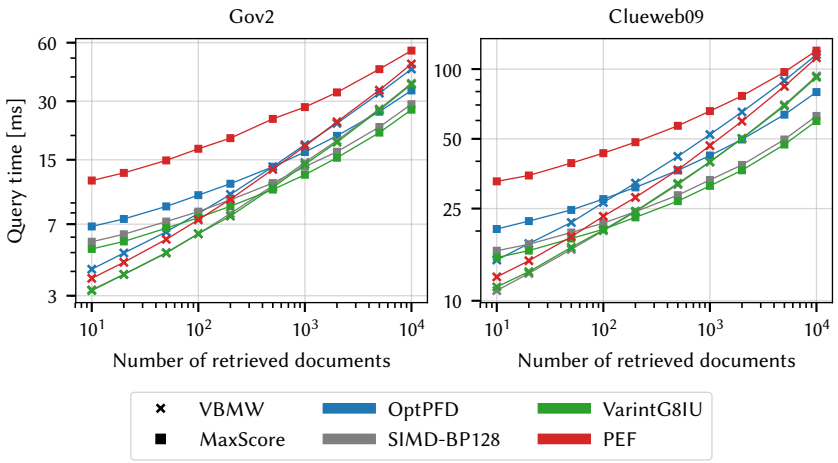

Fig. 2. Query times of MaxScore and VBMW for different k.

3.2 Improving MaxScore for large k

Much previous work has focused on smaller values of k, such as 10 or 100. However, when query processing algorithms are used for subsequent reranking by a complex ranker, more results are needed. The results for a range of values of k are shown in Figure 2 on a log-log scale for better readability. First, we notice a significant time increase with larger k across all encoding techniques. We find this increase to be faster for VBMW. Also, note that at $k = 10,000$ even the performance of PEF, which previously performed well only for VBMW, is similar for both algorithms. While we intend to explore and gain a deeper insight on the motivations behind VBMW results in order to study a solution able to overcome these issues, we acknowledge that MaxScore might be better suited for some cases of top-k document retrieval. We believe that there is a potential for further speeding up MaxScore in scenarios where k is large. As mentioned in Section 2.2, nonessential lists are only used for lookups in order to

refine the document scores. It would be interesting to analyze the possibility of redesigning an innovative solution to iterate and test document IDs membership over inverted lists with the use of probabilistic data structures.

4 Conclusions

Top-k document retrieval has been gaining increasing attention in recent years due to the spread of complex ranking techniques that would result impracticable on the entire collection of documents. This aspect brings in front of the researchers many opportunities for improving the efficiency of this essential phase. In this paper, we illustrate our initial results and propose different approaches that may concur to bring top-k document retrieval several improvements.

Acknowledgements. This research was supported by NSF Grant IIS-1718680 and a grant from Amazon.

References

[1] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Softw. Pract. Exper.*, 40:131–147, 2010.
[2] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, pages 426–434, 2003.
[3] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, pages 1–1, 2009.
[4] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proc. SIGIR*, pages 993–1002, 2011.
[5] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw. Pract. Exper.*, 45(1):1–29, 2015.
[6] Antonio Mallia and Elia Porciani. Faster blockmax WAND with longer skipping. In *Proc. ECIR*, pages 771–778, 2019.
[7] Antonio Mallia, Giuseppe Ottaviano, Elia Porciani, Nicola Tonellotto, and Rossano Venturini. Faster BlockMax WAND with Variable-sized blocks. In *Proc. SIGIR*, pages 625–634, 2017.
[8] Antonio Mallia, Michal Siedlaczek, Joel Mackenzie, and Torsten Suel. PISA: performant indexes and search for academia. In *Proc. OSIRRC@SIGIR*, pages 50–56, 2019.
[9] Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano indexes. In *Proc. SIGIR*, pages 273–282, 2014.
[10] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. Vectorized VByte decoding. *CoRR*, abs/1503.07387, 2015.
[11] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.
[12] Larry H Thiel and HS Heaps. Program design for retrospective searches on large data bases. *Information Storage and Retrieval*, 8:1–20, 1972.
[13] Howard R. Turtle and James Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, 1995.
[14] Sebastiano Vigna. Quasi-succinct indices. In *Proc. WSDM*, pages 83–92, 2013.
[15] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.
[16] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, page 59, 2006.