Fast Disjunctive Candidate Generation Using Live Block Filtering

Antonio Mallia
antonio.mallia@nyu.edu
New York University
New York, US

Michał Siedlaczek
michal.siedlaczek@nyu.edu
New York University
New York, US

Torsten Suel
torsten.suel@nyu.edu
New York University
New York, US

ABSTRACT

A lot of research has focused on the efficiency of search engine query processing, and in particular on disjunctive top-k queries that return the highest scoring k results that contain at least one of the query terms. Disjunctive top-k queries over simple ranking functions are commonly used to retrieve an initial set of candidate results that are then reranked by more complex, often machine-learned rankers. Many optimized top-k algorithms have been proposed, including MaxScore, WAND, BMW, and JASS. While the fastest methods achieve impressive results on top-10 and top-100 queries, they tend to become much slower for the larger k commonly used for candidate generation.

In this paper, we focus on disjunctive top-k queries for larger k. We propose new algorithms that achieve much faster query processing for values of k up to thousands or tens of thousands. Our algorithms build on top of the live-block filtering approach of Dimopoulos et al [12], and exploit the SIMD capabilities of modern CPUs. We also perform a detailed experimental comparison of our methods with the fastest known approaches, and release a full model implementation of our methods and of the underlying live-block mechanism, which will allows others to design and experiment with additional methods under the live-block approach.

ACM Reference Format:
Antonio Mallia, Michał Siedlaczek, and Torsten Suel. 2021. Fast Disjunctive Candidate Generation Using Live Block Filtering. In *Proceedings of the Fourteenth ACM International Conference on Web Search and Data Mining (WSDM '21), March 8–12, 2021, Virtual Event, Israel*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3437963.3441813

1 INTRODUCTION

Large search engines need to answer tens of thousands of user queries per second with latencies of fractions of a second. Such engines incur significant hardware and energy costs to meet demands, motivating research on techniques that can increase query processing efficiency. In particular, a lot of work has focused on the disjunctive top-k query processing problem, where given a query we have to retrieve the k results with highest score among all documents containing at least one of the query terms. Here, the scoring function is typically a simple aggregation of per-term impact scores, where impact scores are either precomputed and stored in the index, or computed at run-time from data stored in the index.

Many optimized algorithms for disjunctive top-k query processing have been proposed, including MaxScore [34], WAND [4], various methods based on block-max scores [5, 9, 12–14, 16, 23, 30], and the recent JASS [18] approach. The fastest state-of-the-art methods use various optimizations in order to avoid accessing and scoring most of the index postings for the query terms, and achieve very low latencies for smaller values of k – typically under 10ms for $k = 10$ on the GOV2 and ClueWeb09B collections of 25M and 50M documents, respectively. They tend to be less efficient for larger k, such as $k = 1{,}000$ or $k = 10{,}000$, where they require up to an order of magnitude more time.

However, this case is now arguably more important in practice than the case of small k, since disjunctive top-k queries are primarily used in current search engines to retrieve an initial set of candidate results that is then reranked using more complex machine-learned ranking functions. Under this approach, often called cascade ranking [3, 6, 19, 20, 35, 37], the initial candidate set contains anywhere from several hundred to more than ten thousand candidates, thus leading to poor performance for existing algorithms.

In this paper, we propose new methods for disjunctive top-k query processing that obtain significant performance improvements for larger values of k, up to at least $k = 10{,}000$. Our algorithms are based on the so-called *live-block* filtering approach proposed in [12], which uses SIMD operations on fine-grain block-max data to determine a relatively small set of *live blocks*, which are areas of the index that might contain top-k results. In particular, we propose one method that integrates the MaxScore algorithm into the live-block approach, and another that applies a novel SIMD-optimized version of brute-force Term-at-a-Time (TAAT) to live blocks. We also provide a detailed experimental comparison with the fastest known algorithms that shows significant improvements for the new methods, and release a complete implementation of the algorithms and underlying live-block mechanism.

The remainder of this paper is organized as follows. Next, we provide background and discuss related work. Section 3 describes our new methods and their implementation. Section 4 describes the experimental setup, and Section 5 presents our experimental results. Finally, Section 6 provides some concluding remarks.

2 BACKGROUND AND RELATED WORK

We now provide technical background and discuss related work. Throughout this paper, we assume a document collection C, where each document is identified by an integer document ID (docID).

WSDM '21, March 8–12, 2021, Virtual Event, Israel
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8297-7/21/03…$15.00
https://doi.org/10.1145/3437963.3441813

2.1 Indexes and Scoring Functions

An inverted index is a data structure commonly used in search engines that stores for each term the IDs of the documents containing the term. More precisely, an inverted index consists of inverted lists, one for each distinct term in the collection, where each inverted list is a sequence of index postings. Each posting contains the docID of a document containing the term, plus some additional information about the occurrences of the term in the document. Unless stated otherwise, we assume that postings in each list are sorted by docID, enabling the use of efficient compression algorithms. This is the most common approach in large-scale search engines; see, e.g., [10].

We consider two posting formats in this paper. In one case, postings are of the form (d, f) where f is the number of times the term occurs in document d. This is a common choice, as many well-known scoring functions such as cosine measures, BM25 [29], and basic language modeling approaches [1, 7, 15, 38] can be efficiently computed from the frequency and some extra data about inverted list and document lengths.

In the second case, a posting is of the form (d, s) where s is a precomputed and suitably quantized impact score, and the score of a document with respect to the query is the sum of its impact scores over the query terms. The advantage of this format is that queries typically run much faster, as we avoid computation of impact scores at query time. (This also allows for machine-learned impact scores that would be infeasible to compute at query time.) On the other hand, it requires us to fix the scoring function at indexing time, and there is some possible loss in precision due to the use of quantized, and thus approximated, impact scores. We refer to the first case as a standard index, and the second one as a quantized index. We mostly focus on quantized indexes in this paper.

2.2 Disjunctive Top-k Query Processing

Given an inverted index and a query q, the disjunctive top-k query processing problem requires us to find the k documents with the highest score with respect to q (with suitable tie breaks, say based on docID). This problem can clearly be solved by exhaustively traversing the inverted lists for the query terms and computing document scores from the frequencies or impact scores, and a number of traversal methods including Term-at-a-Time (TAAT) and Document-at-a-Time (DAAT) have been proposed (see, e.g., [39]).

We focus on optimized methods that find the top-k results without exhaustive traversal of the inverted lists. This is often called *dynamic pruning* or *early termination*, and has been studied extensively over the last two decades. Algorithms that use early termination are considered *safe* if they always return the same top-k results as an exhaustive traversal of the inverted lists, and *unsafe* otherwise. Note that under this definition, a method using a quantized index is considered safe if it returns the same result as an exhaustive traversal of the quantized index structures – while the process of quantization itself introduces an approximation and might result in a reduction in result quality if not done carefully, this is a separate concern. (We also note that ranking functions such as BM25 or language modeling are themselves derived under various simplifying assumptions and approximations.)

Our goal here is to improve the efficiency of safe disjunctive top-k processing, particularly for larger values of k. Many safe and highly optimized algorithms for this problem that have been proposed. Frequently studied approaches are MaxScore [34], WAND [4], and Block-Max WAND [14] and various related methods [5, 9, 12, 13, 23, 24, 30, 31] that use block-wise maximum impact scores to skip parts of the index that cannot contain any top-k results.

A somewhat different approach was taken in the recent JASS algorithm [18], which uses a Score-at-a-Time (SAAT) approach [2] where index postings are organized by quantized impact scores, and accessed in this order during query processing. JASS can be used as an exhaustive algorithm, or early-terminated at any time during execution, resulting in an approximate, but unsafe, result. Follow-up work in [8] compared JASS to the WAND and BMW approaches, and observed that for large k JASS might be a good alternative as its running time does not increase significantly with k. Also, running times for BMW were observed to be somewhat unpredictable, with some otherwise harmless looking queries taking a long time, while JASS exhibited a more stable behavior.

The comparison in [8] motivated our effort here to improve the running time of safe early-termination approaches for large k. We will compare our new methods to an aggressively early-terminated but unsafe version of JASS, the MaxScore and VBMW approaches, which were the fastest in the recent experimental comparison study in [26], and the BMW-LB method in [12], which is the most similar previous approach and also based on the live-block mechanism.

2.3 Thresholds and Document Orderings

We briefly describe two additional methods for speeding up query processing, document reordering and threshold estimation. Document reordering is the idea of assigning docIDs to documents in an optimal manner. It has been shown that assigning docIDs based on URL ordering [32] or recursive bipartite partitioning [11] can significantly decrease both index size and query processing times.

Threshold estimation is the problem of estimating the top-k threshold for a query in time significantly less than the cost of issuing the query. It is known that having a good initial estimate for the threshold of a query leads to significantly faster execution times for many algorithms, compared to the case where the threshold starts from zero and slowly increases as higher-scoring documents are discovered during index traversal [14, 27, 28, 36].

Table 1: Performance comparison of query time (in ms) for MaxScore and VBMW w.r.t TREC 2005 and 2006 queries on GOV2, for $k = \{10, 1000, 10000\}$ in three different settings: unquantized index with initial threshold 0 (U), quantized index with initial threshold 0 (Q), and quantized index with initial threshold (Q_T) estimated using the Q_k^3-*log* method.

	TREC 2005			TREC 2006		
	U	Q	Q_T	U	Q	Q_T
			Top-10			
MaxScore	4.29	1.92	1.64	6.38	3.11	2.88
VBMW	1.96	1.85	1.57	4.51	4.08	3.85
			Top-1,000			
MaxScore	10.07	6.07	4.18	14.83	9.53	7.04
VBMW	8.29	7.31	5.05	16.23	14.63	11.56
			Top-10,000			
MaxScore	21.78	16.59	9.78	32.11	25.72	17.02
VBMW	23.78	20.93	12.50	40.98	37.76	26.77

In our implementation and experimental comparison, we use document reordering by recursive graph bisection (BP) [11], using the implementation in [21], and threshold estimation using the safe Q_k^3-*log* method in [27] (building on an approach in [36]). For fairness, we also integrate these techniques into all the methods we compare to, with the exception of JASS where thresholds are not useful and where the global ordering is only applied within each impact score segment.

Very recently, Mackenzie and Moffat [22] showed how threshold estimation and quantization can speed up query processing and, in the process, reduce the improvement obtained by block max-based methods over other approaches. We show in Table 1 for our setup how quantization and (to a lesser degree) threshold estimation give larger benefits for a simpler algorithm such as MaxScore, resulting in MaxScore matching and often outperforming an optimized VBMW implementation, especially for larger k and query logs containing longer queries (TREC 2006).

2.4 Live-Block Filtering

Our approach is based on the *live-block filtering* approach proposed in [12], and thus we describe this approach briefly. Recall that in block max-based approaches, we store a maximum impact score for each block of consecutive postings in an inverted list, and these *block maxes* are used to skip over postings that cannot result in a score above the top-k threshold. Here, each block might contain a fixed number of postings, say 32 or 128 [5, 14], or a variable number of consecutive postings with similar impact scores [23]. Smaller blocks allow us to skip more postings during index traversal. However, they also lead to additional overheads as block boundaries are not aligned in docID space between the different inverted lists.

As an alternative, [12, 13] proposed to align the block boundaries between different lists, by defining block boundaries in terms of docIDs. In particular, we choose a global parameter b, and store a maximum impact score for each group of consecutive postings in an inverted list whose docIDs only differ in the least significant b bits. Thus, for each inverted list we get an array of $\lceil |C|/2^b \rceil$ maximum impact scores, where $|C|$ is the number of documents in the collection. Given a query and a current estimate for its top-k threshold – the value needed for a result to make it into the top k – we say that a block is *dead* if the sum of the corresponding block maxes for the query terms is less than the threshold, and *live* otherwise. Thus, we can perform a vector add operation between the block-max arrays of the query terms to determine which blocks are live or dead. This operation can be performed extremely efficiently using SIMD command sets such as SSE, AVX2, or AVX-512 on current CPUs, allowing use of fairly small values of b with little slowdown.

The problem with this idea is that it would seem to require storing an array of $\lceil |C|/2^b \rceil$ maximum impact scores for every inverted list, including very short lists where most of the block maxes are zero. This is addressed in [12, 13] by only precomputing the block-max arrays for lists above a certain length and/or frequently occurring in queries, and materializing the other lists on-the-fly from the index as needed. It was shown in [13] that this can support blocks as small as $2^5 = 32$ with space overhead less than half of the index size, and with average on-the-fly materialization costs of a few hundred microseconds per query.

As described in [12], the entire live-block mechanism of precomputed block-max arrays, on-the-fly materialization, and vector-add live-block computation can be implemented as a lower-level black-box mechanism that gives higher-level query processing algorithms access to live-block information and block maxes without having to deal with the underlying complexities. In fact, as shown in [12], a number of well-known query processing algorithms can be significantly accelerated by simply replacing the standard forward seek mechanism in inverted lists (sometimes called nextGEQ()) with an implementation that seeks forward to the next posting contained in a live block, leaving the algorithm itself unchanged. In our experiments, we compare to BMW-LB, the fastest of the algorithms in [12], which uses this modified forward seek as well as direct access to the block maxes in the arrays.

In preliminary experiments in Figure 1, we show the percentage of live blocks for different block sizes and values of k, for random and BP document reordering and for estimated and real top-k thresholds. We see that BP reordering results in fewer live blocks, and that estimated thresholds do almost as well as the ideal (real) threshold. As expected, the percentage of live blocks increases with the block size and with k.

3 OUR METHODS AND IMPLEMENTATIONS

We now describe our two new methods, which we call Range-MaxScore and Range-DRAAT, and discuss some details of the implementation.

Range-MaxScore: This is a very simple approach that almost directly applies the MaxScore algorithm to the live blocks produced by the underlying live-block mechanism. That is, we apply an optimized MaxScore to each docID range of size 2^b that is *live*, where we separately select essential lists for each block based on block-max scores rather than per-list maxscores. We take some care to minimize any overheads in invoking MaxScore, since it is called many times on small amounts of data.

Range-DRAAT: In this approach, we basically apply an optimized Term-at-a-Time algorithm to each live block. Thus, in contrast to Range-MaxScore, this method accesses all postings that exist in a live block, but saves the overhead of selecting and handling essential lists.

One aspect that distinguishes our version of TAAT is the use of SIMD commands for handling the direct-mapped accumulator array. As shown in Figure 1, the live-block mechanism works best for small values of b, such as $b = 5$ and $b = 6$. For these cases, the direct-mapped accumulator arrays for Range-DRAAT, which have 2^b entries, become small enough to be handled inside the SIMD registers of current CPUs. This means that we can usually initialize the accumulators in one SIMD command, and later identify any accumulated values above the threshold in a few SIMD commands. Moreover, our approach does not use a top-k heap, but instead collects any results above the (estimated) threshold in a memory-based array that is sorted at the end to obtain the top-k.

More precisely, given large enough vector registers, in a single operation we can compare all the accumulator entries with the threshold to identify any entries above the threshold, and generate a corresponding bitmask. Next, we can prune the non-exceeding elements in the SIMD vector, using table lookups and an efficient shuffle instruction, such that the exceeding ones become compacted and can then be copied into the result array. In the end, a highly

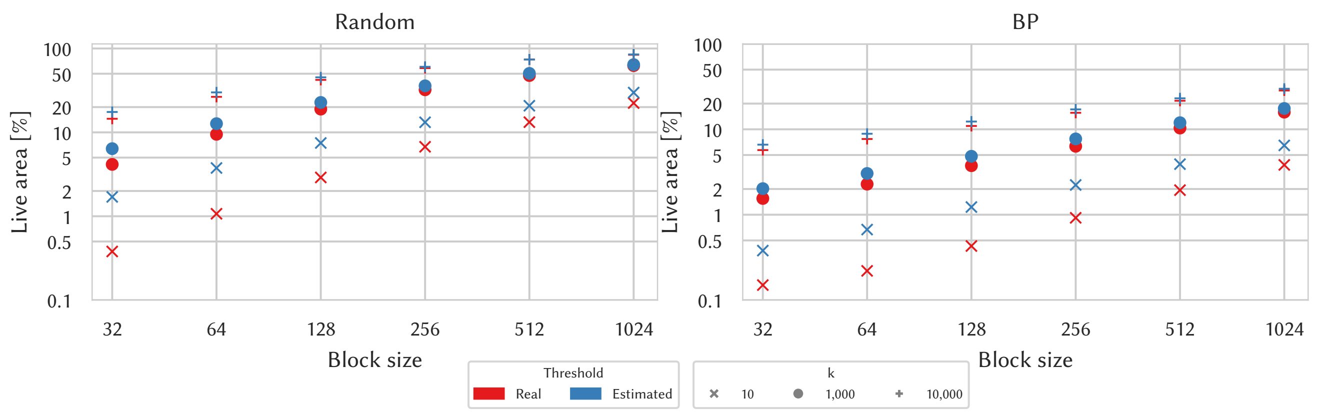

Figure 1: Average percentage of live blocks per query for several block sizes w.r.t. TREC 2005 queries on GOV2 under random and BP document ordering, using real and estimated thresholds for $k = \{10, 1000, 10000\}$.

tuned radix sort is used to sort array entries in descending order by score.

Live-Block Implementation: We implemented the live-block mechanism as an extension to the PISA open-source platform for indexing and query processing research. Our implementation followed the description in [12], but we did not implement the additional *posting bits* mechanism, which according to [12] gave minor benefits for randomly ordered indexes but no benefits for the reordered index structures that we use here.

We experimented with different implementations of the live-block computation, a scalar one, one using SSE, one using AVX2, and one using the AVX-512 instruction set. As we show later, the latter two significantly outperformed the others. We also note that an early implementation of our methods took a long time on some of the queries for large values of k. After inspecting these cases, we noticed that these were queries with very short inverted lists, where the (estimated) disjunctive top-k threshold was zero. Thus, all blocks were identified as live, and significant overhead was incurred by calling Range-MaxScore and Range-DRAAT on blocks that had no postings at all. This issue was easily resolved by requiring live blocks to have a score strictly above zero, but it illustrates how subtle implementation details can have major consequences.

Threshold Estimation and Consequences: We implemented a top-k threshold estimation technique called Q_k^3*-log* recently described in [27], which builds on an approach in [36]. In this method, top-k thresholds are precomputed for all terms in the index, and for all pairs and triplets of terms that occur in a training set of queries, in our case a set of 10M queries from the AOL query log. Thresholds are precomputed by simply issuing a disjunctive top-k query for every selected term, pair, and triplet. At query time, the highest threshold of any available term, pair, or triplet contained in the query is used as the estimate. This is a safe estimate in that it will never overestimate the threshold; i.e., that there is no danger of having to rerun a query with a lower threshold due to obtaining less than k results above the estimated threshold.

We note that the recent progress in threshold estimation techniques enabled another unexpected simplification in our code. We already discussed how we were able to get rid of the top-k heap in Range-DRAAT. This was fairly easy to implement because our threshold estimate is strong enough to be used throughout the index traversal, i.e., with no need to refine the threshold after traversing part of the index. The same idea was also used in the live-block computation, where we use the initial threshold estimate to immediately compute the live blocks of the entire document range. This is in contrast to [12, 13], which starts with a very weak initial threshold (in fact, 0 in their experiments), and then performs several rounds of live block computation over parts of the document collection using progressively higher threshold values. Our solution resulted in a much simpler implementation that achieves almost the same number of live blocks as a solution that starts with a strong threshold estimate that is further increased over time.

Compression of block-max arrays: To efficiently store the block-max arrays, we implemented a simple but effective compression mechanism that works well on sparse block-max data and is also extremely fast to decompress. We define a *superblock* as consisting of 256 consecutive block-max values. For each superblock we store the number of block maxes greater than zero, and for each such block max we store a pair (p, s), where p is the offset in the superblock and s is the block-max score. As shown later, for certain list lengths, this takes less space than storing a precomputed block-max vector and is much faster to convert back into a block-max vector than on-the-fly computation. (We note that this method is different from, and faster than, a method for block-max compression described in [12].)

4 EXPERIMENTAL SETUP

We now describe our setup and some more implementation details.

Testing Setup All algorithms are implemented in C++17 and compiled with GCC 7.5.0 with the highest optimization settings. In particular, we compare our methods to the following previously known algorithms: MaxScore, VBMW, BMW-LB, and JASS. For VBMW, the variable-block sizes depend on a parameter λ; we apply binary search over λ to obtain an average block size of 40 ± 0.5. For JASS, we used the original implementation[1] from the authors with a fairly aggressive early termination policy where we stop

[1]https://github.com/andrewtrotman/JASSv2/

after accessing just 10% of the index postings, as suggested in [18]; note that this is an unsafe early termination method, while all other methods are safe. For BMW-LB and Range-MaxScore, we set the size of the blocks of the block-max arrays to 128; for Range-DRAAT we use 32 instead. These choices minimized the average query processing costs for each method.

All algorithms were provided with a strong initial threshold estimate based on the Q_k^3-*log* method in [27], with the exception of JASS where this does not apply.

The experiments were conducted on a single core of a machine with Intel Core i7-7820X Skylake cores clocked at 3.60GHz, with 64GB RAM, running GNU/Linux 4.15. The CPU supports Intel SSE4.2, Intel AVX2 and Intel AVX-512 instruction-set extensions.

Indexes: All indexes were saved to disk after construction, and memory mapped to be queried, so that there are no hidden space costs due to loading of additional data structures in memory. The posting lists are compressed using SIMD-BP128 [17], which was shown in [26] to provide a good trade-off between size and speed. JASS uses an inverted index compressed with Group Elias Gamma SIMD [33] that is significantly larger than the inverted index used by the other methods. Before timing the queries, we ensure that the required posting lists were fully loaded in memory.

Unless specified otherwise, indexes were reordered using BP [11, 21], and were build as quantized indexes using 8-bit linear quantization. Block maxes used for live-block computation were also quantized in the same way. Note that in principle we could use different quantizations for indexes and block maxes, or even use unquantized inverted indexes with quantized block maxes, where we would need to round up block-max values to the next quantile.

Documents and Queries: We performed experiments on three standard document collections, GOV2, ClueWeb09B and ClueWeb12B. The terms in the collection were lowercased and stemmed using the Porter2 stemmer; no stopwords were removed. Table 2 summarizes basic statistics of the collections and the sizes of the resulting compressed inverted indexes (used by all the algorithms with the exception of JASS).

Table 2: Basic statistics for the test collections.

	Documents	Terms	Postings	Index Size
GOV2	25,205,179	32,407,061	5,426,935,226	8.25 GB
CW09B	50,220,110	87,896,391	15,426,727,424	27.54 GB
CW12B	52,343,021	133,248,235	14,620,401,885	26.69 GB

To evaluate query processing speed, we use TREC 2005 and TREC 2006 Terabyte Track Efficiency Task queries. From each sets of queries, we randomly selected 1,000 queries.

Our source code is publicly available[2] for readers interested in further implementation details or in replicating the experiments. Our code is based on the PISA platform [25].

5 EXPERIMENTAL RESULTS

In this section we analyze the performance of Range-MaxScore and Range-DRAAT with an extensive experimental evaluation in a reproducible setting, using state-of-the-art baselines, standard benchmark text collections, and large query logs.

[2]https://github.com/pisa-engine/pisa/tree/live-block

Most of the preliminary experiments related to live-block computation are performed on GOV2 using TREC 2005 query log. Later ClueWeb09B and ClueWeb12B with TREC 2005 and TREC 2006 query logs are added to fully evaluate our algorithms, as they are more representative of real-world web search collections.

5.1 Live-block Computation

Space overhead and computational time trade-offs. Recall that we precompute and store block maxes for all inverted lists above a certain length $\mathcal{L}_{min}$, and compute the block-max arrays on-the-fly for shorter lists. Table 3 shows the amount of space required to store (uncompressed) precomputed block-max arrays, given a minimum list length. As expected, small minimum list sizes and small block sizes result in large space requirements for the block maxes, often multiples of the index size.

Next, Table 4 shows the average time per query spent on materialization of block-max arrays, using on-the-fly computation for lists shorter than $\mathcal{L}_{min}$. Thus, by choosing an appropriate value of $\mathcal{L}_{min}$ in Tables 3 and 4, we can trade off time and space. For example, for Range-DRAAT we use blocks of size 32, so if we set $\mathcal{L}_{min}$ in Tables 3 and 4 to 2^{18}, we would need 2.55 GB of space and an average of 471 microseconds overhead per query. We could decide to use more space and reduce the time overhead, by setting $\mathcal{L}_{min} = 2^{14}$, which would require 12.24 GB of additional memory but only an average of 70 microseconds overhead per query.

Table 3: Space overhead (in GB) to store precomputed block-max arrays for several block sizes with different minimum posting list lengths ($\mathcal{L}_{min}$), on GOV2.

	Block size					
$\mathcal{L}_{min}$	32	64	128	256	512	1024
2^{11}	55.26	27.63	13.82	6.91	3.46	1.73
2^{12}	30.23	15.12	7.56	3.78	1.89	0.95
2^{13}	19.06	9.53	4.77	2.38	1.19	0.60
2^{14}	12.24	6.12	3.06	1.53	0.77	0.38
2^{15}	8.02	4.01	2.00	1.00	0.50	0.25
2^{16}	5.68	2.84	1.42	0.71	0.36	0.18
2^{17}	4.12	2.06	1.03	0.52	0.26	0.13
2^{18}	2.55	1.28	0.64	0.32	0.16	0.08

We noticed that posting lists with lengths in the range $[2^{14}, 2^{18}]$ use a lot of space when precomputed, and a lot of time when computed on-the-fly. They are also somewhat sparse in that the majority of their block maxes are 0. Next, we explore what happens when we precompute and store these arrays using the compression format proposed in Section 3. Results are shown in Table 5, where the first row shows the compressed size of only these lists, and the second row the average cost per query of decompressing block-max arrays.

For our example with block size 32, we see that block-max compression can greatly reduce space requirements while keeping on-the-fly costs small. In particular, if we precompute block-max arrays for lists longer than 2^{18} using 2.55 GB, precompute and store compressed block maxes for lists in the range $[2^{14}, 2^{18}]$ using an additional 2.03 GB, and materialize block maxes on-the-fly for the remaining lists, we obtain a total space overhead of $2.55 + 2.03 = 4.58$ GB and an average materialization and decompression cost of $70 + 48 = 118\mu s$, much better than before.

Table 4: Average block-max on-the-fly computation time (in μs) per query, for several block sizes with different posting list lengths ($\mathcal{L}_{min}$), for TREC 2005 queries on GOV2.

	Block size					
$\mathcal{L}_{min}$	32	64	128	256	512	1024
2^{11}	26	12	5	3	3	2
2^{12}	34	15	7	5	4	3
2^{13}	46	21	11	8	6	5
2^{14}	70	34	19	16	14	13
2^{15}	105	57	38	34	30	29
2^{16}	167	104	77	70	68	66
2^{17}	280	205	168	156	150	149
2^{18}	471	385	344	329	324	319

Table 5: Space overhead (in GB) to store compressed block-max arrays, and time (in μs) to decompress them for several block sizes, for posting lists with lengths in the range $[2^{14}, 2^{18}]$ on GOV2.

	Block size					
	32	64	128	256	512	1024
Space	2.03	1.74	1.51	1.29	1.09	0.87
Time	46	28	19	12	7	5

Live-block computation. Next, we look at the cost of the actual live-block computation once the block-max arrays for a query are materialized. The data in Table 6 indicates that the use of SIMD instructions is absolutely essential to efficiently compute the live-block bit-vector. On the other hand, the difference between the SIMD instruction sets was less pronounced, with even SSE obtaining decent results. Finally, we note that Table 6 shows that costs increase significantly with smaller block sizes. For block size 32 we have 8 times more blocks to add up than for block size 256, but the cost increases by a factor of about 25. The reason appears to be cache effects – for larger block sizes the entire block-max array may already reside in memory after on-the-fly computation or decompression, while for the smaller block sizes the array does not fit into the same cache level.

Table 6: Average live-block computation time per-query (in μs) for TREC 2005 queries on the GOV2 collection, using different instruction set extensions.

	Block size					
	32	64	128	256	512	1024
Scalar	2,444	1,220	610	304	152	75
SSE	289	155	60	22	10	5
AVX2	199	81	36	12	6	3
AVX-512	180	67	27	7	4	2

5.2 Overall Comparison

Next, we compare all methods in terms of overall efficiency, for different query lengths. The results are presented in Table 7, which shows average query latencies for $k = \{10, 1000, 10000\}$, for two query logs on three different datasets. We observe that JASS is substantially slower than the other methods, with some exceptions for queries longer than 5 terms on the GOV2 dataset, and for TREC06 with $k = 10{,}000$. VBMW and MaxScore perform similarly, with the former being more efficient for short queries and small k values. Overall, our new methods, Range-MaxScore and Range-DRAAT, are significantly faster than previous methods in most cases. In particular, Range-MaxScore is fast for small k values, while Range-DRAAT excels on larger values.

In Figure 3 we summarize query latencies for all techniques across all collections, and for several k values, in Tukey's box-and-whisker plots. For most of the algorithms, the spans covered by the whiskers are quite broad, indicating high variance in query latencies. Range-MaxScore and particularly Range-DRAAT exhibit a more predictable behaviour than most other methods.

Finally, in Figure 2 we plot the running times of the methods for different k values, ranging from $k = 10$ up to 100,000. We see that Range-DRAAT significantly outperforms all other methods for the largest values of k. Surprisingly, while JASS does well for $k = 10{,}000$, its performance degrades significantly as we increase k further. We caution here that we are not sure why this is happening, and it may be that JASS was not built for such extreme values.

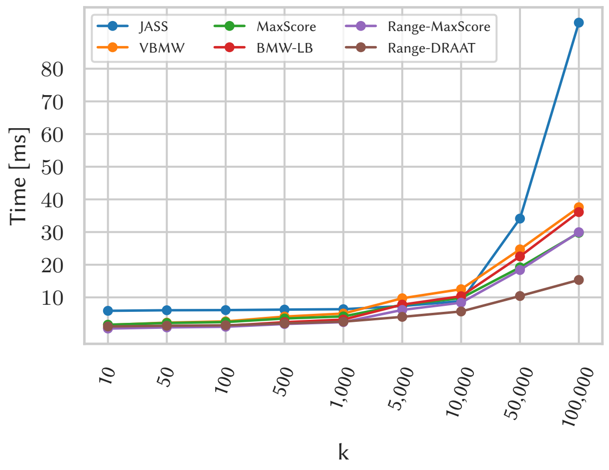

Figure 2: Visualization of average query latencies (in ms) of different query processing strategies for TREC 2005 queries on GOV2, for different k values.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we studied algorithms for safe disjunctive top-k query processing, and proposed two new methods based on the live-block approach in [12]. Our experimental results show that these methods achieve significant improvements over the best previously proposed approaches. We also provide a model implementation of our methods, and the underlying live-block mechanism, in the PISA framework that will allow others to easily experiment with new algorithms under the live-block approach.

Our work leaves a number of open questions for future research. For example, one could try to further improve the speed of the Range-DRAAT approach by designing new compressed index formats that enable the use of SIMD commands for score accumulation, or that support faster on-the-fly creation of block-max arrays. There is also the potential to design other types of algorithms under the live-block approach that outperform our methods.

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

Table 7: Query times (in ms) of different query processing strategies for several query lengths and average query times w.r.t. TREC 2005 and TREC 2006 Terabyte Track Efficiency Task queries on GOV2, ClueWeb09B and ClueWeb12B for $k = \{10, 1000, 10000\}$.

			TREC05						TREC06					
			2	3	4	5	6+	Avg	2	3	4	5	6+	Avg
GOV2	Top-10	JASS$_{10\%}$	3.11	6.17	7.76	9.03	11.36	5.89	3.33	6.54	7.98	9.58	11.76	7.63
		VBMW	0.60	1.11	1.50	2.61	6.23	1.57	0.49	1.58	2.44	3.44	10.97	3.77
		MaxScore	1.44	1.40	1.45	2.24	2.88	1.64	1.25	2.01	2.47	2.88	5.42	2.81
		BMW-LB	0.23	0.37	0.54	1.16	2.49	0.61	0.21	0.61	1.20	1.74	5.48	1.83
		Range-MaxScore	**0.16**	**0.29**	**0.46**	**0.89**	**1.86**	**0.46**	**0.16**	**0.48**	**0.88**	**1.33**	**3.63**	**1.28**
		Range-DRAAT	0.29	0.50	0.80	1.78	6.13	1.10	0.36	0.95	1.83	3.07	13.17	3.83
	Top-1,000	JASS$_{10\%}$	3.48	6.70	8.43	9.57	11.96	6.38	4.20	7.20	9.22	11.04	12.79	8.61
		VBMW	2.01	3.73	4.76	7.56	20.11	5.05	2.16	4.55	7.30	11.54	32.92	11.57
		MaxScore	2.96	3.49	4.25	5.94	9.59	4.18	2.89	4.22	5.59	7.63	14.75	6.97
		BMW-LB	1.27	2.34	3.01	5.10	12.61	3.21	1.46	2.86	4.97	7.94	22.01	7.75
		Range-MaxScore	1.05	1.89	2.62	**4.03**	**8.08**	**2.43**	1.12	2.47	3.92	**5.84**	**12.77**	**5.18**
		Range-DRAAT	**0.73**	**1.37**	**2.13**	4.13	13.14	2.57	**0.87**	**1.78**	**3.36**	5.93	19.93	6.28
	Top-10,000	JASS$_{10\%}$	4.57	8.99	11.80	14.20	**17.60**	8.87	5.23	9.21	11.99	15.14	**17.62**	11.42
		VBMW	6.52	10.94	12.19	16.89	39.46	12.50	6.21	23.28	17.16	24.01	63.54	27.19
		MaxScore	6.72	9.85	10.06	12.46	20.13	9.78	6.48	19.34	12.23	15.42	29.39	16.95
		BMW-LB	5.86	9.68	9.89	13.73	29.57	10.36	5.56	23.19	14.36	19.18	49.69	22.87
		Range-MaxScore	5.15	8.47	8.28	11.21	19.92	8.38	4.63	20.05	11.53	14.41	28.94	16.40
		Range-DRAAT	**2.28**	**3.91**	**5.71**	**8.92**	22.33	**5.69**	**2.35**	**5.34**	**7.17**	**11.48**	30.06	**11.18**
ClueWeb09B	Top-10	JASS$_{10\%}$	18.52	29.89	37.57	45.11	48.47	29.47	14.29	28.41	34.45	41.38	46.28	28.61
		VBMW	3.00	6.49	6.87	12.19	29.09	7.73	3.24	5.22	9.81	14.72	39.26	14.27
		MaxScore	4.75	5.09	4.96	6.87	8.72	5.42	3.55	4.60	6.45	8.15	14.35	7.37
		BMW-LB	0.66	1.64	2.05	3.64	9.43	2.20	0.81	1.45	3.33	5.24	16.15	5.32
		Range-MaxScore	**0.46**	**1.08**	**1.83**	**3.09**	**6.79**	**1.65**	**0.59**	**1.24**	**2.92**	**4.57**	**13.57**	**4.52**
		Range-DRAAT	0.86	2.18	3.63	6.76	31.22	4.99	1.01	2.68	6.11	11.02	50.23	14.01
	Top-1,000	JASS$_{10\%}$	18.72	30.16	37.70	45.46	48.68	29.69	14.72	30.00	34.84	41.91	46.63	29.63
		VBMW	8.50	15.69	20.03	34.73	74.68	20.61	10.42	13.97	26.37	40.03	103.77	38.34
		MaxScore	8.12	9.24	11.89	16.46	23.01	11.12	8.44	9.05	14.37	20.20	**36.81**	17.51
		BMW-LB	3.28	6.43	9.53	18.00	36.42	9.45	6.64	6.62	14.09	22.16	59.10	21.31
		Range-MaxScore	2.62	5.16	8.78	**14.81**	**23.00**	**7.29**	5.28	5.94	12.56	**19.05**	41.29	**16.52**
		Range-DRAAT	**1.91**	**4.39**	**7.66**	15.15	54.46	9.53	**2.35**	**4.91**	**11.53**	21.10	74.01	22.39
	Top-10,000	JASS$_{10\%}$	20.35	33.35	42.24	50.83	55.08	32.94	15.41	30.97	37.99	45.79	**51.26**	**31.83**
		VBMW	16.32	28.90	37.61	62.67	135.99	38.09	18.40	31.21	46.12	74.97	190.28	71.28
		MaxScore	13.17	17.15	22.99	32.59	**47.60**	20.65	14.18	21.46	26.14	38.67	69.53	33.69
		BMW-LB	9.35	17.44	24.44	43.02	83.75	23.67	12.71	22.79	31.13	50.76	126.93	48.31
		Range-MaxScore	8.41	14.60	22.79	35.87	56.58	19.14	10.85	20.98	27.88	42.54	84.74	37.09
		Range-DRAAT	**4.88**	**10.04**	**16.33**	**31.45**	86.16	**17.93**	**5.06**	**11.15**	**21.04**	**37.38**	111.35	36.62
ClueWeb12B	Top-10	JASS$_{10\%}$	16.03	27.92	34.12	40.08	46.44	26.78	12.46	25.84	31.21	37.84	43.75	25.95
		VBMW	1.90	4.62	5.39	9.39	24.81	5.93	1.53	3.82	7.72	11.75	32.25	11.29
		MaxScore	4.17	4.46	4.29	6.22	7.83	4.79	3.04	4.03	5.72	6.89	13.04	6.51
		BMW-LB	0.39	1.06	1.37	2.59	6.71	1.50	0.45	1.03	2.39	3.89	12.60	4.02
		Range-MaxScore	**0.31**	**0.77**	**1.20**	**2.23**	**5.22**	**1.19**	**0.38**	**0.90**	**2.16**	**3.35**	**10.75**	**3.46**
		Range-DRAAT	0.65	1.56	2.61	4.70	23.58	3.69	0.76	1.90	4.45	7.42	37.03	10.19
	Top-1,000	JASS$_{10\%}$	16.38	28.22	34.46	40.53	46.80	27.12	12.59	26.45	31.89	38.48	44.13	26.20
		VBMW	7.32	12.14	15.40	26.94	61.94	16.64	6.37	10.00	19.91	30.08	80.29	28.94
		MaxScore	8.03	8.21	9.82	13.86	20.94	10.08	5.95	7.69	12.60	17.14	**31.89**	14.89
		BMW-LB	2.73	4.76	6.85	12.41	28.25	7.13	3.57	4.67	10.18	15.59	44.54	15.47
		Range-MaxScore	2.17	4.01	6.17	**10.24**	**18.58**	**5.51**	2.81	4.41	9.49	**13.93**	32.68	**12.49**
		Range-DRAAT	**1.71**	**3.37**	**5.68**	10.55	42.89	7.39	**1.82**	**3.58**	**8.39**	14.36	57.12	16.80
	Top-10,000	JASS$_{10\%}$	17.62	30.79	38.24	45.34	52.31	29.80	13.65	28.43	34.50	42.04	**49.46**	28.21
		VBMW	13.30	23.61	29.48	49.23	113.70	30.98	11.97	29.37	36.76	55.89	145.35	55.59
		MaxScore	12.13	15.76	19.93	28.21	**43.05**	18.58	11.43	23.43	22.92	31.94	59.16	29.83
		BMW-LB	7.64	13.59	18.90	32.02	67.40	18.64	8.13	24.46	23.81	36.72	94.38	37.52
		Range-MaxScore	6.97	12.02	17.54	26.66	46.52	15.33	7.04	22.60	22.00	32.47	67.81	30.50
		Range-DRAAT	**4.06**	**7.91**	**12.74**	**22.39**	71.01	**14.28**	**4.02**	**8.92**	**16.00**	**26.83**	86.40	**28.07**

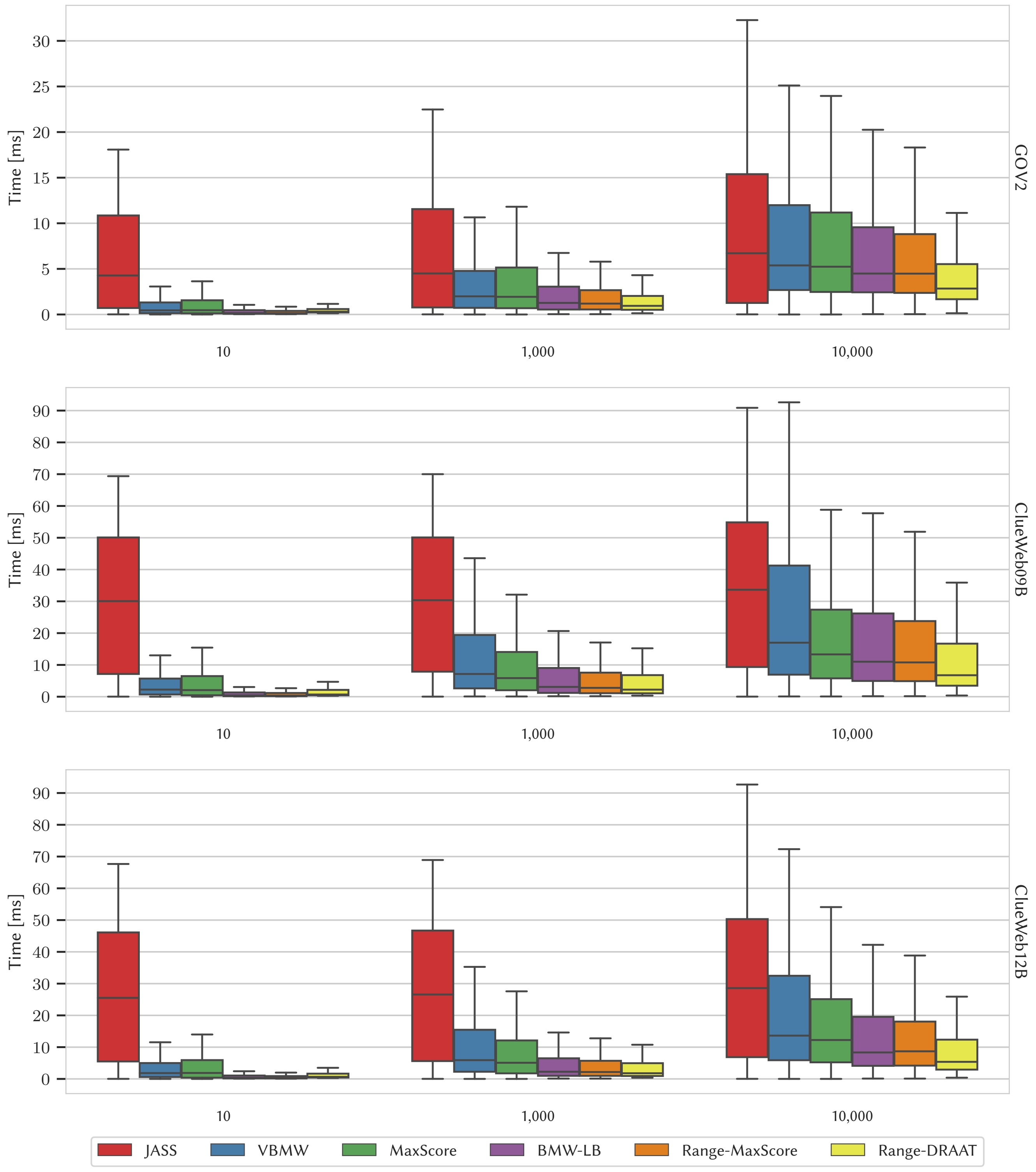

Figure 3: Visualization of query latencies (in ms) using Tukey's box-and-whisker plots of different query processing strategies w.r.t. TREC 2005 queries on GOV2, ClueWeb09B, ClueWeb12B for $k = \{10, 1000, 10000\}$.

REFERENCES

[1] Gianni Amati and Cornelis Joost Van Rijsbergen. 2002. Probabilistic models of information retrieval based on measuring the divergence from randomness. *ACM Transactions on Information Systems (TOIS)* 20, 4 (2002), 357–389.

[2] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. 2001. Vector-space ranking with effective early termination. In *Proceedings of the 24th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 35–42.

[3] Nima Asadi and Jimmy Lin. 2013. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 997–1000.

[4] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth International Conference on Information and Knowledge Management.* 426–434.

[5] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. 2011. Interval-based pruning for top-k processing over compressed lists. In *2011 IEEE 27th International Conference on Data Engineering.* 709–720.

[6] Ruey-Cheng Chen, Luke Gallagher, Roi Blanco, and J Shane Culpepper. 2017. Efficient cost-aware cascade ranking in multi-stage retrieval. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 445–454.

[7] Stéphane Clinchant and Eric Gaussier. 2010. Information-based models for ad hoc IR. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval.* 234–241.

[8] Matt Crane, J Shane Culpepper, Jimmy Lin, Joel Mackenzie, and Andrew Trotman. 2017. A comparison of document-at-a-time and score-at-a-time query evaluation. In *Proceedings of the tenth ACM International Conference on Web Search and Data Mining.* 201–210.

[9] Lídia Lizziane Serejo de Carvalho, Edleno Silva de Moura, Caio Moura Daoud, and Altigran Soares da Silva. 2015. Heuristics to improve the BMW method and its variants. *Journal of Information and Data Management* 6, 3 (2015), 178–178.

[10] Jeffrey Dean. 2009. Challenges in Building Large-Scale Information Retrieval Systems: Invited Talk. In *Proceedings of the second ACM International Conference on Web Search and Data Mining.*

[11] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* 1535–1544.

[12] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. 2013. A candidate filtering mechanism for fast top-k query processing on modern cpus. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 723–732.

[13] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. 2013. Optimizing top-k document retrieval strategies for block-max indexes. In *Proceedings of the sixth ACM International Conference on Web Search and Data Mining.* 113–122.

[14] Shuai Ding and Torsten Suel. 2011. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 993–1002.

[15] Hui Fang and ChengXiang Zhai. 2005. An exploration of axiomatic approaches to information retrieval. In *Proceedings of the 28th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 480–487.

[16] Omar Khattab, Mohammad Hammoud, and Tamer Elsayed. 2020. Finding the Best of Both Worlds: Faster and More Robust Top-k Document Retrieval. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval.* 1031–1040.

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

[18] Jimmy Lin and Andrew Trotman. 2015. Anytime ranking for impact-ordered indexes. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval.* 301–304.

[19] T-Y. Liu. 2009. Learning to Rank for Information Retrieval. 3, 3 (2009), 225–331.

[20] C. Macdonald, R. L. Santos, and I. Ounis. 2013. The Whens and Hows of Learning to Rank for Web Search. 16, 5 (2013), 584–628.

[21] Joel Mackenzie, Antonio Mallia, Matthias Petri, J Shane Culpepper, and Torsten Suel. 2019. Compressing inverted indexes with recursive graph bisection: A reproducibility study. In *European Conference on Information Retrieval.* 339–352.

[22] 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.*

[23] Antonio Mallia, Giuseppe Ottaviano, Elia Porciani, Nicola Tonellotto, and Rossano Venturini. 2017. Faster BlockMax WAND with variable-sized blocks. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 625–634.

[24] Antonio Mallia and Elia Porciani. 2019. Faster BlockMax WAND with longer skipping. In *European Conference on Information Retrieval.* 771–778.

[25] Antonio Mallia, Michał Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for Academia. In *CEUR Workshop Proceedings Vol-2409.* 50–56.

[26] Antonio Mallia, Michał Siedlaczek, and Torsten Suel. 2019. An experimental study of index compression and DAAT query processing methods. In *European Conference on Information Retrieval.* 353–368.

[27] Antonio Mallia, Michal Siedlaczek, Mengyang Sun, and Torsten Suel. 2020. A Comparison of Top-k Threshold Estimation Techniques for Disjunctive Query Processing. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management.*

[28] Matthias Petri, Alistair Moffat, Joel Mackenzie, J Shane Culpepper, and Daniel Beck. 2019. Accelerated query processing via similarity score prediction. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval.* 485–494.

[29] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. 1995. Okapi at TREC-3. *Nist Special Publication Sp* 109 (1995), 109.

[30] Cristian Rossi, Edleno S de Moura, Andre L Carvalho, and Altigran S da Silva. 2013. Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 183–192.

[31] Dongdong Shan, Shuai Ding, Jing He, Hongfei Yan, and Xiaoming Li. 2012. Optimized top-k processing with global page scores on block-max indexes. In *Proceedings of the fifth ACM International Conference on Web Search and Data Mining.* 423–432.

[32] Fabrizio Silvestri. 2007. Sorting out the document identifier assignment problem. In *European Conference on Information Retrieval.* 101–112.

[33] Andrew Trotman and Kat Lilly. 2018. Elias Revisited: Group Elias SIMD Coding. In *Proceedings of the 23rd Australasian Document Computing Symposium.* 1–8.

[34] Howard Turtle and James Flood. 1995. Query Evaluation: Strategies and Optimizations. *Inf. Process. Manage.* 31, 6 (1995), 831–850.

[35] Lidan Wang, Jimmy Lin, and Donald Metzler. 2011. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 105–114.

[36] Erman Yafay and Ismail Sengor Altingovde. 2019. Caching Scores for Faster Query Processing with Dynamic Pruning in Search Engines. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management.* 2457–2460.

[37] Dawei Yin, Yuening Hu, Jiliang Tang, Tim Daly, Mianwei Zhou, Hua Ouyang, Jianhui Chen, Changsung Kang, Hongbo Deng, Chikashi Nobata, et al. 2016. Ranking relevance in yahoo search. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* 323–332.

[38] Chengxiang Zhai and John Lafferty. 2004. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems (TOIS)* 22, 2 (2004), 179–214.

[39] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM computing surveys (CSUR)* 38, 2 (2006), 6–es.