Optimizing Top-k Document Retrieval Strategies for Block-Max Indexes

Constantinos Dimopoulos
Polytechnic Institute of NYU
constantinos@cis.poly.edu

Sergey Nepomnyachiy
Polytechnic Institute of NYU
snepom01@students.poly.edu

Torsten Suel
Polytechnic Institute of NYU
suel@poly.edu

ABSTRACT

Large web search engines use significant hardware and energy resources to process hundreds of millions of queries each day, and a lot of research has focused on how to improve query processing efficiency. One general class of optimizations called early termination techniques is used in all major engines, and essentially involves computing top results without an exhaustive traversal and scoring of all potentially relevant index entries. Recent work in [9, 7] proposed several early termination algorithms for disjunctive top-k query processing, based on a new augmented index structure called Block-Max Index that enables aggressive skipping in the index.

In this paper, we build on this work by studying new algorithms and optimizations for Block-Max indexes that achieve significant performance gains over the work in [9, 7]. We start by implementing and comparing Block-Max oriented algorithms based on the well-known Maxscore and WAND approaches. Then we study how to build better Block-Max index structures and design better index-traversal strategies, resulting in new algorithms that achieve a factor of 2 speed-up over the best results in [9] with acceptable space overheads. We also describe and evaluate a hierarchical algorithm for a new recursive Block-Max index structure.

1. INTRODUCTION

Large web search engines have to answer hundreds of millions of queries per day over tens of billions of documents. To process this workload, such engines use hundreds of thousands of machines distributed over multiple data centers. In fact, query processing is responsible for a significant part of the cost of operating a large search engine, and a lot of industrial and academic research has focused on decreasing this cost. Major families of techniques that have been studied include *caching* of full or partial results or index structures at various levels of the system, *index compression* techniques that decrease both index size and access costs, and *early termination* (or pruning) techniques that, using various shortcuts, try to identify the best or most promising results without an exhaustive evaluation of all candidates. Previous work can also be divided into work that improves the efficiency of the overall distributed architecture (say through smart query routing, data partitioning, and distributed caching), and work that focuses on the throughput and response time of individual nodes in the system.

In this paper, we are interested in early termination techniques for optimizing performance within individual query processing nodes, assuming some distributed architecture has successfully partitioned the problem over many machines. More precisely, we study *safe* early termination (ET) techniques for (ranked) *disjunctive queries*. We define these terms later, but *disjunctive queries* basically means queries with a simple term-based ranking function (e.g., Cosine, BM25) that is applied to all documents containing at least one of the query terms, and *safe* means that the technique must return exactly the same results as an exhaustive evaluation [19].

While state-of-art search engines use highly complicated ranking functions based on hundreds of features, simple ranking functions as discussed above are still important for performance reasons. This is because it would be much too costly to evaluate every candidate result using the full ranking function. Instead, search engines typically first use a simple ranking function to narrow the field of candidate results, and then apply more and more complicated ranking functions in later phases [23].

In the initial phase, search engines may use either disjunctive queries, as discussed above, or conjunctive queries where only documents containing all query terms are considered (or possibly a mix of the two). Disjunctive queries are known to return higher-quality results in many scenarios; however, they are also significantly more expensive to compute than conjunctive queries that can discard any documents that are missing even a single query term.

This has motivated some amount of previous work on early termination techniques for disjunctive queries that attempts to narrow the efficiency gap, including [22, 6, 2, 20, 9, 7]. In particular, two recent papers [7, 9] independently proposed an augmented inverted index structure called *Block-Max Index* [9], and showed how it can be used for faster early termination. In a nutshell, a Block-Max Index augments the commonly used inverted index structure, where for each distinct term t we store a sorted list of the IDs of those documents where t occurs, with upper-bound values for blocks of these IDs. That is, for every say 64 IDs of documents containing a term t, we store the maximum term-wise score of any of these documents with respect to t. This then allows algorithms to quickly skip over blocks of documents that score too low to make it into the top results.

1.1 Our Contribution

In this paper, we study and evaluate new and (modifications of) existing early-termination algorithms for disjunctive queries that use such Block-Max Indexes. The previous work in [7, 9] proposes essentially the same structure, but describes different algorithms to exploit it. In particular, [9] proposes algorithms based on the WAND approach in [6], while [7] describes algorithms based on the Maxscore approach described in [22]. The algorithms achieve significant speedups over previous work, and thus motivate additional study of Block-Max Indexes and how to best exploit them.

We will start out by performing a direct experimental comparison of WAND- and Maxscore-based algorithms with and without Block-Max Indexes, which leads to some new and unexpected observations. Next, we build on these observations by designing and implementing new techniques for exploiting Block-Max Indexes, in particular docID-oriented block selection schemes, on-the-fly generation of Block-Max Indexes, and a new recursive query processing algorithm that uses a hierarchical partitioning of inverted lists into blocks. Overall, we achieve roughly a factor of two improvement in running time over the fastest previous results in [9], though we believe that additional improvements are possible by extending our techniques.

2. BACKGROUND AND RELATED WORK

In this section, we provide some technical background knowledge on inverted indexes, query processing, and early termination techniques, and discuss related work.

2.1 Inverted Indexes and Index Compression

Current search engines perform query processing using an inverted index, which is a simple and efficient data structure that allows us to find documents containing particular terms [24, 27]. Given a collection of N documents, we assume each document is identified by a unique document ID (docID) between 0 and $N-1$. An inverted index contains many inverted lists, where each inverted list L_w is a list of postings describing all places where term w occurs in the collection. More precisely, each posting contains the docID of a document that contains the term w, the number of occurrences of w in the document (called frequency), and sometimes the exact locations of these occurrences in the document or other context such as font size etc. Postings in each list are typically sorted by docID, or sometimes in some other way as described later. Thus, in the case where we store docIDs and frequencies, each posting is of the form (d_i, f_i). We focus on this case, but our techniques also apply to cases where positions, context information, or quantized impact scores are stored.

The inverted lists of common query terms may consist of many millions of postings. To allow faster access and limit the amount of memory needed, search engines use various compression techniques that significantly reduce the size of the inverted lists [27]. Compression is crucial for search engine performance and there are many compression techniques in the literature; see [26, 25, 18] for some recent work. In this paper we use a version of the PForDelta compression method [28] described in [26], but all ideas also apply to other methods.

Because lists can be very long, we often want to skip parts of the lists during query processing. To allow this, inverted lists are usually split into blocks of, say, 64 or 128 postings, such that each block can be decompressed individually. This requires an additional table that stores for each block the uncompressed maximum (or minimum) docID and the block size. The size of this extra table is small compared to the overall index. For performance reasons, we store the data in each a block as 64 or 128 docIDs followed by the corresponding frequencies.

2.2 Query Processing

The simplest form of query processing is called *Boolean query processing*. A query *apple AND orange* looking for all documents containing both *apple* and *orange* can be implemented by intersecting the docIDs in the inverted lists for *apple* and *orange*. In simple *ranked queries*, a ranking function is used to compute a score for each document passing a Boolean filter, such as the union or intersection of the query terms, and then the k top-scoring documents are returned. The ranking function should be efficiently computable from the data in the inverted lists (frequencies and maybe positions), plus maybe a limited amount of other statistics stored outside the inverted index (e.g., document lengths or global scores such as Pagerank). Many classes of simple ranking functions similar to BM25 or Cosine have been studied; see [3].A common property of many of these is that they are decomposable into term scores, i.e., a sum or other simple combination of per-term scores.

Real search engines use ranking functions based on hundreds of features. However, such functions are quite expensive to evaluate. To achieve efficiency, search engines commonly separate the ranking process into two or more phases. In the first phase, a very simple and fast ranking function such as BM25 is used to get, say, the top 100 or 1000 documents. Then in the second and further phases, increasingly more complicated ranking functions with more and more features are applied to documents that pass through the earlier phases. Thus, the later phases only examine a fairly small number of result candidates, and a significant amount of computation is still spent in the first phase. In this paper, we focus on executing such a simple first-phase function, a problem that has been extensively studied in the literature.

Recall that simple ranked queries consist of a Boolean filter followed by ranking the documents that pass this filter. The most commonly used Boolean filters are conjunctive (AND) and disjunctive (OR). Disjunctive queries return better results than conjunctive queries in many scenarios, but are much more expensive to compute as they have to evaluate many more documents. For this reason, web search engines try to use conjunctive queries as much as possible, but improvements in the speed of disjunctive queries could make these a more attractive alternative.

2.3 Early Termination

When processing simple ranked queries on large document collections, the main performance challenge is caused by the lengths of the inverted lists and the large number of candidates that pass the Boolean filter and thus need to be evaluated. Early termination is a set of techniques that addresses this problem. Following [9], we say that a query processing algorithm is *exhaustive* if it fully evaluates all documents that satisfy the Boolean filter condition. Any other algorithm uses *early termination* (ET).

We note that early termination can be achieved in many different ways, including by (1) *stopping early*, where each inverted list is arranged from most to least promising posting and traversal is stopped once enough good results are found, (2) *skipping*, where inverted lists are sorted by docIDs, and thus promising documents spread out over the lists, but we can skip over uninteresting parts of a list, or (3) *partial scoring*, where candidate documents are only partially or approximately scored. But the above definition of ET can also include techniques such as static pruning [5, 11] and index tiering [15].

In this paper we focus on *safe early termination* [19], where we get exactly the same results as an exhaustive algorithm, i.e., the same documents in the same order with the same scores. Also, we focus on main memory-based indexes as motivated in [20, 8]. Most major engines either keep the entire index in memory, or at least a large enough part of it such that any I/O bottlenecks can be hidden using fairly standard caching techniques.

2.4 Index Layout and Access

As discussed, some ET techniques may require inverted index structures to be arranged in specific ways. In particular, many techniques assume a layout such that the most promising documents appear early in the inverted lists, which can be done by either reordering the postings in a list, or partitioning the index into several layers. In general, most techniques use one of the following layouts:

- **Document-Sorted Indexes**: The standard approach for exhaustive query processing, where postings in each inverted list are sorted by docID.
- **Impact-Sorted Indexes**: Postings in each list are sorted by their impact (or term score), that is, their contribution to the document score.
- **Impact-Layered Indexes**: Each list is partitioned into several layers, such that all postings in layer i have a higher impact than those in layer $i+1$. In each layer, postings are sorted by docID.

Impact-sorted and impact-layered indexes are very popular for ET algorithms as they place the most promising postings close to the start of the lists [14, 10, 2, 1, 4, 20, 13, 22]. A problem with impact-sorted indexes is that index compression suffers when docIDs are not in sorted order. In this case, an impact-layered index that uses a small number of layers often provides a better alternative.

In contrast, in document-sorted indexes the most promising results are spread out over the inverted list, and the goal in ET is to find ways to skip most of the postings in between. Few ET algorithms use document-sorted indexes; see, e.g., [6, 22, 21, 9, 7]. However, recent work in [7, 9] showed very good results for document-sorted indexes through the use of an additional Block-Max structure, and this motivates our work here where we build on these results. Thus, unless stated otherwise, we assume document-sorted indexes throughout the paper.

One advantage of document-sorted indexes is that they allow simple and fast index traversal during query execution based on a *Document-at-a-time* (DAAT) approach. Here, each list has a pointer to a *current* posting, and all pointers move forward in parallel as a query is processed. At any point in time, all postings with docID less than some value have already been processed, while documents with larger docID are yet to be considered. In contrast, impact-sorted and impact-layered indexes employ other, often more complicated, traversal strategies.

DAAT traversal is commonly implemented using the following methods reminiscent of file system or database access: First, there are methods for opening and closing inverted lists for reading. Second, a forward-seek operation called *nextGEQ* can be used to move the read pointer in an opened list forward to the first posting with a docID at least some specified value. Finally, there is an operation for returning the term score of the posting at the current pointer position. All our algorithms perform DAAT traversal using these methods, and we report the number of nextGEQ operations and term score computations in some of our results.

2.5 Basic ET Algorithms

We now describe previous safe early termination algorithms for disjunctive queries. We focus on methods using document-sorted indexes and DAAT traversal that are closely related to our approaches, in particular the WAND algorithm in [6], the Maxscore algorithm in [22], and recent enhancements of these algorithms based on Block-Max Indexes in [7, 9].

WAND: The WAND algorithm proposed by Broder et al. in [6] maintains for each query term a pointer to the current posting in the list (as done in DAAT). It also stores for each inverted list the highest impact score of any posting in the list, called the *maxscore* of the list. Whenever the algorithm moves forward in the inverted lists, it does so in three steps, pivot selection, alignment check, and evaluation.

In the first step, the current list pointers are sorted by docID. Then a *pivot* term is selected by summing up the maxscores of the query terms in ascending order of list pointer docID, until the sum becomes larger or equal to the threshold θ that is needed to be part of the current top-k results. The term where this happens is selected as the pivot, and used to drive query processing. This is illustrated in Figure 1, where *cat* is selected as the pivot term. The crucial observation is that no docID smaller than the current docID of the pivot term can make it into the top-k results, unless it was already considered earlier.

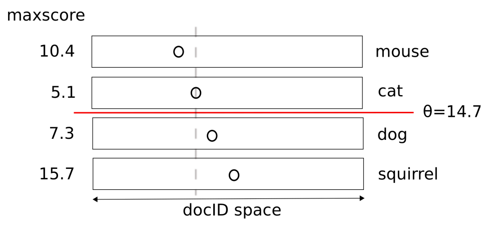

Figure 1: Pivot selection in the WAND algorithm with 4 **terms, where the current threshold for the top-k results is** 14.7**. The list for *cat* is selected as pivot since** $10.4+5.1 > 14.7$.

In the second step, we try to align the terms above the pivot term with the docID of the pivot term, called the pivot ID. To do this, we take these terms one after the other (in the figure, just the list for *mouse*) and do a forward seek using nextGEQ to the pivot docID. If such a seek results in a docID strictly larger than the pivot docID (because the list does not have a posting for the pivot docID), then we go back to the first step and re-sort and select another pivot. If all seeks result in a posting at the pivot docID, then in the third step we fully evaluate this docID, check if it needs to be inserted into the current top-k, move all pointers forward to at least this docID plus one, and then continue.

Maxscore: This algorithm has been known for a long time but was first described in detail in [22]. There are several versions of Maxscore based on both DAAT and TAAT traversal. The version described here is similar to those presented in [21, 12]. Like WAND, this algorithm also exploits knowledge of the maxscore of a list, hence the name. However, instead of using sorting by current docID and pivot selection to dynamically partition the lists into lists above and below the pivot term, Maxscore distinguishes between essential and non-essential lists.

Suppose the lists for the query terms are sorted from top to bottom by their maxscore, as shown in Figure 2. We again have a threshold θ that a document score has to reach to make it into the current top-k results. We now select as non-essential lists a set of lists such that their maxscores sum up to less than θ, by adding lists starting from the list with the smallest maxscore. The other lists are called essential. Observe that no document can make it into the top-k results just using postings in the non-essential lists, and at least one of the essential terms has to occur in any top-k document.

Thus, we can safely execute a top-k query by essentially performing a union operation over the essential lists, and then performing lookups into the non-essential lists to get the precise document scores. In every step we select the next larger docID in the essential lists, but instead of fully scoring each such candidate in all lists, we perform partial scoring as follows. We first evaluate the scores of this docID in the essential lists. If the sum of these scores plus the maxscores of the non-essential lists is less than θ, then we can abort the evaluation. Otherwise, we score the candidate in the non-essential lists until either we have a complete score, or the maximum possible score drops below the threshold. Whenever we find a new top-k result, we also check if the resulting increase in the threshold θ means that another list can be added to the non-essential lists.

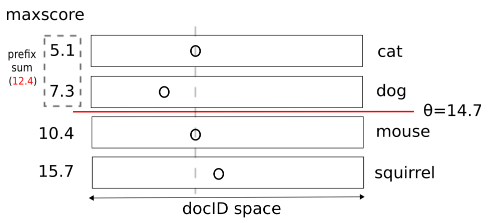

Figure 2: Selecting non-essential lists in the Maxscore algorithm. Here, the lists for *cat* and *dog* are non-essential as their maxscores sum up to $5.1 + 7.3 < 14.7$.

To optimize this algorithm, we need to decide how to select the non-essential lists and in which order to evaluate postings in the non-essential lists. We could select non-essential lists with the smallest maxscore (minimizing the number of essential lists), or select the longest lists (minimizing the total size of the essential lists). Not surprisingly, the longest lists are almost always the ones with the smallest maxscore in common ranking functions such as BM25, and thus selecting by maxscore usually achieves both objectives.[1] When evaluating scores in the non-essential lists, it is best to go from highest to lowest maxscore.

Block-Max Indexes and Algorithms: Two recent papers [9, 7] independently proposed an augmented inverted index structure called a *Block-Max Index* and used it to speed up the WAND [9] and Maxscore [7] approaches. The basic idea of the structure is very simple: Since inverted lists are often compressed in blocks of say 64 or 128 postings, we store for each block the maximum term score within the block, called *block maxscore*. Thus each inverted list maintains a piece-wise constant upper bound approximation of the term scores, as shown in Figure 3.

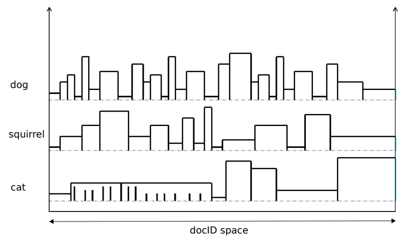

Figure 3: Three inverted lists are piecewise upper-bounded by the maximum scores in each block. As shown for one block in the bottom list, inside each block we have various values, including many zero values, that can be retrieved by evaluating the postings.

Since both WAND and Maxscore use the maxscore of a list to enable skipping, the simplest idea for using the Block-Max Index is to try to use block maxscores instead of list maxscores whenever possible. In fact, this is basically what the approaches in [9, 7] do, but there are some technical challenges that need to be overcome to make this work. These challenges are due to the fact that block maxscores are only valid within a block, and thus we have to be careful whenever we move over one or more block boundaries. Moreover, block boundaries in different lists are not aligned.

In the following, we refer to the approach in [9] as *Block-Max WAND* (BMW), and to the approach in [7] as *Block-Max Maxscore* (BMM). BMW and BMM differ in two main ways. First, BMW builds on WAND [9] while BMM builds on Maxscore. Second, BMW deals with block boundaries in an online fashion as they are encountered, while BMM uses a preprocessing step that detects and aligns block boundaries.

More precisely, BMW selects a candidate pivot using list maxscores as in WAND, but then uses block maxscores to check if this is really a viable pivot or if its score is too low to make it into the top-k results. In general, a combination of block maxscores and partial evaluation in selected lists is used to quickly rule out many pivots. Thus, block boundary issues are avoided by using list maxscores for pivot selection, and only using block maxscores once a pivot candidate has been selected. We refer to [9] for more details.

BMM first preprocesses the block boundaries in order to partition the docID space into intervals, such that we have an interval boundary whenever any of the participating lists has a block boundary. Then BMM runs Maxscore within each interval, selecting non-essential lists and doing partial scoring using block maxscores instead of list maxscores.

[1]We observed that this correlation between list length and maxscore value is quite strong over most of the range of lists lengths, but breaks down for lists with only a few postings.

More details and some additional variations of this basic idea are discussed in [7]. Also, followup work to [9, 7] in [16] shows how to deal with ranking functions that add a query-independent score such as Pagerank to the term scores.

Our work in this paper starts from the work in [9, 7] and asks how we can further improve query processing with Block-Max Indexes, by either refining the BMW or BMM approaches or designing new algorithms. It seems to us that there are a lot of additional benefits that can be obtained by using Block-Max Indexes and related structures.

3. PRELIMINARY EXPERIMENTS

In this section, we implement and evaluate a set of baseline algorithms derived from previous work. While the algorithms are not really novel, we are not aware of any previous comparison.

In our experiments, we use the TREC GOV2 dataset consisting of 25.2 million web pages with an uncompressed size of 426GB. We build inverted index structures with 64 docIDs and frequencies per block, using the version of PForDelta used in [26]. The resulting compressed index size is 8.75GB. We use 1000 queries selected at random from the TREC 2006 Efficiency track queries, and 1000 random queries from TREC 2005 as our testing sets.

All algorithms were implemented using C++ and compiled using gcc with -O3 optimization. Algorithms were carefully optimized for performance. We use BM25 as ranking function, and return top-10 results unless stated otherwise. All experiments were conducted on a single core of a 2.27Ghz Intel Xeon (E5520) CPU. All data structures are memory-resident.

3.1 Methods without Block-Max Indexes

We start by comparing the methods without Block-Max Indexes, in particular WAND, Maxscore, and the method of Strohman and Croft (SC) [20]. For SC, we use the numbers reported in [9] which were on the same data and machine. All other methods were re-implemented for this paper.

The implementation of WAND follows closely the description in [6] and the implementation in [9]. Recall that the method first selects a pivot, then aligns the lists, and then evaluates. We tried using partial scoring techniques in the evaluation phase, and interleaving alignment and scoring, but found no performance benefits in this. For Maxscore, we follow the description in Section 2.5, with careful use of partial scoring techniques.

In Table 1 we compare the performance of SC, WAND, and Maxscore on the TREC 2006 and 2005 queries. We show average times over all queries, and average times for queries with 2, 3, 4, 5, and more than 5 terms. The most important observation is that Maxscore consistently and significantly outperforms the SC and WAND algorithms, by roughly a factor of 2. This is surprising because SC and WAND were intended to be performance optimizations over the older Maxscore algorithm. Comparing WAND and Maxscore, we believe there are two main costs that slow down WAND, the pivot selection via sorting (which we carefully optimized by updating the sorted order), and the large number of calls to *nextGEQ* as part of the alignment process as shown in Table 2. We also note that Maxscore is especially strong on queries with 5 or more queries, where it outperforms the others by more than a factor of 2.

TREC 06						
Algorithm	avg	2	3	4	5	>5
SC (from [9])	64.3	12.2	36.7	75.6	117.2	226.3
WAND	75.43	20.59	42.42	88.25	146.73	240.44
Maxscore	34.17	10.67	22.76	41.71	60.32	91.05
TREC 05						
WAND	51.4	18.19	34.82	55.33	102.14	321.29
Maxscore	20.13	9.42	17.51	25.85	35.33	87.84

Table 1: Running times in ms per query for methods without Block-Max Index structure.

TREC 06			
Algorithm	Time (ms)	# evals	# nextgeq
WAND	75.43	$448,209$	$1,211,034$
Maxscore	34.17	$673,862$	$722,867$
TREC 05			
WAND	51.4	$366,063$	$775,581$
Maxscore	20.13	$409,444$	$429,835$

Table 2: Number of term score evaluations and calls to *nextGEQ* per query for methods without Block-Max Index structure.

3.2 Block Max-Based Methods

Next, we compare basic methods that use Block-Max Indexes, in particular BMW, BMM, and some modifications of these. We note that previous work [9, 7, 16] chooses block boundaries in the Block-Max Index by placing a fixed number of postings, e.g., 64, in each block. This is a natural choice for two reasons: First, since blocks are also used in index compression, it seems natural to piggy-back onto this mechanism and add another value, the block maxscore. Second, this allows easy control over the space overhead of the structure. We refer to such an approach as a Posting-Oriented Block-Max Index, and use this approach for our first experiments. However, in later sections we revisit this assumption and look at other choices of block boundaries that result in better performance.

For the experiments, we re-implemented BMW as described in [9] and re-used some code from the core compression and index access mechanisms. Our baseline in this study is the BMW algorithm in [9], which to the best of our knowledge, achieves the best reported query processing times. We also implemented a Maxscore-based method (BMM), but with some changes over the implementation described in [7]. In particular, we removed the preprocessing for aligning block boundaries and the selection of non-essential lists in each interval as described in [7], as we found this to be a performance bottleneck. We note here that while the numbers reported in [7] are higher than those in [9] and for a smaller data set, they include some disk access costs and make somewhat different architectural assumptions. Thus, it is difficult to directly compare the numbers.

In the implementation of BMM, we select the essential lists as in Maxscore, based on list maxscores, but use a series of filters based on block maxscores to rule out most candidates. In particular, after selecting the next unprocessed posting from the essential lists, we first check if the (precomputed) sum of the list maxscores of the non-essential lists plus the sum of the block maxscores of the essential lists is above the threshold. If yes, then we replace the list maxscores of the non-essential lists with block maxscores, and check again. Next, we get the term scores of the candidate in the essential lists, and check if the potential score is still above the threshold. If so, we then start scoring the non-essential lists until all terms have been evaluated or the candidate has been ruled out.

In case the first (or second) filter above fails, we can in fact not just skip the current posting, but move directly to the end of the shortest (or shortest essential) block. (This is similar to the GetNewCandidate() method described in [9].) We note that in many cases, we may be able to skip even beyond the end of that block, for example when the block is followed by one with a smaller block maxscore. We can exploit this by a more aggressive optimization, where we skip over block boundaries until we arrive at the next *live* block where the sum of the block-max scores of all terms is larger than θ. We note that [7] has a similar idea for skipping over dead areas. We refer to the version of BMM that uses this optimization as *Block-Max Maxscore-Next Live Block* (BMM-NLB).

Finally, initial experiments found that some methods are better for queries with few terms, and some for queries with many terms. Since all algorithms use exactly the same data structures, it is easy to call the best method for any number of query terms; this combined method is called BM-OPT. We did not see any way to implement a good Block-Max based version of SC, which takes a very different, non-DAAT based, approach from the other algorithms.

In Table 3, we report the performance of the various algorithms on the TREC 2006 and 2005 queries. We observe that BMW obtains very large gains over WAND, as reported in [9]. BMM, however, achieves only slight improvements over our optimized version of Maxscore, which already did much better than WAND. Overall, BMW now performs better than BMM. The BMM-NLB version that tried to skip additional block boundaries actually performs worse than BMM except for 2-term queries; the reason is that there are not that many dead areas for multi-term queries, so the benefit is limited while the added complexity slows down performance. We also observe that BMM outperforms BMW for queries with many terms, and for this reason BM-OPT performs slightly better on average than BMW.

Finally, Table 4 shows the average number of term score evaluations and calls to *nextGEQ* per query. We can see that BMW evaluates significantly fewer documents and makes fewer calls than all other algorithms. However, while BMM performs many more evaluations and calls, its running time was only slightly slower than BMW. This indicates that these evaluations and calls are not the only indicator of performance. In fact, it seems that BMW spends a fair amount of effort trying to avoid evaluations and calls, while BMM does not spend as much time thinking about avoiding them and instead keeps the control structure simple.

For the remainder of the paper we report the performance of our algorithms only for the TREC 06 query traces, due to space limitation.

4. A HIERARCHICAL ALGORITHM

Next, we describe a new query processing algorithm that uses a modified hierarchical Block-Max Index. The basic intuition for this is very natural; instead of using only one size of blocks, we partition inverted lists into a hierarchy of blocks and store block maxscores at all different levels. We would expect that different terms and queries lead to different optimal choices for the block size, and that even for a particular term and query, a hierarchical structure would enable large skips whenever possible while allowing us to drill into smaller block sizes if needed. As indicated before, this structure is now completely decoupled from the blocks used in index compression.

TREC 06						
Algorithm	avg	2	3	4	5	>5
BMW	27.44	4.14	12.09	32.83	55.11	108.47
BMM	32.77	10.35	21.95	39.8	57.73	87.41
BMM-NLB	51.48	9.62	30.24	64.32	99.09	159.08
BM-OPT	25.95	4.14	12.09	32.83	55.11	87.41
TREC 05						
BMW	16.25	3	10.38	17.55	33.66	127.61
BMM	19.97	9.33	17.35	25.66	35.18	86.45
BMM-NLB	29.46	8.85	25.19	41.02	61.37	168.15
BM-OPT	13.83	3	10.38	17.55	33.66	86.4

Table 3: Running times in ms per query for basic methods with Block-Max Index Structures.

TREC 06			
Algorithm	Time (ms)	# evals	# nextgeq
BMW	27.44	27,440	431,817
BMM	32.77	661,072	667,642
BMM-NLB	51.48	573,630	581,442
TREC 05			
BMW	16.25	17,555	240,756
BMM	19.97	404,441	407,331
BMM-NLB	29.46	309,047	312,552

Table 4: Number of term score evaluations and calls to *nextGEQ* for the different algorithms.

We start with a basic partitioning into fairly small blocks containing, say, 8 or 16 postings. We then build a binary tree structure on top of these blocks, where each higher-up block stores as its block maxscore and right block boundary the larger of the two childrens' block maxscores and right block boundaries, respectively. Finally, the root contains the list maxscore and has as its right boundary the value N (where all docIDs are between 0 and $N-1$. We also implicitly treat the actual postings, and the empty spaces between them, as child nodes of the basic blocks at the bottom of the tree; this is called the posting level of the tree.

Given this structure, we now design an algorithm that uses it. The basic idea is to perform a traversal of the trees for the query terms, reminiscent of branch-and-bound computations. During traversal, we maintain a value *cub* that is an upper bound on the scores of any documents in the intersection of the current blocks. For each term, we have a stack or other mechanism (say successor pointers) for efficient tree traversal. We first place the roots of the trees for the query terms on a stack, and initialize *cub* to the sum of their maxscores.

Now we repeatedly check if *cub* is above the current threshold for top-k results. If yes, there are two cases: (1) If we have reached the posting level in all lists, this means that we have completely evaluated a docID and can place it into the top-k, and then continue with the next docID by popping any blocks associated with postings (but not those representing space between postings). (2) In the other case, there is the possibility of a top-k result, but we need to go deeper into the trees to check. To do so, we select one of the lists, say the one with the farthest right boundary, pop it from the stack, and instead push its children on the stack and update *cub* as needed. Once we reach the posting level, we do not immediately place all postings in the block on the stack, but materialize them one at a time from the inverted index using *nextGEQ* and scoring of terms.

If the *cub* was below the threshold, this means that no top-k result can be in the intersection of the current nodes, and we can skip to the end of the block with the closest right boundary, by popping this block from the stack and updating *cub*. The algorithm stops when the stacks are empty. We note that this algorithm cannot easily be classified as being WAND or Maxscore based. There are also various ways to modify and optimize this algorithm, but due to space constraints we have to omit some of the details.

We show performance results for the new algorithm in Table 5. The performance of the algorithm depends on the size of the basic blocks at the bottom of the tree, and for this case we used a size of 4 postings per block for lists shorter than 1500000 postings and 8 postings per block for longer ones. Thus, the resulting hierarchical Block-Max structure is comparable to the size of the entire inverted index. Despite this size, the algorithm only outperforms the BMW algorithm by a moderate amount. The reason is that there is a fair amount of overhead for pushing and popping nodes onto the stack, and for selecting which nodes to push and pop. We will revisit the performance of this algorithm later in the context of reordered indexes.

Algorithm	avg	2	3	4	5	>5
BMW	27.44	4.14	12.09	32.83	55.11	108.47
HIER	24.47	2.14	9.06	31.03	50.40	100.75

Table 5: Running times in ms per query for BMW and for the new algorithm with a hierarchical Block-Max Index, for 2, 3, 4, 5, and more than 5 query terms.

In the following section, we describe new algorithms that move beyond the posting-oriented Block-Max Index structures used here. This means that we decouple the choice of blocks for storing block maxscores from the choice of blocks for inverted index compression, and the reader should think about the Block-Max Index as a structure that is separate from the inverted lists. In some cases, we may choose much smaller blocks than before, resulting in better pruning power but much larger Block-Max Index structures, and part of the challenges we address are about how to limit this space overhead.

5. DOCID-ORIENTED BLOCKS

We now go back to the non-hierarchical case and try to further improve the BMW and BMM approaches. As in the hierarchical case, we decouple the Block-Max structure from the blocks used for index compression, thus allowing us to choose block boundaries in a way that optimizes performance. In particular, we could define block boundaries based not on the number of postings in a block, but based on docID space. For example, if we choose to have one block maxscore say for every 1024 docIDs, then given the docID of a candidate, we can find the location of its corresponding block maxscore by just shifting the docID by 10 bits to the right.

We call such blocks that are based on a simple partitioning of docID space docID-oriented Block-Max structures. Their main benefit is extremely fast lookup. The main challenge is how to select the block sizes. Having one block for every 1024 docIDs might be feasible for longer lists, but would be a huge waste of space for a term with only a handful of postings, where most block maxscores would be zero. We now consider three ways to select docID-oriented block boundaries, choosing a fixed size, choosing sizes based on the expected number of postings in a block, and choosing variable block sizes. Some of the methods result in very large Block-Max structures, and we will address the space problem in the next section.

Fixed Block Size: We first consider docID-oriented Block-Max structures that split the docID space into blocks of the same fixed size for all inverted lists. We consider docID ranges that are powers of two, to achieve very fast block maxscore lookups. Figure 4 shows the running times of the BMW, BMM, and BMM-NLP algorithms from Section 3.2 as we vary the block size from 2^3 to 2^{19} docIDs. We note that the methods do not perform very well for very large or extremely small blocks. For large blocks, we get very little pruning power from block maxscores, and we would be better off just using the Maxscore algorithm. For very small blocks, we only get very small skips and have a lot of memory accesses to fetch block maxscores. However, in between, in the range from 2^6 to 2^8, we see very good performance. In particular, BMM-NLB achieves a running time of less than $13ms$ per query for blocks of size 64 docIDs.

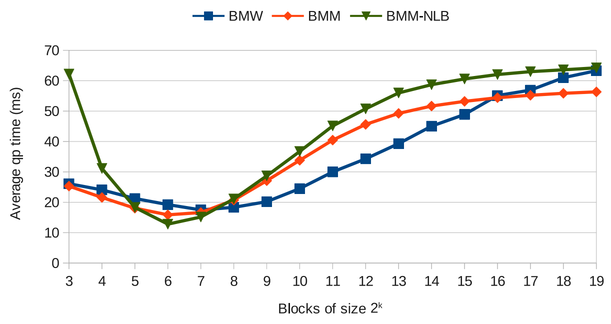

Figure 4: Times per query for fixed block sizes ranging from 2^3 to 2^{19} docIDs.

However, if we actually store one 4-byte block maxscore for every 64 docIDs in docID space, then for the GOV2 collection of 25.2 million documents this means about $1.5MB$ of data for each inverted list. Since there are more than 20 million distinct terms in the collection, most of them having only a few postings, the space requirement would be about $35TB$.

Expected Block Size: Another natural approach would be to choose larger docID ranges for short lists, and smaller docID ranges for long lists. In particular, we could pick the docID range for each inverted list such that the expected number of postings in each block (total number of postings divided by total number of blocks) is kept constant. This gives us direct control over the total size of the structure, similar to the case of posting-oriented blocks before.

In Figure 5 we show the running time of our algorithms as we vary the expected number of postings per block from 1 to 2^{19}. More precisely, given a target b, we choose the block size as the smallest number 2^x such that the expected number of postings per block is at most b. We observe that BMW performs well for values of b up to 2^5, while BMM and BMM-NLB only do well for even smaller values of b. However, none of the methods achieves running times below $20ms$. Thus, this method limits the size of the Block-Max

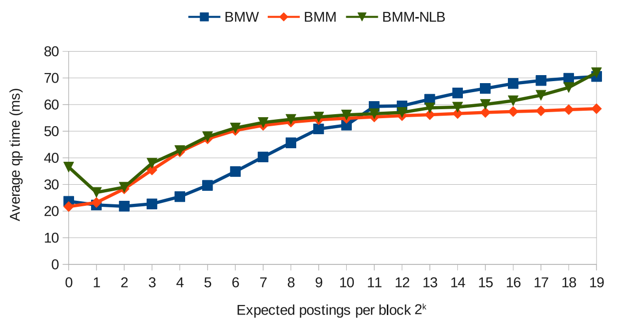

Figure 5: Times per query for expected number of postings per block ranging from 1 to 2^{19}.

Algorithm	Time (ms)	# evals	# nextgeq
BMW	18.44	$2,230$	$369,230$
BMM	16.16	$45,060$	$443,789$
BMM-NLB	12.73	$45,060$	$57,391$

Table 6: Running times for one particular variable block size setting.

structure, but performance degrades compared to the fixed block size method.

Variable Block Size: As seen, both fixing the block size and fixing the expected number of postings per block has problems. The former uses too much space, and the latter is slow. We now check if there is some way to get the same performance as in the fixed case by using (slightly) less space and a variable block size. Experiments showed that we can basically match the best running times for the fixed case, as shown in Table 6 where the best method, BMM-NLB, achieves a time of $12.73ms$.

Note that these results are meaningless without more information about the settings and space consumption. In particular, the results in Table 6 use a block size of 2^{10} docIDs for any inverted list of up to 2^{10} postings, then 2^6 up to length 2^{12}, 2^7 up to length 2^{15}, 2^8 up to length 2^{17}, 2^7 up to length 2^{18}, and 2^6 for any list longer than 2^{18}. The astute reader will notice that this still uses block sizes of at most 2^{10} docIDs for all lists, including for lists with only a few postings. Thus, this can save at most a factor of $2^4 = 16$ over the earlier setting that required more than $35TB$. In fact, this setting still requires about $2.25TB$ of space! We will show in the next section how we can reduce this by a factor of 1000 to obtain a more realistic structure, without significant increases in running time.

6. ENGINEERING THE SPACE ISSUE

In the previous section, we discovered some settings that achieve very fast running times, but at the cost of an unrealistic space consumption. In this section, we address this problem, and study two simple ways to significantly reduce the required space, thus making these settings feasible in practice.

On-The-Fly Block-Max Generation: As mentioned before, most of the inverted lists in a collection are very short, consisting of only a few postings. In fact, 96.95% of all distinct terms in our dataset occur in less than 128 documents. However, for such terms, it is quite feasible to create the docID-oriented Block-Max structure on-the-fly, that is, during query processing. Note that this requires three steps, initializing an array to hold the Block-Max structure, decompressing the entire inverted list and evaluating its postings, and using the uncompressed docIDs and computed scores to set the block maxscores in the array. The first step involves zeroing out an array of about $25,000$ numbers in the case of block size 2^{10} docIDs on GOV2, and modern CPUs can do this in a few microseconds. The other steps depend on the list length, but we will see that this is efficient for lists of up to a few ten thousand postings.

In Table 7, we show the total space for the variable docID-oriented Block-Max index in Table 6 in the last section, which had a raw size of about $2.25TB$, and the resulting additional cost in ms per query of generating Block-Max structures on the fly. If we limit on-the-fly generation to lists of at most 2^{11} postings, we still have a space requirement of over $94GB$ for the Block-Max structure. Performing on-the-fly generation for lists up to length 2^{17} brings the size down to $8.41GB$, about the same size as the inverted index, but adds several ms to the running time of a query. However, doing on-the-fly generation for lists up to length 2^{15} brings us to a space of about $11GB$ with less than $0.5ms$ in overhead. We note that the small overhead per query is due to the speed of one-the-fly generation and the fact that not every query has a query term with a short enough list.

Method	Space (GBs)	Overhead (ms/q)
on-the-fly BMG$_{<2^{11}}$	94.53	0.02
on-the-fly BMG$_{<2^{13}}$	24.52	0.10
on-the-fly BMG$_{<2^{15}}$	11.07	0.44
on-the-fly BMG$_{<2^{17}}$	8.41	3.42
No on-the-fly BMG	2249.66	–

Table 7: Space/time trade-off for one-the-fly block-max generation.

Block-Max Score Quantization: We can further decrease the space requirements by replacing the 4-byte floating point representation of the maxscores with 1-byte unsigned chars. To do this, we need to quantize the floating point numbers to 256 distinct values. Quantization is a standard technique in information retrieval, and a number of methods have been proposed for quantizing term scores.

Our problem here is actually a little simpler since the block maxscores are only used as a rough initial filter, and not for the final ranking of the results. In this case, a simple linear quantization is good enough, where for each inverted list we choose a basic quantile z and then represent each block maxscore m by the smallest i such that $i * z \geq m$. If z is chosen by dividing the list maxscore by 255, then each i can be stored in one unsigned char. When we access a block maxscore, we simply multiply i with the (term-dependent) quantile z. The rounding-up to the next multiple of z results in slightly larger block maxscores and thus slightly less pruning, but the difference is negligible.

Experimental Results: We now present the space/time trade-off that are achieved when using both on-the-fly Block-Max generation and block maxscore quantization in our query processing algorithms, using the same block size settings as in Table 6.

We start by looking at the performance of our fastest algorithm, BMM-NLB. Table 8 shows the performance of BMM-NLB with on-the-fly generation and with and without quantization. We note that there are two factors that can increase the running time of the algorithm, the time for the on-the-fly generation, and the cost of accessing the

Method	Space (GBs)	Time (ms)
on-the-fly BMG$_{<2^{13}}$	24.52	12.83
on-the-fly BMG$_{<2^{13}}$ + BMQ	6.13	14.34
on-the-fly BMG$_{<2^{15}}$	11.07	13.17
on-the-fly BMG$_{<2^{15}}$ + BMQ	2.76	14.64
No on-the-fly BMG and, no BMQ	2249.66	12.73

Table 8: Space/time trade-off for on-the-fly block-max generation and quantization in BMM-NLB.

quantized values and multiplying them with the quantile z. We see that with on-the-fly generation only for lists shorter than 2^{13} and no quantization, we get a running time of 12.83 ms and a space requirement of $24.52GB$. At the other hand of the spectrum, we get a running time of $14.64ms$ with a space requirement of $2.76GB$, about 30% of the size of the inverted index, by using quantization and on-the-fly generation for lists up to length 2^{15}.

Algorithm	Time (ms)	# evals	# nextgeq
BMW	19.82	1,646	369,571
BMM	17.44	46,207	444,569
BMM-NLB	14.64	46,207	58,809
BM-OPT	14.32	45,905	74,027
Posting-oriented BMW	27.44	27,440	431,817

Table 9: Performance of the algorithms when both on-the-fly generation and quantization are used.

Next, in Table 9 we show the performance of all three basic algorithms when both on-the-fly generation and quantization is applied, using the setting from Table 8 that results in the smallest space requirement. We see that all algorithms are significantly faster than the state-of-the-art postings-oriented BMW. Finally, Table 10 shows the results for different numbers of query terms.

Algorithm	avg	2	3	4	5	>5
BMW	19.82	3.79	10.08	22.73	39.67	73.4
BMM	17.44	4.4	9.98	21.36	32.71	54.66
BMM-NLB	14.64	5.59	9.54	17.44	24.55	42.23
BM-OPT	14.32	3.79	9.54	17.44	24.55	42.23

Table 10: Time in ms per query (ms) for queries with 2, 3, 4, 5, and more than 5 terms.

7. ADDITIONAL RESULTS

In this section, we provide a few additional results. In particular, we look at the performance of the methods for reordered and impact-layered indexes, and at how performance changes when we need top-100 or top-1000 results rather than just the top-10. Due to space constraints, we only present some selected results.

Document ID Reordering: A lot of work over the last decade has looked at how to improve the compressibility of inverted indexes by reordering them, that is, by assigning docIDs to documents in a way that minimizes index size. In general, this is done by assigning consecutive docIDs to similar documents; one particularly simple and effective way to do this is to assign docIDs to web pages based on an alphabetic sorting of their URLs [17]. Recent work in [25] showed that this not only decreases index size, but also accelerates conjunctive query processing, and [9] extended this observation to the BMW algorithm for disjunctive queries.

Thus, we decided to check how our new methods fare under reordering, using URL-sorting as suggested in [17] and used in [9]. Table 11 shows the resulting running times for our new docID-oriented Block-Max Index structures with quantization and on-the-fly generation. Note that the best previous result for this case in [9] was $8.89ms$, for the old posting-oriented version of BMW. As we see, reordering also gives a significant boost to the new algorithms, though BMM-NLB benefits less than the other methods.

Algorithm	Time (ms)	# evals	# nextgeq
BMW	8.34	4,324	129,560
BMM	7.02	17,476	169,709
BMM-NLB	10.33	17,476	18,964

Table 11: Impact of docID reordering on the query processing time of BMW, BMM, and BMM-NLB with quantization and on-the-fly generation.

In Table 12 we show the performance of the hierarchical method from the previous sections under reordering. Somewhat surprisingly, the hierarchical method did extremely well under reordering and significantly outperformed all others. We believe that this is due to the ability of the hierarchical method to exploit the larger but also the highly variable (between different queries and different regions of the docID space) skips that are possible under document reordering. We note that it also has the largest space requirement for its hierarchical Block-Max structure, and we give results for two settings with sizes of $6GB$ and $10GB$. (These numbers can be reduced using quantization and on-the-fly generation of hierarchical Block-Max structures.)

Algorithm	Time (ms)
HIER 6GB	4.85
HIER 10GB	4.29

Table 12: Impact of docID reordering on the query processing time of the hierarchical method.

7.1 Document ID Reordering and Layering

The work in [9] also proposed a version of BMW that uses an impact-layered index structure with two layers. This means that each inverted list longer than some threshold is split into two layers, the first layer containing postings with high term scores, and the second layer containing postings with low scores. Then the resulting lists are fed into the query, which treats the two layers as two distinct inverted lists. We implemented layered versions of our BMW, BMM, and BMM-NLB algorithms with quantization and on-the-fly generation, and evaluated the results. For the non-reordered case, there were no significant improvements over the non-layered version, but in the case of reordered indexes we obtained the results in Table 13. We can see that BMW benefits significantly from layering, while BMM and BMM-NLB actually perform worse by adding layers in the index-reordered case. For comparison, the best result for this case in [9] was $7.4ms$.

Algorithm	Time (ms)	# evals	# nextgeq
BMW	5.7	2,085	72,314
BMM	12.53	58,560	287,494
BMM-NLB	21.63	58,560	69,027

Table 13: Impact of docID reordering and layering on query processing times with quantization and on-the-fly generation.

Increasing k: Finally, we look at the performance of our methods when we are interested not just in the top-10, but the top-100 or top-1000 results. Getting more than 10 results is necessary in scenarios where results are re-ranked in subsequent phases, as done in current web search engines. (We note however that even in that case, we might only need top-10 results from each of 100 query processing nodes.)

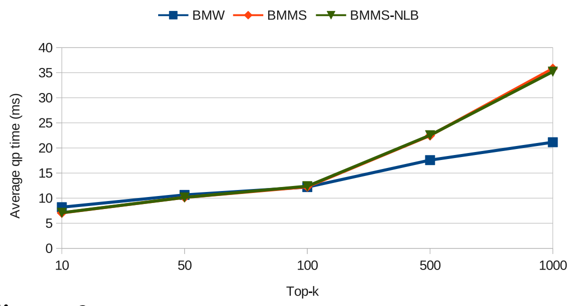

Figure 6: Performance of top-k query processing as we increase k.

We show results for the case of reordered indexes in Figure 6. We observe that while BMM and BMM-NLB show some performance degradation as we increase k, BMW fares much better, achieving around $12ms$ per query for top-100 and around $21.5ms$ per query for top-1000.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed improved algorithms for safe early termination in disjunctive queries based on the recently proposed Block-Max Index structures. Our main contributions are an experimental comparison of several new and existing methods, a new hierarchical algorithm that outperforms existing methods on reordered indexes, and new and extremely fast methods using docID-based block boundaries plus techniques that make them feasible in terms of space requirements. Overall, we achieve about a factor of 2 in speed-up over the fastest previous methods for standard inverted indexes, and slightly less improvement for reordered and layered index structures.

There are still many open problems and opportunities for future research, including additional and better hierarchical algorithms, potential Block-Max structures for pairs of frequently co-occurring terms, or early termination methods that better exploit the data parallel (SSE) instructions available in current generations of processors.

Acknowledgement

This research was supported by NSF Grant IIS-1117829 ”Efficient Query Processing in Large Search Engines”, and by a grant from Google. Sergey Nepomnyachiy was supported by NSF Grant ITR-0904246 ”The Role of Order in Search”.

9. REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proceedings of the 24th Annual Int. ACM SIGIR Conference on Research and Development in Inf. Retrieval*, 2001.

[2] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. of the 29th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2006.

[3] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-K: Index-access optimized top-k query processing. In *Proceedings of the 32th International Conference on Very Large Data Bases*, 2006.

[5] R. Blanco and A. Barreiro. Probabilistic static pruning of inverted files. *ACM Transactions on Information Systems*, 28(1), Jan. 2010.

[6] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th ACM Conf. on Information and Knowledge Management*, 2003.

[7] K. Chakrabarti, S. Chaudhuri, and V. Ganti. Interval-based pruning for top-k processing over compressed lists. In *Proc. of the 27th Int. Conf. on Data Engineering*, 2011.

[8] J. Dean. Challenges in building large-scale information retrieval systems. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, 2009.

[9] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. of the 34th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2011.

[10] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31:2002, 2002.

[11] R. Fagin, D. Carmel, D. Cohen, E. Farchi, M. Herscovici, Y. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual Int. ACM SIGIR Conference on Research and Development in Inf. Retrieval*, 2001.

[12] S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *Proc. of the 33th European Conf. on Information Retrieval*, 2011.

[13] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proceedings of the 29th International Conference on Very Large Data Bases*, 2003.

[14] M. Persin, J. Zobel, and R. Sacks-davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47:749–764, 1996.

[15] K. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In *1st Latin American Web Congress*, 2003.

[16] D. Shan, S. Ding, J. He, H. Yan, and X. Li. Optimized top-k processing with global page scores on block-max indexes. In *Proc. of the Fifth Int. Conf. on Web Search and Data Mining*, 2012.

[17] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of the 29th European Conf. on Information Retrieval*, 2007.

[18] F. Silvestri and R. Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proc. of the 19th ACM Conf. on Information and Knowledge Management*, 2010.

[19] T. Strohman. *Efficient Processing of Complex Features for Information Retrieval.* PhD thesis, University of Massachusetts Amherst, 2007.

[20] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proc. of the 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2007.

[21] T. Strohman, H. R. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proc. of the 28th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2005.

[22] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Processing and Management*, 31(6):831–850, 1995.

[23] L. Wang, J. J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. of the 34th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2011.

[24] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition.* Morgan Kaufmann, 1999.

[25] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. of the 18th Int. Conf. on World Wide Web*, 2009.

[26] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. of the 17th Int. Conf. on World Wide Web*, 2008.

[27] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[28] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22th Int. Conf. on Data Engineering*, 2006.