A Comparison of Document-at-a-Time and Score-at-a-Time Query Evaluation

Matt Crane,[1] J. Shane Culpepper,[2] Jimmy Lin,[1] Joel Mackenzie,[2] and Andrew Trotman[3]
[1] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
[2] Department of Computer Science, RMIT University, Melbourne, Australia
[3] Department of Computer Science, University of Otago, Dunedin, New Zealand
{matt.crane,jimmylin}@uwaterloo.ca, {shane.culpepper,joel.mackenzie}@rmit.edu.au, andrew@cs.otago.ac.nz

ABSTRACT

We present an empirical comparison between document-at-a-time (DAAT) and score-at-a-time (SAAT) document ranking strategies within a common framework. Although both strategies have been extensively explored, the literature lacks a fair, direct comparison: such a study has been difficult due to vastly different query evaluation mechanics and index organizations. Our work controls for score quantization, document processing, compression, implementation language, implementation effort, and a number of details, arriving at an empirical evaluation that fairly characterizes the performance of three specific techniques: WAND (DAAT), BMW (DAAT), and JASS (SAAT). Experiments reveal a number of interesting findings. The performance gap between WAND and BMW is not as clear as the literature suggests, and both methods are susceptible to tail queries that may take orders of magnitude longer than the median query to execute. Surprisingly, approximate query evaluation in WAND and BMW does not significantly reduce the risk of these tail queries. Overall, JASS is slightly slower than either WAND or BMW, but exhibits much lower variance in query latencies and is much less susceptible to tail query effects. Furthermore, JASS query latency is not particularly sensitive to the retrieval depth, making it an appealing solution for performance-sensitive applications where bounds on query latencies are desirable.

Keywords

Efficiency; Experimentation; Measurement

1. INTRODUCTION

Document-at-a-time (DAAT) query evaluation and score-at-a-time (SAAT) query evaluation represent two fundamentally different approaches to top k document retrieval. These two approaches have vastly different mechanics and share strong affinities to different index organizations, namely document-ordered indexes and impact-ordered indexes, respectively. These facts and other implementation details make a fair comparison between DAAT and SAAT strategies difficult. Although both have been extensively studied, the literature can be characterized as having parallel threads: DAAT strategies are for the most part only compared to other DAAT strategies, and SAAT to SAAT. To our knowledge, there has not been a fair comparison between DAAT and SAAT query evaluation within a common framework that allows us to study their performance characteristics. Furthermore, both strategies can provide *approximate* rankings that enable tradeoffs between effectiveness and efficiency, adding to the complexity of any comparison.

In this paper, we present a fair empirical comparison of DAAT and SAAT query evaluation. For DAAT, we consider two specific techniques: WAND [7] and Block-Max WAND (BMW) [8, 12, 14]. For SAAT we consider JASS [20]. Our experiments over modern web collections control for score quantization, document processing, compression, implementation language, implementation effort, and a host of other details, thus isolating performance characteristics that can be directly attributed to the core algorithms. Our major findings can be summarized as follows:

- When considering rank-safe query evaluation, WAND and BMW are comparable in terms of performance. The block-max optimizations in BMW are effective for short queries, but not long queries. This finding adds more nuance to the general understanding in the literature that "BMW is better than WAND". We explain this result in detail.
- When considering exact query evaluation, both WAND and BMW (DAAT) are faster than JASS (SAAT), although the performance gap is narrower than what one might expect considering that JASS exhaustively traverses *all* postings.
- In both exact and approximate query evaluation, JASS exhibits far less variance in query latencies compared to WAND and BMW, both of which suffer from occasional tail queries that take orders of magnitude longer than the median query.
- In both exact and approximate query evaluation, JASS query evaluation latency is less sensitive to the depth of k in top k retrieval than either WAND or BMW, which makes the algorithm attractive for multi-stage retrieval architectures that might need to generate large initial candidate sets.
- Understanding the impact of tail queries is critical when comparing the performance profiles of different query evaluation techniques. We show that simply reporting mean running times for large query sets is not good enough, even when query sets are broken down by length. Mean query latency does not provide a clear understanding of variances in latency. A more careful examination of WAND and BMW shows that tail queries are not obviously predictable (for example, by query length). Though relatively uncommon, these queries can have a significant impact in performance-sensitive applications where bounds on query latencies are desired.

WSDM 2017, February 06 - 10, 2017, Cambridge, United Kingdom

ISBN 978-1-4503-4675-7/17/02…$15.00

DOI: http://dx.doi.org/10.1145/3018661.3018726

Beyond presenting a comparison between DAAT and SAAT query evaluation strategies within a common framework, our work contributes to a better understanding of modern processor architectures in the context of different philosophies to designing efficient algorithms. DAAT strategies have the goal of minimizing the number of overall computations in generating a top k ranking (e.g., reducing the number of postings touched and documents scored). The tradeoff is that such approaches incur the cost of *irregular* computations, such as skipping around in postings lists—which in architectural terms may translate into branch mispredictions and/or cache misses. On the other hand, SAAT strategies in general and JASS in particular aim to regularize computations (by minimizing branch mispredictions and cache misses) even at the cost of incurring additional (perhaps even unnecessary) computations. We seek to understand the effectiveness/efficiency tradeoffs of these approaches in the context of top k retrieval.

2. BACKGROUND AND RELATED WORK

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:

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

where $w_{d,t}$ is 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 as quickly as possible. This formulation captures vector-space models, probabilistic models such as BM25, as well as language modeling and divergence from randomness approaches.

Nearly all modern search engines depend on an inverted index for top k retrieval. 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, and these organizations share affinities with query evaluation strategies.

In *document-ordered* indexes, postings lists are sorted by document ids (docids) in increasing order and term frequencies are stored separately. Such indexes are usually associated with DAAT query evaluation, which achieves high performance by skipping documents in the postings lists that cannot appear in the top k results. In typical implementations the term weights ($w_{d,t}$'s) are computed during query evaluation as functions of term frequency, document frequency, etc. The most popular DAAT strategies today are WAND [7] and its successor BMW [8, 12, 14], both of which are included in our evaluation. More recent work has looked at priming the score of the k·th item in the heap using tiering and other techniques [10]. We do not explore these ideas in this work since they represent orthogonal optimizations, as SAAT-based approaches could also benefit from tiering.

Other related work on DAAT-based strategies includes the work of Fontoura et al. [15] who explored the impact of query length on several related in-memory processing algorithms, improving effectiveness by incorporating various document features at traversal time [33], and better candidate pre-filtering [13].

In *frequency-ordered* indexes [29], docids are grouped by term frequency; within each grouping docids are sorted in increasing order, but the groupings are arranged in decreasing order of term frequency. Such indexes are typically paired with term-at-a-time (TAAT) query evaluation strategies, which as the name suggests, considers each query term in turn [26, 28]. Building on this thread of work, Anh et al. [3] observed that term weights ($w_{d,t}$'s) could be directly stored in such an index instead of the term frequencies. To facilitate compression, weights can be quantized into integer values called impact scores. These *impact-ordered indexes* are typically used in conjunction with a SAAT query evaluation strategy, which processes blocks of postings in decreasing impact score, potentially hopping from term to term. A modern implementation of SAAT is the JASS algorithm of Lin and Trotman [20], which is the third technique that we evaluate in this paper. One variant of JASS was the fastest in a recent reproducibility evaluation comparing several open-source search engines [21], and thus JASS represents the state of the art. Note that the reproducibility evaluation did not include either a WAND or a BMW implementation, so it is unclear how they compare with JASS. Our study answers this question.

3. APPROACH

In this study we compare two DAAT strategies, WAND and BMW, and a SAAT strategy, JASS. There are, of course, a multitude of algorithms that could be included in a performance evaluation, but studying each incurs substantial effort: here, we opted to focus on in-depth analyses of a few representative algorithms as opposed to superficial evaluations of many. WAND and BMW are selected because they represent the most popular DAAT strategies today. We evaluate both because, contrary to claims in the literature, we find that WAND and BMW are comparable in terms of performance—we explain in detail why this is the case. JASS was selected as the representative of SAAT query evaluation due to its performance in the recent reproducibility study discussed above [21]. We specifically did not evaluate TAAT query evaluation because those algorithms have been largely superseded by score-at-a-time approaches. The choice of the three algorithms also illustrates our point about the design of modern processor architectures, as we discuss later.

3.1 Overview of Different Techniques

We begin with an overview of WAND, BMW, and JASS, the three algorithms in our study.

WAND. A popular dynamic pruning algorithm which performs DAAT traversals during scoring is WAND. The key idea in WAND is to use a small amount of pre-computed "impact" information in order to minimize scoring operations while traversing the postings lists for all of the query terms simultaneously. WAND processing is straightforward: First, pick an additive ranking model such as BM25 and construct a document-ordered index. For each postings list, pre-compute a value U_t which represents the maximum contribution term t can have for the scoring model. When processing a query, perform a standard DAAT traversal of the postings with the following twist:

1. Set a *finger* pointing to the first unevaluated document in each postings list.
2. Set the term processing order of the lists using the docids: smallest to largest.
3. While the sum of the U_t scores is less than the score of the k·th item in the heap, step to the next list.
4. As soon as the heap score is exceeded, the docid in the current list is selected as the *pivot*.
5. A finger search is initiated for all lists evaluated before the pivot, and the current pivot docid is scored if all of the previous lists' docids match the pivot.
6. If the true document score exceeds the minimum value in the k heap, it is added to the heap, and the current threshold is set to the new minimum heap score.

Algorithm 1 WAND processing with partial scoring.

```
    function WAND(q, I, k, θ)
        for t ← 0 to |q| − 1 do
            U[t] ← max_d{w_d | (d, w_d) ∈ I_t}
            (c_t, w_t) ← first_posting(I_t)
 5:     end for
        φ ← −∞                                   // current threshold
        Ans ← { }                                // k-set of (d, s_d) values
        while the set of candidates (c_t, w_t) is non-empty do
            permute the candidates so that c_0 ≤ c_1 ≤ ··· c_{|q|−1}
10:         score_limit ← 0
            pivot ← 0
            while pivot < |q| − 1 do
                tmp_s_lim ← score_limit + U[pivot]
                if tmp_s_lim > φ × θ then
15:                 s ← tmp_s_lim
                    break, and continue from step 21
                end if
                score_limit ← tmp_score_lim
                pivot ← pivot + 1
20:         end while
            if c_0 = c_pivot then                // score document c_pivot
                t ← 0
                while t < |q| and c_t = c_pivot do
                    s ← s − U[t]  // remove upper bound estimate
25:                 s ← s + w_t   // add real contribution to score
                    if s < φ then
                        break, and reset all pointers to c_pivot + 1.
                    end if
                    (c_t, w_t) ← next_posting(I_t)
30:                 t ← t + 1
                end while        // s is the score of document c_pivot
                if s > φ then    // and is a possible top-k answer
                    Ans ← insert(Ans, (c_pivot, s))
                    if |Ans| > k then
35:                     Ans ← delete_smallest(Ans)
                        φ ← minimum(Ans)
                    end if
                end if
            else                                 // can't score c_pivot (yet)
40:             for t ← 0 to pivot − 1 do
                    (c_t, w_t) ← seek_to_document(I_t, c_pivot)
                end for   // all pointers are now at c_pivot or greater
            end if
        end while
45:     return Ans
    end function
```

The performance of WAND can be further improved by adding an additional check during document scoring where the current upper bound score is lowered by subtracting U_t from the upper bound sum, and adding the real term document score as it is computed. Algorithm 1, which is adapted and simplified from Petri et al. [30], shows how to accomplish this [25, 31]. At Line 15, the current score is set to the upper bound estimate during pivot selection. When scoring begins the estimate is removed (Line 24) and the real contribution for the term is added (Line 25). As soon as the current score drops below θ (the score of the k·th item in the heap) scoring can stop and all posting pointers can be updated accordingly. This simple enhancement can lead to a measurable reduction in the number of documents being fully scored, and the benefits quickly increase with query length. We will show in Section 4 that this small change in the WAND implementation greatly reduces the performance gaps between WAND and BMW reported in previous work. To achieve faster but approximate query evaluation, the current upper-bound score can be boosted by a constant, θ (Line 14). Clearly, setting $\theta = 1$ will result in score-safe processing.

BMW. The BMW processing algorithm is a practical enhancement to WAND. The key observation of the BMW algorithm is that postings in inverted indexes are compressed as blocks, and every i·th posting is left uncompressed in order to support skipping [26]. Instead of just using a single global U_t, a series of local block scores $U_{b,t}$ can be pre-computed and used to refine pivot selection as the postings lists are traversed. The goal is to only decompress a block to compute the real document score when necessary. Decompressing postings blocks adds to the overall query latency and so limiting the number of blocks that must be fully processed can overcome the additional overhead of recomputing upper bound estimates using the $U_{b,t}$ scores. Several subsequent studies have confirmed that BMW requires fewer documents to be scored than WAND, and our results confirm this. However, achieving high performance with BMW requires careful software engineering, making it difficult to find publicly available implementations of the algorithm that are robust, reliable, and reusable.

JASS. Query evaluation in JASS [20] begins by first looking up the postings corresponding to all query terms. Each postings list comprises a sequence of decreasing impact scores, each of which is associated with a run of delta-compressed sorted docids (which we call a segment). Segments from all postings lists are sorted in decreasing impact score and processed in that order. For each segment, its impact score is loaded into a CPU register, and for each docid in the segment, the register's value is added to its accumulator. Thus, the final result is an unsorted list of accumulators (holding the document scores). Note that in an impact-ordered index the impact score is stored once for each segment and not once for each docid, and therefore the number of integers read per segment can be nearly half the number read in a document-ordered index.

To avoid sorting the accumulator list, a heap of the top k can be maintained during processing. That is, after adding the current impact score to the accumulator, 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 docid.

For SAAT query evaluation, several approaches to accumulator organization and management have been previously proposed [1, 3, 16, 26]. In this work, we implement the approach of Jia et al. [16]. Since our impact scores are 9 bits (discussed later), and queries are generally short, it suffices to allocate an array of 16-bit integers, one per document, indexed by the docid. Modern hardware has ample memory to keep the accumulators in memory. This approach is much simpler than other accumulator management strategies focused on accumulator pruning, e.g., [1, 26], which made sense when memory was scarce.

Note that the default JASS algorithm *exhaustively* processes all postings. That is, the query evaluation algorithm considers *all* documents that have at least one query term. This represents a substantial amount of work in terms of processor operations, but the computations are highly regular, at least compared to the memory access patterns of WAND and BMW. However, since we are processing postings segments in decreasing impact order, we can terminate at any time—yielding approximate rankings. By definition, the segments are processed in decreasing importance: larger score contributions will be added earlier such that the ranking is gradually refined as query evaluation progresses. This effectiveness/efficiency tradeoff is controlled by a parameter ρ that specifies

the maximum number of postings to process. Lin and Trotman [20] describe how ρ can be mapped into wall-clock query execution time using a simple linear model, thus producing an *anytime* algorithm where JASS can be tuned to meet an arbitrary time budget.

3.2 Common Framework

To conduct a fair evaluation between WAND, BMW, and JASS, we have built a common framework that factors out many issues that are orthogonal to the performance of query evaluation. In particular, we have taken care to address the following issues:

Score quantization. One major difference between DAAT and SAAT strategies is that the latter is only practical with pre-computed quantized impact scores. In contrast, document-ordered indexes typically used for DAAT strategies store term frequencies in the postings and compute document scores with respect to a scoring model (e.g., BM25) during query evaluation. The substantive difference is that computing document scores for SAAT involves only integer additions (summing the impact scores), whereas computing something like BM25 requires many floating point operations. To factor out these differences, we have modified WAND and BMW to work with impact scores (exactly the same as those of JASS, see more details below), so that in all cases the exact same number of operations are required to compute an individual document score.

It should also be noted that both WAND and BMW make a priori assumptions about scoring algorithms and parameter selection. For example, if BM25 is used, every posting must be pre-scored in order to compute U_t and $U_{b,t}$, and scoring function parameters cannot be easily changed after indexing occurs. Macdonald et al. [24] showed that approximations can be used rather than the exact scores, but this may result in solutions that are not safe-to-k or traversals that process more postings than if the true upper-bounds were computed. Changing a single parameter requires the entire index to be re-scored in order to guarantee that U_t and $U_{b,t}$ are correct. Furthermore, considering the cost of computing scores on the fly for every pivot document at query time, it is not clear why score quantization is not more widely used in current WAND and BMW systems, especially since such systems are commonly used for early-stage candidate retrieval in a multi-stage ranking architecture. This is not a novel idea, in fact Dimopoulos et al. [12] explored both quantization and document reordering in previous work, but to our knowledge most current systems do not take advantage of these techniques. Using quantization can result in a 30% to 40% improvement in performance, and document reordering approaches provide similar benefits as well. WAND also benefits from these enhancements and we shall see shortly that in certain contexts it benefits proportionally more than BMW does.

Document processing. Document processing, which includes tokenization, stemming, and stopword removal, can have a significant impact on both the effectiveness and efficiency of different query evaluation strategies. We factor out these differences by using the ATIRE system [35] to first construct a master index, from which all our individual experiments derive. That is, in a pre-processing step, our WAND, BMW, and JASS implementations read the ATIRE index and rewrite the data in their internal formats. The master index quantizes document scores in the range 1–511, following the technique of Crane et al. [9], using BM25 as the scoring model. All pre-processing for each individual technique includes all necessary postings sort operations and computing auxiliary data, such as the block max values for BMW. Thus, our experiments have factored out differences due to document processing.

Compression. It is fairly obvious that different compression techniques have material impact on the performance of query evaluation techniques, and thus it is necessary to factor out their effects. In our framework, we implemented two different compression codecs. The first is QMX compression [34], which can be thought of as an extension of the Simple family [2] that takes advantage of SSE (Streaming SIMD Extensions) instructions in the x86 architecture. Experiments [34] have shown QMX to be more efficient to decode than SIMD-BP128 [18] (a related SIMD-based technique) and competitive with all SIMD and non-SIMD techniques in terms of size. In addition, we have also integrated the FastPFor library of Lemire and Boytsov [18] and corresponding optimizations [19].

For document-ordered indexes, postings are organized around blocks of 128 integers, both in terms of the skipping entry points for WAND and for computing the block-max scores for BMW. This organization fits perfectly with block-based compression codecs such as OPT-PFor or SIMD-BP128. Although QMX is not block based, we retain the same organization for comparability to the FastPFor codec. In all cases, docids were compressed after applying gap encoding. For JASS, the impact scores are stored uncompressed (since they are not repeated in an impact-ordered index), while in WAND and BMW they were compressed without gap encoding.

We ran experiments with both QMX and FastPFor, and the conclusions are the same. For brevity, we report the results using QMX only because it has proven to be faster. Compression is an orthogonal issue to what we are interested in, but our experiments show that it does not really matter anyway.

Language and implementation effort. All implementations are written in C++, which factors out differences that are attributed to the programming language. Beyond the implementation language, one common criticism of studies that attempt to compare very different techniques is unequal implementation effort: more optimization effort is spent on one technique than another. To alleviate this concern: our team includes researchers that have published extensively advocating for DAAT approaches and researchers that have published extensively advocating for SAAT approaches. We bring to this work codebases that have formed the basis of many previous papers [20, 22, 25, 30, 31, 35] and one contribution of this work is the non-trivial integration of these codebases to eliminate the various factors discussed above. The original source of our WAND and BMW implementations are heavily inspired by code[1] of Dimopoulos et al. [12], which has been further refined to work within a unified framework. The code used in all experiments reported in this paper has been made available[2,3] for others to replicate and build on our results.

Name	# Docs	TREC Topics
Gov2	25,205,179	701–850 (‘04–‘06)
ClueWeb09b	50,220,423	51–200 (‘10–‘12)
ClueWeb12-B13	52,343,021	201–300 (‘13–‘14)

Table 1: Summary of TREC collections and topics used.

4. EXPERIMENTAL SETUP

Our experiments used three standard TREC web test collections: Gov2, ClueWeb09 (category B), and ClueWeb12-B13 (“category B”). Details for these collections are provided in Table 1, showing their sizes and the corresponding TREC topics used to evaluate effectiveness. We also used the recently-created UQV test collection [5]. This collection contains $5{,}764$ queries and has shallow

[1] https://github.com/dimopoulosk/WSDM13

[2] https://github.com/jmmackenzie/Quant-BM-WAND/

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

judgments for all of the queries on ClueWeb12-B13. After normalization, the average query length is 5.3 terms, and 5,593 queries (97%) have 10 terms or less. This larger test collection is more suitable for the evaluation of query latency when effectiveness is not under consideration.

All experiments used an identical underlying index generated by the ATIRE system, as described in Section 3.2. For simplicity, we kept collection processing to a minimum: for each document, all malformed UTF-8 sequences were converted into spaces, alphabetic characters were separated from numeric characters and stripping stemming was applied; no additional document cleaning was performed except for XML comment and tag removal. We did not remove stopwords as these are often used in higher order term proximity feature models. As previously described, we post-processed the ATIRE index into a representation appropriate to each query evaluation strategy, with the same compression techniques to the extent possible. All document scores were quantized to 9-bit values based on BM25. Although ATIRE performs multithreaded indexing by default, in all our experiments we indexed on a single thread in order to preserve the original collection document order. As previously discussed, in these experiments we report results with QMX compression because it has proven to be faster than other state-of-the-art compression algorithms [34], but the choice of compression does not alter our findings.

Experiments were conducted in memory on a Red Hat Enterprise Linux Server v7.2 (Maipo) with two Intel Xeon E5-2630 CPUs and 256 GB of RAM. All algorithms are single threaded and thus executed on one core in an otherwise idle machine. We measured query latency in milliseconds. To capture query variance, we primarily report results using a 'Tukey' boxplot, in which the box bounds the 25th and 75th percentiles and the bar inside the box denotes the median. The whiskers correspond to data within $1.5\times$ IQR and outliers are plotted as points. For convenience, we add a red diamond in these plots to represent the mean. Space usage is measured in GB (10^9 bytes) and includes only postings. In terms of effectiveness, for Gov2 we compute RBP [27] and average precision, for ClueWeb09b we compute RBP and nDCG@20, and for ClueWeb12-B13 we compute RBP and nDCG@10. We used $p = 0.8$ for all RBP computations. Note that we intentionally use shallow metrics for the two ClueWeb collections as computing deep recall-based metrics such as AP are not recommended given the shallow pooling depth used when producing the original relevance judgments; see Lu et al. [23] for a more comprehensive discussion about the effects of metric choice with different collections. The high residuals for RBP even with $p = 0.8$, shown in parenthesis in Table 2, suggest that caution should still be used when interpreting effectiveness results on the ClueWeb collections.

Our first set of experiments examined exact (rank-safe) query evaluation. However, WAND, BMW, and JASS all support *approximate* query evaluation where effectiveness can be traded for efficiency. For WAND and BMW, this is controlled with θ, a pruning threshold which can be interpreted as the extent to which the query is treated as an AND or an OR. For JASS, the effectiveness/efficiency tradeoff is controlled by ρ, which is the number of postings to process. In our second set of experiments we sweep the θ and ρ parameters to better understand the performance characteristics in approximate query evaluation.

5. RESULTS

We split the presentation of results into three subsections. First we consider exact query evaluation, where all the techniques produce the same output and compete solely on query latency. Next, we consider approximate query evaluation, where techniques are able to trade off effectiveness for efficiency. Finally, we reflect on the characteristics of the different query evaluation algorithms in light of modern processor architectures.

Gov2					
System	**Mean**	**Median**	**Space**	**AP**	**RBP**
WAND$_e$	41.6	23.3	17	0.2899	0.5862 (0.0028)
BMW$_e$	30.8	17.4	17	0.2899	0.5862 (0.0028)
JASS$_e$	34.9	20.2	15	0.2899	0.5862 (0.0028)
ClueWeb09b					
System	**Mean**	**Median**	**Space**	**nDCG**	**RBP**
WAND$_e$	190.8	73.5	51	0.1380	0.2588 (0.1265)
BMW$_e$	166.3	61.4	52	0.1380	0.2588 (0.1265)
JASS$_e$	170.5	117.8	51	0.1380	0.2588 (0.1265)
ClueWeb12-B13					
System	**Mean**	**Median**	**Space**	**nDCG**	**RBP**
WAND$_e$	210.5	114.3	65	0.1240	0.2639 (0.3555)
BMW$_e$	210.3	107.9	65	0.1240	0.2639 (0.3555)
JASS$_e$	221.9	148.7	63	0.1240	0.2639 (0.3555)

Table 2: Effectiveness and efficiency results for exact query evaluation for TREC topics with $k = 1000$: median and mean latencies measured in ms, space measured in GB (10^9 bytes), and standard effectiveness metrics.

5.1 Exact Query Evaluation

Our main top-level results are presented in Table 2, which shows mean and median query latencies, index sizes, and effectiveness of exact rankings with $k = 1000$ produced by WAND, BMW, and JASS (we use the subscript $_e$ to denote *exact*). The index sizes show no substantial differences in space usage between the different index organizations. The effectiveness metrics confirm that our implementations are correct and that they compare favorably to similar results reported in the literature.

An initial look at the results suggests that WAND and BMW are generally faster than JASS, although JASS is faster than WAND on Gov2. On the ClueWeb collections, JASS is substantially slower if we consider *median* query latency, but compares favorably in terms of *mean* query latency. However, a deeper dive into the experimental results reveals many interesting findings, which we organize below around major themes:

Variance in query latencies. In Figure 1 we summarize query latencies for all techniques across all collections in 'Tukey' box-and-whiskers plots, as described earlier. For both WAND and BMW, we find outliers and in general broader spans covered by the whiskers, which indicate greater variance in query latencies. In contrast, JASS query latencies exhibit fewer outliers and are more tightly clustered. Thus, even though JASS is slower in terms of median query latency, the mean latencies for WAND and BMW are greatly impacted by the outliers. This seems like an important observation for operational search engines from the perspective of the user experience, and indeed designers of production systems have devoted substantial effort to reducing "tail latencies" [11, 17].

Are there any "obvious" predictors of tail latencies? Figure 2 breaks down query latencies by query length for the UQV queries on ClueWeb12-B13, with different values of $k = \{10, 100, 1000\}$. We used the UQV queries since we are only focused on query latency and the larger number of available queries allows more mean-

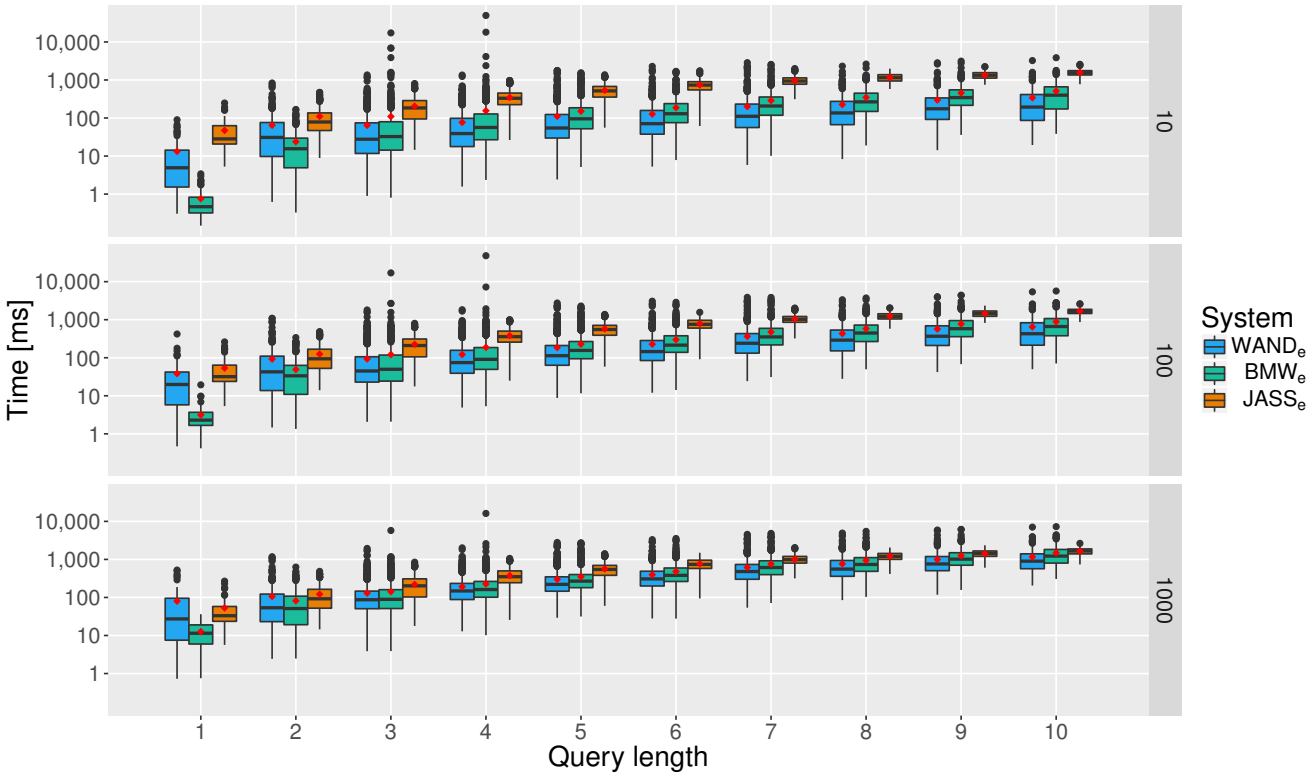

Figure 2: Distribution of query latencies for UQV queries on ClueWeb12-B13, broken down by query length with $k = \{10, 100, 1000\}$.

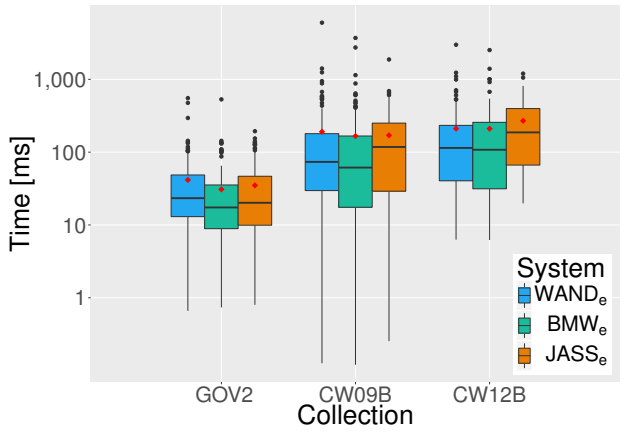

Figure 1: Visualization of query latencies in Table 2 (TREC topics, $k = 1000$) as 'Tukey' box-and-whiskers plots. JASS exhibits less variance in latency than the other approaches.

ingful analysis. One obvious hypothesis is that the outliers tend to occur with longer queries, but this is clearly not the case: the worst offenders are actually 3–4 term queries. We certainly do not make the claim that tail queries are in principle unpredictable with WAND and BMW, simply that there are no obvious correlates from our experimental results; such a prediction task would be interesting future work. In contrast, we see that query latencies for JASS are all much more tightly bounded for all query lengths.

The slowest query for BMW (across all values of k) was "*the interpretation of dream*". An interesting anomaly is that this query ran slower for smaller values of k (Figure 2, Figure 3, and Table 3). A failure analysis reveals that this was due to a particular term distribution, an observation that has been previously noted by Petri et al. [30]. That is, BMW skips more often, but much less effectively, as k decreases. This translates into a large overhead of calculating and inducing skips, but for little reward. Additionally, other design choices such as quantization, stemming, and/or stopping also contribute to such tail latencies.

From Figure 2 we see that JASS is relatively slow for one-term queries because it does not perform any special optimizations. For one term queries it still sorts all postings segments and inserts document scores into the heap—both of which are not necessary. This is a simple optimization that we save for future work.

To further examine the impact of the search depth k, Figure 3 shows the summary distribution of query latencies for UQV queries on ClueWeb12-B13 with $k = \{10, 100, 1000\}$; summary statistics for this figure are presented in Table 3. We see that JASS is relatively insensitive to k, but the other two techniques grow slower as k increases. For JASS, the only difference in query evaluation for different values of k is the number of documents retained in the heap, and heap operations are dominated by the time taken to traverse postings (exhaustively), and hence k does not have much impact. For multi-stage ranking algorithms [4, 36] that may require large values of k, JASS may be a good choice.

In summary, our experimental results show that the DAAT approaches are faster, particularly for small values of k, but JASS has much more predictable query latency.

Additionally, a methodological lesson from these results is the inadequacy of reporting only summary figures such as those in Table 2, since they may hide important differences between techniques such as outliers, as we have shown here. However, more

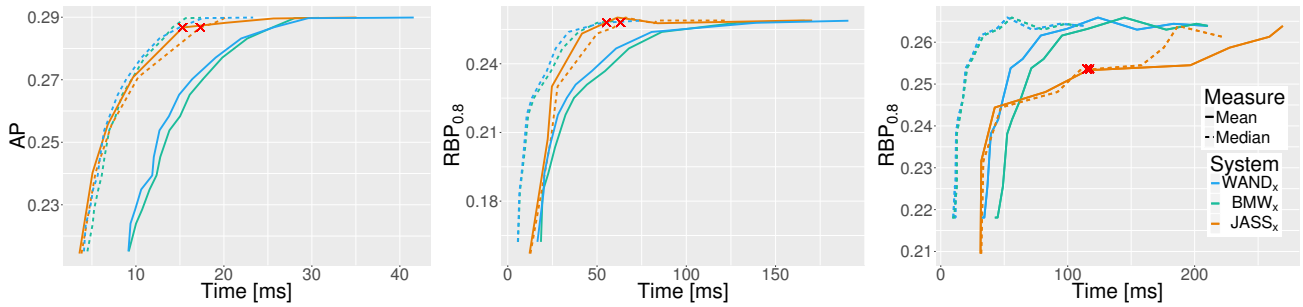

Figure 4: Effectiveness/efficiency tradeoffs on Gov2 (left), ClueWeb09b (middle), and ClueWeb12B-13 (right). Curves represent sweeps across the θ and ρ parameter space, with $k = 1000$. Effectiveness for Gov2 is reported in AP; for ClueWeb09b and ClueWeb12-B13, RBP with $p = 0.8$. Red crosses represent the JASS heuristic of setting ρ to 10% of the collection size.

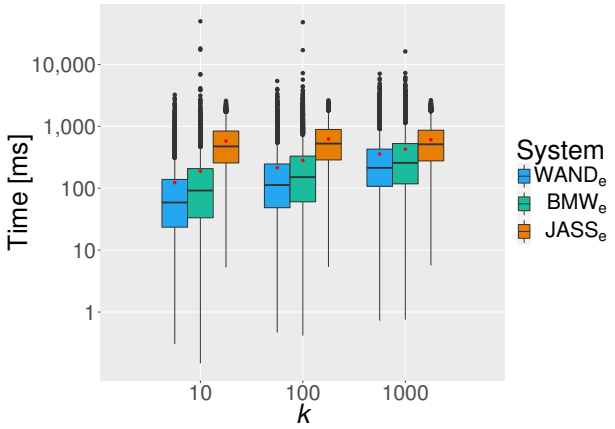

Figure 3: Distribution of query latencies for UQV queries on ClueWeb12-B13 with $k = \{10, 100, 1000\}$.

$k = 10$				
Algorithm	**Min**	**Max**	**Mean**	**Median**
WAND$_e$	0.3	3236.5	136.7	61.8
BMW$_e$	0.1	49680.0	209.6	95.7
JASS$_e$	5.3	4886.6	621.4	495.1
$k = 100$				
Algorithm	**Min**	**Max**	**Mean**	**Median**
WAND$_e$	0.5	5397.5	241.7	118.0
BMW$_e$	0.4	48063.6	321.0	157.5
JASS$_e$	5.4	4929.4	671.4	543.2
$k = 1000$				
Algorithm	**Min**	**Max**	**Mean**	**Median**
WAND$_e$	0.7	7066.4	406.2	221.9
BMW$_e$	0.7	16181.2	493.1	264.6
JASS$_e$	5.7	5194.7	656.7	531.9

Table 3: The query time statistics corresponding to Figure 3, tabulated for convenience. Times are reported in ms.

detailed breakdown of latency figures, such as box-and-whiskers plots, have not become standard practice in information retrieval research. They should be.

The impact of block-max optimizations. According to the information retrieval literature, there is the general sense that BMW represents a definitive improvement over WAND in terms of query latency, but this result is not borne out in our experiments. For example, consider Table 2 and Figure 3. On older collections such as Gov2 and even ClueWeb09b, BMW appears to be consistently better than WAND. However, the gap narrows in ClueWeb12-B13. When we move to longer queries as found in the UQV collection, the trend actually reverses. A closer look at Figure 2 provides further insight into what is happening: quantization and incremental scoring changes the relative performance profiles of BMW and WAND. As queries get longer and the collection size grows, WAND begins to outperform BMW. On short queries BMW is clearly superior, but its complex logic for computing skips suffers from diminishing returns as queries become longer. Longer queries are more likely to benefit from partial scoring, but the number of additional cache misses incurred traversing multiple postings lists to compute block score refinements also increases.

Taking into account all these results, there is a much less convincing case that BMW should always be used instead of WAND, particularly for long queries and quantized indexes. At the very least we suggest a more nuanced conclusion than simply "BMW is better than WAND".

5.2 Approximate Query Evaluation

Our next set of experiments consider *approximate* query evaluation where we trade off effectiveness for efficiency. For WAND and BMW, this tradeoff is controlled by the pruning threshold θ, and for JASS, the tradeoff is controlled by ρ, which is the number of postings to process.

In Figure 4, we sweep across the θ and ρ parameter space to understand the effectiveness/efficiency tradeoffs of WAND, BMW, and JASS. In all cases, we set $k = 1000$; for Gov2, we report AP, for ClueWeb09b and ClueWeb12-B13 we report RBP with $p = 0.8$. For JASS, Lin and Trotman [20] suggest a heuristic of setting ρ to be 10% of the collection size—these are shown by the red crosses on the JASS curves. We call this the JASS$_a$ setting. This heuristic appears to yield a good setting for Gov2 and ClueWeb09b, but less so for ClueWeb12-B13. Note, however, the y scale in the ClueWeb12-B13 figure, which exaggerates the effectiveness loss.

To support a fair comparison between approximate WAND, BMW, and JASS, we selected settings of θ for WAND and BMW that yields the same effectiveness as the JASS$_a$ setting (10% heuristic). That is, from the red crosses in the JASS curves in Figure 4, we draw a horizontal line and note where it intersects with either the WAND or BMW performance curves. We take the closest θ value and call these settings WAND$_a$ and BMW$_a$ respectively.

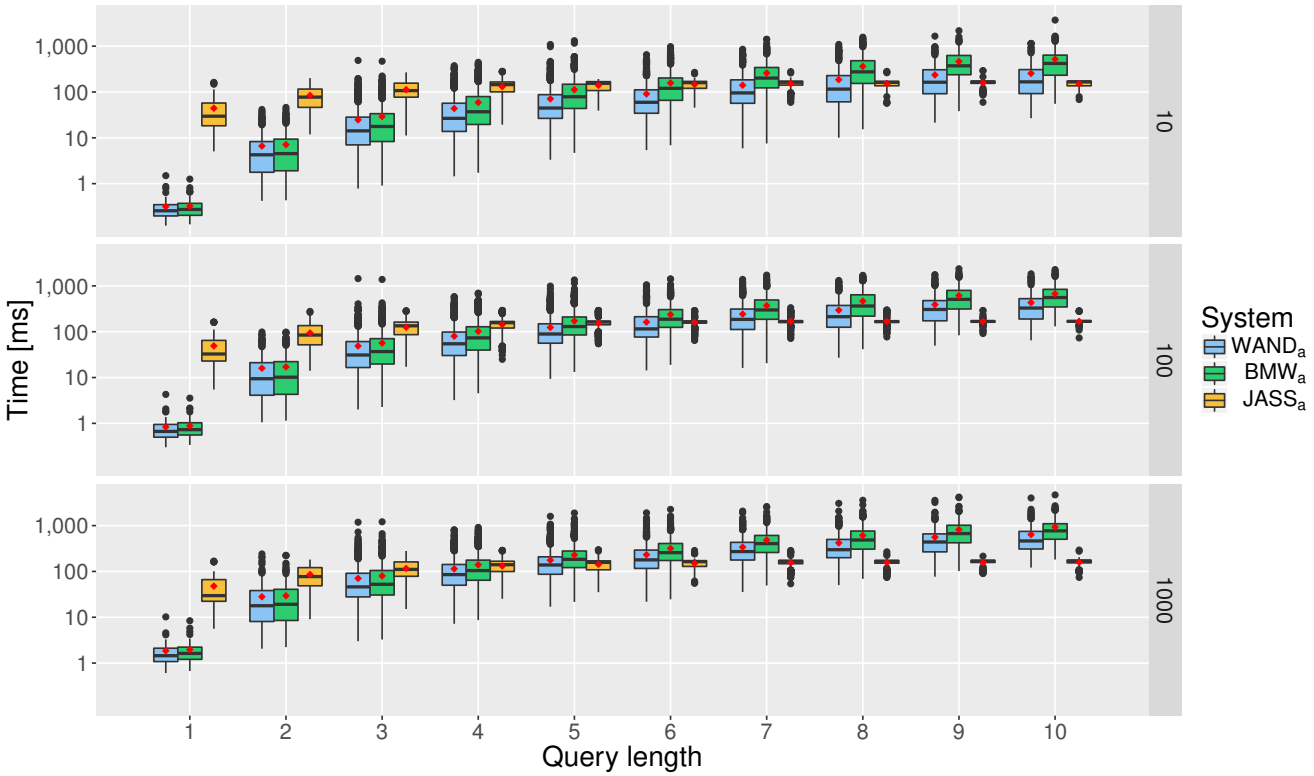

Figure 6: Distribution of query latencies for approximate query evaluation techniques for UQV queries on ClueWeb12-B13, broken down by query length with $k = \{10, 100, 1000\}$.

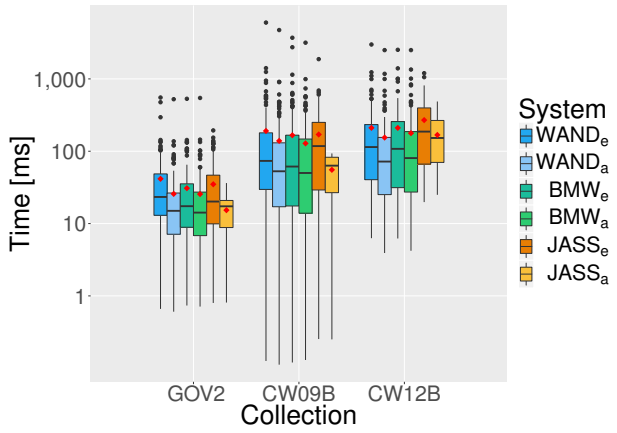

Figure 5: Distribution of query latencies for exact and approximate variants of WAND, BMW, and JASS, with $k = 1000$.

Figure 5 shows the distribution of query latencies for WAND$_a$, BMW$_a$, and JASS$_a$ as defined above. For reference, the exact query evaluation conditions from Figure 1 are replicated here. We see that although the approximate variants of WAND and BMW do indeed decrease query latency, they do little to reduce the variance in latencies. Furthermore, outliers are still present, which means that WAND and BMW are not particularly effective for controlling tail latencies. In contrast, approximate JASS reduces the span of the whiskers and tightly controls tail latencies—this is indeed in its design. Once JASS traverses ρ postings, it terminates, and since the number of traversed postings is linear with respect to query evaluation latency, JASS provides a very responsive "knob" for controlling query latencies.

In Figure 6 we further analyze query latencies of the approximate variants WAND$_a$, BMW$_a$, and JASS$_a$ for the UQV queries on ClueWeb12-B13 with $k = \{10, 100, 1000\}$. The box-and-whiskers plots break down query latencies by query length. Apparent here is the contrast between JASS$_a$ and the approximate DAAT variants. Query latencies increase for WAND$_a$ and BMW$_a$ as the queries become longer, but not for JASS$_a$, since it sets an upper bound on the number of postings processed. JASS is able to effectively control tail latencies, consistent with the experiments above.

Finally, in Figure 7 we examine the effects of different values of k for the approximate variants. The latency distribution of the exact algorithms from Figure 3 are replicated for reference. The conclusions here are consistent with exact query evaluation: JASS is insensitive to k and approximate variants of WAND and BMW do not address the issue of tail latencies.

5.3 Modern Processor Architectures

Finally, these results teach us interesting lessons about modern processor architectures. The insight behind nearly all performance optimizations in DAAT query evaluation is skipping documents that cannot possibly be in the top k. This is realized in WAND via dynamic pruning; block-max optimizations introduce more complex logic that enable the skipping of more documents. Skipping, however, has associated costs that may be non-trivial on modern processor architectures. Skipping logic necessarily involves branches, which can potentially translate into pipeline stalls when branch prediction fails. Skipping postings creates irregular memory ac-

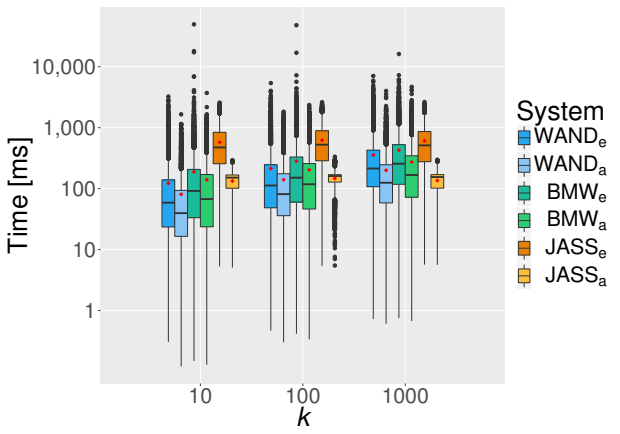

Figure 7: Distribution of query latencies for approximate query evaluation techniques for UQV queries on ClueWeb12-B13 with $k = \{10, 100, 1000\}$.

cess patterns that might create cache misses and hence incur additional memory latencies. These are potential performance issues in WAND that are made even more severe in BMW due to the increase in the complexity of the skipping logic.

The important question is, what is the cost of deciding if postings can be safely skipped relative to the cost of simply traversing the postings exhaustively? Contrast the approach of WAND$_e$ and BMW$_e$ with JASS$_e$, which exhaustively traverses all postings for every query. However, the traversal is set up such that all memory accesses are predictable, and hence benefit from pre-fetching, with minimal branching, thus exploiting modern pipelined processor architectures to the extent possible.

In Table 4 we show the mean and median numbers of postings that are processed for the UQV queries on ClueWeb12-B13. As expected, BMW processes substantially fewer postings than WAND, but this does not *definitively* translate into faster queries, as we have discussed above. Most interesting is the fact that JASS processes orders of magnitude more postings, but is not substantially slower. This finding illustrates the cost of *irregular* computations (the logic in deciding what postings to skip) and *irregular* memory access patterns (the skipping itself) in modern processor architectures. Modern processors are capable of tremendous instruction throughput, provided that the computations are organized in a *regular* manner, as JASS attempts to do. This is not a new observation, as the lesson is well-known in the database community [6, 32], but our experiments nicely summarize the performance characteristics of modern processor architectures in the context of information retrieval algorithms. A pithy summary of this finding might be: it takes as much time to decide what work you can skip as it takes to just do the work. Recall that JASS typically does very little work per posting: it has the impact score already in a CPU register, loads a docid and adds the register to the docid position of the accumulators array (resulting in the intermediate score being in a CPU register), then compares that value to the bottom of the heap (also in a CPU register), typically not needing to update the heap.

6. CONCLUSION

In summary, our work has reached several interesting conclusions. First, WAND and BMW appear to exhibit superior performance compared to SAAT-based methods such as JASS when considering median query latency. However, a more careful analysis shows

Algorithm	Mean	Median
WAND$_e$	191,269	133,264
BMW$_e$	126,858	98,260
JASS$_e$	60.1M	47.4M
WAND$_a$	108,661	82,515
BMW$_a$	101,544	78,308
JASS$_a$	4.3M	4.8M

Table 4: The number of postings processed for the UQV queries on ClueWeb12-B13, for exact and approximate query evaluation techniques.

that the first two methods have one important weakness that is often overlooked—the performance of tail queries, which may take orders of magnitude longer than the median query to execute. Second, it is not clear that the superior performance of WAND and BMW is a significant advantage for very small values of k as simple bag-of-words systems incorporating these techniques are often first stage retrieval systems. Thus, retrieving hundreds or even thousands of results for some queries may be necessary. JASS is insensitive to k, and can benefit from increasing the ρ approximation *and* retrieve many results simultaneously. JASS also allows the number of postings scored to be controlled. This means that query latency can be tightly controlled. Even in an approximate configuration, there is no obvious way to avoid tail latencies in BMW and WAND, which can occur at any time. Query length alone does not appear to be a good predictor of tail queries in WAND and BMW, although it would be interesting to explore a broader range of features that might predict these outliers.

Finally, the performance profiles of BMW and WAND are dependent on query length. On short queries, BMW is clearly superior due to its fine-grained ability to dynamically prune away documents that need not be considered, at the level of postings blocks. However, BMW suffers from diminishing returns as queries become longer and hence more complex. The additional branch mispredicts and other side effects of irregular computations do not make up for the fact that BMW scores fewer documents.

Despite decades of research on query evaluation for top k retrieval, these observations have been elusive for information retrieval researchers due to very different index organizations and query evaluation mechanics. Only after carefully controlling for score quantization, document processing, compression, implementation language, implementation effort, and a number of details are we able to arrive at an empirical evaluation that fairly characterizes the performance of document-at-a-time and score-at-a-time query evaluation. Our common framework can serve as the basis of additional follow-up explorations.

Acknowledgments. This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Australian Research Council's *Discovery Projects* Scheme (DP170102231). Shane Culpepper is the recipient of an Australian Research Council DECRA Research Fellowship (DE140100275).

References

[1] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR*, pages 372–379, 2006.

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

[3] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with

effective early termination. In *SIGIR*, pages 35–42, 2001.

[4] N. Asadi and J. Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *SIGIR*, pages 997–1000, 2013.

[5] P. Bailey, A. Moffat, F. Scholer, and P. Thomas. UQV100: A test collection with query variability. In *SIGIR*, pages 725–728, 2016.

[6] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008.

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

[8] K. Chakrabarti, S. Chaudhuri, and V. Ganti. Interval-based pruning for top-k processing over compressed lists. In *ICDE*, pages 709–720, 2011.

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

[10] 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. *IP&M*, 52(5):855–872, 2016.

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

[12] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *WSDM*, pages 113–122, 2013.

[13] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. A candidate filtering mechanism for fast top-k query processing on modern CPUs. In *SIGIR*, pages 723–732, 2013.

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

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

[16] X.-F. Jia, A. Trotman, and R. O'Keefe. Efficient accumulator initialisation. In *ADCS*, pages 44–51, 2010.

[17] S. Kim, Y. He, S.-W. Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *WSDM*, pages 7–16, 2015.

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

[19] D. Lemire, L. Boytsov, and N. Kurz. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience*, 46(6):723–749, 2016.

[20] J. Lin and A. Trotman. Anytime ranking for impact-ordered indexes. In *ICTIR*, pages 301–304, 2015.

[21] 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 *ECIR*, pages 408–420, 2016.

[22] X. Lu, A. Moffat, and J. S. Culpepper. On the cost of extracting proximity features for term-dependency models. In *CIKM*, pages 293–302, 2015.

[23] X. Lu, A. Moffat, and J. S. Culpepper. The effect of pooling and evaluation depth on ir metrics. *IRJ*, 19(4):416–445, 2016.

[24] C. Macdonald, I. Ounis, and N. Tonellotto. Upper-bound approximations for dynamic pruning. *TOIS*, 29(4):17:1–17:28, 2011.

[25] J. Mackenzie, F. M. Choudhury, and J. S. Culpepper. Efficient location-aware web search. In *ADCS*, pages 4:1–4:8, 2015.

[26] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *TOIS*, 14(4):349–379, 1996.

[27] A. Moffat and J. Zobel. Rank-biased precision for measurement of retrieval effectiveness. *ACM Transactions on Information Systems*, 27(1):2:1–2:27, 2008.

[28] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *IP&M*, 30(6):733–744, 1994.

[29] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *JASIS*, 47(10):749–764, 1996.

[30] M. Petri, J. S. Culpepper, and A. Moffat. Exploring the magic of WAND. In *ADCS*, pages 58–65, 2013.

[31] M. Petri, A. Moffat, and J. S. Culpepper. Score-safe term dependency processing with hybrid indexes. In *SIGIR*, pages 899–902, 2014.

[32] K. A. Ross, J. Cieslewicz, J. Rao, and J. Zhou. Architecture sensitive database design: Examples from the Columbia group. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 28(2):5–10, 2005.

[33] D. Shan, S. Ding, J. He, H. Yan, and X. Li. Optimized top-k processing with global page scores on block-max indexes. In *WSDM*, pages 423–432, 2012.

[34] A. Trotman. Compression, SIMD, and postings lists. In *ADCS*, pages 50:50–50:57, 2014.

[35] A. Trotman, X.-F. Jia, and M. Crane. Towards an efficient and effective search engine. In *Workshop on Open Source Information Retrieval*, pages 40–47, 2012.

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