Examining the Additivity of Top-k Query Processing Innovations

Joel Mackenzie
The University of Melbourne
Melbourne, Australia

Alistair Moffat
The University of Melbourne
Melbourne, Australia

ABSTRACT

Research activity spanning more than five decades has led to index organizations, compression schemes, and traversal algorithms that allow extremely rapid response to ranked queries against very large text collections. However, little attention has been paid to the interactions between these many components, and the *additivity* of algorithmic improvements has not been explored. Here we examine the extent to which efficiency improvements *add up*. We employ four query processing algorithms, four compression codecs, and all possible combinations of four distinct further optimizations, and compare the performance of the 256 resulting systems to determine when and how different optimizations interact. Our results over two test collections show that efficiency enhancements are, for the most part, additive, and that there is little risk of negative interactions. In addition, our detailed profiling across this large pool of systems leads to key insights as to *why* the various individual enhancements work well, and indicates that optimizing "simpler" implementations can result in higher query throughput than is available from non-optimized versions of the more "complex" techniques, with clear implications for the choices needing to be made by practitioners.

KEYWORDS

Query Processing, Dynamic Pruning, Experimentation, Additivity

ACM Reference Format:
Joel Mackenzie and Alistair Moffat. 2020. Examining the Additivity of Top-k Query Processing Innovations. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340531.3412000

1 INTRODUCTION

Large web search engines serve results to many thousands of queries each second, against collections containing billions of documents. While the cost of each search is inconsequential [52], in aggregate there are enormous expenditures involved, for hardware and for electricity (the latter for both processing and cooling). Query processing cost minimization is thus a critical business driver for search vendors. The deployment of low-latency systems is also an important factor in terms of user satisfaction and retention [6, 46].

In any search system, there is a trade-off between *efficiency* and *effectiveness*. To boost the latter, more complex similarity models are employed, with more extensive processing steps involved for each query. Effectiveness is often measured using test collections comprising a fixed set of documents, a fixed set of queries, and a set of relevance judgments, with numeric scores calculated using an agreed effectiveness metric. That framework allows large-scale repeated experimentation across techniques, and like-for-like comparison of alternatives; it has also supported long-running projects such as TREC and NTCIR. In a retrospective survey undertaken in 2009, Armstrong et al. [5] examined the *additivity* of a set of orthogonal techniques for improving effectiveness, finding that while heuristics are broadly additive, it should not be assumed that improvements always "add up". Other researchers have also explored additivity across effectiveness enhancements [1, 25].

Here we examine the extent to which *efficiency* improvements can be said to "add up". We employ four query processing algorithms, four compression codecs, and all possible combinations of four distinct further optimizations, and compare the performance of the 256 resulting systems on two large document collections to determine when and how different optimizations interact. Query processing time (50th and 95th percentiles) for the key task of top-k retrieval (that is, identifying and reporting the k highest-scoring documents as identified by a defined similarity heuristic) is the primary yardstick used for comparison, but index size is also of interest, plus, in the final analysis, implementation complexity as a surrogate for programming cost.

Given this context, a range of questions can be articulated:

- RQ1: Do efficiency enhancements provide more benefit to weak algorithms than to strong algorithms?
- RQ2: Are efficiency enhancements additive?
- RQ3: Is there any "volatility risk" introduced when efficiency enhancements are added?
- RQ4: Can query throughput and index space be traded against each other in a useful way?
- RQ5: What trade-offs exist beyond throughput and index space?

The remainder of this paper considers these five areas.

2 BACKGROUND

2.1 Additivity, Efficiency, and Reproducibility

As already noted, additivity of effectiveness has been explored in some detail [1, 5, 25]. There has been less attention given to additivity in the area of efficiency, and the focus has instead been on *reproducibility* as an area of interest. For example, a sequence of SIGIR workshops has explored ways in which researchers might benefit by strategically building on each others' work, and proposals towards that goal have also emerged [22, 29]. In their study of selective search, Kim et al. [26] explicitly consider the interaction between topical sharding and dynamic pruning as a question of efficiency additivity, and find that they provide independent benefit.

CIKM '20, October 19–23, 2020, Virtual Event, Ireland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3412000

Recent work also demonstrated the additivity of static and dynamic pruning techniques [34].

The earlier work of Armstrong et al. [5] also raised the issue of baselines in experimentation, in comments intended as a guide to researchers working in effectiveness, but equally pertinent to efficiency-related activities. Commentary in regard to the use of competitive baseline systems has also been provided for neural ranking systems [61] and for recommender systems [14].

2.2 Scalable Query Processing

Modern search engines operate within strict latency constraints. However, effective ranking models are often computationally expensive, with hundreds of scoring features per document [32]. A common approach to balance these trade-offs is to utilize *ranking cascades* [56], where a simple ranker is used to quickly find a set of possibly relevant documents (*candidate generation*), which are then re-ranked via more expensive models. The focus in this paper is on first-phase ranking.

Bag-of-Words Similarity Scoring. Typical candidate generation models assume term independence, with documents scored by accumulating independent term-document contributions. Given an n-term query $Q = \{t_1, t_2, \ldots, t_n\}$, and a document d, the score of d with respect to Q is computed as a contribution sum over terms:

$$\mathcal{S}(Q, d) = \sum_{i=1}^{n} C(t_i, d)\,,$$

where $C(t, d)$ represents the score contribution to document d derived from term t. Document- and query- independent weights can also be incorporated, with many similarity models then covered, including the frequently-used BM25 model [45].

Efficient Index Traversal. To facilitate bag-of-words ranking an *inverted index* is used, with a *postings list* stored for each unique term t and containing a sequence of document identifiers and their term-frequency (*tf*) information [63]. Of the alternative storage options we assume that a *document-ordered* arrangement is in use.

Document-ordered indexes store postings in ascending document identifier order, facilitating Document-at-a-Time (DaaT) traversal and scoring [13]. A min-heap of the top-k documents is maintained, together with a threshold θ that tracks the lowest score amongst those k. Exhaustive RankedOR processing proceeds by computing a score for every candidate document as the query terms' postings lists are merged; checking that score against θ; and updating the heap (and hence θ) if necessary.

While simple, naïvely evaluating every document that contains one or more of the terms is expensive, and a range of *dynamic pruning* algorithms have been developed to reduce the number of documents scored. In particular, Broder et al. [8] observed that if per-term upper-bound contributions are pre-computed, they can be used to quickly eliminate from consideration any documents with no prospect of reaching a score of θ. To implement this idea, the largest $C(t, d)$ in each postings list is stored as U_t. During processing, partial sums of the U_t values guide the selection of documents which might enter the top-k. Petri et al. [42] explain this WAND mechanism in more detail, and provide pseudocode.

Ding and Suel [18] made further gains with their block-max indexing approach. Instead of storing only one global U_t bound per postings list, which can lead to generous overestimates of true scores, Ding and Suel suggest storing upper bounds on a per-block basis (denoted by $U_{t,b}$), with multiple bounds within each postings list. The resultant block-max WAND (BMW) algorithm operates in the same way as WAND, but with the more accurate block-wise upper-bound scores further accelerating query processing.

Most recently, Mallia et al. [35] showed that allowing the blocks to be *variable* in size generates even tighter upper-bound scores. The resulting VBMW algorithm represents the (current) state-of-the-art for fast top-k query processing [35, 37], particularly when k is small. We note the existence of other algorithms and index organizations such as both DaaT and TaaT MaxScore mechanisms [55] and Score-at-a-Time index traversal [13], but focus on the WAND-family of optimizations in this work. More details can be found in the recent survey of Tonellotto et al. [52].

Compression Codecs. The representation of the stored index is a second important facet of efficient system design. Integer compression techniques allow the index size to be substantially reduced, and, at the same time, allow query throughput to be increased. A wide range of such methods have been developed, each with differing blends of attributes [44, 58, 63].

Mallia et al. [37] examine eleven approaches in the context of DaaT processing, including comparing four representative codecs in detail. Following their lead we employ the same four codecs: *OptPForDelta* (OptPFD) [60]; *Partitioned Elias-Fano* (PEF) [41]; *Vectorized Binary Packing* (SIMD-BP128) [27]; and a derivative of *Variable Byte* known as Varint-G8IU [50]. Details of these methods can be found in the survey by Pibiri and Venturini [44].

3 EFFICIENCY INNOVATIONS

We now turn to four distinct optimizations which can be applied individually or in combinations with any of the four processing regimes (RankedOR, WAND, BMW, and VBMW) that were summarized in Section 2. Table 1 lists these optional "add-ons".

3.1 Factor P: Predicting the Heap Threshold

Dynamic pruning algorithms use the current heap threshold θ as a target value to identify documents for scoring. The higher that value, the fewer documents scored. In ordinary processing, θ is initialized to the lowest score that can be assigned to any document, denoted by ∇, typically either zero or negative infinity. During query processing, θ then monotonically increases towards its final value Θ_k, corresponding to the k th highest scoring document for the query Q.

A key observation then follows: initializing θ to some value closer to Θ_k (but not exceeding it) will likely result in fewer documents being scored, and hence faster query processing. This leads to methods for *predicting* the min-heap threshold to determining a suitable initialization $\theta = \hat{\Theta}_k$ in order to accelerate query processing. Figure 1 illustrates this idea.

Table 1: The four add-on optimizations that are explored.

Optimization	Key	Description
Prediction	P	At index time, the k th highest score for each postings list is computed and stored. At query time, the maximum of these values across the query terms is used to initialize the min-heap threshold.
Quantization	Q	At index time, the score of every posting is computed and quantized into a single b-bit integer value. At query time, scoring involves adding the corresponding quantized scores together.
Reordering	R	At index time, the document identifiers are re-ordered, resulting in better compression and term locality.
Stopping	S	At query time, stopwords are removed from the query if possible, allowing fewer postings to be processed.

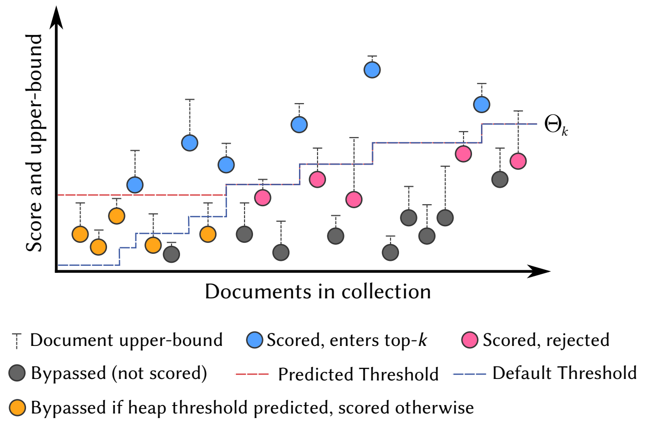


Figure 1: Threshold *prediction*, with $k = 3$ documents being identified. The blue line shows normal processing, where the threshold is initialized to a minimum value; the red line shows the effect of using an initial score threshold. Orange dots depict documents which would be scored when the default threshold is used, but would be skipped if the higher initial value was used. As long as the initial threshold is lower than the final value Θ_k, the answer set is guaranteed to be correct (adapted from Petri et al. [43]).

Prediction Mechanisms. A range of methods for estimating $\hat{\Theta}_k$ have been explored, including both *safe* approaches (which guarantee that $\hat{\Theta}_k \leq \Theta_k$), and *unsafe* approaches (where $\hat{\Theta}_k > \Theta_k$ might only be discovered *after* the query has been processed).

A cheap-yet-effective approach which is deterministic and safe was noted by Daoud et al. [15] and explored in detail by Kane and Tompa [24]. It involves storing the k th highest scores for a predetermined and fixed set such as $k \in \{10, 100, 1000\}$. Then, upon receiving a query Q, the threshold θ can be initialized as the maximum of the appropriate set of k th scores for the candidate terms, denoted as Q_k, exploiting the fact that there must be k documents in the collection with a score $\geq Q_k$ because of that term alone. In the case where the search-time k is not among the set of indexed k, the next largest should be used: for example, if 20 documents are to be retrieved and $k \in \{10, 100, 1000\}$ were pre-computed at index time, the largest of the $k = 100$ term thresholds should be used as Q_k. Other deterministic methods are possible, including those which require various forms of index tiering, stratification, and preliminary evaluation [9, 16, 20, 26, 51]. Other authors consider prediction in the context of caching [59].

Unsafe methods have also been explored [11, 38]. One approach that has been successful is to employ machine-learning to predict $\hat{\Theta}_k$ for each query. The downside of unsafe methods is that if $\hat{\Theta}_k > \Theta_k$, the query may not contain the correct top-k documents, and follow-up processing must be conducted if correctness is to be assured [43].

Configurations Measured. We examine prediction (factor "P") via the deterministic Q_k method, setting the initial heap threshold to the maximum of the k th highest "next greater than or equal to k" scores across the query terms. The absence of prediction corresponds a default "cold-start" setting, with θ initialized to ∇.

3.2 Factor Q: Index Quantization

The computation of document/query similarity scores, required to accurately rank documents, is another expensive aspect of query processing. The underlying models used to compute similarity scores often rely on floating point arithmetic, including divisions, and hence require many CPU cycles per document scored.

Anh et al. [4] note that this cost can be mitigated by computing all similarity contributions offline, and storing each such *impact score* in the inverted index, rather than the raw elements of the computation. Since similarity functions typically generate real-valued contributions, it is expensive to store them for each posting; instead, they are *quantized* into b-bit integers in the range $[1, 2^b - 1]$. Then, at query time, processing a top-k query involves summing the corresponding b-bit integers, with integer addition requiring far fewer clock cycles than floating point multiplication and division.

Various quantization methods have been explored, including those which provide higher fidelity in the lower or higher score ranges. However, the best approximation of the original score distribution can be computed with *uniform quantization* [4, 12], which approximates the score distribution of the impacts across b buckets, resulting in very similar effectiveness to the original *tf* index when a sufficiently large value of b is used [12].

Configurations Measured. We examine quantization (factor "Q") via the *uniform* quantization approach of Anh et al. [4], with the quantized impacts being stored in lieu of the *tf* values. At query time, scoring involves summing these quantized values. The absence of quantization corresponds to a default *tf* index, with score contributions computed during query processing.

3.3 Factor R: Document Reordering

Postings lists store document numbers in increasing order, with most compression techniques exploiting the fact that in this arrangement the differences between successive identifiers (the d-gaps) are, on average, very small. Further savings arise if each term's occurrences are non-random through the collection – the ideal situation being when they are clustered into widely separated dense regions, with runs of small d-gaps. Document *reordering*, also known as document identifier reassignment, is the process of reassigning the document identifiers, so that when the index is formed, documents that are "like" each other (in that they have many terms in common) are placed near to each other [49].

In addition to saving index space, document reordering can also improve query throughput [19, 37, 60], with newer approaches jointly optimizing for both space and time [57]. While improvements to query processing efficiency from reordering are not well studied, the basis for gains is clear: in conjunctive processing, document reordering produces longer *skips* and fewer decompressions [60]; and the more compact a representation, the more likely it will be cache friendly.

Document Reordering Approaches. Early work focused on using the Traveling Salesman Problem (TSP) as a heuristic to find tours across a graph of document-to-document similarity values, thereby clustering similar documents together [7, 19, 47]. Silvestri [48] explores a series of straightforward heuristics, showing that ordering a collection by the underlying document URLs provides improved compression rates compared to random ordering. Sorting by URL works well because documents from the same website usually share many common terms, increasing the clustering of the underlying postings lists.

Recently, Dhulipala et al. [17] describe a method for directly optimizing the expected storage cost of the encoded index. They construct a bipartite graph between the terms and the documents of the index, and then split the graph into two halves. After each split, an estimate is made of the compression gain generated by swapping each document from one partition to the other. When positive gains are observed, the algorithm swaps documents between the two partitions and iterates, stopping after a fixed number of cycles. The process then recurses on the two partitions, stopping at a predefined depth. This elegant approach provides the current state-of-the-art for both graph and index reordering [17, 33].

Configurations Measured. We examine reordering (factor "R") via Dhulipala et al.'s BP algorithm [17, 33], applying it as a pre-indexing step. The absence of reordering corresponds to an index built with randomly shuffled document identifiers. We note that while other reference orderings are possible [37], randomizing the ordering prevents any artifacts that might arise in the experimentation as a result of (the often unknown) document collection strategy.

3.4 Factor S: Stopping

One of the simplest possible optimizations, stopping involves removing commonly observed terms from either documents (during indexing) or queries (as they are parsed prior to being evaluated). Applying stopping during indexing allows for a smaller index, but is inherently lossy; queries composed of stop words such as *"to be or not to be"* cannot be answered on a stopped index. As such, a more flexible approach is to apply the stop list to incoming queries. Stopping is not safe, and the result of executing a stopped query may differ from that obtained for the original query.

Table 2: Details of test collections.

Corpus	Documents	Unique Terms	Postings
Gov2	25,205,179	39,180,841	5,880,709,591
ClueWeb12B	52,343,021	165,309,502	15,319,871,265

Configurations Measured. We examine query-time stopping (factor "S") via the Indri stop list, which contains 418 common English terms.[1] The absence of stopping simply means the queries are unchanged. Queries which were empty after stopping were discarded from both query logs to avoid biasing the experimental analysis.

3.5 Modes, Enhancements, and Additivity

The key theme explored in this work is determining which combinations of these techniques – the four query processing modes described in Section 2, and the four possible enhancements to them that have been described in this section – are complementary, and hence additive. The alternative is for different "improvements" to be exploiting the same underling inefficiencies in different ways, and hence be substitutes for each other and non-additive when combined. A further dimension is introduced when the question of compression codec is added, since decompression costs while processing the index also involve a trade-off between time and space. The next section describes detailed experiments over a large number of implementation combinations to explore these issues.

4 EXPERIMENTS

4.1 Experimental Setup

Hardware and Software. All experiments were performed entirely in-memory on a Linux machine with two 3.50 GHz Intel Xeon Gold 6144 CPUs and 512 GiB of RAM. Timings are reported as the average of three independent runs. The document collections were indexed using Indri 5.11 with Krovetz stemming, and then converted to a format readable by PISA [36], which was used to conduct all subsequent indexing and query processing experiments.[2]

Datasets and Queries. Two public collections are used:

- Gov2,[3] around 25 million .gov sites crawled during 2004.
- ClueWeb12B,[4] the "B" portion of the 2012 ClueWeb crawl, containing around 52 million web documents.

Table 2 reports statistics after indexing by Indri.

The publicly available *Million Query Track* (MQT) queries from the 2007, 2008, and 2009 TREC Million Query Track [2, 3, 10] were used, with 5,000 queries sampled from the 60,000 candidates.[5] The sampling process selected 1,000 queries of each length from 1 to 4, plus a further 1,000 queries containing 5 or more terms.

[1] http://www.lemurproject.org/stopwords/stoplist.dft
[2] https://github.com/pisa-engine/pisa/ → Commit 16a9c33, 27th of March, 2020.
[3] http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm
[4] http://lemurproject.org/clueweb12/
[5] https://trec.nist.gov/data/million.query.html

Table 3: Space (GiB) for the postings lists of the ClueWeb12B index using two orderings, four compression codecs, and either *tf* form (top) or quantized form with $b = 9$ (bottom).

Ordering	Term Frequency Index			
	OptPFD	PEF	SIMD-BP128	Varint-G8IU
Random	20.1	18.5	25.2	37.0
BP	13.9	13.0	20.0	35.0
Ordering	**Quantized Index**			
	OptPFD	PEF	SIMD-BP128	Varint-G8IU
Random	27.0	27.4	28.1	37.6
BP	21.4	22.4	25.1	35.5

Fixed Parameters and Settings. For document ranking, the BM25 [45] model was used, as it is a cheap-yet-effective approach that permits dynamic pruning [42]. Recent work has explored subtle variations of BM25 [23, 54]; the PISA overview [36] provides a precise definition of the computation employed. Parameter values of $k_1 = 0.4$ and $b = 0.9$ were used [53].

Where block-based compression was used, postings lists were encoded as fixed-length blocks of 128 integers, with blocks of less than 128 values encoded with Binary Interpolative coding [39]. The block-max score structures used by BMW and VBMW employ, respectively, fixed and variable length blocks, containing an average of 40 postings each [35].

Where quantized indexes were employed, term-document contributions were mapped into 256 buckets ($b = 8$) for Gov2, and into 512 buckets ($b = 9$) for ClueWeb12B. These choices provide sufficient resolution to avoid effectiveness loss [12, 13].

4.2 Space Consumption

Space consumption was measured for two index orderings (Random and BP), four compression codecs, and for both plain *tf* values and quantized impacts. In the interests of brevity only the ClueWeb12B results are shown, with the same trends observed for Gov2.

Postings Lists. Table 3 reports the size, in GiB, of the postings lists for all sixteen different ClueWeb12B indexes. Two trends emerge: first, applying the BP reordering improves the space occupancy of the inverted indexes, irrespective of the compression codec employed; and second, the quantized indexes are larger than the *tf*-based ones, as expected. The OptPFD and PEF codecs provide the best compression, with PEF slightly smaller over *tf* indexes, and the converse true for quantized indexes.

The Varint-G8IU codec is less sensitive to both index ordering and to quantization. This is because Varint-G8IU is a byte-aligned encoder, and thus only realizes gains when the *d*-gaps in the document identifier lists are reduced [37].

Indexes based on a URL-sorted arrangement were also measured, and were between 500 MiB and 5 GiB larger than the equivalent BP index, depending on the codec used. An interesting aspect of these results is that they both reproduce the findings of Mallia et al. [37] and also extend the comparison to include quantized indexes, previously only examined in isolation [12, 13].

Table 4: Space overhead (GiB, ClueWeb12B) to support dynamic pruning, with the U_t entry the cost of storing upper-bound scores (32 bits per postings list), and the Q_k entry the total cost of storing the k th highest scores in each sufficiently long postings list, with $k \in \{10, 100, 1000\}$. The other entries are the cost of storing block-level upper-bound scores, with 64 bits required per block (32 bits for an upper-bound score, and 32 bits for a block-bounding document number). The bounds in a quantized index have very similar costs.

U_t	Q_k	Mean Block Size						
		16	32	40	64	96	128	256
0.62	0.05	8.21	4.71	4.01	2.96	2.38	2.09	1.66

Upper-Bound Scores and Thresholds. Upper-bound scores must be kept on a per-list (WAND) or per-block (BMW, VBMW) basis, where blocks may or may not be of fixed size. Table 4 summarizes the cost of these bounds for ClueWeb12B. The list-wise upper-bounds, denoted U_t, cost only 32 bits per postings list, but are stored for every term. On the other hand, the block-wise upper-bounds must also be accompanied by an "endpoint" value describing the documents spanned by the block, and cost 64 bits per block. The total cost of storing block-wise upper-bounds then depends on the total number of blocks stored, and hence on their average size. Larger blocks result in a smaller memory footprint, but also generate smaller throughput improvements [35].

In the experimental configurations, block-wise upper-bounds added an overhead of between 3% to 67% for Gov2 (not shown), and of between 4% to 63% for ClueWeb12B (Table 4), with the exact amount determined by both the compression codec and block configuration; for example, the worst-case overhead for ClueWeb12B arises when using a mean block size of 16 and a BP ordered *tf* index with PEF encoding. There is a clear trade-off between the compression codec used and the space available for metadata such as block-wise upper-bound scores; and all other things being equal, better compression may permit smaller block sizes.

Table 4 also lists the cost of adding up to three "k th highest scores" to each postings list. Large values of k require less storage overhead, as fewer postings lists are likely to have a length $\geq k$ elements [24]. For example, across its 39,180,841 unique terms, the Gov2 index maintains 3,336,298 scores for $k = 10$; another 818,295 for $k = 100$; and just 117,532 scores for $k = 1000$. In total, keeping k th-largest scores for $k \in \{10, 100, 1000\}$, adds an overhead of between 0.1% to 0.5% for Gov2 (not shown), and between 0.1% to 0.4% for ClueWeb12B, with the percentages taken relative to the smallest and largest compressed indexes.

4.3 Query Processing Latency

The next experiments compare query latency, considering each of the four query processing mechanisms (Section 2), taken with and without each of the four enhancements (Section 3, considering all combinations), and coupled with each of the four compression codecs. That is, a total of $4 \times 2^4 \times 4 = 256$ systems are explored. For brevity, in this section results are only reported for the SIMD-BP128 codec, as it was found to be the fastest (and also by Mallia et al. [37]). The other codecs are explored further in Section 5.2.

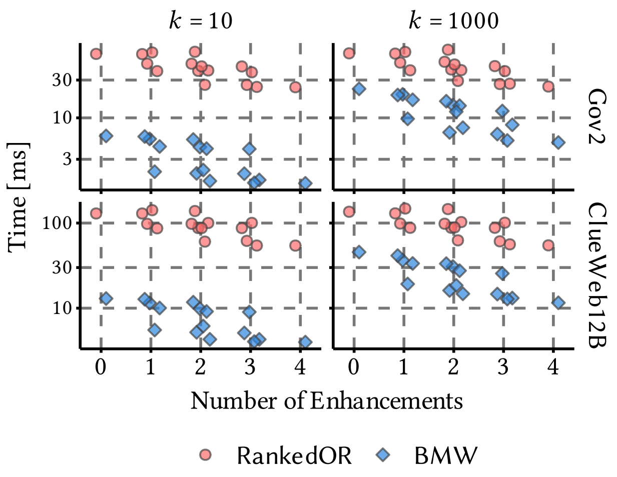


Figure 2: Median processing latency (millisec) as a function of the number of enhancements, with BMW a representative dynamic pruning technique. Similar trends arise for WAND and VBMW.

Overview. As a first "high altitude" view, Figure 2 presents median query latency for 32 of the systems, those employing RankedOR and BMW processing, measured using two different ranking depths k, and two different document collections. The slightly unorthodox horizontal scales in each pane count the number of enhancements applied, drawn from the P, Q, R, S palette (Table 1), starting with none, and ending with all of them. Hence there are 1, 4, 6, 4, and 1 points plotted against the five axis values for each system.

Within each pane, the downward trends show that adding enhancements to a given algorithm does indeed tend to result in improved throughput, with more enhancements leading to larger gains. Some exceptions are evident for the RankedOR algorithm, which does not employ dynamic pruning when traversing the index. That is, certain combinations of enhancements do not benefit RankedOR, and can even lead to slight throughput decreases. We delve deeper into these overall outcomes shortly. Irrespective of the processing method, the fastest configuration was always the one that employed all four enhancements, indicating that this set of optimizations does indeed behave additively.

Table 5 provides more detail, reporting median and tail latencies for each distinct algorithm/enhancement combination when applied to ClueWeb12B. Substantial improvements can be achieved by employing the method-agnostic enhancements. For example, the exhaustive RankedOR algorithm obtains a 2.5× speedup (median) and a 4.0× speedup at the tail. The dynamic pruning algorithms show even greater gains, of between 3.6× to 4.7× at the median, and 3.1× to 4.6× at the tail.

Working down each column in the table is also informative. For example, the single most effective optimization for RankedOR is stopping, followed by quantization. On the other hand (and perhaps unsurprisingly), the dynamic pruning approaches derive the greatest relative benefit from index reordering. The overall benefit accruing from each enhancement is considered further in Section 5.1.

Comparing Algorithms. While most configurations follow the expected pattern of RankedOR being less efficient than WAND, WAND being less efficient than BMW, and BMW being less efficient than VBMW, there are also some interesting exceptions. Firstly, with the default configuration, we note that WAND is only slightly slower than both BMW and VBMW, and actually outperforms them when considering tail latency. Similar outcomes are present for all single optimizations with the exception being reordering, which greatly improves the block-based algorithms. Similar trends can be observed in the two-optimization group, where WAND outperforms its more advanced counterparts in two of seven cases for median latency, and in four of seven cases for tail latency. Again, WAND outperforms BMW and VBMW on median latency for one and two configurations (of four) when considering three-optimizations, and is only narrowly beaten when all four optimizations are considered.

Further analysis, across both algorithms and enhancements, reveals that WAND with all optimizations enabled outperforms every instance of BMW which used two or fewer optimizations, and also outperforms two of the four BMW configurations which employ three optimizations. Furthermore, WAND with all optimizations outperforms most instances of VBMW using two or fewer optimizations, and even outperforms VBMW with P+Q+S. This behavior can be observed when comparing the tail latency of RankedOR with full optimizations to the default and some single-optimization instances of BMW and VBMW.

These relative outcomes raise key questions for practitioners, who must balance the effort spent adding enhancements to a current implementation against the likely benefit of commencing a completely new implementation of a nominally "better" underlying processing approach. Section 6 discusses this issue further.

5 UNDERSTANDING THE INTERACTIONS

We now turn our focus to the way that optimizations and algorithms interact, and in particular, to whether there is a risk of overall throughput gains masking increased volatility in the cost of individual queries. To facilitate that analysis, for each of the four enhancements (Section 3) all possible system pairs are formed, first without that enhancement, and then with it. Let S_B denote a baseline system that does not include a particular optimization $O \in \{P, Q, R, S\}$, and S_I denote S_B with the further inclusion of O. For example, S_B might be WAND+P+Q, in which case there are two possible S_I systems: WAND with stopping added as well, to yield the WAND+P+Q+S system; and the other being the WAND+P+Q+R combination. For each optimization $O \in \{P, Q, R, S\}$ there are thus eight possible S_B, S_I pairs for each of the four underlying processing regimes. For each such pair, the *execution deltas* between the S_B and S_I systems are measured on a per-query basis, with positive deltas representing improvements resulting from O being added, and negative deltas representing degradations.

5.1 Latency and Profiling

Figure 3 shows execution deltas for three different evaluation cost metrics: median query latency; the number of postings process; and the number of postings blocks decoded over ClueWeb12B.

Predicting Thresholds. The leftmost group in each pane summarizes improvements accruing from score prediction, factor P. It

Table 5: Query processing latency (millisec) for four underlying methods, grouped into five categories based on the number of optimizations employed, for $k = 1000$ and ClueWeb12B. Speedups, shown in brackets, are computed with respect to the *Default* systems in the first row. The most efficient run in each category is highlighted in blue, and the most efficient run in each column is highlighted in red.

Factors	Median Latency (P_{50})				Tail Latency (P_{95})			
	RankedOR	WAND	BMW	VBMW	RankedOR	WAND	BMW	VBMW
Default	135.7	49.1	45.6	45.6	1425.6	263.3	385.0	366.2
P	130.0 (×1.0)	45.5 (×1.1)	41.4 (×1.1)	39.1 (×1.2)	1395.4 (×1.0)	262.1 (×1.0)	380.0 (×1.0)	363.4 (×1.0)
Q	98.7 (×1.4)	32.0 (×1.5)	36.5 (×1.2)	33.3 (×1.4)	1132.4 (×1.3)	224.4 (×1.2)	317.6 (×1.2)	307.8 (×1.2)
R	148.0 (×0.9)	26.6 (×1.8)	19.2 (×2.4)	17.9 (×2.6)	1367.0 (×1.0)	146.0 (×1.8)	158.3 (×2.4)	133.5 (×2.7)
S	87.9 (×1.5)	37.0 (×1.3)	33.5 (×1.4)	34.8 (×1.3)	501.8 (×2.8)	191.5 (×1.4)	231.6 (×1.7)	249.6 (×1.5)
P+*Q*	98.1 (×1.4)	29.4 (×1.7)	33.3 (×1.4)	30.3 (×1.5)	1153.3 (×1.2)	222.6 (×1.2)	314.1 (×1.2)	308.1 (×1.2)
P+*R*	145.9 (×0.9)	20.7 (×2.4)	16.2 (×2.8)	13.3 (×3.4)	1318.6 (×1.1)	138.5 (×1.9)	153.6 (×2.5)	126.4 (×2.9)
P+*S*	88.0 (×1.5)	34.1 (×1.4)	30.8 (×1.5)	31.2 (×1.5)	503.8 (×1.0)	190.7 (×1.4)	230.0 (×1.7)	246.5 (×1.5)
Q+*R*	88.9 (×1.5)	20.7 (×2.4)	18.5 (×2.5)	16.0 (×2.8)	1054.8 (×1.4)	119.7 (×2.2)	144.9 (×2.7)	123.3 (×3.0)
Q+*S*	62.6 (×2.2)	24.9 (×2.0)	27.6 (×1.7)	26.4 (×1.7)	415.3 (×3.4)	161.2 (×1.6)	207.5 (×1.9)	207.3 (×1.8)
R+*S*	102.7 (×1.3)	20.3 (×2.4)	14.8 (×3.1)	13.5 (×3.4)	478.6 (×3.0)	106.6 (×2.5)	95.5 (×4.0)	88.9 (×4.1)
P+*Q*+*R*	88.0 (×1.5)	15.8 (×3.1)	14.6 (×3.1)	11.8 (×3.9)	1052.7 (×1.4)	113.3 (×2.3)	135.9 (×2.8)	116.3 (×3.1)
P+*Q*+*S*	61.0 (×2.2)	23.5 (×2.1)	25.6 (×1.8)	23.9 (×1.9)	412.5 (×3.5)	160.0 (×1.6)	201.9 (×1.9)	205.9 (×1.8)
P+*R*+*S*	100.9 (×1.3)	17.1 (×2.9)	12.8 (×3.6)	10.9 (×4.2)	479.0 (×3.0)	103.8 (×2.5)	93.7 (×4.1)	86.4 (×4.2)
Q+*R*+*S*	56.2 (×2.4)	15.5 (×3.2)	13.1 (×3.5)	11.8 (×3.9)	358.9 (×4.0)	86.0 (×3.1)	88.0 (×4.4)	82.9 (×4.4)
P+*Q*+*R*+*S*	54.7 (×2.5)	13.8 (×3.6)	11.6 (×3.9)	9.7 (×4.7)	356.1 (×4.0)	84.4 (×3.1)	86.5 (×4.5)	80.3 (×4.6)

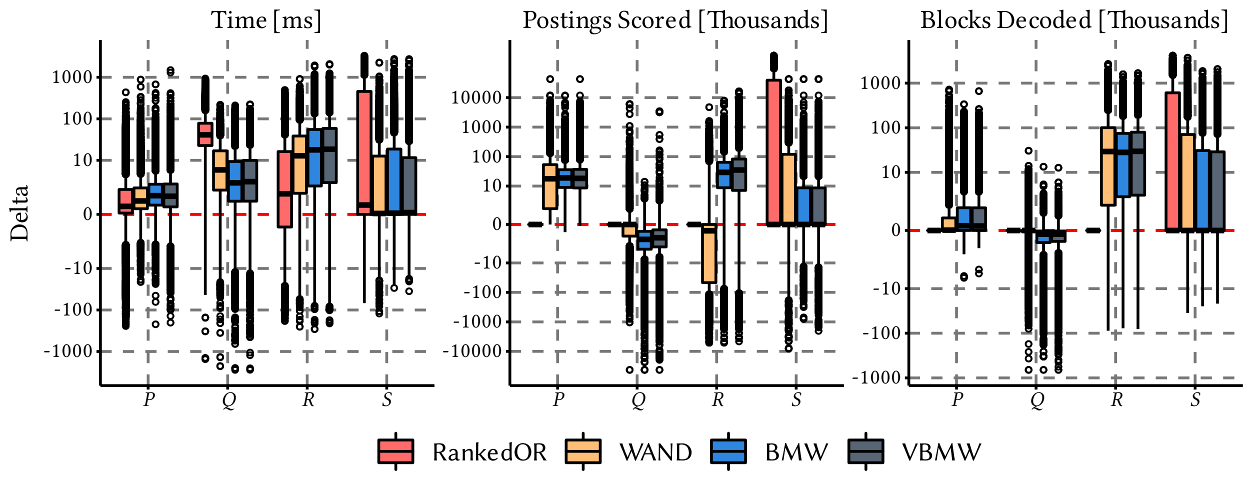


Figure 3: Execution deltas over all queries for each S_B, S_I pair, with each box/whisker element representing $8 \times 5{,}000$ individual deltas. Three different facets of query execution (ClueWeb12B, $k = 1{,}000$) are shown: median latency (left); postings evaluated (middle); and blocks decoded (right). Note that the y-axis has been transformed using an inverse hyperbolic arcsine function, which exhibits log-like properties but is defined for values of $y \leq 0$.

results in a small increase in query throughput when aggregated across all S_B, S_I pairs, with a slightly larger effect as the algorithm becomes more complex. This trend arises because as the algorithms become better at estimating true upper-bound scores, more pruning takes place, and fewer blocks are decoded. However the only benefit that RankedOR receives is fewer heap operations.

Quantization. The second group in each pane, quantization, also has a positive impact on all algorithms, but becomes less effective for more the more complex algorithms, with RankedOR benefiting the most. This trend is accounted for by the volumes of postings being scored (center pane); methods which score the most postings benefit the most from quantization.

Quantization induces *more* scoring and decoding in the dynamic pruning algorithms on average, with the time saved in each operation still allowing net reductions in query latency. Quantization can also lead to highly negative execution deltas, with some queries taking one second *longer* than if they were processed without quantization. Detailed investigation revealed this to be a consequence of large numbers of *score ties* in the rankings, compounded by the min-heap implementation in the code-base used for these experiments admitting documents based on the liberal test $\mathcal{S} \geq \theta$. The likelihood of documents having tied scores is much higher when quantization is being used, especially with single-term queries. While there is no right answer regarding specific tie-breaking functionality [28], changing to a strict $\mathcal{S} > \theta$ heap entry requirement would be one simple way to accelerate retrieval [62].

Index Reordering. This third factor has a strong positive effect on the dynamic pruning algorithms, especially for the block-based methods. Reordering also improves RankedOR speed – there is no impact on the volume of postings or blocks examined, but the reordered indexes are smaller and hence faster to decode.

Index reordering induces more scoring for WAND, but at the same time yields a large reduction in the number of blocks decoded, suggesting that reordering improves the locality of documents – when one document must be scored in a (compression) block, there is a high chance of scoring others in that block.

To quantify the extent of the clustering in the postings lists, a *postings accuracy* is computed for every posting in each index, defined as that posting's impact as a fraction of the associated block-max impact, accuracy$(t, d) = C(t, d)/U_{t,b}$. Figure 4 plots these accuracies as cumulative distributions across the Random and BP orderings of Gov2 and ClueWeb12B, for both Fixed and Variable block indexes. The notable and consistent relationship between the blue lines (random ordering) and red lines (reordered) illustrates why reordering has such a pronounced positive effect on the block-based dynamic pruning algorithms.

Stopping. While stopping leads to savings on average, it seems counterintuitive that stopping queries could lead to *increases* in postings scored and blocks decoded (Figure 3). In fact, in some cases stopping *does* make query processing slower, because it can remove highly discriminatory term combinations. Particularly egregious cases arise when queries are stopped down to single terms. For example, *"signs of ms"* $\rightarrow$ *"signs"*, which removes much of the benefits of dynamic pruning, especially for WAND which then only has a single upper-bound available to it. This particular example leads to WAND evaluating around 8.7 million extra documents, and adds up to 120 ms to the processing time. Practitioners might thus wish to consider special optimizations for single-term queries.

In terms of effectiveness, the risk of stopping can be quantified by computing the *overlap* between a stopped ranking and the corresponding unstopped ranking, noting that one important use-case of these methods is in the first phase of a multi-stage ranker, where it is the set of document that is required, and not their exact order. At $k = 1{,}000$ the mean (over queries, for ClueWeb12B) overlap between non-stopped processing and stopped processing is 96.7%, with 1,564 of the queries affected. The risk of effectiveness loss resulting from quantization can be similarly computed: quantization alone also results in an overlap of 96.7%, and quantization plus

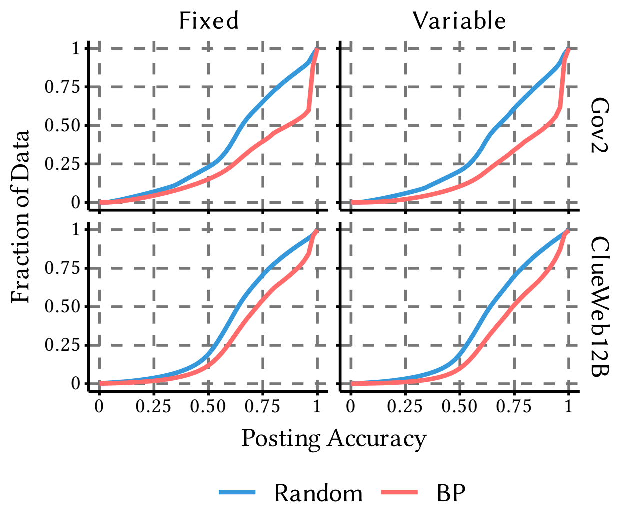
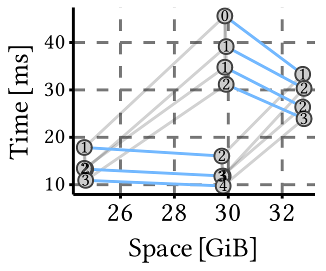
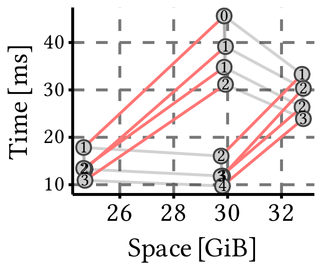


Figure 4: Posting accuracy for Random and BP index orderings, Fixed and Variable blocks, and Gov2 and ClueWeb12B. Reordering the collection improves accuracy, since similar documents are clustered near each other in the postings lists.



(a) Quantization



(b) Reordering

Figure 5: Median latency and space differences due to quantization (left), and reordering (right), for VBMW with $k = 1{,}000$ across ClueWeb12B. Each point is marked by a number between 0 and 4 to indicate the number of enhancements in operation, with 0 denoting the *Default* arrangement, and along each edge, the corresponding count increases by one. The short grey vertical edges indicate the orthogonal threshold prediction and stopping options, which add negligible space but reduce the latency.

stopping achieves 94.1%. Failure cases tend to involve aggressive stopping, for example *"what to do for burns"* $\rightarrow$ *"burns"*, or short queries with flat term distributions, for example *"news"*.

5.2 Time Versus Space

Trade-Offs. Reduced processing latency is often achieved at the cost of increased index space, meaning that the trade-off between them is also of critical interest. Figure 5 shows the interaction between time and space for ClueWeb12B and VBMW processing, in the left pane highlighting the eight S_B, S_I pairs that correspond to quantization, and in the right pane highlighting the eight S_B, S_I pairs that correspond to reordering. The efficiency gains due to

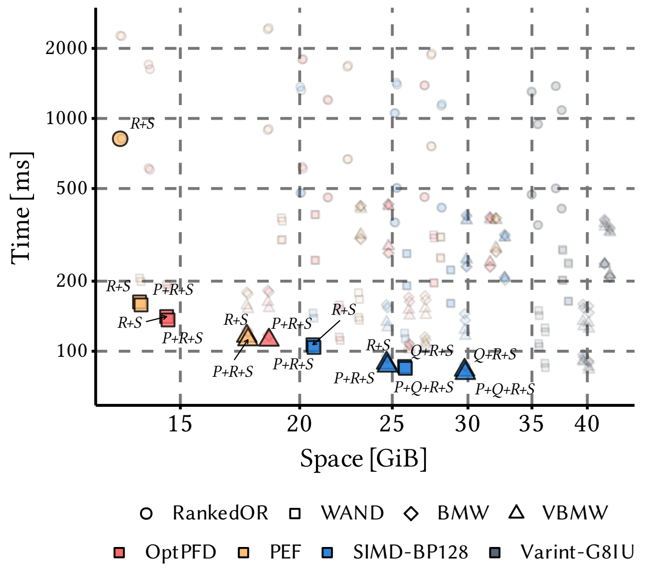


Figure 6: Tail latency (95th percentile, millisec, for $k = 1{,}000$) versus total index space (GiB) for 256 combinations of processing mode, enhancements, and compression codec, for ClueWeb12B. The Pareto frontier is defined by the larger points, each labeled with a configuration description. Note the log scale on both axes.

quantization, which are in the range of 2 ms per query (*P+R+S* $\rightarrow$ *P+Q+R+S*) to 17 ms (*Default* $\rightarrow$ *Q*) require a non-trivial amount of extra space. However, index reordering has the opposite effect, resulting in both faster processing *and* smaller indexes.

The Big Picture. Figure 6 compares all 256 system configurations (four processing modes, 2^4 combinations of four enhancements, and four compression codecs), plotting tail latency in milliseconds as a function of index space, for ClueWeb12B and with $k = 1{,}000$. The labeled markers represent the Pareto frontier, with the unlabeled grayed-out markers indicating other systems shadowed by the frontier; and with marker shapes indicating processing modes, and marker colors indicating compression codecs.

At the "fast" end of the frontier, the SIMD-BP128 codec is dominant, whereas at the "compact" end, PEF and OptPFD are the best choices; with Varint-G8IU not appearing at all. Likewise, no instance of BMW appears on the frontier, as it is always edged out by VBMW, which uses the same amount of space and is slightly faster. However, WAND occurs several times on the frontier – it performs well when tail latency is important (Table 5). In terms of the enhancements, the frontier is dominated by the *R* (which is safe) and *S* (which is not safe) options, indicating that they provide the best bang-for-buck with respect to time and space trade-offs for dynamic pruning algorithms. If speed is the primary objective, then *Q* (also not safe) and *P* (safe) can be added as well.

6 DISCUSSION AND CONCLUSION

Flexibility Considerations. Dynamic pruning algorithms are typically tightly coupled with a particular similarity computation, since the list- and block-bound information must be computed in advance. If a different ranking calculation is required for some reason, the index can still be used in RankedOR mode, simply ignoring the bound information. Another option is to compute and store multiple list bounds (to support WAND processing using a defined suite of similarity computations) and one set of block bounds (the primary similarity option, used for BMW or VBMW processing). A further compromise is to store the maximal *tf* value for each postings list, allowing (less precise) bounds to be computed on-the-fly [30, 31]. For example, flexible bounds are used in Lucene [21].

Similar heuristics could be extended to score threshold prediction, which also relies on pre-computing score impacts offline. Index quantization reduces flexibility; and while recent work has shown that quantization can still be used with append-only collections [40], any change to the ranking model requires totally re-scoring and quantizing all postings. On the other hand, index reordering is a one-time-only operation that is undertaken prior to indexing; hence, given the gains that are available, it should be the first priority for any practitioner. Stopping also provides efficiency gains without impacting flexibility, but needs to be applied judiciously.

Implementation Complexity. Programming effort is another dimension that has practical significance, and there are multiple instances in our results of optimization to "simpler" processing modes being viable alternatives to the more complex processing modes. Hence, there may be pragmatic considerations that suggest that RankedOR or WAND (relatively straightforward to implement) should be preferred to BMW/VBMW (complex to implement) unless there is a clear efficiency or resource bottleneck observed. Attention must also be paid to the type of optimizations that are available. For example, it may be wiser to spend some available space by shifting from the PEF or OptPFD codecs to the SIMD-BP128 codec than it is to apply the same (or more) space to quantization.

Conclusion. We posed five research questions in Section 1, with one over-riding concern, expressed in the paper's title: whether efficiency improvements are additive.

As a result of this project, we are now able to provide answers. Having combined a palette of four processing modes, four orthogonal enhancements, and four compression codecs to create a smorgasbord of 256 systems, and then having run them over 5,000 queries and two large document collections, we have arrived at a set of qualified "yes" answers – that the efficiency improvements explored are indeed broadly additive; that there is little volatility risk involved in adding enhancements to existing systems; that query latency and index space can be traded against each other in interesting ways; and that there are other subjective trade-offs that should also be taken into consideration when designing an implementation.

Acknowledgment. This work was supported by the Australian Research Council Discovery Project DP200103136.

REFERENCES

[1] M. Akcay, I. S. Altingovde, C. Macdonald, and I. Ounis. On the additivity and weak baselines for search result diversification research. In *Proc. ICTIR*, pages 109–116, 2017.

[2] J. Allan, B. Carterette, J. A. Aslam, V. Pavlu, B. Dachev, and E. Kanoulas. Million query track 2007 overview. In *Proc. TREC*, 2007.

[3] J. Allan, J. A. Aslam, B. Carterette, V. Pavlu, and E. Kanoulas. Million query track 2008 overview. In *Proc. TREC*, 2008.

[4] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. SIGIR*, pages 35–42, 2001.

[5] T. G. Armstrong, A. Moffat, W. Webber, and J. Zobel. Improvements that don't add up: Ad-hoc retrieval results since 1998. In *Proc. CIKM*, pages 601–610, 2009.

[6] X. Bai, I. Arapakis, B. B. Cambazoglu, and A. Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Trans. Inf. Sys.*, 36(2):1–42, 2017.

[7] R. Blanco and A. Barreiro. TSP and cluster-based solutions to the reassignment of document identifiers. *Inf. Retr.*, 9(4):499–517, 2006.

[8] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, pages 426–434, 2003.

[9] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *Proc. SIGIR*, pages 30–38, 1995.

[10] B. Carterette, V. Pavlu, H. Fang, and E. Kanoulas. Million query track 2009 overview. In *Proc. TREC*, 2009.

[11] C. L. A. Clarke, J. S. Culpepper, and A. Moffat. Assessing efficiency-effectiveness tradeoffs in multi-stage retrieval systems without using relevance judgments. *Inf. Retr.*, 19(4):351–377, 2016.

[12] M. Crane, A. Trotman, and R. O'Keefe. Maintaining discriminatory power in quantized indexes. In *Proc. CIKM*, pages 1221–1224, 2013.

[13] M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman. A comparison of Document-at-a-Time and Score-at-a-Time query evaluation. In *Proc. WSDM*, pages 201–210, 2017.

[14] M. F. Dacrema, P. Cremonesi, and D. Jannach. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In *Proc. RecSys*, pages 101–109, 2019.

[15] C. M. Daoud, E. S. de Moura, A. Carvalho, A. S. da Silva, D. Fernandes, and C. Rossi. Fast top-k preserving query processing using two-tier indexes. *Inf. Proc. & Man.*, 52(5):855–872, 2016.

[16] C. M. Daoud, E. S. de Moura, D. Fernandes, A. S. da Silva, C. Rossi, and A. Carvalho. Waves: A fast multi-tier top-k query processing algorithm. *Inf. Retr.*, 20(3):292–316, 2017.

[17] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proc. KDD*, pages 1535–1544, 2016.

[18] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. SIGIR*, pages 993–1002, 2011.

[19] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *Proc. WWW*, pages 311–320, 2010.

[20] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proc. VLDB*, 4(12):1213–1224, 2011.

[21] A. Grand, R. Muir, J. Ferenczi, and J. Lin. From MaxScore to Block-Max Wand: The story of how Lucene significantly improved query evaluation performance. In *Proc. ECIR*, pages 20–27, 2020.

[22] S. Hofstätter and A. Hanbury. Let's measure run time! Extending the IR replicability infrastructure to include performance aspects. In *Proc. OSIRRC at SIGIR 2019*, pages 12–16, 2019.

[23] C. Kamphuis, A. P. de Vries, L. Boytsov, and J. Lin. Which BM25 do you mean? A large-scale reproducibility study of scoring variants. In *Proc. ECIR*, pages 28–34, 2020.

[24] A. Kane and F. W. Tompa. Split-lists and initial thresholds for WAND-based search. In *Proc. SIGIR*, pages 877–880, 2018.

[25] S. Kharazmi, F. Scholer, D. Vallet, and M. Sanderson. Examining additivity and weak baselines. *ACM Trans. Inf. Sys.*, 34(4), 2016.

[26] Y. Kim, J. Callan, J. S. Culpepper, and A. Moffat. Does selective search benefit from WAND optimization? In *Proc. ECIR*, pages 145–158, 2016.

[27] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.*, 41(1):1–29, 2015.

[28] J. Lin and P. Yang. The impact of score ties on repeatability in document ranking. In *Proc. SIGIR*, pages 1125–1128, 2019.

[29] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. Toward reproducible baselines: The open-source IR reproducibility challenge. In *Proc. ECIR*, pages 408–420, 2016.

[30] C. Macdonald and N. Tonellotto. Upper bound approximation for BlockMaxWand. In *Proc. ICTIR*, pages 273–276, 2017.

[31] C. Macdonald, I. Ounis, and N. Tonellotto. Upper-bound approximations for dynamic pruning. *ACM Trans. Inf. Sys.*, 29(4):17.1–17.28, 2011.

[32] C. Macdonald, R. L. T. Santos, I. Ounis, and B. He. About learning models with multiple query-dependent features. *ACM Trans. Inf. Sys.*, 31(3):11.1–11.39, 2013.

[33] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel. Compressing inverted indexes with recursive graph bisection: A reproducibility study. In *Proc. ECIR*, pages 339–352, 2019.

[34] J. Mackenzie, Z. Dai, L. Gallagher, and J. Callan. Efficiency implications of term weighting for passage retrieval. In *Proc. SIGIR*, pages 1821–1824, 2020.

[35] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. Faster BlockMax WAND with variable-sized blocks. In *Proc. SIGIR*, pages 625–634, 2017.

[36] A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel. PISA: Performant indexes and search for academia. In *Proc. OSIRRC at SIGIR 2019*, pages 50–56, 2019.

[37] A. Mallia, M. Siedlaczek, and T. Suel. An experimental study of index compression and DAAT query processing methods. In *Proc. ECIR*, pages 353–368, 2019.

[38] A. Mallia, M. Siedlaczek, M. Sun, and T. Suel. A comparison of top-k threshold estimation techniques for disjunctive query processing. In *Proc. CIKM*, 2020. To appear.

[39] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[40] S. Mohammed, M. Crane, and J. Lin. Quantization in append-only collections. In *Proc. ICTIR*, pages 265–268, 2017.

[41] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. SIGIR*, pages 273–282, 2014.

[42] M. Petri, J. S. Culpepper, and A. Moffat. Exploring the magic of WAND. In *Proc. Aust. Doc. Comp. Symp.*, pages 58–65, 2013.

[43] M. Petri, A. Moffat, J. Mackenzie, J. S. Culpepper, and D. Beck. Accelerated query processing via similarity score prediction. In *Proc. SIGIR*, pages 485–494, 2019.

[44] G. E. Pibiri and R. Venturini. Techniques for inverted index compression, 2019. arXiv:1908.10598.

[45] S. E. Robertson and H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Found. Trnd. Inf. Retr.*, 3:333–389, 2009.

[46] E. Schurman and J. Brutlag. Performance related changes and their user impact. Velocity, 2009.

[47] W.-Y. Shieh, T.-F. Chen, J. J.-J. Shann, and C.-P. Chung. Inverted file compression through document identifier reassignment. *Inf. Proc. & Man.*, 39(1):117–131, 2003.

[48] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. ECIR*, pages 101–112, 2007.

[49] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proc. SIGIR*, pages 305–312, 2004.

[50] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proc. CIKM*, pages 317–326, 2011.

[51] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proc. SIGIR*, pages 219–225, 2005.

[52] N. Tonellotto, C. Macdonald, and I. Ounis. Efficient query processing for scalable web search. *Found. Trnd. Inf. Retr.*, 12(4-5):319–500, 2018.

[53] A. Trotman, X.-F. Jia, and M. Crane. Towards an efficient and effective search engine. In *Proc. OSIR at SIGIR 2012*, pages 40–47, 2012.

[54] A. Trotman, A. Puurula, and B. Burgess. Improvements to BM25 and language models examined. In *Proc. Aust. Doc. Comp. Symp.*, pages 58–65, 2014.

[55] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Proc. & Man.*, 31(6):831–850, 1995.

[56] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. SIGIR*, pages 105–114, 2011.

[57] Q. Wang and T. Suel. Document reordering for faster intersection. *Proc. VLDB*, 12(5):475–487, 2019.

[58] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, San Francisco, second edition, 1999.

[59] E. Yafay and I. S. Altingovde. Caching scores for faster query processing with dynamic pruning in search engines. In *Proc. CIKM*, pages 2457–2460, 2019.

[60] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.

[61] W. Yang, K. Lu, P. Yang, and J. Lin. Critically examining the "neural hype": Weak baselines and the additivity of effectiveness gains from neural ranking models. In *Proc. SIGIR*, pages 1129–1132, 2019.

[62] Z. Yang, A. Moffat, and A. Turpin. How precise does document scoring need to be? In *Proc. Asia Info. Retri. Soc. Conf.*, pages 279–291, 2016.

[63] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):6:1–6:56, 2006.