A Candidate Filtering Mechanism for Fast Top-K Query Processing on Modern CPUs

Constantinos Dimopoulos, Sergey Nepomnyachiy, Torsten Suel
Computer Science & Eng.
Polytechnic Institute of NYU
constantinos@cis.poly.edu, snepom01@students.poly.edu, suel@poly.edu

ABSTRACT

A large amount of research has focused on faster methods for finding top-k results in large document collections, one of the main scalability challenges for web search engines. In this paper, we propose a method for accelerating such top-k queries that builds on and generalizes methods recently proposed by several groups of researchers based on Block-Max Indexes [15, 10, 13]. In particular, we describe a system that uses a new filtering mechanism, based on a combination of block maxima and bitmaps, that radically reduces the number of documents that have to be further evaluated. Our filtering mechanism exploits the SIMD processing capabilities of current microprocessors, and it is optimized through caching policies that select and store suitable filter structures based on properties of the query load. Our experimental evaluation shows that the mechanism results in very significant speed-ups for disjunctive top-k queries under several state-of-the-art algorithms, including a speed-up of more than a factor of 2 over the fastest previously known methods.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval

General Terms

Algorithms, Performance, Experimentation

Keywords

top-k query processing, early termination, block-max inverted index, docID-oriented block-max index, candidate filtering mechanism, posting bitset, live area computation

1. INTRODUCTION

One of the major problems for large search engines is to keep up with the tremendous growth in the size of the web and the number of queries submitted by users. As discussed in [12], each of these measures increased by several orders of magnitude over the course of a decade. This creates performance challenges in many parts of the search engine architecture, including data acquisition, data analysis, and index maintenance. However, these challenges are particularly acute in the query processing component, whose workload grows with both data size and query load. In fact, one might naively expect query processing costs to increase with the product of these measures, though in practice various techniques are used to limit this increase. In this paper, we focus on such techniques for improving query processing efficiency.

Query processing in search engines is a fairly complex process, typically involving hundreds of features used for ranking, multiple phases that identify promising documents based on subsets of the features, and a distributed architecture that routes queries and results within and between clusters of thousands of machines. However, most systems appear to process a query by first evaluating on each machine a fairly simple ranking function over an inverted index. This results in an initial set of say a few thousand results that is then further processed to identify the best ten or more results to return to the user [26]. We focus on improving this initial step, which is responsible for a significant fraction of the overall work. We assume that this initial step is a ranking function of the form $r(q, D) = \sum_{t \in q} s(t, D)$, i.e., it is the sum of the scores of all the terms occurring in the query. This is a fairly common assumption, and accommodates popular functions such as BM25 as well as approaches based on automatically learning term scores that approximate more complex ranking functions [1].

A straightforward way to execute such a simple disjunctive ranking function is to completely traverse the index structures of all the query terms and compute or retrieve the scores of all the postings. Under this *exhaustive* method, the cost of each query increases linearly with the number of documents, making it very expensive for large collections. To overcome this problem, many researchers have proposed so-called *early-termination* techniques, i.e., techniques for finding the top results without computing or retrieving all posting scores [25, 9, 6, 23, 15, 10]. We focus on *safe* early-termination techniques, which must return the same top results as the exhaustive method [22].

Content of this Paper: Recent work by several groups of researchers [10, 15, 19, 13] has shown significant improvements in performance based on a *Block-Max Index*, first proposed in [10, 15]. This is an index structure that stores maximum scores for groups of postings, thus allowing quick

SIGIR'13, July 28–August 1, 2013, Dublin, Ireland.

skipping, without full evaluation, of sets of postings with low scores. In this paper, we build on this approach to derive new algorithms with additional very significant performance gains. In particular, our contributions are as follows:

(1) We design a general mechanism that can improve the performance of a whole class of query processing algorithms. Given a space budget, we maintain a set of block-max and posting-bitmap structures that can be used to transparently accelerate any DAAT index traversal algorithm. We show how to exploit the aggregated information from the auxiliary structures for aggressive candidate filtering.

(2) We present a space efficient caching policy for storing an optimal set of structures given a query distribution.

(3) We propose an optimized implementation of the mechanism that exploits SIMD instructions of modern CPUs and restricts the critical data structures to L1 cache.

(4) We provide an extensive experimental evaluation that shows significant improvements in query throughput for multiple algorithms. Our fastest results are more than a factor of two faster than the best previous results.

The remainder of this paper is organized as follows. In Section 2 we outline the background and related work. Section 3 describes the proposed filtering mechanism that uses Block-Max Indexes. Next, in Section 4 we present how to augment the mechanism with posting information. Section 5 explains our proposed solutions for the space overhead of the auxiliary structures and Section 6 presents preliminary results on our proposed methods. In Section 7 we show the performance of the filtering mechanisms on several scenarios, whereas in Section 8 we discuss caching policies for our mechanism. Finally, Section 9 presents additional results and Section 10 concludes and discusses future work.

2. BACKGROUND AND PREVIOUS WORK

In this section, we provide some background on inverted indexes, query processing and early termination techniques, Block-Max Indexes, and describe previous work.

2.1 Inverted Indexes

Inverted Indexes: An inverted index [27, 30] is a simple and efficient data structure widely used by large search engines for query processing. This structure allows finding the documents where specific terms occur and can be formally defined as follows. 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 an inverted list L_w for each distinct term w in the collection. Each L_w is a list of postings describing all documents where term w appears in the collection. More specifically, each posting contains the ID of the document (docID) that contains the term w, the number of occurrences of w in the document (the frequency), and potentially additional information such as the exact positions of these occurrences of term w in the document (positions) or other data. Unless stated otherwise, we store docIDs and frequencies such that each posting is of the form (d_i, f_i). There are several ways to layout the inverted lists, but in this work we focus on document-sorted indexes, where postings in each list are sorted by their docIDs d_i; see [18, 16, 6, 8, 23] for work using other layouts.

Index Compression: The inverted lists of frequent terms may contain millions or even billions of postings. To decrease space requirements, inverted lists are usually kept in compressed form. A common approach for docID-sorted indexes is to store the differences between consecutive docIDs in a list in suitably compressed form [30]. Index compression is crucial for search engine performance, and many compression methods have been proposed; see [29, 28, 21] for some recent work. For fast access, inverted lists are usually compressed in blocks of, say, 64 or 128 postings, such that each block can be accessed and decompressed individually. This is done by storing the uncompressed last docIDs and sizes of all blocks in a separate array that can then be used to locate and decompress individual blocks. Here, we compress the inverted lists into blocks of 64 docIDs followed by the corresponding 64 frequencies.

Index Quantization: We usually store each posting as a pair (d_i, f_i). However, it may sometimes be preferable to store postings as (d_i, s_i) where s_i is a score of d_i with respect to w. There are two main motivations for this: (1) to save the cost of computing scores from frequency values and other statistics such as document size and global term frequency, and (2) for scores that cannot be easily computed on the fly as they are derived from complex ranking functions with hundreds of features, using techniques similar to those in [1]. However, storing each s_i as a 32- or 64-bit number would result in very large index sizes. To reduce the size, so-called *index quantization* techniques are used that basically round the floating point score values to one out of a fixed set of, say, 256 distinct values. This may lead to a slight loss in precision during ranking, but guarantees that each score can be stored in only 8 bits. There are several quantization techniques in the literature; see [2, 3, 5, 4]. In this paper, we use the *Global-by-Value* technique proposed in [5], and also used in [11], using 8 bits for each score. All necessary score accumulations can be done directly in integer space, as in [2]. We refer to such an index structure as a *quantized index*, in contrast to a *standard index* storing frequencies values.

Index access: Document-sorted indexes allow fast index traversal during query execution based on a *Document-At-A-Time* (DAAT) approach. In DAAT index traversal, every list has a pointer to a current posting, and all pointers move forward in a synchronized manner as a query is processed. Thus, all postings (and thus documents) to the left of the pointers have already been processed, while postings to the right are yet to be considered. In DAAT, we typical have functions for opening and closing inverted lists for reading, and a function *nextGEQ* that, given a docID d and an open inverted list L, moves the pointer in L forward to the first posting with docID greater or equal to d. In addition, there is a function for retrieving other data (such as frequency or score) associated with the current posting. All decompression operations in the inverted lists are hidden within these functions. In this work, we focus on algorithms that perform DAAT traversal on document-sorted indexes.

2.2 Query Processing and Early Termination

The simplest way to query an inverted index is to ask for all documents containing some (disjunctive) or all (conjunctive) of a set of query terms. Of course, this would return too many results on large document collections, and thus IR

systems supporting ranked queries return the highest scoring documents among those that satisfy the condition. Many simple ranking functions have been proposed for this task, including BM25 and the Cosine measure [7]. These functions are typically of the form $r(q, D) = \sum_{t \in q} s(t, D)$, meaning that the score of a document with respect to a query is the sum (or some other simple combination) of the query term scores of the document. In this paper, as in almost all previous work, we assume disjunctive queries with a ranking function of this form, that is, our goal is to return the highest scoring among all documents containing at least one of the query terms.

We note that such disjunctive queries tend to be more expensive than conjunctive queries that only need to score documents containing all query terms. For this reason, many search systems try to use conjunctive queries whenever possible. However disjunctive queries are known to return better results in many situations, and thus it is important for search engines to also support these more expensive types of queries.

Cascade Ranking: The above simple ranking on top of a disjunctive or conjunctive filter is only the first phase of query processing in current search engines. These engines will typically take a limited number k of top-scoring results from the simple ranking, say a few thousand overall, or as little as tens per node on a large cluster of machines, and evaluate a second, more complicated but more precise, ranking function on these results. This process may be repeated with an even more evolved ranking function applied to the top results from this ranking. Thus, we expect our top-k algorithms to be the first phase of such an architecture, and will look at the effect of varying k from 10 to a few thousand.

Early Termination: The main performance bottleneck that we have to deal with is the length of the inverted lists, which can grow to many MBs or even GBs on large collections. As mentioned, a simple algorithm for this problem would compute the score of any document containing at least one of the query terms; we call such an algorithm exhaustive. To avoid scoring many of these documents, early-termination (ET) algorithms have been proposed. We say that an ET algorithm is safe if it outputs exactly the same top k results as an exhaustive algorithm [22]. (In the case of a quantized index, safeness is of course defined with respect to an exhaustive algorithm on the full quantized index.)

There are many safe and unsafe ET algorithms in the literature; see, e.g., [25, 9, 6, 23, 15, 10] for a small sample. Some algorithms keep inverted lists sorted by docID, while others reorganize the lists by score $s(q, D)$, so that all high-scoring postings are encountered early during query processing. In this paper, we focus on safe early termination. All our algorithms keep the inverted lists sorted by docID and use DAAT for index traversal – but our running times also outperform all safe methods with other index layouts.

2.3 ET Algorithms

In this part, we first describe a simple exhaustive algorithm and then outline two basic safe ET algorithms that use DAAT traversal on document-sorted indexes, the WAND algorithm in [9] and the Maxscore algorithm in [25]. Both algorithms store for each inverted list the highest impact score of any posting, called the *maxscore* of the list. Recall that in DAAT traversal, we maintain a pointer to the current posting of each list. Also, we use θ to refer to the current threshold, which is the smallest score that can still make it into the top k results during the execution of the algorithm.

Exhaustive: The simplest query processing algorithm first picks the smallest current docID across all query terms, called *cID*. This can be done using either a loop or a heap, and cID is then fully evaluated. Afterwards, all pointers are moved to at least the next posting after docID cID. We also maintain a heap of the current top-k results, and add cID to this heap if it has a score larger than the current k^{th} highest score.

WAND: This algorithm, proposed in [9], consists of three phases: pivot selection, alignment check, and evaluation. In every iteration, a pivot term is selected by summing up the maxscores of the participating lists in order of increasing current docID, until the sum becomes larger or equal to θ. Next, WAND tries to align the current docIDs of the preceding lists in the ordering with the pivot docID, by moving the pointers in these lists forward to the pivot docID. If this succeeds, and all pointers up to the pivot point to the same docID, then this docID is evaluated, if necessary inserted into the top-k heap, and the current pointer in the pivot list is moved forward. Then we go to the next iteration.

Maxscore: This algorithm was first described in [25]. We focus here on the DAAT versions of the algorithm in [24, 17, 13]. Maxscore splits the query terms into essential and non-essential terms as follows. We sum up the maxscores of the lists from lowest to highest while the sum remains less than θ. The terms that contribute to this sum form the non-essential terms, and the others are the essential terms. The basic idea is that any document that can make it into the top k results must contain at least one essential term. Now we select as candidate, the smallest current docID of any essential list, and evaluate it. Thus, we basically run the exhaustive algorithm on the essential lists only. Of course, as the threshold θ for making it into the top k increases during execution of the algorithm, we may move additional terms from the essential to the non-essential set.

2.4 Block-Max Indexes

The idea of storing and exploiting the maximum impact score of each inverted list was recently and independently extended by two research groups [10, 15], who proposed an augmented index structure called *Block-Max Index* (BM). This structure stores the highest impact score for each block in the inverted lists, where the blocks are defined by the index compression method. Recently in [13], the blocks of the Block-Max structures were decoupled from the index compression and defined on docID space rather than posting distribution. We will refer to Block-Max structures whose blocks are defined by postings as *posting-oriented*, while the ones defined on docID space as *docID-oriented*.

2.5 Previous Work Using Block-Max Indexes

In this paper, we build on the work of [15, 10, 13], where we generalize and extend these ideas by proposing a filtering mechanism that uses Block-Max Indexes to further improve the query processing for disjunctive queries.

More specifically, [10, 15] proposed the posting-oriented BM structure to approximate the score of a block in posting space. The Block-Max Indexes provide score approximations in posting block resolution that enables skipping of compressed blocks. The approach in [15] proposed an enhancement of the WAND [9] algorithm that uses the Block-

Max index, called BMW, and operates as follows. After the sorting and pivot selection step as in [9], there is an additional block-max check that tests whether the pivot docID can make it into the top-k results. In case the block-max test fails, BMW can safely skip until the end of the blocks. The work in [10] suggested an algorithm that is built on top of the Maxscore and also uses posting-oriented BM structures. In particular, given a query, the algorithm reads the stored BM information, generates aligned intervals in the docID domain, and computes upper bound scores for each interval. During query processing, each interval's upper bound score is checked whether is larger or equal to the k^{th} highest scoring document seen so far and if the test fails, the algorithm can skip blocks and safely save computations.

Both previous studies show experimentally that these auxiliary indexes provide fast query response times with small space requirements. However, the skipping power offered by posting-oriented BM indexes is limited because the blocks are defined by the index compression method. More recently, a new layout of the BM was proposed [13], where the blocks are defined on the docID domain. The blocks of the BM structure are decoupled from the index compression and their block size can be assigned per-term using the fixed, the expected or the variable block size selection method. The docID-oriented BM indexes provide very fast lookups, because the block-max access can be performed with bit shift operations. The authors in [13] use the variable size selection scheme to deal with the space overhead of the structure and show that several algorithms perform faster using the proposed BM index. We refer to [15, 10, 13] for more details on Block-Max indexes and query processing algorithms using such structures.

In this work, we extend the idea of constructing aligned intervals in docID space and computing interval upper bounds as in [10] by generalizing it on the docID-oriented BM structure [13] with fixed block size. The result is aligned BM blocks for all lists, allowing fast block-maxscore accesses and aggregation of "aligned block" upper bounds. We build on this scheme and propose a filtering mechanism that utilizes docID-oriented BM indexes to discover the "live" blocks in consecutive windows of docID space, which leads to significant performance gains over a series of DAAT algorithms for disjunctive queries. For the remainder of the paper we assume docID-oriented BM using the fixed block size selection method.

3. LIVE BLOCK MECHANISM

In this section we describe our proposed candidate filtering method that exploits the Block-Max index.

Live Area Computation: We can model DAAT query processing algorithms as a sequence of rounds, where in each one we examine whether a specific candidate docID can make it into the top-k results (by evaluating it or safely early terminating). Since our goal is to retrieve the top-k results fast, we can reduce the cost of query processing by minimizing the number of candidate docIDs. In previous works [10, 15, 13], the BM structure is utilized for candidate filtering in an online fashion, meaning that in each round there is a block-maxscore test checking whether the candidate has a chance to be added in the result heap. However, in our approach the aggregation of the upper bound block-maxscores across query terms for consecutive aligned blocks is performed in batch mode and then exploited to aggressively skip blocks. Due to the docID-oriented BM layout, there is no need for the alignment step as needed in [10].

In typical DAAT query processing algorithms, at the end of each round, the candidate docID for the next round is selected. The selection of this document is always preceded by invocation of the nextGEQ() on some lists. Since nextGEQ() usually demands expensive block decompression, we would prefer to make this decision based on the available block-max information. Therefore, given a query with m terms and a threshold θ, we define a block as *live* if the sum of block-max scores across m terms is greater or equal to θ. Thus, we can safely skip the block and avoid redundant decompressions and score calculations when a block is "dead". Remark that in [13] a similar optimization was suggested by the BMM-NLB algorithm, but in their setting there is no enforced block-max alignment.

We extend this idea by computing the live blocks of a *window* of docID space, where window refers to a contiguous docID region. Thus, we calculate block-maxscore sums for all blocks of a window. By definition, only blocks that summed up to more or equal than θ (of the specific window) are live blocks. We will refer to this procedure as *Live Area computation (LA)* of a window. This per-window liveness information is stored in a structure called *Liveness Bitset (LB)*, where each bit corresponds to a block of the window. LB is used for candidate filtering by skipping "dead" blocks. In particular, our mechanism first obtains the next live block from the LB and then provides the first docID of this block to the nextGEQ(). The live area computation is performed prior to query processing to provide efficient filtering oracle for candidate selection. Note that at the end of query processing of each window, we re-use the memory occupied by the LB structure to avoid the costly memory initializations.

Windows: The window size under this setting is crucial since the number of live blocks depends on the most updated θ. When the size of the window is too small, we pay for the construction overhead (LA). On the other hand, if the window size is too large, the effectiveness of the LB would drop due to outdated thresholds. In practice we observed that the appropriate window size should be a function of L1 or L2 cache size. As you advance in docID space (and θ grows), LB becomes more sparse, and hence, more effective. In preliminary experiments we observed that the convergence rate of θ is quite fast.

SIMD: The computation of live areas is performed during query processing (online) per window, thus it must be fast. This process includes the summation of the aligned block-maxscores across terms, the comparison of this sum with θ, and the encoding of the liveness information in LB. Since all inverted lists block-maxes are aligned in docID space, the LA calculation is vectorizable and SIMD capabilities of modern CPUs can be exploited to accelerate this computation. Our calculation of LA is coded for the SSE instruction set and performs the following operations: (i) load 4 consecutive block-maxscores from each query term, (ii) sum the block-maxscores vectors, (iii) compare the results with θ, and (iv) set the corresponding bits in LB. In a nutshell, LA's calculation overhead is negligible because of the natural speed-up of vectorized operations (SSE) and the limitation of LB to L1 cache. Note that SSE also applies to the integer operations used in the quantized index scenario.

In Figure 1, we see an example of LA computation for the 2-term query "cat squirrel" of 4 consecutive block-maxscores.

In this example, the BM scores for both terms are shown in grey, whereas their sum is colored green if the block is live, otherwise it is red. Under this scenario, only the third bit in LB is set. Since we are using SSE for the computation, every step is performed in one instruction. Thus, for any 2-term query we only need 5 instructions instead of 20 to compute the liveness of 4 consecutive blocks. The obtained information can filter all blocks in the window that could not make it into the top-k results and therefore, save decompressions and evaluations.

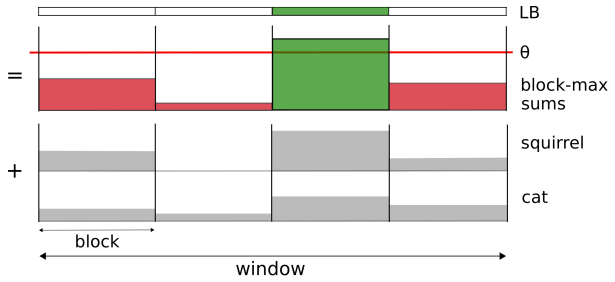

Figure 1: LA computation using terms' BM structures.

Varying Block Size and Threshold: The window-based LA calculation is computed using the most updated θ. The early knowledge of a good threshold makes the filtering mechanism very effective since less blocks will be live. In Figure 2, we show the percentage of live blocks for various block sizes and for different fixed thresholds (θ is not updated). We observe that the number of live blocks decreases when (i) decreasing the size of the block and (ii) providing higher θ. Since larger blocks provide worse block-score approximations, with higher threshold they have better chances of being live. These observations provide a strong evidence about the effectiveness of the proposed filtering mechanism. In practice, when θ is updated per window, the decrease in liveness is more significant (not linear).

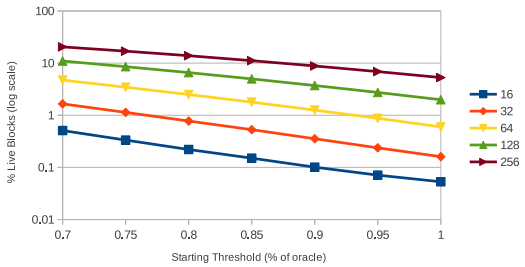

Figure 2: Live block percentage varying block size and fixing the starting threshold, for the TREC06 query trace.

Filtering Mechanism Interface: Our proposed window-based candidate filtering mechanism provides a simple interface that can be easily used by any DAAT query processing algorithm. We now describe the various low-level components of our framework so its adaptation to existing or new algorithms becomes trivial.

In this part, we outline the computation of LB structure and how to utilize it during query processing. First, the LA computation is performed whenever the termination condition of DAAT algorithms occurs. More specifically, these algorithms terminate when the docID of the candidate document is greater than the largest possible docID in the collection. At this point, the algorithms using our mechanism should be adapted to operate in the boundaries of the current window. Thus, whenever the docID of the candidate document exceeds the current window boundaries, the LA computation should be performed with the current θ. This consists of block-maxscores summation, comparison to θ for all blocks of the window, and setting the appropriate bits in the LB. Note that if the BM scores are not available, our mechanism needs direct access to the index to compute them. The liveness information of the LB can be accessed by a core function called *nextPotentialLiveDid*, which given a docID returns the first docID of the current or the next live block. The returned docID and a specific list are then provided to the nextGEQ(). We hide the complexity of this procedure by introducing a new function called *newNextGEQ*, which encapsulates both nextLivePotentialDid() and nextGEQ(). Note that the original nextGEQ() performed on the "live" candidate can still return a docID in a "dead" block. Thus, instead of calling the nextGEQ() the algorithms using our filtering mechanism could just use the new function newNextGEQ.

To sum up, algorithms could use our filtering mechanism by first changing the halting condition to be windows aware, then performing the described LA computation whenever the candidate docID is out of the current window boundary, and finally, by calling the newNextGEQ() instead of nextGEQ(). Hence, our mechanism offers a simple interface to query processing algorithms, by hiding all the complexity behind few functions.

Architecture of the Filtering Mechanism: We now describe how our technique communicates with the available data during query processing. Figure 3 shows our query processing pipeline, which consists of the Index, the Query Processor, the Filter Mechanism and the Filter Cache. The Filter Mechanism has direct bulk access to the Index so that it can generate the augmented structures on-the-fly for lists out of the Filter Cache, as we will describe in Section 5. Hence, the Filter Cache contains augmented structures (such as the BM Index and the LB) that are accessed and exploited by the Filter Mechanism for effective skipping during query processing. The Query Processor uses the core function nextGEQ() to access the Index and move the pointers of the inverted lists. It may also use the Filter Mechanism to request the liveness bit of a docID or the maxscore of a block. In that case, the information is served by the Filter Cache and returned to the Query Processor. Remark that the communication between the Query Processor and the Filter Mechanism can in fact be encapsulated within the newNextGEQ() operator.

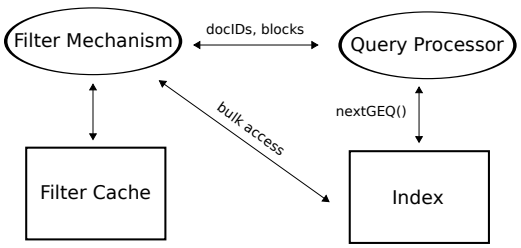

Figure 3: Filtering mechanism architecture.

4. POSTING BITSET (PB)

The LB provides a smart exploitation of the BM structure with performance gains as we will experimentally show in the next section. Its effectiveness to filter candidate docIDs relies solely on the approximation quality of the block-maxscores. The early termination methods of algorithms using BM structures are mainly based on the score approximation. However, we could also filter candidate docIDs using additional information about the postings existence in a block. For example, a block without postings does not contribute to the block-maxscore aggregation in LA. In fact, a block-maxscore is non-zero if and only if the block contains postings. In order to exploit the additional posting existence information of a term we encode it as follows: for a contiguous docID region called *sub-block*, we store a bit describing whether the term contains any posting in it. Thus, for each term we store a *Posting Bitset* (PB), where each bit represents the posting liveness of a specific sub-block. This encoding offers extremely fast accesses to the PB structure (because sub-blocks are docID-oriented).

The PB can be seen as a per-term filtering method that query processing algorithms can benefit from. In particular, this filtering mechanism consists of a basic function called *nextLivePotentialPosting* that given a docID and a list, returns a docID belonging to the current or next populated sub-block of the given list. Moreover, this function can be used to answer whether a term has any postings in a specific region, thus saving redundant computations such as decompressions of blocks and score calculations. Similar to the LB mechanism, the PB filtering technique also requires direct access to the index for construction. The PB structure can be easily combined with the BM index during LA computation to provide finer granularity candidate filtering as we will shortly see.

4.1 LB-PB Mechanism

In the previous section, we proposed the PB structure that provides information about the posting liveness. This structure can be utilized during the LA computation to provide finer granularity filtering. In order to obtain a useful combination of the BM index and the PB, we need to meaningfully select the block size of the sub-blocks. Note that having a sub-block larger than the BM block does not introduce new information. We experimentally observed that a good size for the sub-blocks is 2^{m-3}, where 2^m is the size of the BM block. For the rest of the paper, we assume that the sub-block size is 2^{m-3}.

Our assumption about the sub-block size implies that every BM block is split into 8 sub-blocks. Therefore, for each BM block we have additional information about the posting liveness of its sub-blocks. Utilizing the PB structure in LA computation requires the following steps: first every block-maxscore is duplicated 8 times and masked with 8 sub-block bits, then the sub-block level scores are summed across terms, and finally, we compare the sums with the current most updated θ and set the appropriate bit in the liveness bitset. Note that in our case, the temporary PB structure is 8 times larger than the corresponding structure we used for LB. We will refer to this structure that provides an effective candidate filtering mechanism for the query processing algorithms, as *LB-PB*. Similarly to LB, during query processing the algorithm can provide a docID to the LB-PB to obtain the next live sub-block in the window, avoiding redundant computations such as decompressions and score calculations. Given a docID, the mechanism returns a docID in the current or the next live sub-block. Remark that this filtering technique can exploit both nextLivePotentialDid and nextLivePotentialPosting functions, that now operate on sub-block level, and its complexity can be hidden as well behind the newNextGEQ().

In Figure 4, we describe the live area computation of the query "cat squirrel" for 4 BM blocks using the LB-PB mechanism. The BM scores of both terms are colored in grey, and only the PB of the third block is shown. The non-empty sub-blocks are shaded. The available PB information enables finer block-maxscore granularity on sub-block level. Focusing on the third block, the LA-PB computation includes the multiplication of the posting bitset of the third block of each term with the corresponding BM score. The result is depicted as grey sub-blocks, whereas the summation of the aligned sub-blocks across terms is shown as red or green sub-blocks sums. We can alternatively think this procedure as an alignment step in sub-block resolution. After the LB-PB filtering, only the sixth sub-block of third block was found to be live, while the original LB mechanism would label the entire third block (8 sub-blocks) as live. This example shows the effectiveness of the LB-PB mechanism and its superiority over the simple LB filtering technique.

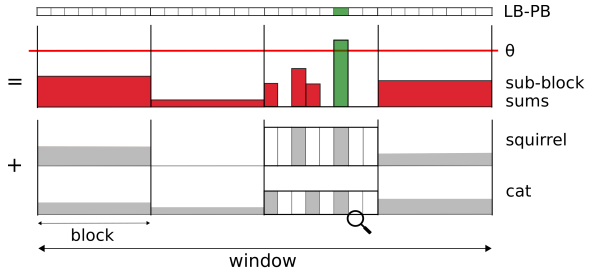

Figure 4: Example of the LB-PB filtering mechanism.

SIMD: Similarly with the LA computation, the calculations of LA when combined with the PB structure can exploit the SIMD opportunities of modern CPUs. All steps of the LA computation of the LB-PB mechanism are vectorizable and therefore, the SSE instruction set is used in the implementation to speed up the calculations.

Varying Block Size and Threshold: Figure 5 presents the percentages of the live sub-blocks for various block sizes and different fixed thresholds. As in Figure's 2 observations, smaller block sizes and higher starting threshold result in less live sub-blocks, as expected. Comparing the percentages of Figure 2 and 5, we observe that the combination of LB and PB leads to less live blocks. These statistics provide evidence that the proposed candidate filtering mechanisms could give significant performance gains.

5. SPACE OVERHEAD

In this section we discuss how to deal with the space overhead of the auxiliary structures.

5.1 Block-Max Index

So far we have not mentioned the space overhead of the BM index needed during the LA computation and we assumed BM resides in memory. In practice, the BM structure

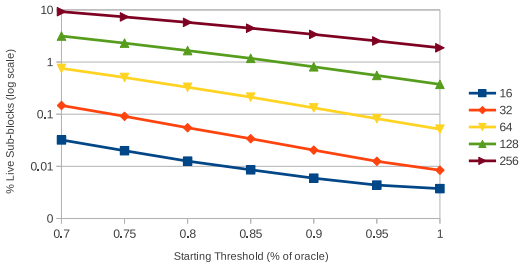

Figure 5: Live sub-block percentage varying block size and fixing the starting threshold, for the TREC06 query trace.

with fixed block size is too large to be memory-resident, so we address the space overhead using the approach in [13], and we will refer to this as *Length-based*. Under this method, the BM for terms with few postings are not stored at all, but generated on-the-fly (OTF), while the BM scores for the rest are undergoing linear quantization (BMQ), such that every block-maxscore is 1-byte (instead of 4). The difference in our setting is that the OTF is now performed per window. The window-based OTF computation involves (i) the decompression of the blocks (docIDs and frequencies), (ii) the score calculation of these docIDs, and (iii) the Block-Max Generation (BMG). We further extend the length-based approach by compressing the quantized Block-Max values (BMQC) of the medium-sized lists. More specifically, we first create a skiplist for non-zero values and then we compress them using a version of PForDelta [29]. This requires decompression of the compressed quantized values during query processing, but its overhead is minimal compared to its space gains.

5.2 Posting Bitset

The size of the sub-block plays a crucial role on the effectiveness of the posting bitset structure. Very large sub-blocks provide poor posting existence information, whereas too small sub-blocks add significant space overhead. Overall, we address the space overhead of the posting bitset structure by following a length-based approach as in the case of the BM structure. A first observation is that for long lists we expect most sub-blocks to be non-empty, hence a PB with all bits set is a decent approximation for it. Instead of storing the actual PBs for such lists, we "fake" them by pointing to a static array of 1's, thus saving this space. We experimentally observed that this approximation of PB does not impair filtering abilities and does not cause a slow down. Moreover, since we already decompress the short lists in OTF BMG, we can use the docIDs to construct the PBs on-the-fly as well. The PB structures are stored in main-memory only for the medium-sized terms. Note that the construction of PB should be also considered under the window-based scenario as described in the previous section.

We will revisit the length-based approach for both structures in Section 8 to achieve a better space/time tradeoff.

6. PRELIMINARY EXPERIMENTS

In this section, we evaluate and compare the performance of the simplest query processing algorithm, the exhaustive, when our filtering mechanisms are applied, with the best previously reported numbers to our knowledge [13] under the scenario of a standard index.

6.1 Experimental Setup

For our experiments, we use the TREC GOV2 dataset that contains 25.2 million web pages. We build inverted index structures with 64 docIDs and frequencies per block, using a version of the PForDelta compression method [31] described in [29], but our ideas also apply to other compression methods. The size of the uncompressed size of the data set is 426GB, whereas its compressed standard index size is 8.75GB and its compressed quantized index 11.97GB. We evaluate our methods using 1000 queries randomly selected from the TREC 2006 and 2005 Efficiency track query set.

We implemented all the algorithms in C++ using BM25 as our ranking function, and return top-10 results unless stated otherwise. For the rest of the paper, we assume docID-oriented BM using the fixed block size selection method with 64 docIDs per block. The experiments were conducted on a single core of an Intel Xeon server with 2.27Ghz and all data structures reside in memory.

6.2 Baseline algorithms

Setup: We begin with a performance comparison of the exhaustive algorithm and its versions that use the LB and the LB-PB mechanisms, called *EX*, *EX-LB* and *EX-LB-PB* respectively, on the standard index. For this experiment we use the following length-based policies for the BM: (i) OTF for lists with length less than 95042, (ii) BMQC for lists between 95042 and 2^{18}, and (iii) BMQ for the remaining ones. Similarly, for the PB structure we use the subsequent length-based rules: (i) OTF for lists with length less than 179758, (ii) maintaining in main memory the posting liveness information for lists with length between 179758 and 2^{22}, and (iii) faking of bitsets for lists larger than 2^{22}. The space requirement for the described policy for the GOV2 data set is $2GBs$, less than 25% of the compressed index, while the use of PB increases the space to $4GBs$. We also compare the performance of the exhaustive algorithm when the filtering methods are applied with the fastest previous results to our knowledge [13]. More specifically, the approach in [13] uses docID-oriented BM structures, variable block size selection methods and both OTF BMG (up to lists of size 2^{15}) and BMQ techniques (on the remaining lists), with $2.76GBs$ space overhead. The authors of [13] were willing to share the code for their fastest algorithm, called BM-OPT, which selects based on the number of query terms the fastest algorithm between BMM, BMM-NLB and BMW.

Exhaustive speed-up: In Table 1, we present the query response times for the BM-OPT algorithm and all versions of the exhaustive algorithm for the TREC06 query trace, when the number of query terms varies. We can see that the LB filtering mechanism makes the exhaustive algorithm faster by a factor of 10, whereas the availability of PB boosts the exhaustive 16 times! The exhaustive algorithm is significantly accelerated by the availability of the LB mechanism achieving average query response time of $13.47ms$. When the LB-PB is used it further reduces the average time to $8.59ms$, 16 times faster over EX. The performance boost of the mechanisms is consistent across varying number of query terms. Table 2 shows the radical decrease in term score evaluations and calls to nextGEQ() when the proposed mechanisms are applied. The reason for the significant reduction in query response times, evaluations and nextGEQ() is the following: at the end of each iteration of any query processing algorithm, the pointers of some or all lists must

be moved using the expensive nextGEQ() (which usually includes decompression). Our mechanism checks the LB (or the LB-PB) structure in order to obtain the next live block (or sub-block) and then provides to the nextGEQ() the first docID of this block (or sub-block). Therefore, this test before the call of nextGEQ() moves the pointers much further in docID space and results in much less redundant expensive decompressions, which is the main reason for the significant performance speed-up of our mechanisms.

Comparison with BM-OPT: Furthermore, we see that the performance of both EX-LB and EX-LB-PB outperforms that of BM-OPT. Concerning the space requirements of the algorithms, as mentioned, the EX-LB occupies $2GBs$, less than the $2.76GBs$ needed by the best previous approach. Thus, our filtering method uses less space to achieve faster times compared to the state-of-the-art algorithm, BM-OPT. When the PB structure is also available the space increases to $4GBs$ and EX-LB-PB achieves much faster query response times compared to BM-OPT. Remark that our filtering mechanisms accelerate significantly the simplest query processing algorithm, the exhaustive, which outperforms the state-of-the-art BM-OPT algorithm.

Algorithm	avg	2	3	4	≥ 5
EX	138.98	37.68	97.06	166.39	301.24
EX-LB	13.47	2.71	5.89	16.63	35.08
EX-LB-PB	8.59	2.65	5.07	10.72	18.69
BM-OPT	14.96	3.81	9.91	18.8	34.29

Table 1: Performance of the exhaustive algorithm on the availability of our mechanisms for the standard index and the TREC06 query trace.

Algorithm	Time (ms)	# nextGEQ/query	# evals/query
EX	138.98	4,489,430	4,489,430
EX-LB	13.47	195,080	194,741
EX-LB-PB	8.59	86,792	86,454

Table 2: Number of term score evaluations and calls to nextGEQ for the standard index and the TREC06 query trace.

7. EXPERIMENTS

In this section, we present a performance comparison of widely used DAAT query processing algorithms when our filtering techniques are applied. These algorithms either use (BMM-NLB, BMW) or not (WAND, Maxscore) the BM structure for early termination during query processing.

7.1 LB-PB on DAAT algorithms

In particular, the WAND algorithm was implemented based on [9], Maxscore and BMM-NLB were coded according to [13], while the BMW algorithm based on [15]. All algorithms were modified in order to use the filtering mechanisms interface as previously described.

The performance comparison of these algorithms for the TREC06 and TREC05 query trace when we vary the number of query terms is presented in Table 3 and 4. The significant speed-up of our candidate filtering mechanism over several query processing algorithms shows the effectiveness of the filtering. Table 3 shows that Maxscore outperforms WAND when any filtering method is applied. On algorithms that use the BM index to early terminate documents, BMW is consistently faster from BMM-NLB. We observe that the performance of Maxscore is similar to BMM-NLB and the reason is that BMM-NLB spends too much time during its several cascading filtering steps, which are not very useful when our mechanisms are used. BMW outperforms all algorithms with $6.45ms$ using the LB and $5.74ms$ when PB is also available. A similar performance behavior is observed for the TREC05 query trace in Table 4. Enforcing candidate selection with LB (and LB-PB) in all algorithms consistently accelerates them. However, this was apparent from preliminary experiments, where the simplest algorithm outperformed the state-of-the-art BM-OPT. Due to space limitation, for the remainder of the paper we omit any results for the TREC05.

LB ($2GBs$)					
Algorithm	avg	2	3	4	≥ 5
WAND	11.53	2.51	4.97	13.86	31.15
Maxscore	8.1	2.3	4.17	10.08	19.4
BMM-NLB	8.29	2.13	3.97	10.2	20.38
BMW	6.45	2.03	3.45	7.63	15.44
LB-PB ($4GBs$)					
WAND	8.19	2.59	4.7	10.05	18.23
Maxscore	6.22	2.34	3.93	7.89	12.68
BMM-NLB	6.21	2.21	3.73	7.88	13.06
BMW	5.74	2.18	3.59	7.04	12.03

Table 3: Performance comparison of several DAAT query processing algorithms when LB and LB-PB mechanisms are applied on standard index, for the TREC06 query trace.

LB ($2GBs$)					
Algorithm	avg	2	3	4	≥ 5
WAND	10.22	2.51	5.02	7.56	47.19
Maxscore	5.62	2.18	4.13	5.75	20.37
BMM-NLB	6.19	2.03	3.98	5.66	24.88
BMW	5.35	1.96	3.58	4.89	20.79
LB-PB ($4GBs$)					
WAND	7.66	2.62	5.02	6.9	29.98
Maxscore	5	2.32	4.21	5.41	15.03
BMM-NLB	5.21	2.18	4.06	5.24	17.22
BMW	5.12	2.15	3.93	5.03	17.11

Table 4: Performance comparison of several DAAT query processing algorithms when LB and LB-PB are applied on standard index, for the TREC05 query trace.

8. SPACE/TIME TRADEOFF

In this section, we discuss how we can further optimize both space and time constraints by moving from simple length-based caching policies to cost-based.

8.1 Length-based Caching

In the previous experiments, the performance of our methods and their space requirements were based on decisions governed by the length of the lists, called *Length-based* policies. These policies include (i) OTF computation of BM and PB for short lists, (ii) storing compressed quantized BM for medium lists (BMQC), (iii) faking of PB for long lists, and (iv) storing quantized BM and PB for all remaining lists.

For the previous experiments, the length-based parameters were fixed to specific values using acceptable space requirements to show the effectiveness of our mechanism. Although the length-based caching provides a good space/time tradeoff, it is solely depends on the length of the list. More specifically, it forces short lists' augmented structures (BM and PB) to be computed OTF, whereas it always stores in the Filter Cache these structures for large-sized lists. This is

not always optimal, since it is likely that some short terms will occur frequently in the query trace and thus, it may be more beneficial to store their augmented structures rather than computing them OTF. Therefore, we would like to move to policies that take into account the frequency of terms in the query workload and offer better space/time tradeoffs.

8.2 Cost-based Caching

It is common that large web search engines use caching mechanisms so their performance adapts and optimizes to the query workload. Such mechanisms maintain statistics about the terms appearances in the query logs and based on them, decide whether to cache the additional information. The main idea behind such approach is to achieve performance speed-up by using less space smartly. Therefore, we try to maximize the obtained performance gains by following optimal policies for each term according to query statistics and under specific space requirements. We will refer to the proposed policies that are based on a benefit function that takes into account the frequency of the terms in the training query trace, as *Cost-based* policies. In particular, if a short term occurs very frequently in the query log, it would be reasonable to maintain its augmented structures in memory. On the other hand, if a long term does not appear too often in the queries, we refrain from storing it in memory.

The proposed cost-based caching policy follows the next steps: first, for each term we compute a rank under various policies P_i as follows $\frac{time(OTF)-time(P_i)}{space(P_i)} * freq(term)$. The policies for the BM index include the length-based policies (i) OTF, (ii) BMQC and (iii) BMQ. Similarly, the policies for the PB structure consist of (i) OTF, (ii) PB, and (iii) faking of PB. Then, given a space budget, we select the terms (from highest benefit rank to lowest) and store their structures in the Filter Cache. This selection procedure ends when the space requirements are met and thus, for all terms that were not selected by the caching policy, we use OTF (regardless of their length). We obtained the frequency of terms in the query log using as our training set the $99k$ queries from the TREC 2006 Efficiency track query trace and then, applied our method to the remaining $1k$ queries.

In Figure 6, we compare the space/time tradeoff of both caching methods on WAND, BMM-NLB and BMW, when only the LB filtering mechanism is available for the standard index. As expected, given a specific space budget, the cost-based policy always achieves faster times for all the algorithms compared to the length-based approach. Our proposed method is very effective even in scenarios with very limited space constraints. In particular, for space budget of $0.5GBs$, BMW achieves $15.75ms$ using the length-based policy, whereas $10.22ms$ using the cost-based. For the remainder of the paper we will use the cost-based caching approach. Figure 7 shows that when more space is available, using the PB structure accelerates the query processing time. Moreover, we observe that the quantized index can further reduce the query response times, for example Q-BMW-LB-PB for $2GBs$ achieves $4.36ms$.

9. EXTENSIONS

In this section, we provide some additional results. More specifically, we look the performance of our methods for reodered indexes and the impact of increasing top-k on the performance of our techniques.

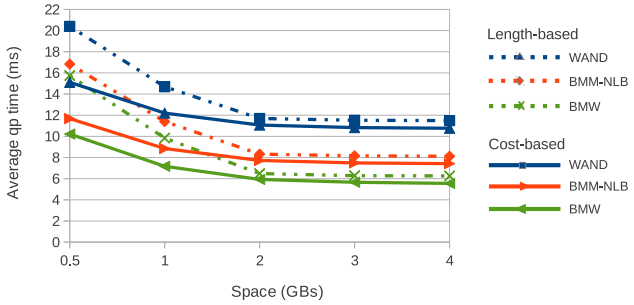

Figure 6: Space/Time tradeoff of Length-based and Cost-based caching on the standard index.

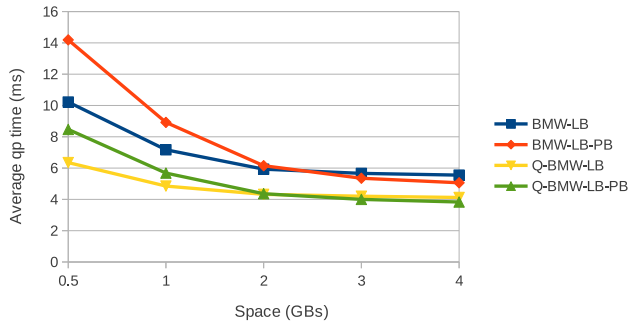

Figure 7: Space/Time tradeoff of Cost-based caching on the availability of LB and LB-PB on the standard and the quantized index.

DocID Reordering: There has been a huge amount of work to reduce the size of the compressed inverted indexes by reordering the docIDs. Under docID reordering, the documents are assigned docIDs using different techniques in order to minimize the index size. The main idea is to assign closeby docIDs to documents sharing lots of terms. A simple and effective method proposed by [20], which first sorts the URLs alphabetically and then assigns docIDs based on this ordering. More sophisticated reordering techniques were proposed in [14], whereas [28] showed that reordering speeds up the conjunctive query processing. Recent work in [15, 13] showed that docID reordering also accelerates the query response times of disjunctive queries. In Table 5 we show the performance of BMW using the cost-based caching policy under the simple reordering method of [20] when varying the number of query terms. For the remainder of the paper, the space requirement of the algorithms using the LB structure is $2GBs$, and $4GBs$ when the PB is also used. As expected reordering gives a speed-up for both filtering mechanisms. In particular, BMW-LB achieves $3.02ms$ average times, whereas BMW-LB-PB performs similarly with $3.04ms$. The reason the performance of BMW-LB-PB is not better is that the reordering changes the distribution of postings in docID space.

Algorithm	avg	2	3	4	≥ 5
BMW-LB	3.02	1.31	2.16	3.25	6.06
BMW-LB-PB	3.04	1.33	2.23	3.3	5.89

Table 5: Impact of docID reordering on BMW on the standard index.

Varying k in Top-k: In this experiment, we show the performance of our methods when k is increased. In Table 6 we see the performance of the filtering mechanisms on BMW as we increase k and vary the number of query terms for the standard and the quantized index. We observe that both mechanisms scale very well, but LB-PB performs better as k increases in both indexes. BMW-LB returns the top-100 results in $13.5ms$. In the quantized index scenario, the query times are further reduced since calculations operate in the integer domain. In particular, Q-BMW-LB-PB returns the top-10 results in $3.83ms$, while the top-1000 in $24.8ms$.

Algorithm	10	50	100	500	1000
BMW-LB	5.93	10.31	13.5	26.88	36.27
BMW-LB-PB	5.06	8.53	11.23	23.33	32.34
Q-BMW-LB	4.32	7.59	10.01	19.72	26.32
Q-BMW-LB-PB	3.83	6.65	8.89	18.09	24.8

Table 6: Impact of top-k on BMW performance on the standard and the quantized index.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a general filtering mechanism based on docID-oriented Block-Max indexes that significantly improves the performance of several widely used algorithms for safe early termination in disjunctive queries. We present a simple interface that can be easily used by several query processing algorithms. Moreover, we augment the mechanism with an auxiliary structure that maintains posting liveness information for each term and show how it can be exploited to achieve faster query times. We also propose caching techniques that address the space/time tradeoff of our mechanisms and overall, we achieve more than a factor of 2 speed-up over the fastest previous methods.

The future work will introduce additional query processing algorithms that exploit the advanced parallel instruction sets available in modern CPUs (AVX) and faster compression methods that use SIMD operations of current CPUs.

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".

11. REFERENCES

[1] D. Agarwal and M. Gurevich. Fast top-k retrieval for model based recommendation. In *Proc. of the Fifth Int. Conf. on Web Search and Data Mining*, pages 483–492, 2012.

[2] 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.

[3] V. N. Anh and A. Moffat. Impact transformation: effective and efficient web retrieval. In *Proc. of the 25th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 3–10, 2002.

[4] V. N. Anh and A. Moffat. Improved retrieval effectiveness through impact transformation. In *Proc. of the 13th Australasian Database Conference*, 2002.

[5] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proc. of the 28th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 226–233, 2005.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Flexible and efficient ir using array databases. *VLDB J.*, 17(1):151–168, 2008.

[12] 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.

[13] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proc. of the Sixth ACM International Conference on Web Search and Data Mining*, 2013.

[14] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *Proc. of the 19th Int. Conf. on World Wide Web*, 2010.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.