Inf Retrieval J (2017) 20:199–220
DOI 10.1007/s10791-016-9291-5

INFORMATION RETRIEVAL EFFICIENCY

The role of index compression in score-at-a-time query evaluation

Jimmy Lin[1] · **Andrew Trotman**[2]

Received: 26 May 2016 / Accepted: 16 December 2016 / Published online: 25 January 2017

Abstract This paper explores the performance of top k document retrieval with score-at-a-time query evaluation on impact-ordered indexes in main memory. To better understand execution efficiency in the context of modern processor architectures, we examine the role of index compression on query evaluation latency. Experiments include compressing postings with variable byte encoding, Simple-8b, variants of the QMX compression scheme, as well as a condition that is less often considered—no compression. Across four web test collections, we find that the highest query evaluation speed is achieved by simply leaving the postings lists uncompressed, although the performance advantage over a state-of-the-art compression scheme is relatively small and the index is considerably larger. We explain this finding in terms of the design of modern processor architectures: Index segments with high impact scores are usually short and inherently benefit from cache locality. Index segments with lower impact scores may be quite long, but modern architectures have sufficient memory bandwidth (coupled with prefetching) to "keep up" with the processor. Our results highlight the importance of "architecture affinity" when designing high-performance search engines.

1 Introduction

Modern superscalar processors are able to dispatch multiple instructions per clock cycle, and at today's clock speeds, this translates potentially into a throughput of a dozen or more instructions per nanosecond. The speed at which processors can execute instructions is no longer the limiting factor in the performance of many applications. Instead, the primary bottlenecks today are memory latencies and processor stalls. Both are related in that the processor must wait for memory references to resolve (e.g., when manipulating pointers)

✉ Andrew Trotman
andrew@cs.otago.ac.nz

[1] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada

[2] Department of Computer Science, University of Otago, Dunedin, New Zealand

before instructions can be fully executed. Another source of stalls is branch mispredicts in pipelined systems, where the wrong execution path is selected and thus instructions need to be "unwound".

Fast query evaluation (specifically, top k retrieval) in main-memory search engines requires designers to effectively manage instruction execution, memory latencies, and memory bandwidth. Index compression techniques provide a case study of how these different architectural issues play out in practice. Compression reduces the amount of data transferred from the memory to the processor, but at the cost of additional instructions necessary for decompression. These extra instructions increase the potential for processor stalls—the classic example being the decoding of the stop bit in variable-byte encoding, where branch mispredicts represent a source of latency (Dean 2009). Advances in integer compression, e.g., the Simple family (Anh and Moffat 2005a, 2010) and PForDelta (Zhang et al. 2008; Yan et al. 2009), have focused on reducing the need for branches via loop unrolling techniques, and recent work based on SIMD instructions (Stepanov et al. 2011; Trotman 2014; Lemire and Boytsov 2015) is very much in the same spirit.

In this paper, we focus on a particular class of algorithms for top k document retrieval: score-at-a-time (SAAT) query evaluation on impact-ordered indexes in main memory. To better understand execution efficiency in the context of modern processor architectures, we examine the role of index compression on query evaluation latency, i.e., how fast we can compute a top k ranking. Experiments include compressing postings with variable byte encoding, Simple-8b, variants of the QMX compression scheme, as well as a condition that is less often considered—no compression. We find that the highest query evaluation speed is achieved by simply leaving the postings lists uncompressed, although the performance advantage over a state-of-the-art compression scheme is relatively small and the increase in index size is considerable.

Beyond documenting this result, we provide an explanation as to *why*. The intuition is as follows: In score-at-a-time query evaluation, index segments with document ids that have the same pre-computed and quantized impact scores are processed in order of decreasing impact score. Index segments with high impact scores are usually short and inherently benefit from cache locality. Furthermore, most modern integer compression techniques operate on relatively large blocks of integers, and thus are not well-suited for compressing these short segments. Index segments with lower impact scores may be quite long, but modern architectures have sufficient memory bandwidth (coupled with prefetching) to keep up with the processor. Thus, in this specific setting, we find that uncompressed postings yield slightly faster query evaluation than compressed postings, albeit at the cost of a substantial increase in index size.

This finding is of interest to information retrieval researchers for a number of reasons. Most previous index compression studies considered only the case where postings reside on disk (Williams and Zobel 1999; Scholer et al. 2002; Anh and Moffat 2005a; Brewer 2005). Main memory has become sufficiently plentiful that in-memory indexes are considered the common (if not the default) setting for search engines today. Although Büttcher and Clarke (2007) considered in-memory indexes, their focus was on random access, which is not the dominant memory access pattern for score-at-a-time query evaluation. Most previous studies also assume document-ordered indexes, e.g., all of the citations above with the exception of Anh and Moffat (2005a). In general, impact-ordered indexes are under-explored in the literature, and we argue that modern block-based compression schemes are not well-suited for such indexes. Uncompressed postings obviously consume considerably more space, so whether trading memory consumption for slightly faster query evaluation is worthwhile depends on the application. However, our work highlights the

need for detailed empirical evaluations to justify fundamental decisions in designing efficient retrieval algorithms and the importance of architecture affinity. Because processor architectures and query evaluation algorithms interact in complex ways, generalizations across different retrieval settings (e.g., document-at-a-time query evaluation vs. score-at-a-time query evaluation, on-disk vs. in-memory indexes) may not hold.

2 Background and related work

2.1 Modern processor architectures

It is useful to begin with an overview of modern processor architectures and the associated challenges of designing low latency applications.

One major challenge is the so-called "memory wall" (Patterson 2004), where increases in processor speeds have far outpaced decreases in memory latency. This means that, relatively, RAM is becoming slower. In the 1980s, memory latencies were on the order of a few clock cycles; today, it could be several hundred clock cycles. To hide latency, computer architects have introduced hierarchical caches, built on the assumption of reference locality—that at any given time, the processor repeatedly accesses only a (relatively) small amount of data. The fraction of memory accesses that can be fulfilled directly from the cache is called the *cache hit rate*, and data not found in cache is said to cause a *cache miss*, incurring much longer access latencies.

Managing cache content is a complex challenge, but there are two main principles that are relevant for a software developer. First, caches are organized into cache lines (typically 64 bytes), which is the smallest unit of transfer between cache levels. That is, when a program accesses a particular memory location, the entire cache line is brought into (L1) cache. This means that subsequent references to nearby memory locations are very fast, i.e., a cache hit. Therefore, it is worthwhile to organize data structures to take advantage of this fact. Second, if a program accesses memory in a predictable sequential pattern (called striding), the processor will prefetch memory blocks and move them into cache, before the program has explicitly requested them (and in certain architectures, it is possible to explicitly control prefetch in software). There is, of course, much more complexity beyond this short description; see Jacob (2009) for an overview.

Another salient property of modern CPUs is pipelining, where instruction execution is split between several stages (modern processors have between one and two dozen stages). At each clock cycle, all instructions "in flight" advance one stage in the pipeline; new instructions enter the pipeline and instructions that leave the pipeline are "retired". Pipeline stages allow faster clock rates since there is less to do per stage. Modern *superscalar* CPUs add the ability to dispatch multiple instructions per clock cycle (and out of order) provided that they are independent.

Pipelining suffers from two dangers, known as "hazards" in VLSI design terminology. *Data hazards* occur when one instruction requires the result of another (that is, a data dependency), such as when manipulating pointers. Subsequent instructions cannot proceed until we first compute the memory location and the processor stalls (unless there is another independent instruction that can be executed). *Control hazards* are instruction dependencies introduced by branches (i.e., conditional jumps in assembly). To cope with this, processors use *branch prediction techniques* to predict which code path will be taken.

However, if the guess is wrong, the processor must "undo" the instructions that occurred after the branch point (called "flushing" the pipeline).

2.2 Inverted indexes and query evaluation

Following the standard formulation of ranked retrieval, we assume that the score of a document d with respect to a query q can be computed as an inner-product:

$$S_{d,q} = \sum_{t \in d \cap q} w_{d,t} \times w_{q,t} \quad (1)$$

where $w_{d,t}$ represents the weight of term t in document d and $w_{q,t}$ represents the weight of term t in the query. The goal of top k retrieval is to return the top k documents ordered by S. Typically, w's are a function of term frequency, document frequency, collection frequency, and the like. This formulation is sufficiently general to capture traditional vector-space models (Salton 1971), probabilistic models such as BM25 (Robertson et al. 1995), as well as language modeling (Ponte and Croft 1998) and divergence from randomness approaches (Amati and van Rijsbergen 2002). Note that modern web search engines typically adopt a multi-stage retrieval architecture (Cambazoglu et al. 2010; Wang et al. 2011; Tonellotto et al. 2013; Asadi and Lin 2013) where top k retrieval supplies the initial candidate documents that are then reranked with a multitude of features (static priors, graph-based and editorial features, phrase and term proximity features, etc.), usually in a learning-to-rank context. In this paper, we focus on the initial retrieval stage, specifically on the efficiency of candidate generation.

Nearly all retrieval engines today depend on an inverted index for top k retrieval. Abstractly, an inverted index maps from vocabulary terms to lists of postings, each of which represents a document that contains the term. Each posting comprises a document id and a payload such as the term frequency. Query evaluation involves looking up postings that are associated with query terms and processing them in a particular order (more details below). As is common today, we assume that the entire index and all associated data structures reside in main memory.

The literature describes a few ways in which inverted indexes can be organized: In *document-ordered* indexes, postings lists are sorted by document ids in increasing order. Term frequencies (and whatever information is necessary for computing the term weights) are stored separately. In *frequency-ordered* indexes, document ids are grouped by their term frequencies; within each grouping, document ids are usually sorted in increasing order, but the groupings themselves are usually arranged in decreasing order of term frequency. In *impact-ordered* indexes, the actual score contribution of each term (i.e., the $w_{d,t}$'s) is pre-computed and quantized into what are known as *impact scores*. We refer to a block of document ids that share the same impact score as a postings (or index) segment. Within each segment, document ids are arranged in increasing order, but the segments themselves are arranged by decreasing impact score.[1] The focus of this work is impacted-ordered indexes.

The basic proposition behind index compression is a smaller index size, which means less data to transfer from storage (originally, disk, but nowadays, main memory). However, compressed indexes incur the cost of additional instructions needed to recover the original

[1] As a detail, note that in this treatment query weights are often ignored, which is justified for a few reasons: typical search queries do not have repeated terms (usually the source of query weights); in fact, although BM25 does include weighting for query frequency, in most implementation that weight is ignored.

postings for query evaluation; these instructions increase the risk of control hazards. It has been shown empirically (see references in the introduction) that for on-disk indexes, this tradeoff is worthwhile, and that index compression decreases query evaluation latency. We explore a different point in the design space: score-at-a-time query evaluation using in-memory impact-ordered indexes.

Early compression techniques applied to information retrieval include Elias, Golomb, Rice, and variable byte encoding (Witten et al. 1999). The first three are bit-wise techniques, which have largely fallen out of favor because they are much slower than byte- and word-aligned techniques to decompress (Scholer et al. 2002). Variable byte encoding (vbyte for short) remains popular today due to its simplicity; integers are coded using one to four bytes, using 7 bits of each byte for storing data and a stop bit to tell the decoder when the current integer ends.

Whereas most early work on postings compression focused on achieving the maximum compression ratio, researchers eventually realized that fast query evaluation also depends on efficient decoding (Scholer et al. 2002; Trotman 2003). Word-aligned techniques may not yield indexes that are as compact as bit-aligned techniques, but they are usually much faster to decode. One well-known example is the Simple family of techniques (Anh and Moffat 2005a, 2010). Variants share in the idea of trying to pack as many integers into a machine word as possible, reserving some of the bits as "selectors", which tell the decoder how to interpret data packed in the "payload" (e.g., how many bits each integer occupies). Follow-on extensions include Simple-16 (Zhang et al. 2008), Simple-4b and Simple-8b (Anh and Moffat 2010), and VSEncoding (Silvestri and Venturini 2010).

PForDelta (Zukowski et al. 2006; Zhang et al. 2008; Yan et al. 2009) adds the idea of "overflow bits" that are "patched in" after decompression to handle large outliers that would otherwise degrade the compression effectiveness of word-based techniques. PForDelta was designed to handle blocks of 128 integers at a time and takes advantage of hard-coded decompression functions for each configuration, where the inner loop is unrolled to achieve maximum performance (i.e., avoiding branches completely).

The most recent work on compression takes advantage of SIMD instructions on modern processors, e.g., varint-G8IU (Stepanov et al. 2011), SIMD-BP128 (Lemire and Boytsov 2015), and QMX (Trotman 2014). For example, SSE (Streaming SIMD Extensions) instructions in the x86 architecture support operations on special 128-bit SSE registers, which allow a single instruction to operate concurrently on four 32-bit integers. In many ways, these techniques can be viewed as a natural extension of loop unrolling to decompress integers packed into machine words. At a high level, they strive to reduce processor stalls (by eliminating as many branches as possible) so as to operate as close to the maximum instruction throughput as possible.

Different query evaluation strategies are best suited for different index organizations. Document-at-a-time (DAAT) strategies (Turtle and Flood 1995; Moffat and Zobel 1996; Strohman et al. 2005; Ding and Suel 2011; Fontoura et al. 2011; Rossi et al. 2013; Wu and Fang 2014), which are the most popular today, work well with document-ordered indexes and term-at-a-time (TAAT) strategies (Turtle and Flood 1995; Moffat and Zobel 1996; Persin et al. 1996; Fontoura et al. 2011) work well with frequency-ordered indexes. Similarly, score-at-a-time (SAAT) strategies (Anh et al. 2001; Anh and Moffat 2005b, 2006; Strohman and Croft 2007; Lin and Trotman 2015) typically operate on impact-ordered indexes. DAAT approaches are attractive because the system only needs to keep track of the top k documents, and therefore the memory footprint is small. Although in its most basic form, DAAT query evaluation needs to process all postings, the addition of skip lists (Moffat and Zobel 1996), max_score (Turtle and Flood 1995), block-max (Ding

and Suel 2011; Rossi et al. 2013), document prioritization (Wu and Fang 2014), and other optimizations allow large sections of postings lists to be skipped without affecting ranking correctness. In contrast, TAAT strategies have larger memory requirements, although researchers have proposed many pruning techniques (Moffat and Zobel 1996; Persin et al. 1996) and with modern hardware, memory footprint is usually not an issue. In addition, TAAT strategies better exploit data locality when decoding postings since the system is not "jumping around" different query terms. SAAT strategies require impact-ordered indexes and process postings segments in decreasing order of their impact scores. This work focuses on SAAT query evaluation.

3 Why compression?

We begin by discussing the role of compression for SAAT query evaluation on impact-ordered indexes in main memory and attempt to convey our intuition as to why compression may not necessarily increase performance.

Modern compression techniques are most effective when operating on relatively large blocks of integers, primarily for two reasons: First, loop unrolling and SIMD instructions are efficient only if there is a lot of "work" to do. For example, since the Simple family tries to pack as many integers as it can in a machine word, if there are fewer integers to compress than there is room for, the standard solution is to add "padding", which is basically wasting processor cycles. An even more extreme example is PForDelta (Zhang et al. 2008; Yan et al. 2009), where standard implementations operate on blocks of 128 integers at a time. In many implementations, if there are fewer than 128 integers, vbyte is used instead. Furthermore, in the context of information retrieval, integers represent document ids and are usually sorted; systems compress the differences between consecutive document ids (called *d*-gaps) instead of the document ids themselves. Given a fixed collection size, more term occurrences yield smaller average gaps, which increase compression effectiveness.

For the reasons discussed above, modern compression techniques are well suited for document-ordered indexes, since the document ids are coded together and the payload separately. The block structure additionally provides convenient entry points to skip pointers (Moffat and Zobel 1996) and other techniques to avoid decompressing entire blocks (Ding and Suel 2011). Postings lists of impact-ordered indexes, on the other hand, are organized into segments that hold document ids with the same impact score. In general, segments with higher impact scores are shorter—and these are the postings that contribute most to the document ranking. Short segments (runs of document ids) go against the design assumptions of modern block-based index compression techniques. Furthermore, fewer document ids to compress (selected from a fixed-sized collection) means larger *d*-gaps overall, which reduces compression effectiveness.

Let us empirically illustrate this point using the Gov2 collection, a web crawl ($\sim$25 million pages) commonly used in information retrieval experiments. Figure 1 shows the distribution of document frequencies, which quantifies the total length of each postings list in total number of documents. For more details, see the experimental setup in Sect. 5. As expected, we see a Zipf-like distribution.[2] The vertical red line shows the cutoff of 128 integers, below which PForDelta would not be used.

[2] The artifacts in the graph come from boilerplate terms that occur in subsets of the collection.

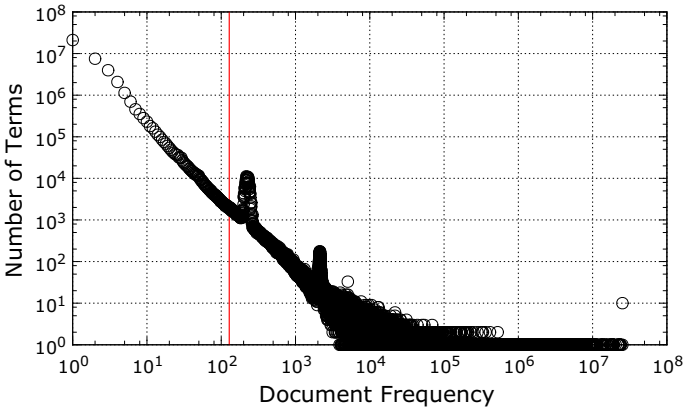

Fig. 1 Distribution of document frequencies for the Gov2 collection

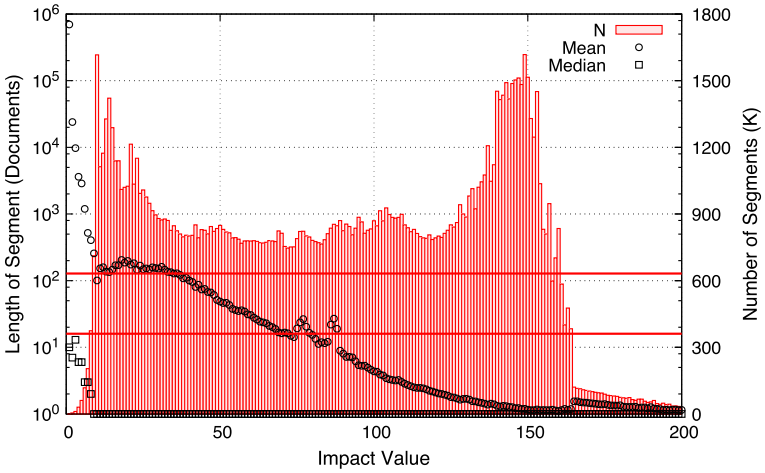

Fig. 2 Distribution of postings segments for an impact-ordered index on the Gov2 collection: the *red bars* show the number of segments that have a particular impact score and the *circles/squares* plot the mean/median length of each segment (Color figure online)

In Fig. 2, we analyze the impacted-ordered index constructed from the same collection (once again, details in Sect. 5). The x-axis shows impact scores and the bar chart (red) shows the number of postings segments with that particular impact score (axis label on the right, unit in thousands). Note that each term typically comprises multiple segments, so it contributes to multiple bars in the chart. In this case, the maximum impact score is 255 (to fit into 8 bits), but the graph cuts off at 200 for better readability since few segments have scores above 200. The circles and squares show the mean and median lengths (respectively) of postings segments with the particular impact score (measured in number of documents). The axis label for these points are on the left; note that the scale is logarithmic. In nearly all cases, the median length is one and the mean is quite short. This figure includes all terms in the collection, even those that appear only once; however, removing those terms does not qualitatively change the shape of the bar chart and has

minimal effect on the mean/median scores. We see that postings segments with high impact scores are generally short. For reference, the 128 length cutoff for PForDelta is shown, as well a length value of 16 (more below).

So why might compression *not* increase query evaluation performance? Consider the case where all document ids that have the same impact score fit on a cache line. We assume four byte integers, which should be sufficient to code document ids in a particular partition (in the case of large-scale distributed search architectures). Assuming proper alignment, 16 integers would fit into a standard 64-byte cache line. In SAAT query evaluation, postings segments holding document ids that share the same impact score are processed one at a time. If there are fewer than 16 documents in such a segment, then all data will be transferred from main memory at once—in this case, it is not clear that compression helps and may in fact incur overhead. We would expect compression to be most helpful if it reduces the number of cache lines the postings segment spans: for example, if compression reduces 32 integers (128 bytes) to 72 bytes, it is unclear if we would get faster query evaluation performance because the integers still span two cache lines.

There are additional complexities to consider. Our calculation above assumes that postings are aligned to cache lines (and alignment requires padding, which increases index size). Intel processors implement a technique called adjacent cache line prefetch, where if one cache line is fetched, the next cache line is prefetched. If this feature is turned on, it would double the breakeven point we've computed above.

In an impact-ordered index, relatively few documents have high impact scores, and these tend to be the segments that will be processed first in a SAAT approach. With early termination optimizations, the hope is that longer segments with lower impact scores might not be processed at all (but see results later). We can illustrate this point empirically in the following manner: for a given query, we can look up all postings segments associated with all the query terms and then sort them by decreasing impact score (breaking ties, let's say, by length of segment in decreasing order). In an impact-ordered query evaluation strategy, this tells us the length of the first segment that will be processed, length of the second, third, fourth, etc. We can then aggregate these statistics across a collection of queries: this is shown in Fig. 3 for TREC topics 701–850 on the Gov2 collection (details in Sect. 5). The x axis denotes the order in which postings segments will be processed, while the y axis denotes the mean (circles) and median (squares) lengths of those segments (in number of

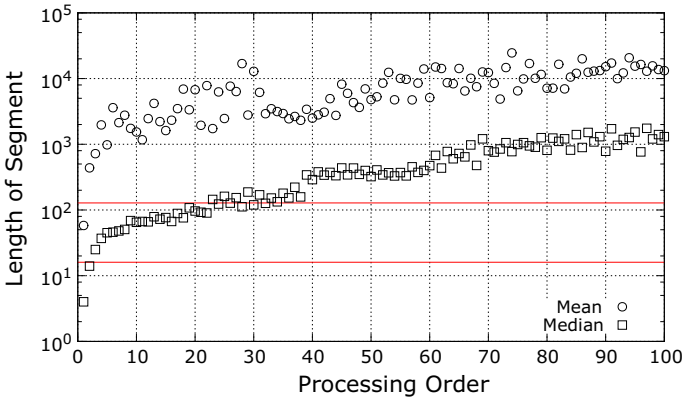

Fig. 3 Length of postings segments for TREC topics 701–850 on Gov2, where the x axis denotes the order in which segments are processed in impact-ordered query evaluation

documents, note log scale). The two horizontal lines are at 128 (PForDelta block) and 16 (integers that fit on a cache line), respectively.

As expected, we see that the early segments are relatively short. This graph poignantly illustrates that PForDelta, which has in many ways emerged as the best practice for compressing document-ordered indexes, is not appropriate for impact-ordered indexes. We see that up until around the 30th postings segment to be processed, the median length still hovers around the 128 threshold. In very rough terms, even if we compress the index using PForDelta, we're actually processing vbyte-compressed postings about half the time, at least for the early postings segments.

Depending on the effectiveness of early termination optimizations, query evaluation might eventually need to process long postings segments with low impact scores. What happens then? We believe that in these cases there will be sufficient striding of memory locations such that the hardware prefetchers will "kick in" and deliver cache lines before the processor explicitly requests them. At that point, it simply boils down to the question of whether the memory bus can "keep up", and we believe that modern memory architectures have sufficient bandwidth to do so (and indeed we show this empirically).

Beyond memory architecture issues, there are additional IR-specific details we must consider: because we're typically compressing *d*-gaps, we need to reconstruct the document ids for query evaluation. This, of course, requires extra instructions. Furthermore, note that *d*-gaps are typically coded with respect to the previous document id, which creates a chain of computational dependencies that does not allow opportunities for out-of-order execution (although in Sect. 4.2 we discuss the SIMD solution to this problem). Finally, for compression schemes that work with blocks of integers, if there are fewer than the block size, we must "signal" that fact to the decoder so that it will not generate document ids that do not exist. The most naïve way would involve an underflow check (i.e., a branch), which defeats much of the point of loop unrolling techniques, the very optimization that makes block-based compression schemes fast.

To summarize: there are a multitude of factors that conspire to reduce the efficiency and effectiveness of compression techniques in impact-ordered indexes. It remains an empirical question how these issues play out in practice, and the remainder of this paper is devoted to this exploration.

4 System design

In this section, we describe the design of our score-at-a-time query evaluation algorithm, which follows the basic approach of Lin and Trotman (2015). We take as a starting point the standard inner-product formulation of ranked retrieval described in Sect. 2.2. In this work, we adopt BM25 term weighting.

Document- and frequency-ordered indexes typically store the term frequency in the index, from which $w_{d,t}$ can be computed. In our impact-ordered indexes, the $w_{d,t}$'s are pre-computed and quantized into b bits (i.e., its impact score). The literature discusses a number of techniques for quantizing the term weights (Anh et al. 2001; Anh and Moffat 2005b; Crane et al. 2013), and in this work we adopt the *uniform* quantization method (Anh et al. 2001), which is an index-wide linear scaling of the term weights:

$$i_{d,t} = \left\lfloor \frac{w_{d,t} - \min(\mathrm{w_{d,t}})}{\max(\mathrm{w_{d,t}}) - \min(\mathrm{w_{d,t}})} \times 2^b \right\rfloor \quad (2)$$

where b is the number of bits used to store the impact. In our implementation we set $b = 8$. Effectiveness results in Sect. 6.1 show that this setting is statistically indistinguishable from exact term weights (and is consistent with previous work). This setting works well because the scores are exactly one byte; otherwise, we would either need to worry about wasted bits or compressing the impact scores.

After quantization, query evaluation becomes a matter of looking up postings corresponding to the query terms, traversing the postings segments, and summing all the impact scores for documents encountered along the way. This requires only integer arithmetic and lends itself to several optimizations, detailed below.

4.1 Query evaluation

In SaaT query evaluation, postings lists segments associated with query terms are sorted by their impact scores (in descending order) and processed in that order (Lin and Trotman 2015). For each document id in a segment, the impact score is added to the accumulator, and thus the final result is an unsorted list of accumulators.

Several methods have been proposed to avoid sorting the list. Anh and Moffat (2006) introduced an approach that Ors high impact postings, then Ands lower impact postings if they can affect the top k documents, before finally applying Refine to obtain the correct ordering of the top k documents. In Or mode, a hash table is used, and for the remaining processing modes the accumulators are converted into a sorted list and merge operations are used. The approach of Anh et al. (2001) keeps an array (keyed on the accumulator score) of lists of accumulators sharing the same score, along with a set of pointers to keep track of the top score and the lowest score of the top k. They outlined how to update these pointers and how to ensure the top k is correctly computed.

We adopt a far simpler approach, based on the work of Jia et al. (2010). Since the impact scores are 8 bits and queries are generally short, it suffices to allocate an array of 16-bit integers, one per document, indexed by the document id; modern hardware has ample memory to keep the accumulators in memory. To avoid sorting the accumulators, a heap of the top k can be maintained during processing. That is, after adding the current impact score to the accumulator array, we check to see if its score is greater than the smallest score in the heap; if so, the pointer to the accumulator is added to the heap. The heap keeps at most k elements, and we break ties arbitrarily based on document id.

Jia et al. (2010) showed that the initialization of these accumulators (i.e., `memset`) is quite expensive and can consume a substantial portion of the query evaluation time. The maximum number of non-zero accumulators is the union of all postings, which is usually small compared to the size of the collection. The solution to this problem is an approach similar to how the operating system manages its memory page table. Although the accumulators occupy a contiguous region in memory, the array is logically broken down into a number of "pages", each with an associated dirty bit. At the start of query evaluation, the dirty bits are all set to indicate that all pages are dirty. When adding to an accumulator, the dirty bit for that page is checked and if set, the entire page is zeroed (i.e., with `memset`). The dirty bit is cleared and query evaluation proceeds as normal. This optimization allows us to initialize only the pages containing the accumulators that are needed. Jia et al. provided a probabilistic derivation of the optimal page size based on the document frequency of terms in the query, but we adopt a simpler approach by setting the page size to

$\exp(2, \text{floor}(\log_2 \sqrt{D}))$ where D is the number of documents in the collection. The square root roughly equalizes the number of pages and the number of accumulators in each page, and working with powers of two allows us to determine the page number from the document id with a bit shift.

Our approach naturally lends itself to an early termination optimization. At the start of query evaluation, the maximum possible score that any document might achieve, $\max_i$, is the sum of the maximum impact scores for each term in the query (Turtle and Flood 1995). After processing the first segment, the associated impact score is subtracted from $\max_i$ and that term's remaining highest impact score is added (or 0 if there are no more segments to be processed). This $\max_i$ might be used in several ways: if the exact ordering of documents in the top k is not required, we can terminate when the difference between the accumulator score for the $(k + 1)$-th document and the k-th document is larger than $\max_i$. That is, given the remaining impact scores, it is not possible for any lower-ranked documents to make it into the top k. If the correct document ranking of the top k is desired (as in our case), we must perform this check for all of the documents in the top k, i.e., if the difference between any two consecutive elements in the top $k + 1$ is larger than $\max_i$ then query evaluation can terminate because the top k ranking can no longer change.

There is one final optimization worth mentioning: in our approach, the heap holding the top k documents only needs to store a pointer to the accumulator holding the impact score. That is, the size of the heap is k times the size of a pointer, which is 80 bytes when pointers are 8 bytes and k is 10. We have no need to explicitly store the document id alongside the impact score as an accumulator object (or struct) because we can compute the document id via pointer arithmetic given the start of the accumulator array, which in turn holds the score. Thus, our heap is very compact and will likely reside in cache throughout query evaluation.

Note that although there are many different techniques to accumulator management for score-at-a-time query evaluation, and we adopt an approach that is quite different in many ways, the differences here are unlikely to affect the outcome of our experiments. This machinery is the same across all our experimental conditions, since we vary only the compression codec.

4.2 Index organization and compression

Our indexes are organized in a relatively standard manner. The dictionary provides the entry point to each postings list; each term points to a list of tuples containing (impact score, start, end, count). Each tuple corresponds to a postings segment with a particular impact score, the start and end are pointers to the beginning and end of the segment data (which contain the document ids), and the count stores the number of documents in that segment. Segments for each term are ordered in decreasing impact score and within each segment documents are ordered by increasing document id.

Document ids in each postings segment are represented using the techniques described below. Note that we specifically did not evaluate PForDelta because it is not appropriate for impact-sorted indexes, as discussed in Sect. 3.

Uncompressed: In this condition, we simply leave the document ids uncompressed and stored in 32-bit integers. Note that there is no point in computing d-gaps since it does not save any space.

Variable byte encoding: In variable byte encoding (Witten et al. 1999) (vbyte for short), an integer is broken into 7-bit chunks and each chunk is written to a separate byte. The remaining bit of each byte is reserved as a stop bit to indicate if the decoder has reached the

end of the integer: the value of this bit is zero in all cases except the final byte where one is stored. Following standard practice, we code the differences between consecutive document ids (i.e., the *d*-gaps). Trotman (2014) notes that vbyte actually refers to a family of related techniques that vary in implementation choices (e.g., endian, placement of the stop bit, etc.) that impact performance. To be clear, we use the implementation of vbyte from the open-source ATIRE system (Trotman et al. 2012).

Simple-8b: Instead of coding one integer at a time, Anh and Moffat (2005a) proposed packing as many integers as possible into fixed-sized words. In the Simple-9 scheme, they break a 32-bit integer into two parts: a 4-bit selector and a 28-bit payload. There are nine possible ways to pack integers into 28-bits if all integers must take the same number of bits (28 one-bit integers, 14 two-bit integers, 9 three-bit integers, etc.), and hence the name—the selector codes how the payload is packed. Subsequent refinements by Anh and Moffat (2010) extended this encoding to 64-bit integers: their Simple-8b approach, which we adopt, maintains a 4-bit selector, but uses a 60-bit payload. This compression scheme has been shown to be more space effective and decoding efficient than both vbyte and Simple-9, representing the best overall approach in the Simply family of techniques.

QMX: The most recent generation of integer compression techniques take advantage of SIMD instructions (Stepanov et al. 2011; Trotman 2014; Lemire and Boytsov 2015). In this work, we use QMX (Trotman 2014), which has been shown to be more efficient to decode than SIMD-BP128 (Lemire and Boytsov 2015) (previously the most decoding efficient overall) on short and long postings lists (but not medium-length postings lists) and competitive with all SIMD and non-SIMD techniques in terms of size.

QMX is similar in spirit to Simple-8b, but with a number of improvements. Integers are packed into 128-bit payloads; the selectors are divided into two parts: four bits encode the packing and four bits encode a run-length of that packing. Unlike in the Simple family, all the payloads are stored first and then the run-length encoded selectors are stored afterwards. Thus, all payloads are 16-byte word-aligned for processing by SIMD instructions (and thus decoding is very fast); selectors are byte-aligned. Critically, QMX is *not* block-based, which makes it well-suited for impact-ordered indexes (see discussion in Sect. 3).

QMX allows for different possibilities for coding *d*-gaps. The standard approach of computing differences between consecutive document ids is called D1, following the terminology introduced by Lemire et al. (2014). In D4, gaps are computed with respect to the document id four positions earlier, i.e., the fifth integer encodes the difference relative to the first, the sixth relative to the second, etc. This approach takes advantage of a SIMD instruction in modern Intel processors that concurrently performs four 32-bit integer additions between two SSE registers, thus allowing four gaps to be decoded in one instruction. The downside, however, is that D4-based gaps are usually larger than D1-based gaps (and thus take more space). For completeness, we also introduced a D∞ condition, where the document ids are compressed directly—this provides a reference point to assess the overall effectiveness of gap encoding. Thus, for QMX, we have the D4, D1, and D∞ variants.

5 Experimental setup

The SaaT query evaluation approach and compression techniques described in the previous section were implemented in JASS (Lin and Trotman 2015), an open-source search engine written in C, with some components in C++. Instead of building a fully-featured search engine, we took advantage of inverted indexes built by the ATIRE system (Trotman et al.

2012), which saved us from having to write a separate indexer—our system reads the index structures generated by ATIRE and recompresses the postings in the appropriate manner. Since we wish to examine the effects of compression, our experiments used exactly the same query evaluation algorithm, varying only the compression scheme applied: no compression, vbyte, Simple-8b, and variants of QMX. All source code necessary to replicate our experiments are available online under an open-source license.[3]

Our experiments used four standard TREC web test collections: Wt10g, Gov2, ClueWeb09 (category B), and Clue-Web12-B13 (i.e., "category B"). Details for these collections are provided in Table 1, showing the sizes of each and the corresponding topics used to evaluate effectiveness (both the TREC topics and the years from which they were drawn). For simplicity, we kept collection processing to a minimum: for each document, all non UTF-8 characters were converted into spaces, alphabetic characters were separated from numeric characters (but no stemming was applied); no additional document cleaning was performed except for XML comment removal (i.e., no HTML tag removal, JavaScript removal, etc.). We did not remove stopwords.

Experiments were conducted on a server with dual 10-core Intel Xeon E5-2670 v2 (Ivy Bridge, launched Q3 2013) running at 2.5 GHz with 244 GiB RAM on Amazon's EC2 service (instance type r3.8xlarge). This is the same processor used in a recent reproducibility study (Lin et al. 2016), which facilitates comparability of results in absolute terms. Our primary metric is query latency, the time it takes for our query evaluation engine to produce the top k ranking, measured with the `gettimeofday()` system call. Our measurements exclude file I/O costs, i.e., we keep track of the time it takes to materialize the top k documents in main memory, but do not include the time taken to write the output files for evaluation. We also exclude one-time startup costs such as loading dictionaries and other auxiliary data structures. Upon startup in each condition, the query engine initializes the relevant data structures, loads the query postings into main memory, and runs through the query set twice. We only measure query latencies for the second run through: this is to ensure we're not measuring latencies associated with virtual memory page faults (and other memory-related idiosyncrasies). Note that our indexes are memory resident, but *not* cache resident since the postings are much larger than the cache.

To test different compression schemes, we cycled through each of them in a batch. The experimental conditions were arranged in a Latin square, where the absolute order of each condition varied from batch to batch. This design also counterbalances the relative order between pairs of conditions to minimize possible interactions. The server on which we ran the experiments was otherwise idle. We ran a total of 32 trials for each condition and all reported results represent aggregates over these runs.

Index size is straightforwardly measured in GB (10^9 bytes). These statistics include only the postings, and do not include auxiliary data structures such as the vocabulary and the mapping from internal document ids (integers) to the collection document ids (strings).

6 Results

6.1 Effectiveness

Our first set of experiments verified the correctness of our implementation by comparing against the results of the ATIRE search engine: since we simply reused its index, we obtain

[3] https://github.com/lintool/JASS.

Table 1 Summary of TREC collections and topics used in our experiments

Name	# Docs	TREC topics
Wt10g	1,692,096	451–550 ('00–'01)
Gov2	25,205,179	701–850 ('04–'06)
ClueWeb09b	50,220,423	1–200 ('09–'12)
ClueWeb12-B13	52,343,021	201–300 ('13–'14)

identical scores and rankings. Results are shown in Table 2 under the column ATIRE (I); we report both average precision (AP) measured at the standard setting of rank 1000 and also NDCG@10. This experimental design also allowed us to isolate the effects of impact quantization by building standard indexes that store term frequencies, shown under the column ATIRE (tf). As expected, quantization has negligible impact on effectiveness: none of the differences are statistically significant, based on Fisher's two-sided, paired randomization test (Smucker et al. 2007); $p < 0.05$ for this and all subsequent comparisons.

As a third comparison, we evaluated the effectiveness of the state-of-the-art Terrier search engine on the same collections, using "out of the box" settings suggested in the documentation. None of the differences in effectiveness are statistically significant except on ClueWeb09b (for both metrics). The point of this comparison is not to make any statements about the effectiveness of Terrier; rather, our goal is to "sanity check" our results to make sure they are roughly comparable to the state of the art.

For the remaining experiments, we set aside questions of effectiveness and focus exclusively on efficiency. In all cases we have verified the correctness of our algorithms in that they all produce exactly the same result.

6.2 Query latency and index size

Table 3 shows the query evaluation latency for retrieving the top 10 hits under different compression conditions across the four test collections. Latency is reported in milliseconds, with 95% confidence intervals. Note that for ClueWeb12-B13 we only have 100 topics, which explains the larger confidence intervals; correspondingly, for ClueWeb09b we have 200 topics, and hence the confidence intervals are much smaller. Relative differences with respect to the uncompressed condition are also shown. Here, we only show results for QMX-D4 (the fastest of the variants), but explore the other variants in more detail in Sect. 6.3.

For each collection, ANOVA confirms that there are significant differences in the mean latencies. Pairwise t tests between the uncompressed condition and each of the others show that all differences are significant ($p < 0.05$) with the exception of QMX-D4 in Gov2.[4] All the different factors discussed in Sect. 3 play out to yield an interesting result: index compression actually hurts performance, albeit QMX-D4 is only slightly slower. That is, the fastest query evaluation speed is achieved by leaving the postings uncompressed at the expense of a considerably larger index, a finding that is consistent across all four collections.

Leaving aside uncompressed postings, the compression results are consistent with findings reported in the literature. Vbyte provides a reasonable baseline, but a block-based

[4] Note that means can still be significantly different even though confidence intervals overlap (Wolfe and Hanley 2002), as is the case of uncompressed vs. QMX-D4 for ClueWeb12-B13.

Table 2 AP@1000 and NDCG@10 comparing the effectiveness of our reference index (tf) and quantized impacted index (I)

	ATIRE (tf)	ATIRE (I)	Terrier
AP@1000			
Wt10g	0.1730	0.1730	0.1880
Gov2	0.2648	0.2645	0.2697
ClueWeb09b	0.1881	0.1872	0.1697
ClueWeb12-B13	0.2468	0.2407	0.2105
NDCG@10			
Wt10g	0.3036	0.3039	0.3008
Gov2	0.4146	0.4122	0.4242
ClueWeb09b	0.2016	0.1980	0.1733
ClueWeb12-B13	0.2747	0.2731	0.2461

Terrier is shown as a reference

Table 3 Query latency in milliseconds (with 95% confidence intervals) for retrieving top 10 hits across the four test collections

	Wt10g		Gov2	
Uncompressed	5.30 ± 0.05		40.9 ± 1.56	
Vbyte	7.66 ± 0.05	+45%	51.4 ± 1.55	+26%
Simple-8b	6.53 ± 0.05	+23%	45.3 ± 1.30	+11%
QMX-D4	5.61 ± 0.05	+6%	41.4 ± 0.05	+1%

	ClueWeb09b		ClueWeb12-B13	
Uncompressed	110 ± 0.07		191 ± 6.01	
Vbyte	140 ± 0.05	+27%	231 ± 6.02	+21%
Simple-8b	124 ± 0.21	+12%	212 ± 6.19	+11%
QMX-D4	116 ± 0.07	+5%	202 ± 6.57	+6%

technique such as Simple-8b is faster due to reduction in processor stalls. Exploiting SIMD instructions, as in QMX-D4, further enhances query evaluation performance.

Table 4 shows index sizes (GB $= 10^9$ bytes) under each compression condition across the four collections (with relative comparisons to uncompressed indexes). With the exception of Wt10g, which is much smaller than the other collections, the results are consistent: compression cuts the index size in about half. Simple-8b and vbyte achieve roughly the same compression ratio, while indexes with QMX-D4 are slightly larger.

6.3 Analysis of gap compression

Table 5 compares the QMX variants in terms of both query evaluation latency ($k = 10$) and index size. Query latency measurements followed the same methodology as before and are reported in milliseconds with 95% confidence intervals. The uncompressed condition is shown for reference; relative differences are computed with respect to QMX-D4. We only show results for ClueWeb09b and ClueWeb12-B13, although results from the other two collections are similar and do not change our conclusions.

ANOVA detects significant differences in mean latencies for ClueWeb09b, and t tests confirm that the D1 and D∞ variants are significantly slower than QMX-D4. ANOVA does not detect significant differences for ClueWeb12-B13, although the trends are consistent with the other collection: D1 is slower than D4, and D∞ is slower than D1.

Table 4 Index size in GB under different compression conditions across the four test collections

	Wt10g		Gov2	
Uncompressed	2.811		36.07	
Vbyte	1.675	−40.4%	17.48	−51.5%
Simple-8b	1.697	−39.6%	16.48	−54.3%
QMX-D4	2.133	−24.1%	19.61	−45.6%

	ClueWeb09b		ClueWeb12-B13	
Uncompressed	102.6		118.5	
Vbyte	47.65	−53.6%	58.90	−50.3%
Simple-8b	45.65	−55.5%	56.75	−52.1%
QMX-D4	55.10	−46.3%	67.75	−42.8%

Table 5 Query latency (in milliseconds) and index size comparisons (in GB) for QMX variants

	ClueWeb09b			
	Query latency		Index size	
Uncompressed	110 ± 0.07		102.6	
QMX-D4	116 ± 0.07		55.10	
QMX-D1	118 ± 0.44	+1.7%	51.54	−6.5%
QMX-D∞	119 ± 0.07	+2.6%	106.7	+94%

	ClueWeb12-B13			
	Query latency		Index size	
Uncompressed	191 ± 6.01		118.5	
QMX-D4	202 ± 6.57		67.75	
QMX-D1	207 ± 5.95	+2.5%	64.04	−5.5%
QMX-D∞	209 ± 6.17	+3.5%	125.0	+85%

We see that QMX-D1 yields a slightly smaller index than QMX-D4, which makes sense because the *d*-gaps are smaller. However, QMX-D4 is a bit faster due to the use of SIMD instructions to reconstruct the document ids. This appears to represent a time/space tradeoff. Interestingly, QMX-D∞, which applies no gap compression (i.e., QMX-codes the document ids directly) actually takes more space than no compression (4-byte integers). Since the document id space requires 26 bits, the overhead of QMX (e.g., in requiring 16-byte aligned chunks, even when coding very short segments) does not make up for the few bits saved in compression. Therefore, we see no compelling reason to adopt QMX-D∞, although this condition shows that storing *d*-gaps (in whatever form) is a good idea.

6.4 Early termination optimizations

In Sect. 3, we articulated the potential advantages of uncompressed postings: for short postings, inherent cache locality, and for long postings, raw memory bandwidth and prefetching. In SaaT query evaluation, the length of postings segments increases as

evaluation progresses, since higher impact segments tend to be short (see Fig. 3). Thus, we would like to know: Is the performance advantage of uncompressed postings lists coming from cache locality or raw memory bandwidth and prefetching?

In Table 6, we show the number of postings segments that are processed with increasing values of k (across all queries in each collection). The $k = \infty$ setting is simply a value of k that is larger than the number of documents in the collection, thus producing a ranking of all documents that contain at least one query term. Interestingly, we see that our early termination technique is not effective; that is, we are nearly always searching to completion at $k = 10$. Why is this so? If for any topic, any of the documents in ranks one through k differ by an impact score of one, then we cannot early terminate and must process all postings segments. Spot checking of the document scores confirms that this is indeed the case, as impact scores between consecutive ranks are generally small, with frequent ties.

To further explore this issue, we attempted to empirically characterize the processing cost of each postings segment in terms of its length. This is accomplished by breaking all queries in the test collections into single-term queries to eliminate cross-term interactions. These queries are issued with $k = \infty$, which disables early-termination checks. The query evaluation algorithm is instrumented to time the processing of each postings segment, and we dump out execution traces that include the impact score, length, and processing time for each segment; we include all heap manipulations in the timing measurements. Processing time is measured in microseconds, but the absolute scores are meaningless because a substantial portion of the processing time is taken up by the system call that gathers the timing information (i.e., the measurement overhead). However, since this measurement overhead is constant, comparisons *across* different compression techniques are meaningful.

The results of this experiment are shown in Fig. 4 comparing uncompressed indexes with QMX-D4 for ClueWeb09b and ClueWeb12-B13. Each point represents a postings segment: uncompressed in red circles and QMX-D4 in blue diamonds (due to limitations of the medium, there is no possible plot that can avoid significant occlusions; quite arbitrarily here, the blue diamonds lie on top of the red circles). The variance in processing time for postings segments of a particular length (i.e., the vertical spread of the points) can be attributed to a combination of inherent noise in the measurements and the heap manipulation overhead. Naturally, we observe more variance for short segments due to timing granularity issues. Note that some processed postings segments are quite long due to the presence of stopwords that were not removed during indexing.

Interestingly, we see an artifact (more prominent in Clue-Web12-B13) where processing time increases quickly with length, and then the slope tapers off—we believe that this shows the effect of the prefetcher "kicking in" once a predictable memory striding pattern has been established. Overall, we see that processing the considerably larger uncompressed segments is faster than segments compressed with QMX-D4 (i.e., red points

Table 6 Total number of postings segments processed for different scores of k in top k retrieval

k	Wt10g	Gov2	CW09	CW12
10	22,820	31,784	25,900	18,671
20	22,834	31,785	25,900	18,671
100	22,835	31,785	25,900	18,671
1000	22,835	31,785	25,900	18,674
∞	22,835	31,785	25,900	18,674

CW09 = ClueWeb09b, CW12 = ClueWeb12-B13

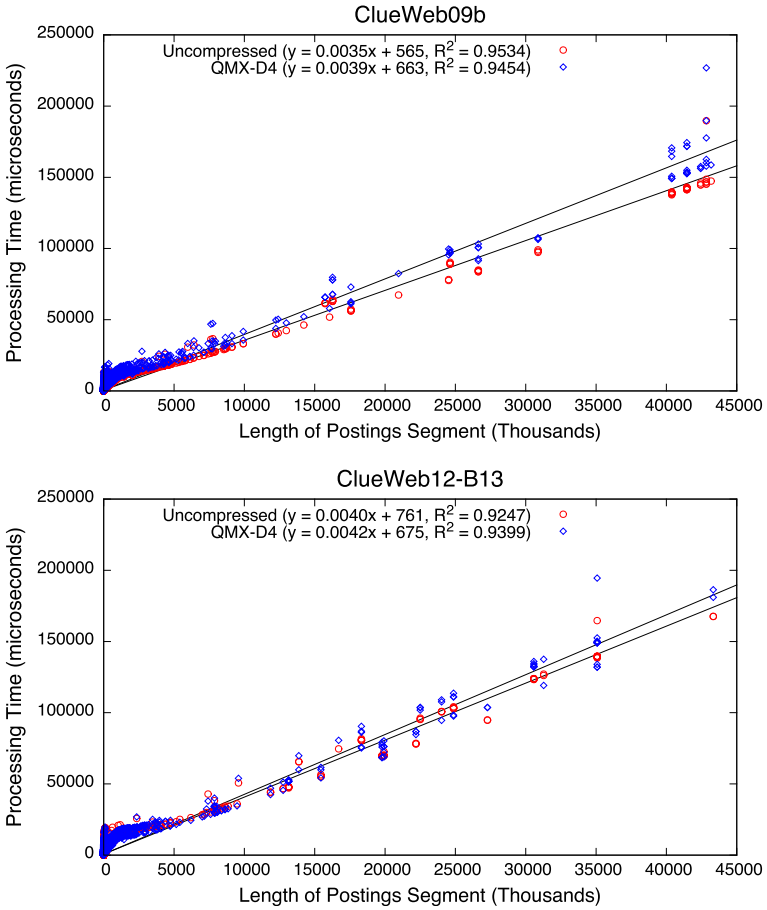

Fig. 4 Processing time (including heap manipulation) with respect to length of postings segment for single-term queries on the ClueWeb09b and ClueWeb12-B13 collections

are below the blue points for the most part). Fitting a linear regression to the points confirms this; the equations of the best fit lines are shown in the figures along with the R^2 value (indicating a good fit). The slopes can be interpreted as the per-posting processing cost, confirming that uncompressed postings are faster despite being larger. This analysis explains the end-to-end query evaluation results reported in Table 3 and provides independent triangulating evidence to confirm our findings. We have also plotted similar graphs for the other compression techniques, which yield the same conclusion; the only difference is that the slopes of the best fit lines are larger, which accounts for their slower query evaluation speeds.

7 Discussion and limitations

What do we make of these results? We have shown that for score-at-a-time query evaluation on impact-ordered indexes in main memory, index compression does *not* increase query evaluation speed, at least for the techniques we examined. Our experiments

consistently show across four collections that uncompressed postings yield the fastest query evaluation, albeit beating QMX-D4 by only a small margin, even though the index is considerably larger then the index of QMX-D4. Although we have not exhaustively explored all possible compression schemes, our techniques do include the state-of-the-art QMX algorithm (which is representative of the latest developments in integer compression), and so we are reasonably confident that our finding will hold if we consider other compression techniques.

The undeniable advantage of compression, of course, is the substantially smaller index size—saving between 40 and 50% on large web collections and thus requiring less memory. Is this tradeoff compelling? In applications where performance considerations dominate (e.g., financial trading), then uncompressed indexes should be worthwhile. From a more balanced economic perspective, the relevant tradeoff would be between a higher clock frequency and more memory (as well as the power implications thereof) to achieve roughly the same performance. Of course, given a fixed amount of memory in a server, smaller index sizes would translate into more space for other uses, for example, result caches—although in a distributed search architecture it is more likely that result caches are managed centrally and not "pushed down" to the individual index partition servers. Overall, we lack the proper context, access to different hardware variants, and detailed knowledge of deployment architectures to fully articulate this complex tradeoff space, but search engine developers working in production environments can straightforwardly evaluate various design choices for specific application scenarios. We simply offer that when considering different designs, the less conventional option of leaving postings lists (or leaving high-impact segments in an impact-ordered index) uncompressed should also be considered.

There are a number of limitations of this present study that should be explicitly noted. We have examined only single-threaded performance: today's processors, however, often have a dozen or more cores. It would be interesting as part of future work to examine query performance under heavier processor utilization with concurrent threads. It is unclear how memory requests from different query execution threads would interact with different compression schemes. Another limitation is the relatively small collection of queries used in our evaluation: TREC topics were selected due to the availability of relevance judgments, which allowed us to establish the effectiveness of our score-at-a-time query evaluation approach with impact score quantization. Although our latency measurements do include 95% confidence intervals, a larger collection of queries would have allowed us to study outliers, i.e., so called "tail latencies" (Dean and Barroso 2013), in more detail. The performance of outlier queries is of particular interest for commercial search engines who strive to maintain a consistent quality of service for all users.

Another interesting finding from our experiments is that a standard early termination optimization for top k SAAT query evaluation is largely ineffective due to the distribution of document scores in the ranked results. This means that query evaluation requires processing nearly all the postings nearly all the time if a correct top k ranking is desired. Summarizing this result more colloquially, if the purpose of early termination is to decide if "work can be saved" during query evaluation, the cost of making that decision is roughly the same as just "doing the work" to begin with. This finding is consistent with present thinking on the design of high-performance applications in general, for example, work in high-speed transactional processing (Thompson and Barker 2010). In terms of raw instruction throughput, modern processors can easily offer billions per second: this makes irregular control flows and random memory accesses exceedingly expensive when they cannot be masked by traditional hardware mechanisms (branch prediction, caching and

prefetching, etc.). Building on this lesson, it would be interesting to expand our study to DaaT query evaluation on document-ordered indexes. The block-max Wand algorithm (Ding and Suel 2011) and variants, e.g., (Rossi et al. 2013)—currently the state of the art—enable skipping of large portions of postings lists, but at the cost of more complex data structure manipulations (e.g., pivoting, searching, etc.). These involve irregular control flows (branches due to search) and more random data access patterns (skipping), introducing many data and control hazards. In contrast, with our SaaT approach, hazards are kept at a minimum with the goal of maximizing processor throughput (especially if we disable the early termination check). It would be interesting to compare DaaT and SaaT techniques from the perspective of architecture affinity.

8 Conclusion

At a high level, this paper provides a reminder that at its core, information retrieval has been and will remain an empirical discipline, and as such, the effectiveness and efficiency of various techniques must be experimentally validated. In the realm of query evaluation performance, this means that generalizations may not carry across different execution settings. For example, compression results for on-disk indexes may not hold for in-memory indexes; observations about document-at-a-time techniques may not carry over to score-at-a-time techniques. As a result, we need to continually refine the design and implementation of query evaluation techniques to ensure that they keep up with progress in hardware design.

References

Amati, G., & van Rijsbergen, C. J. (2002). Probabilistic models of information retrieval based on measuring the divergence from randomness. *ACM Transactions on Information Systems*, *20*(4), 357–389.

Anh, V. N., & Moffat, A. (2005a). Inverted index compression using word-aligned binary codes. *Information Retrieval*, *8*(1), 151–166.

Anh, V.N., & Moffat, A. (2005b). Simplified similarity scoring using term ranks. In *Proceedings of the 28th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2005)* (pp. 226–233). Salvador.

Anh, V. N., & Moffat, A. (2006). Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2006)* (pp. 372–379). Seattle.

Anh, V. N., & Moffat, A. (2010). Inverted index compression using word-aligned binary codes. *Software: Practice and Experience*, *40*(2), 131–147.

Anh, V. N., de Kretser, O., & Moffat, A. (2001). Vector-space ranking with effective early termination. In *Proceedings of the 24th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2001)* (pp. 35–42). New Orleans.

Asadi, N., & Lin, J. (2013). Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proceedings of the 36th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2013)* (pp. 997–1000). Dublin.

Brewer, E. A. (2005). Combining systems and databases: A search engine retrospective. *Readings in database systems*. Cambridge: MIT Press.

Büttcher, S., & Clarke, C.L.A. (2007). Index compression is good, especially for random access. In *Proceedings of the sixteenth international conference on information and knowledge management (CIKM 2007)* (pp. 761–770). Lisbon.

Cambazoglu, B.B., Zaragoza, H., Chapelle, O., Chen, J., Liao, C., Zheng, Z., & Degenhardt, J. (2010). Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the third ACM international conference on web search and data mining (WSDM 2010)* (pp. 411–420). New York.

Crane, M., Trotman, A., & O'Keefe, R. (2013). Maintaining discriminatory power in quantized indexes. In *Proceedings of 22nd international conference on information and knowledge management (CIKM 2013)* (pp. 1221–1224). San Francisco.

Dean, J. (2009). Challenges in building large-scale information retrieval systems. In *Keynote presentation at the second ACM international conference on web search and data mining (WSDM 2009)*. Barcelona.

Dean, J., & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, *56*(2), 74–80.

Ding, S., & Suel, T. (2011). Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34rd annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2011)* (pp. 993–1002). Beijing.

Fontoura, M., Josifovski, V., Liu, J., Venkatesan, S., Zhu, X., & Zien, J. (2011). Evaluation strategies for top-k queries over memory-resident inverted indexes. In *Proceedings of the 37th international conference on very large data bases (VLDB 2011)* (pp. 1213–1224). Seattle.

Jacob, B. (2009). *The memory system: you can't avoid it, you can't ignore it, you can't fake it*. San Rafael: Morgan & Claypool Publishers.

Jia, X.F., Trotman, A., & O'Keefe, R. (2010). Efficient accumulator initialisation. In *Proceedings of the 15th Australasian document computing symposium (ADCS 2010)*.

Lemire, D., & Boytsov, L. (2015). Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, *45*(1), 1–29.

Lemire, D., Boytsov, L., & Kurz, N. (2014). SIMD compression and the intersection of sorted integers. arXiv:1401.6399v9.

Lin, J., & Trotman, A. (2015). Anytime ranking for impact-ordered indexes. In *Proceedings of the ACM international conference on the theory of information retrieval (ICTIR 2015)* (pp. 301–304). Northampton.

Lin, J., Crane, M., Trotman, A., Callan, J., Chattopadhyaya, I., Foley, J., et al. (2016). Toward reproducible baselines: The open-source IR reproducibility challenge. In *Proceedings of the 38th European conference on information retrieval (ECIR 2016)* (pp. 408–420). Padua.

Moffat, A., & Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, *14*(4), 349–379.

Patterson, D. A. (2004). Latency lags bandwidth. *Communications of the ACM*, *47*(10), 71–75.

Persin, M., Zobel, J., & Sacks-Davis, R. (1996). Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, *47*(10), 749–764.

Ponte, J. M., & Croft, W. B. (1998). A language modeling approach to information retrieval. In *Proceedings of the 21st annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 1998)* (pp. 275–281). Melbourne.

Robertson, S. E., Walker, S., Hancock-Beaulieu, M., Gatford, M., & Payne, A. (1995). Okapi at TREC-4. In *Proceedings of the fourth text retrieval conference (TREC-4)* (pp. 73–96). Gaithersburg.

Rossi, C., de Moura, E. S., Carvalho, A. L., & da Silva, A. S. (2013). Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of the 36th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2013)* (pp. 183–192). Dublin.

Salton, G. (1971). *The SMART retrieval system-experiments in automatic document processing*. Englewood Cliffs: Prentice-Hall.

Scholer, F., Williams, H. E., Yiannis, J., & Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2002)* (pp. 222–229). Tampere.

Silvestri, F., & Venturini, R. (2010). VSEncoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of 19th international conference on information and knowledge management (CIKM 2010)* (pp. 1219–1228). Toronto.

Smucker, M.D., Allan, J., & Carterette, B. (2007). A comparison of statistical significance tests for information retrieval evaluation. In *Proceedings of the sixteenth international conference on information and knowledge management (CIKM 2007)* (pp. 623–632). Lisbon.

Stepanov, A. A., Gangolli, A. R., Rose, D. E., Ernst, R. J., & Oberoi, P. S. (2011). SIMD-based decoding of posting lists. In *Proceedings of 20th international conference on information and knowledge management (CIKM 2011)* (pp. 317–326). Glasgow.

Strohman, T., & Croft, W. B. (2007). Efficient document retrieval in main memory. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2007)* (pp. 175–182). Amsterdam.

Strohman, T., Turtle, H., & Croft, W.B. (2005). Optimization strategies for complex queries. In *Proceedings of the 28th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2005)* (pp. 219–225). Salvador.

Thompson, M., & Barker, M. (2010). LMAX: How to do 100k TPS at less than 1ms latency. In *Presentation at QCon San Francisco 2010*

Tonellotto, N., Macdonald, C., & Ounis, I. (2013). Efficient and effective retrieval using selective pruning. In *Proceedings of the sixth ACM international conference on web search and data mining (WSDM 2013)* (pp. 63–72). Rome.

Trotman, A. (2003). Compressing inverted files. *Information Retrieval*, *6*(1), 5–19.

Trotman, A. (2014). Compression, SIMD, and postings lists. In *Proceedings of the 2014 Australasian document computing symposium (ADCS '14)* (pp. 50–57). Melbourne.

Trotman, A., Jia, X.F., & Crane, M. (2012). Towards an efficient and effective search engine. In *Proceedings of the SIGIR 2012 workshop on open source information retrieval*. Portland.

Turtle, H., & Flood, J. (1995). Query evaluation: Strategies and optimizations. *Information Processing and Management*, *31*(6), 831–850.

Wang, L., Lin, J., & Metzler, D. (2011). A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR 2011)* (pp. 105–114). Beijing.

Williams, H. E., & Zobel, J. (1999). Compressing integers for fast file access. *The Computer Journal*, *42*(3), 193–201.

Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing gigabytes: Compressing and indexing documents and images*. San Francisco: Morgan Kaufmann Publishing.

Wolfe, R., & Hanley, J. (2002). If we're so different, why do we keep overlapping? *CMAJ*, *166*(1), 65–66.

Wu, H., & Fang, H. (2014). Document prioritization for scalable query processing. In *Proceedings of 23rd international conference on information and knowledge management (CIKM 2014)* (pp. 1609–1618). Shanghai.

Yan, H., Ding, S., & Suel, T. (2009). Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international eorld wide web conference (WWW 2009)* (pp. 401–410). Madrid.

Zhang, J., Long, X., & Suel, T. (2008). Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international world wide web conference (WWW 2008)* (pp. 387–396). Beijing.

Zukowski, M., Héman, S., Nes, N., & Boncz, P. (2006). Super-scalar RAM-CPU cache compression. In *Proceedings of the IEEE 22nd international conference on data engineering (ICDE 2006)*. Atlanta.