An Experimental Study of Index Compression and DAAT Query Processing Methods

Antonio Mallia, Michał Siedlaczek, and Torsten Suel

Computer Science and Engineering, New York University, New York, US
{antonio.mallia,michal.siedlaczek,torsten.suel}@nyu.edu

Abstract. In the last two decades, the IR community has seen numerous advances in top-k query processing and inverted index compression techniques. While newly proposed methods are typically compared against several baselines, these evaluations are often very limited, and we feel that there is no clear overall picture on the best choices of algorithms and compression methods. In this paper, we attempt to address this issue by evaluating a number of state-of-the-art index compression methods and safe disjunctive DAAT query processing algorithms. Our goal is to understand how much index compression performance impacts overall query processing speed, how the choice of query processing algorithm depends on the compression method used, and how performance is impacted by document reordering techniques and the number of results returned, keeping in mind that current search engines typically use sets of hundreds or thousands of candidates for further reranking.

Keywords: Compression · Query processing · Inverted indexes.

1 Introduction

Over the past few decades, the IR community have been making a continuous effort to improve the efficiency of search in large collections of documents. Advances have been made in virtually all aspects of text retrieval, including index compression and top-k query processing. Although a multitude of authors have reported experimental results, comparing them across different publications poses a challenge due to varying data sets, parameters, evaluation metrics, and experimental setups. We aim to address this issue by providing an extensive experimental comparison across many index compression techniques and several query processing algorithms. Our comparison includes many recent methods, and thus provides a useful snapshot of the current state of the art in this area.

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*. Postings are typically sorted in order of increasing document IDs, although there are other index organizations. We assume document-sorted posting lists throughout this paper.

The first problem we encounter is efficient index representation. In particular, compression of posting lists is of utmost importance, since they account for much of the data size and access costs. In practice, the problem we must solve is efficient encoding of non-negative integers, such as document IDs or their gaps, frequencies, positions, or quantized scores. Some encoding schemes, such as Golomb [23] or Binary Interpolative [34], can be very space-efficient but slow to decode. Other methods achieve very fast decoding while sacrificing compression ratio. In recent years, a significant boost in encoding efficiency has been achieved due to application of SIMD (Single Instruction, Multiple Data) instructions available on many modern CPUs [26, 27, 36, 44].

Likewise, the choice of a retrieval algorithm is crucial to query efficiency. Due to large sizes of most search collections, retrieving all potential matches is infeasible and undesirable. In practice, only the top k highest ranked documents are returned. Ranking methods can be grouped into fast and simple term-wise scoring methods [39, 51], and more complex rankers such as the Sequential Dependence Model (SDM) [31] or learning to rank methods [28, 49]. For term-wise techniques, the score of a document with respect to a query is simply the sum of the partial scores, also called impact scores, of the document with respect to each term. Complex rankers give up this term independence assumption. They are more accurate, but also much more expensive, as they require the evaluation of fairly complex ranking functions on up to hundreds of features that need to be fetched or generated from index and document data. Thus, it would be infeasible to evaluate such complex rankers on large numbers of documents.

To combine the speed of term-wise scoring with the accuracy of the complex rankers, a cascade ranking architecture is commonly deployed: First, a fast ranker is used to obtain $k_c > k$ candidate results for a query; then the k_c candidates are reranked by a slower complex ranker to retrieve the final top k documents. In this paper, we address the first problem, also known as candidate generation, as reranking can be considered a separate, largely independent, problem. In particular, we focus on the performance of different index compression and query processing algorithms for candidate generation. We limit ourselves to safe Document-At-A-Time (DAAT) algorithms for disjunctive top-k queries.

Following the RIGOR *Generalizability* property [3], we focus on assessing how well technology performs in new contexts. There are four dimensions to our comparison study: index compression method, query processing algorithm, document ordering, and the number k of retrieved candidates. Published work proposing new compression methods or query processing algorithms typically only looks at a small slice of possible configurations, say, a new query processing algorithm compared against others using only one or two compression methods and document orderings, and only on a very limited range of k.

Catena et al. [8] showed the impact of compressing different types of posting information on the space and time efficiency of a search engine. Although they investigate several compression methods, their focus is mostly on different variations of the FOR family. They also limit their query processing comparison to exhaustive DAAT retrieval strategy, while we consider dynamic pruning tech-

niques. On the other hand, they study several aspects not considered here, such as compression of different types of index data. Thus, we feel that our study answers different research questions by exploring many combinations of techniques that have never been reported. It can serve as a guide for choosing the best combinations of techniques in a given setup.

Contributions. We make the following major contributions:

1. We provide experimental results for an extensive range of configurations. We include almost all of the state-of-the-art compression techniques, the most commonly used DAAT query processing approaches, and several document reorderings over a wide range of k. To our knowledge, this is the most extensive recent experimental study of this space of design choices.
2. We combine already established open-source libraries and our own implementations to create a code base that provides means to reproduce the results, and that can also serve as a starting point for future research. We release this code for free and open use by the research community.

2 Outline of Our Methods

We now describe the various methods and settings we explore, organized according to the four dimensions of our study: compression method, query processing algorithm, document ordering, and number of candidates k. We decided not to include impact score quantization, another possible dimension, in this paper. Score quantization raises additional issues and trade-offs that we plan to study in a future extension of this work.

2.1 Index Compression Methods

We include in our study a total of 11 index compression methods that we consider to be a good representation of the current state of the art. For each method, we integrated what we believe to be the fastest available open-source implementation. We now briefly outline these methods.

Variable Byte Encoding. These methods encode each integer as a sequence of bytes. The simplest one is Varint (also known as Varbyte or VByte), which uses 7 bits of each byte to encode the number (or a part of it), and 1 remaining bit to state if the number continues in the next byte. Although compression of Varint is worse than that of older bit-aligned algorithms such as Elias [21], Rice [38], or Golomb [23] coding, it is much faster in memory-resident scenarios [40].

Varint basically stores a unary code for the size (number of bytes used) of each integer, distributed over several bytes. To improve decoding speed, Dean [14] proposed Group Varint, which groups the unary code bits together. One byte is used to store 4 2-bit numbers defining the lengths (in bytes) of the next 4 integers. The following bytes encode these integers.

Recently, several SIMD-based implementations of variable-byte encodings have been shown to be extremely efficient [14, 36, 44]. Stepanov et al. [44] analyzed a family of SIMD-based algorithms, including a SIMD version of Group

Varint, and found the fastest to be *VarintG8IU*: 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.

Stream VByte [27] combines the benefits of VarintG8IU and Group Varint. Like Group Varint, it stores four integers per block with a 1-byte descriptor. Thus, blocks have variable lengths, which for Group Varint means that the locations of these descriptors cannot be easily predicted by the CPU. Stream VByte avoids this issue by storing all descriptors sequentially in a different location.

PForDelta. PForDelta [54] 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 $[0, 2^k - 1]$. Several methods have been proposed for encoding the exceptions and their locations. One variant, *OptPForDelta* [50], selects b and k to optimize for space or decoding cost, with most implementations focusing on space. A fast SIMD implementation was proposed in [26].

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 [20, 22] 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 $\mathsf{NextGEQ}(x)$ 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* [35] 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. We use this version, which appears to outperform Elias-Fano in most situations.

Binary Interpolative. Similar to Elias-Fano, Binary Interpolative Coding (BIC) [34] directly encodes a monotonically increasing sequence rather than a sequence of gaps. At each step of this recursive algorithm, the middle element m is encoded by a number $m - l - p$, where l is the lowest value and p is the position of m in the currently encoded sequence. Then we recursively encode the values to

the left and right of m. BIC encodings are very space-efficient, particularly on clustered data; however, decoding is relatively slow.

Word-Aligned Methods. Several algorithms including *Simple-9* [1], *Simple-16* [52], and *Simple-8b* [2] try to pack as many numbers as possible into one machine word to achieve fast decoding. For instance, Simple-9 divides each 32-bit word into a 4-bit selector and a 28-bit payload. The selector stores one of 9 possible values, indicating how the payload is partitioned into equal-sized bit fields (e.g., 7 4-bit values, or 9 3-bit values). Some of the partitionings leave up to three of the 28 payload bits unused. Later enhancements in [2, 52] optimize the usage of these wasted bits or increase the word size to 64 bits.

Lemire and Boytsov [26] proposed a bit-packing method that uses SIMD instructions. The algorithm, called *SIMD-BP128*, packs 128 consecutive integers into as few 128-bit words as possible. The 16-byte selectors are stored in groups of 16 to fully utilize 128-bit SIMD reads and writes.

Very recently, Trotman [46] proposed *QMX* encoding (for Quantities, Multipliers, and eXtractor), later extended by Trotman and Lin [47]. It combines the Simple family and SIMD-BP128 by packing as many integers as possible into one or two 128-bit words. Furthermore, the descriptors are run-length encoded, allowing one selector to describe up to 16 consecutive numbers.

Asymmetric Numeral Systems. Asymmetric Numeral Systems (ANS) [19] are a recent advance in entropy coding that combines the good compression rate of arithmetic coding with a speed comparable to Huffman coding. ANS represents a sequence of symbols as a positive integer x, and depends on the frequencies of symbols from a given alphabet Σ. Each string over Σ is assigned a state, which is an integer value, with 0 for the empty string. The state of a string wa is computed recursively using the state of w and the frequency of symbol a. A more detailed description of ANS is beyond the scope of this paper, and we refer to [19, 33]. Very recently, ANS was successfully used, in combination with integer encodings such as VByte and the Simple family, to encode documents and frequencies in inverted indexes [32, 33].

2.2 Query Processing

Next, we describe the top-k disjunctive DAAT query processing algorithms that we study. We limit ourselves to safe methods, guaranteed to return the correct top-k, and select methods that have been extensively studied in recent years.

MaxScore. MaxScore is a family of algorithms first proposed by Turtle and Flood [48], which rely on the maximum impact scores of each term t (denoted as $\max_t$). 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. More precisely,

for a document found in the essential lists, we can compute a score upper bound by adding its current score (from the essential lists and any non-essential lists already accessed) and the $\max_t$ of those lists not yet accessed; if this bound is lower than T, then no further lookups on this document are needed. There are TAAT and DAAT versions of MaxScore; we only consider the DAAT variant. In this case, we attempt to update p whenever T changes.

WAND. Similar to MaxScore, WAND [6] (Weighted or Weak AND) also utilizes the $\max_t$ values. During DAAT traversal, query terms and their associated posting lists are kept sorted by their current document IDs. We denote this list of sorted terms at step s of the algorithm as $q_s = \langle t_1, t_2, \ldots, t_m \rangle$. At each step, we first find the *pivot* term t_p, where p is the lowest value such that $\sum_{i=1}^{p} \max_{t_i} \geq T$. Let d_p be the current document of t_p. If all t_i with $i < p$ point to d_p, then the document is scored and accumulated, and all pointers to d_p are moved to the next document. Otherwise, no document $d < d_p$ can make it into the top-k; thus, all posting lists up to t_p are advanced to the next document $\geq d_p$, and we resort and again find the pivot. This is repeated until all documents are processed.

Block-Max WAND. One shortcoming of WAND is that it uses 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. Block-Max WAND (BMW) [18] addresses this by introducing block-wise maximum impact scores.

First, regular WAND pivot selection is used to determine a pivot candidate. The candidate is then verified by *shallow pointer movements*: The idea is to search for the block in a posting list where the pivot might exist. This operation is fast, as it involves no block decompression. Shallow pointer movement is performed on each $t_{i<p}$, and the block-wise maximum score is computed. If it is greater than T, then t_p is the pivot. In this case, if all $t_{i\leq p}$ point at d_p, we perform the required lookups, following by advancing the pointers by one; otherwise, we pick the $t_{i<p}$ with the largest IDF, and advance its pointer to a document ID $\geq d_p + 1$. If the block-size maximum score is less than T, we must find another candidate. We consider the documents that are at the current block boundaries for $t_{i\leq p}$, and all the current documents for $t_{i>p}$. We select the minimum document ID among them and denote it as d'. Finally, we select the $t_{i<p}$ with the largest IDF, and move its pointer to d'. We repeat the entire process until all terms are processed.

Variable Block-Max WAND. [30] 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.

Block-Max MaxScore. The idea of using per-block maximum impact scores can also be applied to MaxScore, leading to the Block-Max MaxScore [9] (BMM) algorithm. Before performing look-ups to nonessential lists, we further refine our maximum score estimate using maximum impacts of the current blocks in nonessential lists, which might lead to fewer fully evaluated documents.

2.3 Document Ordering

It is well known that a good assignment of IDs to documents can significantly improve index compression. Many query processing algorithms are also sensitive to this assignment, with speed-ups of 2 to 3 over random assignment observed for some orderings and algorithms.

The problem of finding a document ordering that minimizes compressed index size has been extensively studied [4, 5, 15, 17, 41–43]. Shieh et al. [41] propose an approach based on an approximate maximum travelling salesman problem; they build a similarity graph where documents are vertices, and edges indicate common terms. Blandford and Blelloch [5] use a similarity graph with edges weighted with cosine similarity, and run a recursive partitioning to find the ordering. A considerable downside of such algorithms are their time and space complexity. Silvestri [42] shows that a simple URL-based ordering works as well as more complex methods on many data sets. The simplicity and efficiency of this approach makes it a very attractive choice in practice. Recently, Dhulipala et al. [15] proposed the Recursive Graph Bisection algorithm for graph and index compression, and experiments show their algorithm to exhibit the best compression ratio across all tested indices. We consider three orderings in our study, random assignment, URL-based, and Recursive Graph Bisection.

While most work on document ID assignment focuses on index compression, reordering also impacts query efficiency. Yan et al. [50] found that document reordering can significantly speed up conjunctive queries. Subsequent experiments show similar results for several disjunctive top-k algorithms, and in particular for all the algorithms introduced in the previous subsection [18, 45, 24, 25, 16, 30]. Thus, query processing speeds depend on both compression method and document ordering, though the trade-offs as not yet well understood.

2.4 Choice of k

The final dimension of our study is the choice of the number of results 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, though the optimal value of k varies according to several factors [29]. Suggested values include 20 [10], 200 [53], 1000 [37], 5000 [13], and up to tens of thousands [11], suggesting that the optimal k is context-dependent. Also, recent work in [12] indicates that many top-k algorithms slow down significantly as k becomes larger, but not always at the same rate. Given this situation, we decided to perform our evaluation over a large range of values, from $k = 10$ up to 10000.

3 Experimental Evaluation

Testing Environment. All methods are implemented in C++17 and compiled with GCC 7.3.0 using the highest optimization settings. The tests are performed

	Documents	Terms	Postings
Gov2	24,622,347	35,636,425	5,742,630,292
ClueWeb09	50,131,015	92,094,694	15,857,983,641

Table 1. Basic statistics for the test collections

on a machine with an Intel Core i7-4770 quad-core 3.40GHz CPU, with 32GiB RAM, running Linux 4.15.0. The CPU is based on the Haswell micro architecture which supports the AVX2 instruction set. The CPUs L1, L2, and L3 cache sizes are 32KB, 256KB, and 8MB, respectively. 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 any required lists are fully loaded in memory. All timings are measured taking the results with minimum value of five independent runs, and reported in milliseconds. We compute BM25 [39] scores during retrieval.

Data Sets. We performed experiments on two standard datasets: Gov2 and ClueWeb09 [7], summarized in Table 1. For each document in the collection the body text was extracted using Apache Tika, the words lowercased and stemmed using the Porter2 stemmer; no stopwords were removed.

Document Ordering. We experimented with three document orderings: Random, URL, and BP. The first, with IDs randomly assigned to documents, serves as the baseline. URL assigns IDs in lexicographic order of URLs [42]. BP is based on the Recursive Graph Bisection algorithm [15].

Implementation Details. Our codebase is a fork of the ds2i[1] library, extended with many additional encoding, query processing, and reordering implementations used in this study. The source code is available at `https://github.com/pisa-engine/pisa` for readers interested in further implementation details or in replicating the experiments. We integrated what we believe are the currently best open-source implementations of the various compression algorithms. We use the FastPFor[2] library for implementation of VarintGB, VarintG8IU, OptPFD, Simple16, Simple8b, SIMD-BP128, and StreamVByte. PEF and Interpolative are based on the code of the ds2i library. The QMX implementation comes from JASSv2[3]. We used the reference implementation of ANS[4] with 2d max:med contexts mechanism. All block-wise encodings use blocks of 128 postings per block.

We implemented the original Recursive Graph Bisection algorithm by Dhulipala et al. [15] and validated the results obtained against those reported in their paper. Our implementations of BMW and BMM store maximum impact scores for blocks of size 128, while VBMW uses blocks of average length 40. Both these values were also used in previous work.

Queries. To evaluate query processing speed, we use TREC 2005 and TREC 2006 Terabyte Track Efficiency Task, drawing only queries whose terms are all in the collection dictionary. This leaves us with 90% and 96% of the total TREC

[1] `https://github.com/ot/ds2i`
[2] `https://github.com/lemire/FastPFor`
[3] `https://github.com/andrewtrotman/JASSv2`
[4] `https://github.com/mpetri/partitioned_ef_ans`

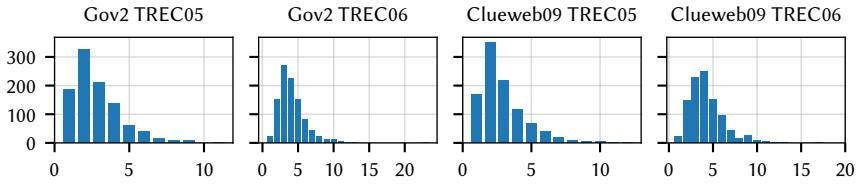

Fig. 1. Query length distributions.

Table 2. Overall space in GB, and average bits per document ID and frequency.

	Random						URL						BP					
	Gov2			ClueWeb09			Gov2			ClueWeb09			Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
Packed+ANS2	7.36	7.71	2.54	19.40	7.65	2.14	4.17	3.96	1.85	14.47	5.36	1.94	3.57	3.25	1.72	13.21	4.80	1.87
Interpolative	7.32	7.58	2.62	19.12	7.52	2.12	4.26	3.80	2.14	13.90	5.15	1.87	3.71	3.11	2.06	12.81	4.65	1.81
PEF	7.65	7.60	3.05	19.68	7.53	2.39	4.65	4.11	2.37	15.95	5.85	2.20	3.97	3.30	2.23	14.66	5.29	2.11
OptPFD	8.09	8.13	3.14	21.47	8.07	2.76	4.92	4.48	2.38	17.04	6.18	2.41	4.28	3.74	2.23	15.57	5.56	2.30
Simple16	9.53	9.43	3.85	25.30	9.40	3.35	5.96	5.34	2.90	19.36	6.92	2.79	5.28	4.62	2.73	17.82	6.34	2.65
Simple8b	9.96	9.24	4.63	26.41	9.18	4.14	6.32	5.53	3.27	21.46	7.36	3.47	5.60	4.77	3.03	20.09	6.87	3.27
QMX	10.21	9.16	5.07	27.10	9.14	4.53	6.71	5.98	3.36	23.27	7.98	3.75	5.92	5.19	3.06	21.64	7.43	3.49
SIMD-BP128	10.35	8.82	5.60	27.05	8.76	4.89	7.00	6.35	3.41	25.08	8.68	3.97	6.03	5.42	2.98	23.02	8.00	3.61
Varint-G8IU	14.51	11.38	8.83	40.51	11.60	8.84	13.75	10.35	8.81	38.82	10.75	8.83	13.57	10.09	8.81	38.35	10.51	8.83
VarintGB	15.64	12.01	9.77	43.58	12.18	9.81	15.02	11.15	9.77	42.10	11.43	9.80	14.87	10.94	9.77	41.77	11.27	9.80
StreamVByte	16.03	12.30	10.04	44.55	12.44	10.03	15.37	11.37	10.04	42.97	11.65	10.03	15.25	11.21	10.04	42.66	11.49	10.03

2005 and TREC 2006 queries for Gov2, and 96% and 98% of the total TREC 2005 and TREC 2006 queries for ClueWeb09. From each sets of queries, we randomly selected 1 000 queries for each query length from 2 to 6+. Figure 1 shows the query length distributions. For Figures 2 and 3, for each data point we sample 500 queries from each query set. We call this set Trec05-06.

4 Results and Discussion

Compressed Index Size. All compression results are summarized in Table 2, sorted by increasing space for the Gov2 collection under the URL ordering. We find that this order is mostly preserved across all tested scenarios. The exceptions are Packed+ANS2 and Interpolative, which compete for the top spot. However, relative differences change quite significantly. For instance, variable byte methods benefit little from URL or BP reordering. This is expected, since virtually all gain comes from decreased document gaps, and no improvement is seen for frequency encodings. On the other hand, packing methods are highly sensitive to ordering, and achieve significantly better compression with URL or BP. For instance, when using Packed+ANS2, the sizes of Gov2 and ClueWeb09 with URL ordering decrease by 43% and 27%, respectively, compared to Random. Further improvements are seen for BP.

Query Efficiency. We first executed the five early termination algorithms in all configurations with $k = 10$. The results are shown in Table 3. As expected, for a fixed ordering there is a clear trade-off between index size and query speed. Variable byte encoding is extremely fast, but also gains the least from reordering. Interestingly, SIMD-BP128 basically matches the performance of Varint-G8IU

Table 3. Query times (in ms) of different query processing strategies on indexes encoded using different encoding techniques.

			Gov2					ClueWeb09				
			MaxScore	WAND	BMM	BMW	VBMW	MaxScore	WAND	BMM	BMW	VBMW
RANDOM	TREC05	Packed+ANS2	13.91	41.60	14.36	22.23	13.73	36.10	117.16	36.34	64.57	46.98
		Interpolative	16.53	55.80	16.93	30.71	18.65	43.32	160.25	43.12	87.72	63.13
		PEF	11.77	16.65	12.55	10.14	6.23	29.62	47.98	31.82	33.06	23.99
		OptPFD	7.43	15.99	8.59	10.91	7.22	19.74	47.28	22.50	34.63	26.61
		Simple16	8.33	19.39	9.38	12.83	8.15	22.28	57.02	24.54	39.10	29.72
		Simple8b	7.05	16.08	8.51	10.98	7.29	18.67	47.54	22.36	34.15	26.17
		QMX	7.50	15.77	8.72	11.50	7.28	20.11	47.77	22.73	35.25	26.47
		SIMD-BP128	6.27	10.68	7.62	8.81	5.75	16.76	32.17	19.79	28.48	21.63
		Varint-G8IU	5.86	10.95	7.43	8.69	5.75	15.46	33.03	19.56	27.74	21.71
		VarintGB	5.96	11.33	7.52	9.00	5.90	15.75	34.26	19.79	28.61	21.86
		StreamVByte	6.44	11.36	7.74	9.23	6.09	17.15	34.32	20.48	29.53	22.70
	TREC06	Packed+ANS2	25.81	95.41	26.14	60.93	41.18	55.19	196.34	68.78	137.39	96.30
		Interpolative	30.84	127.23	30.90	83.35	55.84	67.06	270.54	66.90	185.83	129.58
		PEF	19.08	32.49	20.51	30.56	20.38	41.13	67.24	44.89	71.40	49.53
		OptPFD	12.93	34.24	15.21	32.73	23.04	28.84	73.09	33.78	75.62	55.61
		Simple16	14.72	42.59	16.74	37.72	25.85	32.76	91.10	37.10	84.75	61.31
		Simple8b	12.48	34.86	15.32	32.31	22.83	27.73	74.67	34.20	73.75	55.21
		QMX	13.06	34.17	15.39	33.68	23.10	29.09	73.99	34.49	78.40	55.94
		SIMD-BP128	10.56	21.80	13.20	27.04	18.88	23.95	48.20	29.73	63.25	45.76
		Varint-G8IU	9.96	22.51	13.14	26.32	18.66	22.44	49.62	29.44	61.44	45.46
		VarintGB	10.19	23.36	13.27	26.96	18.92	23.02	51.33	29.78	63.00	45.96
		StreamVByte	10.83	23.40	13.37	27.96	19.69	24.70	51.64	30.32	65.60	48.22
URL	TREC05	Packed+ANS2	9.00	19.91	9.53	9.06	5.14	29.54	79.06	29.42	33.82	20.44
		Interpolative	9.89	23.44	10.31	10.63	5.73	31.14	96.21	31.13	38.75	22.55
		PEF	8.62	9.45	8.90	3.78	2.21	25.76	31.62	26.57	14.19	8.93
		OptPFD	5.08	8.29	5.99	4.22	2.53	16.54	31.15	18.78	16.33	10.45
		Simple16	5.60	9.27	6.31	4.62	2.70	17.32	35.21	19.43	17.26	11.01
		Simple8b	4.60	7.87	5.64	4.01	2.46	14.97	29.64	17.79	15.41	9.94
		QMX	5.16	8.07	6.05	4.23	2.40	16.15	30.42	18.40	15.63	9.82
		SIMD-BP128	4.27	5.68	5.29	3.12	1.91	13.50	19.72	16.01	11.97	7.76
		Varint-G8IU	3.96	5.72	5.02	3.07	1.96	12.47	20.00	15.76	11.88	7.90
		VarintGB	4.02	5.89	5.10	3.17	1.94	12.85	20.87	16.09	12.23	8.00
		StreamVByte	4.48	5.96	5.31	3.22	1.99	13.71	20.89	16.31	12.52	8.11
	TREC06	Packed+ANS2	14.83	39.05	15.46	20.64	11.14	46.20	128.12	45.27	73.88	40.96
		Interpolative	16.23	45.66	16.65	24.26	12.51	49.40	158.23	48.14	84.89	45.85
		PEF	14.32	15.52	14.41	9.71	5.38	39.41	45.35	39.19	33.11	18.43
		OptPFD	8.22	14.12	9.85	10.49	5.95	24.98	47.76	29.42	37.37	21.61
		Simple16	8.81	16.67	10.68	11.18	6.36	26.34	55.21	29.64	39.73	22.42
		Simple8b	7.48	13.78	9.42	10.09	5.80	22.70	45.66	27.51	35.75	20.89
		QMX	8.40	13.88	9.90	10.67	5.69	24.28	45.80	28.03	36.88	20.56
		SIMD-BP128	6.80	8.98	8.64	8.27	4.63	19.96	29.18	24.49	28.56	16.50
		Varint-G8IU	6.31	9.12	8.45	8.02	4.70	18.67	29.81	24.37	28.10	16.62
		VarintGB	6.48	9.57	8.54	8.20	4.75	19.14	31.05	24.54	28.84	16.94
		StreamVByte	6.95	9.51	8.74	8.50	4.86	20.29	31.22	24.90	29.79	17.42
BP	TREC05	Packed+ANS2	8.53	14.91	9.15	6.86	4.85	27.52	59.41	27.98	25.23	17.53
		Interpolative	9.67	18.87	10.18	8.52	5.67	30.14	77.01	30.43	29.96	19.92
		PEF	8.69	8.28	8.81	2.99	2.25	24.92	26.88	25.86	10.98	7.80
		OptPFD	4.97	6.97	5.94	3.43	2.61	15.57	25.47	17.92	12.57	9.15
		Simple16	5.38	7.83	6.23	3.69	2.77	16.57	29.33	18.72	13.44	9.71
		Simple8b	4.45	6.69	5.57	3.24	2.50	14.08	24.25	17.02	11.93	8.74
		QMX	5.01	7.04	6.03	3.45	2.51	15.19	25.38	17.79	11.97	8.56
		SIMD-BP128	4.19	5.21	5.24	2.55	2.00	12.75	17.19	15.43	9.34	6.97
		Varint-G8IU	3.86	5.22	5.02	2.54	2.03	11.87	17.71	14.95	9.24	6.98
		VarintGB	3.93	5.34	5.10	2.63	2.05	12.08	18.37	15.32	9.55	7.06
		StreamVByte	4.26	5.45	5.29	2.65	2.10	12.97	18.12	15.74	9.75	7.20
	TREC06	Packed+ANS2	14.61	26.66	15.67	15.64	10.36	41.24	85.86	41.53	50.79	0.00
		Interpolative	16.50	34.54	17.49	19.30	12.29	45.86	113.44	44.90	60.75	38.32
		PEF	14.88	12.44	15.18	7.65	5.44	37.60	34.73	36.46	24.24	16.21
		OptPFD	8.31	11.15	10.26	8.61	6.03	22.62	35.20	26.18	26.95	18.40
		Simple16	8.93	12.87	10.74	9.23	6.38	24.03	40.47	27.37	28.93	19.62
		Simple8b	7.52	10.77	9.78	8.20	5.82	20.65	33.79	25.28	25.94	18.11
		QMX	8.37	11.28	10.32	8.52	5.78	21.82	34.30	25.94	26.66	17.61
		SIMD-BP128	6.97	7.73	9.06	6.80	4.74	18.09	23.00	23.41	21.21	14.47
		Varint-G8IU	6.39	7.83	8.89	6.68	4.80	16.96	23.56	22.36	20.75	14.64
		VarintGB	6.56	8.15	9.18	6.83	4.79	17.44	24.44	22.54	21.39	14.67
		StreamVByte	7.04	8.25	9.15	7.07	4.96	18.53	24.59	23.24	21.97	15.22

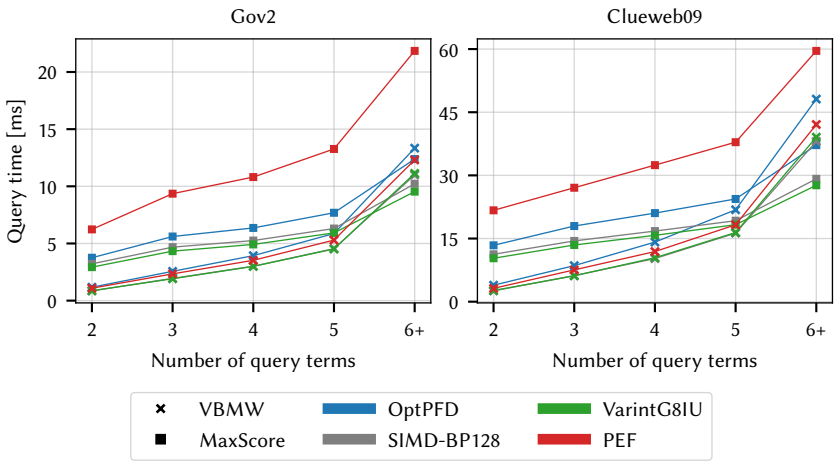

Fig. 2. Query times for different query lengths under URL ordering.

and VarintGB while achieving significantly better compression. Other packing methods—QMX, Simple8b, and Simple16—along with OptPFD, fall slightly yet noticeably behind. PEF is on average less efficient. However, it almost matches the performance of the top four encodings for algorithms utilizing many skips or lookups: BMW and especially VBMW. This is significant, as PEF decreases the index size by more than 50% over the variable byte techniques. Finally, the entropy-based encodings perform the worst across all settings, with Interpolative being by far the slowest. Based on these results, we select a set of four encoding techniques for further analysis: OptPFD, PEF, SIMD-BP128, and Varint-G8IU, each representing a different group of fast algorithms.

Overall, the fastest retrieval algorithm is VBMW. Moreover, it facilitates efficient query processing on a space-efficient PEF-encoded index. Unsurprisingly, it improves upon BMW, which in turn improves upon WAND. These two also fall short of MaxScore, which is the fastest when testing Random ordering but does not benefit as much from reordering. We find BMM to provide no improvement over MaxScore. Given these facts, and due to space constraints, we focus further experiments on the MaxScore and VBMW algorithms.

We also notice a significant difference between different document orderings. Queries on randomly ordered indexes can be almost 3 times slower than on URL ordered ones. Quite interesting, although limited, is the improvement obtained by BP over URL ordering. Even though there is some variability of the gain for Gov2, which depends on the algorithm and encoding used, it is quite evident and constant for ClueWeb09. To the best of our knowledge, this result for BP has not been discussed in the literature, and it would be interesting to further investigate the reasons for the improvement, with the aim of further improvements.

Query Length. Figure 2 shows average query times for different query term counts using Trec05-06 queries, under the URL ordering. Interestingly, MaxScore performs better for long queries (except when PEF encoding is used). For

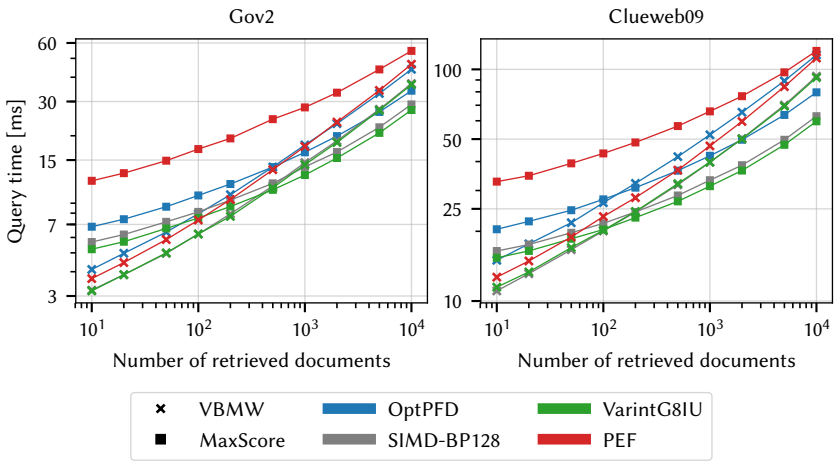

Fig. 3. Query times for different k under URL ordering.

ClueWeb09 the difference is significant: about 10 ms. This could justify a hybrid retrieval method that switches between algorithms based on a query length.

Varying k. The results for a range of values of k (using URL ordering) are shown in Figure 3 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. Both algorithms are roughly equally fast for $k = 100$ using Varint-G8IU or SIMD-BP128, while for larger k MaxScore becomes faster. We note that at $k = 10,000$ even the performance of PEF, which previously performed well only for VBMW, is similar for both algorithms. This suggests that MaxScore might be better suited for some cases of candidate generation.

5 Conclusions

In this paper, we performed an experimental evaluation of a whole range of previously proposed schemes for index compression, query processing, index reordering, and the number of results k. Our experiments reproduce many previous results while filling some remaining gaps and painting a more detailed picture. We confirm known correlation between index size and query speed, and provide comprehensive data that may help to find the right trade-off for a specific application. In particular, we find SIMD-BP128 to perform on a par with the variable byte techniques while providing a significantly higher compression ratio. Moreover, PEF is both space and time-efficient when using the VBMW algorithm. VBMW is the fastest query processing method for small k while MaxScore surpasses it for $k > 100$. Query cost increases significantly with k for DAAT methods, justifying TAAT and SAAT approaches for candidate generation such as [12]. The good performance of MaxScore on long queries motivates hybrid methods that select algorithms based on query length.

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

References

1. Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Information Retrieval **8**(1), 151–166 (2005)
2. Anh, V.N., Moffat, A.: Index compression using 64-bit words. Software: Practice and Experience **40**(2), 131–147 (2010)
3. Arguello, J., Diaz, F., Lin, J., Trotman, A.: SIGIR 2015 workshop on reproducibility, inexplicability, and generalizability of results. In: Proc. of the 38th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval. pp. 1147–1148 (2015)
4. Blanco, R., Barreiro, Á.: Document identifier reassignment through dimensionality reduction. In: European Conference on Information Retrieval. pp. 375–387 (2005)
5. Blandford, D., Blelloch, G.: Index compression through document reordering. In: 2002 Data Compression Conference. pp. 342–351 (2002)
6. Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.: Efficient query evaluation using a two-level retrieval process. In: Proc. of the 12th Intl. Conf. on Information and Knowledge Management. pp. 426–434 (2003)
7. Callan, J., Hoy, M., Yoo, C., Zhao, L.: Clueweb09 data set (2009), `http://lemurproject.org/clueweb09/`
8. Catena, M., Macdonald, C., Ounis, I.: On inverted index compression for search engine efficiency. In: Proc. of the 36th European Conf. on IR Research (2014)
9. Chakrabarti, K., Chaudhuri, S., Ganti, V.: Interval-based pruning for top-k processing over compressed lists. In: Proc. of the 2011 IEEE 27th Intl. Conf. on Data Engineering. pp. 709–720 (2011)
10. Chapelle, O., Chang, Y.: Yahoo! learning to rank challenge overview. In: Proceedings of the Learning to Rank Challenge. pp. 1–24 (2011)
11. Chapelle, O., Chang, Y., Liu, T.Y.: Future directions in learning to rank. In: Proceedings of the Learning to Rank Challenge. pp. 91–100 (2011)
12. Crane, M., Culpepper, J.S., Lin, J., Mackenzie, J., Trotman, A.: A comparison of document-at-a-time and score-at-a-time query evaluation. In: Proc. of the 10th ACM Intl. Conf. on Web Search and Data Mining. pp. 201–210 (2017)
13. Craswell, N., Fetterly, D., Najork, M., Robertson, S., Yilmaz, E.: Microsoft research at trec 2009 web and relevance feedback tracks. Tech. rep., Microsoft Research (2009)
14. Dean, J.: Challenges in building large-scale information retrieval systems: invited talk. In: Proc. of the 2nd ACM Intl. Conf. on Web Search and Data Mining. pp. 1–1 (2009)
15. Dhulipala, L., Kabiljo, I., Karrer, B., Ottaviano, G., Pupyrev, S., Shalita, A.: Compressing graphs and indexes with recursive graph bisection. In: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1535–1544 (2016)
16. Dimopoulos, C., Nepomnyachiy, S., Suel, T.: Optimizing top-k document retrieval strategies for block-max indexes. In: Proc. of the 6th ACM Intl. Conf. on Web Search and Data Mining. pp. 113–122 (2013)
17. Ding, S., Attenberg, J., Suel, T.: Scalable techniques for document identifier assignment in inverted indexes. In: Proc. of the 19th Intl. Conf. on World Wide Web. pp. 311–320 (2010)
18. Ding, S., Suel, T.: 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. pp. 993–1002 (2011)

19. Duda, J.: Asymmetric numeral systems as close to capacity low state entropy coders. CoRR **abs/1311.2540** (2013)
20. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM **21**(2), 246–260 (1974)
21. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory **21**(2), 194–203 (1975)
22. Fano, R.M.: On the number of bits required to implement an associative memory. Massachusetts Institute of Technology, Project MAC (1971)
23. Golomb, S.W.: Run-length encodings (corresp.). IEEE Trans. Information Theory **12**(3), 399–401 (1966)
24. Hawking, D., Jones, T.: Reordering an index to speed query processing without loss of effectiveness. In: Proc. of the 7th Australasian Document Computing Symposium. pp. 17–24 (2012)
25. Kane, A., Tompa, F.W.: 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. pp. 877–880 (2018)
26. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Softw. Pract. Exper. **45**(1), 1–29 (2015)
27. Lemire, D., Kurz, N., Rupp, C.: Stream vbyte: Faster byte-oriented integer compression. Information Processing Letters **130**, 1–6 (2018)
28. Liu, T.Y.: Learning to rank for information retrieval. Foundations and Trends in Information Retrieval **3**(3), 225–331 (2009)
29. Macdonald, C., Santos, R.L., Ounis, I.: The whens and hows of learning to rank for web search. Inf. Retr. **16**(5), 584–628 (2013)
30. Mallia, A., Ottaviano, G., Porciani, E., Tonellotto, N., Venturini, R.: Faster blockmax WAND with variable-sized blocks. In: Proc. of the 40th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval. pp. 625–634 (2017)
31. Metzler, D., Croft, W.B.: A Markov random field model for term dependencies. In: Proc. of the 28th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval. pp. 472–479 (2005)
32. Moffat, A., Petri, M.: ANS-based index compression. In: Proc. of the 2017 Intl. Conf. on Information and Knowledge Management. pp. 677–686 (2017)
33. Moffat, A., Petri, M.: Index compression using byte-aligned ANS coding and two-dimensional contexts. In: Proc. of the 11th ACM Intl. Conf. on Web Search and Data Mining. pp. 405–413 (2018)
34. Moffat, A., Stuiver, L.: Binary interpolative coding for effective index compression. Inf. Retr. **3**(1), 25–47 (2000)
35. Ottaviano, G., Venturini, R.: Partitioned elias-fano indexes. In: Proc. of the 37th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval. pp. 273–282 (2014)
36. Plaisance, J., Kurz, N., Lemire, D.: Vectorized VByte decoding. CoRR **abs/1503.07387** (2015)
37. Qin, T., Liu, T.Y., Xu, J., Li, H.: LETOR: A benchmark collection for research on learning to rank for information retrieval. Inf. Retr. **13**(4), 346–374 (2010)
38. Rice, R., Plaunt, J.: Adaptive variable-length coding for efficient compression of spacecraft television data. IEEE Transactions on Communication Technology **19**(6), 889–897 (1971)
39. Robertson, S.E., Jones, K.S.: Relevance weighting of search terms. Journal of the American Society for Information science **27**(3), 129–146 (1976)

40. Scholer, F., Williams, H.E., Yiannis, J., Zobel, J.: Compression of inverted indexes for fast query evaluation. In: Proc. of the 25th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval. pp. 222–229 (2002)
41. Shieh, W.Y., Chen, T.F., Shann, J.J.J., Chung, C.P.: Inverted file compression through document identifier reassignment. Information Processing & Management **39**(1), 117–131 (2003)
42. Silvestri, F.: Sorting out the document identifier assignment problem. In: Proc. of the 29th European Conf. on IR Research. pp. 101–112 (2007)
43. Silvestri, F., Orlando, S., Perego, R.: Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In: Proc. of the 27th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval. pp. 305–312 (2004)
44. Stepanov, A.A., Gangolli, A.R., Rose, D.E., Ernst, R.J., Oberoi, P.S.: SIMD-based decoding of posting lists. In: Proc. of the 20th Intl. Conf. on Information and Knowledge Management. pp. 317–326 (2011)
45. Tonellotto, N., Macdonald, C., Ounis, I.: Effect of different docid orderings on dynamic pruning retrieval strategies. In: Proc. of the 34th Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval. pp. 1179–1180 (2011)
46. Trotman, A.: Compression, SIMD, and postings lists. In: Proc. of the 2014 Australasian Document Computing Symposium. pp. 50:50–50:57 (2014)
47. Trotman, A., Lin, J.: In vacuo and in situ evaluation of SIMD codecs. In: Proc. of the 21st Australasian Document Computing Symposium. pp. 1–8 (2016)
48. Turtle, H., Flood, J.: Query evaluation: Strategies and optimizations. Information Processing & Management **31**(6), 831–850 (1995)
49. Wang, L., Lin, J., Metzler, D.: Learning to efficiently rank. In: Proc. of the 33rd Ann. Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval. pp. 138–145 (2010)
50. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proc. of the 18th Intl. Conf. on World Wide Web. pp. 401–410 (2009)
51. Zhai, C., Lafferty, J.: A study of smoothing methods for language models applied to information retrieval. ACM Transactions on Information Systems **22**(2), 179–214 (2004)
52. Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: Proc. of the 17th Intl. Conf. on World Wide Web. pp. 387–396 (2008)
53. Zhang, M., Kuang, D., Hua, G., Liu, Y., Ma, S.: Is learning to rank effective for web search? In: SIGIR 2009 workshop: Learning to Rank for Information Retrieval. pp. 641–647 (2009)
54. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: Proc. of the 22nd Intl. Conf. on Data Engineering (2006)