Faster Top-k Document Retrieval Using Block-Max Indexes

Shuai Ding
Polytechnic Institute of NYU
Brooklyn, New York, USA
sding@cis.poly.edu

Torsten Suel
Polytechnic Institute of NYU
Brooklyn, New York, USA
suel@poly.edu

ABSTRACT

Large search engines process thousands of queries per second over billions of documents, making query processing a major performance bottleneck. An important class of optimization techniques called *early termination* achieves faster query processing by avoiding the scoring of documents that are unlikely to be in the top results. We study new algorithms for early termination that outperform previous methods. In particular, we focus on safe techniques for disjunctive queries, which return the same result as an exhaustive evaluation over the disjunction of the query terms. The current state-of-the-art methods for this case, the WAND algorithm by Broder et al. [11] and the approach of Strohman and Croft [30], achieve great benefits but still leave a large performance gap between disjunctive and (even non-early terminated) conjunctive queries.

We propose a new set of algorithms by introducing a simple augmented inverted index structure called a *block-max index*. Essentially, this is a structure that stores the maximum impact score for each block of a compressed inverted list in uncompressed form, thus enabling us to skip large parts of the lists. We show how to integrate this structure into the WAND approach, leading to considerable performance gains. We then describe extensions to a layered index organization, and to indexes with reassigned document IDs, that achieve additional gains that narrow the gap between disjunctive and conjunctive top-k query processing.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance.

Keywords

IR query processing, top-k query processing, early termination, inverted index.

SIGIR'11, July 24–28, 2011, Beijing, China.

1. INTRODUCTION

Due to the rapid growth of the web, more and more people are relying on search engines to locate useful information. As a result, an increasing share of the world's computational resources is spent on search-related tasks. Current large-scale search engines have to be able to answer hundreds of millions of queries per day on tens of billions of web pages. Thus, highly optimized methods are needed to efficiently process all these queries.

One major bottleneck in query processing is the length of the inverted list index structures (described in the next section), which can easily grow to hundreds of MBs or even GBs for common terms (roughly linear in the size of the data set). Given that search engines need to answer user queries within fractions of a second, naively traversing this basic index structure, which could take hundreds of milliseconds or more for common terms, is not acceptable.

This basic problem has long been recognized by researchers, and has motivated a lot of work on optimization techniques including distributed computation [29, 5, 24], index compression [37], caching [7, 22], and early termination [31, 12] (also called pruning or optimized top-k processing). In this paper we focus on early termination, which in a nutshell means returning the best $k = 10$ or 100 results without an exhaustive traversal of the relevant index structures. In particular, we propose to augment the index by adding additional information. Essentially, we add to each compressed block in the inverted lists one value, the maximum impact score. While this is a simple idea, we are not aware of any previous work that stores such information for better query processing. We call this modified index structure a *Block-Max Index*. We also propose a new set of algorithms based on the WAND approach [11] for safe early termination (where exactly the same results as in the naive baseline are returned) using our block-max index structure. One interesting property of our algorithms is that they performs *document-at-a-time (DAAT)* index traversal, based on either document-sorted or impact-layered index structures.

2. BACKGROUND

In this section, we provide background on inverted index structures, query processing, and early termination.

2.1 Inverted Indexes and Index Compression

Current search engines perform query processing based on an inverted index, which is a simple and efficient data structure that allows us to find documents that contain a particular term [37]. Given a collection of N documents, we assume that each document is identified by a unique *document ID* (docID) between 0 and $N-1$. An inverted index consists of 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 (called *positions*), plus maybe other context such as font size etc. Postings in an inverted list are typically sorted by docID, or sometimes by some other measure (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 in this paper, but all our techniques also apply to cases where positions, context information, or precomputed quantized impact scores are stored.

The inverted lists of common query terms may consist of many millions or even billions of postings. To allow faster access to lists on disk, and limit the memory needed, search engines use sophisticated compression techniques that significantly reduce the size of each inverted list [37]. Compression is crucial for search engine performance [14, 36], and there are many compression techniques in the literature; see [36, 35, 27]. In this paper we use the New-PFD compression method, which was shown to perform well in [35], but our ideas also apply to other compression techniques.

As the lists for common terms could be very long, we want to be able to skip most parts of the lists during query processing. To do so, inverted lists are often split into blocks of, say, 64 or 128 docIDs, such that each block can be decompressed separately. To do so, we have an extra table, which stores for each block the uncompressed maximum (or minimum) docID and the block size in this table. The size of this extra table is small compared to the size of the inverted index. Thus, 64 or 128 postings are grouped together as a block where we store 64 or 128 compressed docIDs, followed by the corresponding compressed frequencies.

2.2 Query Processing

Given the inverted index structure mentioned above, the most basic form of query processing is called *Boolean query processing*. A query **(apple AND orange) OR pear** for all documents containing both words **apple** and **orange**, or the word **pear**, can be implemented by first intersecting the docIDs in the inverted lists for **apple** and **orange**, and then merging the result with the inverted list for **pear**.

Search engines use *ranked query processing*, where a ranking function is used to compute a score for each document passing a simple Boolean filter, and then the k top-scoring documents are finally returned. This ranking function should be efficiently computable from the information in the inverted lists (i.e., the frequencies and maybe positions) plus a limited amount of other statistics stored outside the inverted index (e.g., document lengths or global scores such as Pagerank). Many classes of functions similar to BM25 or Cosine have been studied; see [6] for more details.

Current web search engines use ranking functions based on hundreds of features. Such functions are quite complicated and fairly little has been published about how to efficiently execute them on large collections. One "folklore" approach separates ranking into two phases. In the first phase, a simple and fast ranking function such as BM25 is used to get, say, the top 100 or 1000 documents. Then in the second phase a more involved ranking function with hundreds of features is applied to the top documents returned from the first phase. As the second phase only examines a small number of top candidates, a significant amount of the computation time is still spent on the first phase. In this paper we focus on executing such a simple first-phase function, say BM25, a problem that has been extensively studied in the literature.

Recall that ranked query processing consists of a Boolean filter followed by scoring and ranking the documents that pass this filter. The most commonly used Boolean filters are conjunctive (AND) and disjunctive (OR). In general, disjunctive queries have traditionally been used in the IR community while web search engines have often tried to employ conjunctive queries as much as possible. One reason is that disjunctive queries tend to be significantly (by about an order of magnitude for exhaustive query processing) more expensive than conjunctive queries, as they have to evaluate many more documents.

To traverse the index structure, there are two basic techniques, Document-At-A-Time (*DAAT*) and Term-At-A-Time (*TAAT*) [32]: For conjunctive queries, DAAT is often preferred, while many optimized approaches for disjunctive queries use TAAT.

2.3 Early Termination Algorithms

As discussed before, one bottleneck in query processing is the length of the inverted lists. Early termination is one important technique that addresses this problem. We say that a query processing algorithm is *exhaustive* if it fully evaluates all documents that satisfy the Boolean filter condition. Any non-exhaustive algorithm is considered to use *early termination (ET)*. There are four ways in which early termination often happens:

- **Stop early**: In this case, the postings are usually arranged such that the most promising documents appear early. Then we stop the traversal of the index as soon as we (may) have the *top-k* results. Well-known examples are the TA, FA, and NRA algorithms of Fagin [21]; see [8] for a highly optimized implementation of some of these algorithms.
- **Skip within lists**: When the postings in each list are sorted by docIDs, the promising documents are spread out throughout the inverted lists, and thus the standard intuition for "*stop early*" does not apply. There are few published works on early termination techniques under this scenario. An exception is the WAND algorithm in [11], which uses a smart pointer movement technique to skip many documents that would be evaluated by an exhaustive algorithm. More details are provided further below.
- **Omit lists**: One or more lists for the query terms are completely ignored, if they do not affect the final results by much.
- **Score only partially**: We partially evaluate a document by computing only some term scores, or by computing approximate scores. When we find that the document cannot be in the top results, we stop evaluation; an example is [33].

Note that our definition of ET is very general and includes other techniques such as static pruning [10, 19] and tiering [16]. In this paper we focus on *safe early termination* [30], which means we want exactly the same results as in the naive baseline, i.e., the same set of documents in the same order with the same scores. We will ignore other techniques, which try to return search results that are somehow similar, or of similar quality. Also, we focus on memory-based indexes, as for example considered in [30, 17], or at least on the case where disk is not the main bottleneck.

2.4 Index Organizations

Many existing techniques for early termination from the DB and IR communities are based on the idea of reorganizing the inverted index such that the most promising documents appear early in the inverted lists. This can be done by either reordering the postings in each list, or partitioning the index into several layers or tiers. In particular, we can distinguish among the following widely used index organizations:

- **Document-Sorted Indexes**: This is the standard approach for basic exhaustive query processing, where the postings in each inverted list are sorted by document ID.

- **Impact-Sorted Indexes**: Postings in each list are sorted by their impact, that is, their contribution to the score of a document. Postings with the same impact are sorted by document ID. Note that this assumes that the ranking function is decomposable (i.e., a sum or other simple combination of per-term scores), which is true for Cosine, BM25, and many other functions in the literature.
- **Impact-Layered Indexes**: We partition the postings in each list into a number of layers, such that all postings in layer i have a higher impact than those in layer $i+1$, and then sort the postings in each layer by docID.

Impact-sorted and impact-layered indexes are very popular index organizations for early termination techniques, as they place the most promising postings close to the start of the lists [25, 21, 3, 4, 8, 30, 23, 32]. A problem with impact-sorted indexes is that compression could suffer as docID gaps in the inverted lists may be very large. In this case, an impact-layered index that uses a small number of appropriately chosen layers may provide a better alternative. However, impact-sorted indexes are useful when the number of distinct impact scores is small, or frequencies are used as proxies for impacts.

In contrast, document-sorted indexes tend to be less studied for early termination techniques, and only few algorithms use them (e.g., [11]).

2.5 Index Traversal Techniques

For index traversal, the two most commonly used techniques are:

- **Document-At-A-Time (DAAT)** : In DAAT query processing, each list has a pointer that points to a "current" posting in the list. All the pointers move forward in parallel as the query is being processed.
- **Term-At-A-Time (TAAT)**: In TAAT query processing, we first access one term, or one layer from one term, and then move to the next term, or the next layer from the same term or a different term. We use a temporary data structure to keep track of currently active top-k candidates.

Note that TAAT requires additional data structures to store promising candidates seen in some but not all of the lists; this is one of the main differences to DAAT. In this paper we use **TAAT** to refer to any techniques that use nontrivial data structures to keep track of promising candidates (beyond the simple heap structure used for the current top-k results), and thus this includes the original Term-At-A-Time technique [13] as well as Score-At-A-Time in [3].

For *conjunctive* and exhaustive query execution, DAAT is very fast and considered state of the art (at least for queries with a moderate number of queries terms), whereas TAAT-type methods are often bottlenecked by the nontrivial data structures. However, for *disjunctive* queries it is hard to integrate early termination and exploit layered indexes with DAAT. Thus, for this case most early termination algorithms in the literature are based on TAAT that use impact-sorted or layered indexes. In this paper we challenge this assumption and suggest DAAT algorithms may actually do better for early termination even in the case of disjunctive queries. We note that DAAT does have a significant advantage by not having any expensive temporary data structures.

2.6 Two State-of-the-Art Techniques

For disjunctive queries, the fastest existing safe early termination techniques appear to be the approach by Strohman and Croft (SC) in [30] (based on earlier work in [3, 2]), and the WAND approach by Broder et al [11]. Both approaches are safe in that they return exactly the same *top-k* results as the baseline in the same order and with the same score (actually, the original SC algorithm may not return the same scores, but is easily extended to do so).

The approach by Strohman and Croft (SC) uses impact-sorted indexes and assumes a ranking function proposed in [3], where all impacts have one of 8 distinct values; however, it can be extended to ranking functions such as BM25 using an impact-layered index organization. The approach then uses a variation of Term-At-A-Time (TAAT) query processing where we first access the higher layers of the lists and then move to the lower layers with smaller impact scores; when we are guaranteed to have a set containing the top-k results, we will switch from OR mode to AND mode, by only searching for the docIDs already stored in the structure among the remaining layers, to find the final correct results. Figure 1 shows one example of the index layout for SC, with equal-size layers. Note that the original SC applies a different ranking technique, so the size of each layer varies. We explore this and find that it does not make a big difference whether we use variable-size or equal-size layers; we will explain this further below. In SC, a sorted array is used to keep track of candidate documents that have been seen in some but not all term lists. After processing each layer, an extra phase is used to "filter out" candidates that can not make into the top-k. Thus, this approach requires temporary data structures (arrays) to keep track of promising candidates.

The WAND approach, on the other hand, uses a standard document-sorted index, and can thus employ a Document-At-A-Time (DAAT) approach that does not require additional temporary data structures (apart from the small top-k heap used by all methods). The downside is that the promising documents are spread out throughout the inverted lists, and thus the standard intuition for early termination does not apply. Instead, WAND uses an ingenious pointer movement strategy based on *pivoting* that allows it to skip many documents that would be evaluated by an exhaustive algorithm. More precisely, in DAAT query processing each list has a pointer that points to a "current" posting in the list, and that moves forward as the query is being processed. Thus any posting to the left of the pointers have already been processed. Throughout the algorithm, WAND keeps the terms sorted in increasing order by their current docIDs. Assume that at some point during a query "*dog, cat, kangaroo, monkey*", the current docIDs are 609, 273, 9007, and 4866, respectively, as shown in Figure 2, where the lists are arranged from top to bottom according to these docIDs. Suppose also that we know the maximum impact score for each list, and that a total document score of at least 6.8 (the threshold) is needed in order to make it into the current *top-k* results. We now sum up the maximum scores of the lists from top to bottom until we reach a score no smaller than 6.8. In Figure 2, this happens at the third list from the top ($2.3 + 1.8 + 3.3 > 6.8$). We can now claim that the smallest docID that can make it into the *top-k* is 4866. Thus, we can move the top two pointers forward to the first postings in their lists with docIDs at least 4866, enabling skipping in these lists. If docID 4866 appears in both the first two lists then we evaluate this docID. Otherwise, we sort the lists according to the current docIDs and *pivot* again. Thus, WAND achieves early termination by enabling skips over postings that cannot make into the top results. For the threshold value, we use the lowest score in the heap that contains the top-k results found thusfar. Note that WAND also stores one maximum impact score for each list, which could be kept in the term dictionary of the index.

These two methods are different in interesting ways – index organization and index traversal choices: WAND uses a document-sorted index and DAAT as index traversal technique, whereas SC

uses an impact-ordered or layered index and TAAT, using an additional data structure to keep the candidates. For disjunctive queries SC seems to outperform WAND, although the numbers for SC that we report are not as fast as those in [30], due to our use of BM25 as the ranking function and due to us keeping stopwords and removing 1-term queries. WAND is of interest to us as it will form the basis of our improved approaches.

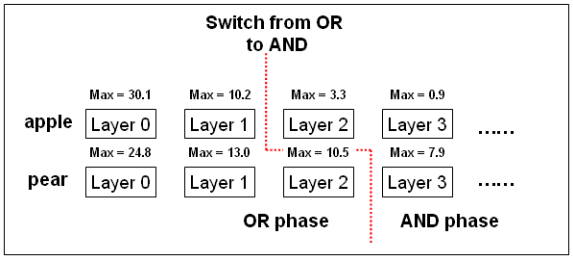

Figure 1: The index layout for SC. The layers are processed in descending order by their maximum impact scores. Inside each layer, postings are sorted by docIDs.

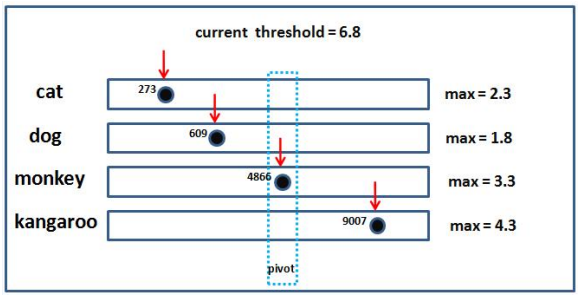

Figure 2: A scenario during the processing of a 4-term query, where the current pointers point to docIDs 273, 609, 4866, and 9007. WAND selects the third list as a pivot, and moves earlier pointers to docID 4866. Then all lists are sorted again according to their current docIDs.

3. OUR CONTRIBUTION

In this paper, we propose new early termination algorithms by building on the WAND approach [11]. Recall that WAND stores the maximum impact for each inverted list. Our initial insight is that skipping in WAND is limited because it uses the maximum impact scores over the entire lists, which can be much larger than average. Recall that we have an extra table to store information allowing us to skip blocks. We propose to augment the index structure by also storing in this table the maximum impact value for each block. We call such an index a *Block-Max Index*. In this way, we get a piece-wise upper-bound approximation of the impact scores in the lists, as shown in Figure 3. This approximation hides the detailed scores, shown in Figure 3 for one block in the kangaroo list, which can only be obtained by decompressing the block. This idea is very simple and easy to implement. As we will see later, this gives many optimization opportunities and leads to large performance gains.

After this slight change to the index structure, resulting in only a small increase in index size, we have to adapt the WAND algorithm to work with it. One obvious idea is to just use the local maximum value for the current block, instead of the global one, in the pivoting phase. Unfortunately, this does not guarantee correctness. To see this, let us look at the example in Figure 4. Looking only at the max scores for the blocks containing the current pointers (i.e., score

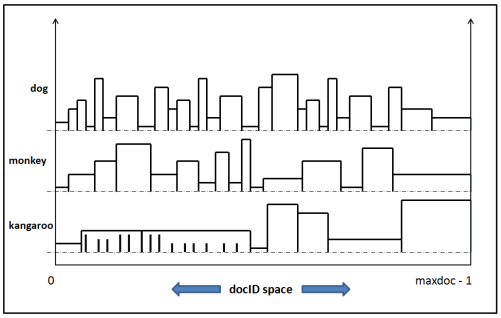

Figure 3: Three inverted lists where lists are piecewise upper-bounded by the maximum scores in each block. As shown for one block in the bottom list, inside each block we have various values, including many (implied) zero values, that can be retrieved by decompressing the block.

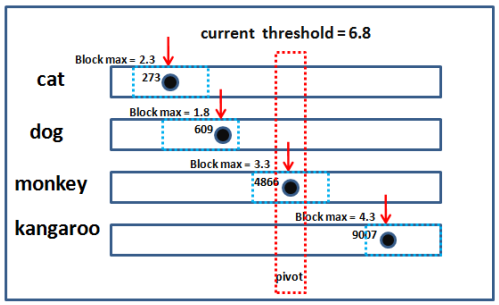

Figure 4: An example showing why directly using block max scores does not work.

2.3 for the first list, and so on), we cannot conclude that 4866 is the smallest docID that can make into it the *top-k*, because it is possible that the next block after docID 273 in the first list (but with docIDs smaller than 4866) has a much higher maximum impact score. Thus, directly applying the local max value does not work. We will describe how to modify the algorithm in later sections.

Overall, we make the following main contributions in this paper:

1. We propose a modified index structure, the Block-Max Index, which only slightly increases the index size. We then study improved techniques for safe early termination based on this index structure and the WAND approach in [11].
2. We show how to extend our techniques to layered indexes, reordered indexes, and conjunctive query processing.
3. We evaluate our techniques on the TREC GOV2 collection of 25.2 million documents, and demonstrate considerable improvements compared to the state-of-the-art techniques.
4. We discuss some interesting open questions resulting from our work.

4. RELATED WORK

Previous work on early termination (ET) techniques can be divided into two fairly disjoint sets of literature. In the IR community, researchers have studied ET techniques for the fast evaluation of vector space queries since the 1980s; some early work appears in [13, 32, 33]. There have been a large number of papers in recent years. Most relevant to us, several recent papers have focused on how to use impact-sorted indexes [2, 3, 30] for early termination, resulting in highly efficient methods for disjunctive queries.

There has also been a lot of work on early termination in the database community; see [21] for a survey and [20] for a formal analysis. Stated in IR terms, the algorithms also assume that postings in the inverted lists are sorted by their contributions and accessed in sorted order. However, the application scenarios are somewhat different, and many (but not all) of the algorithms assume that once a document is found in one inverted list, we can efficiently evaluate it by performing lookups into the other inverted lists. Such random lookups are highly undesirable in most IR scenarios.

Early termination techniques also differ in terms of their assumptions about result quality. We can distinguish between safe (or reliable) early termination techniques that return exactly the same top-k results as the baseline [30, 20], techniques that return mostly the same results, and those that just return results of equivalent quality, as determined by suitable IR measures. We focus on safe early termination of disjunctive queries, where the most relevant previous techniques are the approach of Strohman and Croft [30] and the WAND approach of Broder et al. in [11].

Two other recent ideas in IR query processing are also relevant to our work. First, a number of recent papers show how to decrease inverted index size and query processing costs by optimizing the assignment of docIDs to the documents in the collection [26, 9]. Intuitively, if we assign consecutive docIDs to very similar pages, for example by sorting pages by URL [28] or clustering by textual similarity, we obtain runs of small docIDs gaps that allow better index compression with suitable techniques. Moreover, as shown in [35], reordering significantly increases the speed of conjunctive queries. Our work here shows that, somewhat surprising to us, reordering can help even more for disjunctive queries.

The second relevant idea is that of two-level indexes proposed in [1]. The idea is to cluster documents by similarity and then in the first level only index the clusters (i.e., whether a cluster contains a term), while the second level says which documents in the cluster actually contain the term. Thus the first level basically approximates the overall index, similar to the way in which we use the maximum impact scores in each block to approximate the distribution of impact scores in a list.

Finally, we note that very recently, and independent of our work, Kaushik et al [15] have proposed an index structure very similar to the Block-Max Index in this paper, which also stores maximum impact information for blocks. Their algorithm for disjunctive queries first performs preprocessing to split blocks into intervals with aligned boundaries and to discard intervals that cannot contain any top results. Then a version of the maxScore technique [32] is applied to the remaining intervals. While their and our algorithms are different, they both achieve significant performance improvements based on similar underlying ideas.

5. BLOCK-MAX WAND ALGORITHM

In this section we give our basic algorithm, *Block-Max WAND (BMW)*, which is an extension of WAND to our Block-Max Index.

5.1 The Basic Idea

As described in Section 2, naively using the maximum impact score for each block in the "pivoting" phase will not work, and thus we need to add some additional ideas. In the traditional DAAT query processing, one core function is called *Next(d,list(i))* or *NextGEQ(d, list(i))* [11]; this function receives a docID d and an inverted list *list(i)* as inputs and returns the first docID after the current docID in *list(i)* that is equal to or greater than *d*. The call to this particular function usually involves a decompression of one block in *list(i)*. We call this a **deep pointer movement** due to the reason that it usually involves a block decompression. As we have the max score for each block, we design another function called *NextShallow(d,list(i))* which only moves the current pointer to the corresponding block without decompression (using d and information about the block boundaries in the table). We call this a **shallow pointer movement**. We use two main ideas in our modified algorithm: (i) we use the global maximum scores to determine a candidate pivot, as in WAND, but then use the block maximum scores to check if the candidate pivot is a real pivot, and (ii) we use shallow instead of deep pointer movements whenever possible.

5.2 The Algorithm

The detailed algorithm is shown in Algorithm 1, and we refer to it as **Block-Max WAND (BMW)**. Note that in BMW we still also keep the maximum score for the whole list as in WAND.

```
Initialize();
repeat
    /* sort the lists by current docIDs                    */
    Sort(lists);
    /* same "pivoting" as in WAND using the max
    impact for the whole lists, use p to denote
    the pivot                                               */
    p = Pivoting(lists, θ);
    d = lists[p] → curDoc;
    if (d == MAXDOC) then
        break;
    end
    for i = 0 ... p + 1 do
        NextShallow(d, list(i));
    end
    flag = CheckBlockMax(θ, p);
    if (flag == true) then
        if ( lists[0] → curDoc == d ) then
            EvaluatePartial(d , p);
            Move all pointers from lists[0] to lists[p] by calling
            Next(list, d + 1)
        end
        else
            Choose one list from the lists before lists[p] with the
            largest IDF, move it by calling Next(list, d + 1)
        end
    end
    else
        d' = GetNewCandidate();
        Choose one list from the lists before and including lists[p]
        with the largest IDF, move it by calling Next(list, d')
    end
until Stop;
```

Algorithm 1: Block-Max WAND for disjunctive query processing based on local max values.

As shown in Algorithm 1, the main difference compared with WAND is that before we evaluate one docID, we will first move shallow pointers to check if we indeed have to evaluate this docID or not, based on the maximum scores for blocks. By doing this we filter out most of the candidates and achieve much faster query processing. Also, when the check fails, we can skip further forward using *GetNewCandidate()*, as described later in detail.

The two functions used in Algorithm 1, *NextShallow()* and *CheckBlockMax()*, are listed in Algorithm 2 and Algorithm 3. They are fairly obvious from the context. In *EvaluatePartial()*, we evaluate the document by summing up the scores from *list[0]* until *list[pivot+1]*. As soon as we find that the document can not make it into the top results, we stop the evaluation.

Another important improvement happens when *CheckBlockMax()* returns false, which means the current document d can not make it into the top results. Instead of picking one list (usually the one with the largest IDF) and moving it forward to at least $d + 1$, we

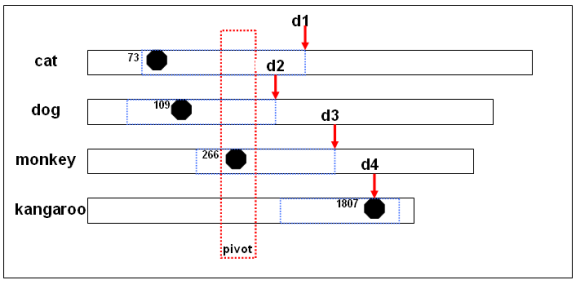

Figure 5: An example showing how *GetNewCandidate()* works. Assume 266 is the pivot and it fails to make it into the top results. In this case, we enable better skipping by choosing min$(d1, d2, d3, d4)$ as the next possible candidate, instead of $266 + 1$

will use the d' returned by *GetNewCandidate()*. The reason is that since the current document was ruled out based on the block maxima, we should skip at least beyond the end of one of the current blocks. This idea behind *GetNewCandidate()* is shown in Figure 5. Assume docID 266 is the pivot; when it fails the *CheckBlockMax()* check, instead of moving one of the first three lists to $266 + 1$, we will move it to $d' = min(d1, d2, d3, d4)$ where $d1, d2, d3$ are the block boundaries plus one of the first three lists, and $d4$ is the current docID in the forth list (equal to 1807 in this case). By doing this, skipping is greatly improved compared to using $d + 1$, while still guaranteeing a *safe* result. The proof should be obvious.

```
while did > list− > blockboundary[current_block] do
  | current_block + +;
end
```

Algorithm 2: NextShallow(list, did)

```
maxposs = 0.0f;
for i = 0 . . . pivot + 1 do
  | maxposs+ = list[i]− > blockmax[current_block];
end
if ( maxposs > threshold ) then
  | return true;
end
else
  | return false;
end
```

Algorithm 3: CheckBlockMax(threshold, pivot)

6. EXPERIMENTS

In this section, we provide a first set of experimental results.

6.1 Experimental Setup

We evaluate our methods on the TREC GOV2 collection. The GOV2 collection consists of 25.2 million web pages crawled from the gov Internet domain. The uncompressed size of these web pages is 426GB. We compress the inverted index using the New-PFD version of PForDelta as described in [35], with 64 docIDs and frequencies in each block.(We also tried other block sizes, but this one gave the best results.) The compressed index consumes 8759MB and the extra information for the Max-Block Index (the maximum score for each block) adds about 400MB (using 32 bits for each score though this could be reduced).

We randomly picked 1000 queries from the TREC 2006 Efficiency queries, and 1000 queries from the TREC 2005 Efficiency

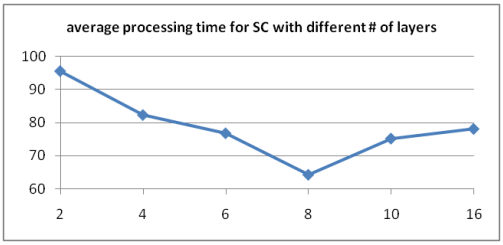

Figure 6: Processing time in ms for different number of layers in SC.

queries as our testing sets. The average numbers of postings per query are 4.67M and 6.07M using 2006 and 2005 set, respectively. We use BM25 as our ranking function. In all our runs, we load the inverted index completely into main memory. Unless stated otherwise, we return top-10 results. Runs are performed on a single core of a 2.27GHz Intel(R) Xeon CPU. All the codes are available by contacting the first author.

6.2 Results

In this section we compare our algorithm BMW with exhaustive OR (using DAAT), WAND, and SC on disjunctive queries. We measure the performance by three criteria – time (average time per query in ms), decoded integers per query and evaluated docIDs (docIDs that are completely scored against all query terms) per query. These criteria are also used in previous work [30, 11].

We reimplemented the SC algorithm based on the code available at http://repo.or.cz/w/galago.git, with BM25 as the ranking function. The original SC has four phases–*OR, AND, Refine, and Ignore*. In this paper we focus on getting safe results; thus we do not have an *Ignore* phase in our implementation (as we need the exact scores). Most of the time is spend in the *OR* phase, so this does not change the query processing time by much.

We partitioned each list into 8 equal-sized layers. Note that in the original SC algorithm in [30], a different ranking function was used that has only 8 distinct impact scores, with each layer dealing with one score, and thus the layers were of different sizes. We also tried different combinations of variable-sized layers We found that equal-sized layers usually did at least as well as the other heuristics, though in principle there is still room for better approaches. Our explanation is that in SC, there is a "switching point", where we are guaranteed to have encountered the docIDs of all the corrects top-k results in at least one list. At this point, we can switch from OR mode (TAAT) to AND mode before starting the next layer. Most of the computation time in SC is spend in OR mode, and things become much faster afterwards. While this "switching point" depends on how we partition the lists into layers, the switching point cannot happen before the point where the *threshold condition* is satisfied in the well-known TA algorithm of Fagin. The optimal partitioning would make a cut right after the threshold condition is satisfied for a query, but since this depends on the query one good choice is to spread out cuts evenly over the lists so that the next cut is never far away.

We also experimented with different numbers of layers. Figure 6 shows SC performance for different numbers of layers for equal-size layers, justifying the use of 8 layers.

Table 1 shows the query processing time using different algorithms. All runs in this paper exclude single-term queries, and no stopwords were removed (changing these assumptions would result in even better times). We observe that for TREC 2006, the effect of removing single-term queries is small, while for TREC 2005 the difference is significant as there are many such queries in the

TREC 2006						
	avg	2	3	4	5	> 5
exhaustive OR	225.7	60	159.2	261.4	376	646.4
WAND	77.6	23.0	42.5	89.9	141.2	251.6
SC	64.3	12.2	36.7	75.6	117.2	226.3
BMW	27.9	4.07	11.52	33.6	54.5	114.2
exhaustive AND	11.4	10.3	10.8	14.0	15.4	15.2
TREC 2005						
	avg	2	3	4	5	> 5
exhaustive OR	369.3	62.1	238.9	515.2	778.3	1501.4
WAND	64.4	23.5	43.7	73.4	98.9	265.9
SC	63.5	14.2	37.5	119.7	172.9	316.9
BMW	21.2	3.5	12.7	25.2	39	104
exhaustive AND	6.86	6.4	7.3	9.2	4.7	5.9

Table 1: Average query processing time in ms for different numbers of query terms, using different algorithms on the TREC 2006 and 2005 query logs. Exhaustive OR, WAND, SC, and BMW are for disjunctive queries, while Exhaustive AND is for conjunctive queries.

log. (But note that most single-term queries are resolved by result caching in real search engines.)

From Table 1 we can see that our *BMW* algorithm improves query processing performance. This is mainly due to the superiority of the DAAT index traversal over TAAT and the large amount of skipping. Also, our implementation for SC is not as fast as the numbers reported in [30], especially on *TREC 2005 query log*. This is because we remove single-term queries but do not remove stopwords as in [30], and also due to the use of BM25 as our ranking function. Still, SC outperforms basic WAND, as also reported in previous work. Overall, our basic *BMW* algorithm achieves much faster query processing but is still much slower than *exhaustive AND* using standard DAAT.

Table 2 shows the other two criteria for different methods on the *TREC 2006 query log* (we omit the 2005 data due to space constraints). We also include the number of deep pointer movements and shallow pointer movements. As these measures are only meaningful for DAAT index traversal, we ignore the numbers for SC which uses TAAT traversal. Note that we did not include numbers for SC for evaluated docIDs, mainly because SC adopts TAAT-like query processing and the definition of evaluated docIDs will be misleading. Also, in BMW each partial evaluation is counted as an evaluated docID, no matter whether it stops early or not.

From Table 2, we see that all techniques improve greatly over exhaustive OR. WAND only evaluates 4.6% of the docIDs compared to exhaustive OR, which approximately matches the numbers in [11]. BMW evaluates even fewer docIDs. This means that BMW should perform even better when we have a more expensive scoring function than BM25, such as the one mentioned in [11]. Another interesting point is that SC decodes less integers compared with the fastest method, BMW, which means that assigning promising docIDs to the first layers does help a lot. However, SC uses an additional data structure to temporarily store candidates, and this is the main drawback for SC and many other TAAT-based techniques.

6.3 Document ID Reassignment

In this section we show results after document ID reassignment. The idea for document ID reassignment is to assign docIDs to documents so that similar pages have close IDs. This idea is extensively explored in [28, 35, 18] and it is shown that after the reassignment, both compressed index size and query processing speed under exhaustive AND are significantly improved.

One natural question is whether reassignment can also help processing speeds for disjunctive query processing. For exhaustive OR, the intuition is that the improvement should be tiny because we fully evaluate the documents in all the lists anyway, no matter how the docIDs are assigned. For SC, the improvement should also be modest because in SC we will assign the postings such that the postings with higher impact scores appear earlier in the list. However, for WAND and BMW, reassignment of docIDs might give some benefits, as the distribution of impact values within each block should become more even, helping both WAND and BMW.

	evaluated docs	decoded ints	dpm	spm
exhaustive OR	3815676	9356032	15.9M	–
WAND	178391	6274432	1.18M	–
SC	–	965248	–	–
BMW	21921	2642752	0.42M	0.76M
exhaustive AND	20026	1939584	0.25M	–

Table 2: The average number of evaluated docIDs, decoded integers, deep pointer movements (dpm), and shallow pointer movements (spm) for different methods on the TREC 2006 query log. Exhaustive OR, WAND, SC, and BMW are for disjunctive queries, while Exhaustive AND is for conjunctive queries.

TREC 2006						
	avg	2	3	4	5	> 5
exhaustive OR	210.6	55.3	156.6	245.9	354.2	583.9
WAND	50.1	17.2	28.7	57.5	94.9	168.9
SC	69.3	14.1	40	80.9	126.7	239.9
BMW	8.89	1.4	3.6	10.2	16.9	37.8
exhaustive AND	6.56	5.5	5.3	7.1	10.8	8.4
TREC 2005						
	avg	2	3	4	5	> 5
exhaustive OR	349.7	56.4	226	495.8	743.1	1411.9
WAND	42.4	18.1	29.4	47.4	64.5	163.3
SC	76.4	12.6	52.9	112.2	162.8	288.9
BMW	7.2	1.3	3.7	8.9	13.5	35.8
exhaustive AND	4.5	4.3	4.7	5.9	2.7	4.1

Table 3: Average query processing times in ms for different numbers of query terms after docID reassignment, on the TREC 2006 and 2005 query logs.

Table 3 shows the query processing times for the different techniques after docID reassignment. In particular, we assign docIDs according to the alphabetical ordering used in [28, 35]. From the table, we see that query processing performance is greatly improved for WAND and especially for BMW. In fact, the gaps between disjunctive and conjunctive queries are significantly narrowed. This means that reassignment succeeds in making the scores inside the blocks block much smoother, thus improving skipping.

Query processing performance is also slightly improved for exhaustive OR. This is mainly because reassignment reduces the compressed size of the inverted index, thus reducing the cost of main memory accesses; see also [14] for more discussion of this issue.

Table 4 shows the corresponding results for evaluated docIDs and decoded integers on the *TREC 2006 query log*. We observe similar trends as in the query processing time, with significant reductions for WAND and BMW.

7. EXTENSIONS

In this section we give some extensions to our BMW algorithm.

7.1 A Layered Version of BMW

As shown in previous section, BMW achieved the best query processing performance for disjunctive queries, and significantly

	evaluated docs	decoded ints	dpm	spm
exhaustive OR	3815676	9356032	15.9M	–
WAND	221926	3472704	0.74M	–
SC	–	715776	–	–
BMW	9308	1181760	0.126M	0.22M
exhaustive AND	20026	951744	0.10M	–

Table 4: The average number of evaluated docIDs, average decoded integers, deep pointer movements (dpm), and shallow pointer movements (spm) for different methods, after docID reassignment on the TREC 2006 query log.

narrowed the performance gap between disjunctive and conjunctive queries. The main advantage for BMW seems to be that it uses DAAT index traversal, and thus does not have to use an expensive data structure to keep track of promising candidate documents.

On the other hand, SC achieves pretty good early termination performance (note from the previous section that SC actually decodes fewer integers than the other algorithms), by using an impact-layered index and assigning the most promising documents to the first layers. This means that the intuition for SC, putting top-scoring documents early in the lists to stop early, does have a lot of merit. A natural question is if we can combine this idea of a layered index with our BMW algorithm and its DAAT traversal mechanism. We now show how to do this. Our basic algorithm is very simple: For each inverted list, we split it into N layers. Then we *treat each layer just as a separate term*. In this case we directly apply the BMW algorithm on the impact-layered index.

The intuition behind this idea is that after we pick out the top-scoring documents from each list, the scores for the remaining docIDs are much smoother. So when we store the maximum impact score for each block, it is less likely that this score will be much larger than the others in the block. It's not difficult to understand that such spiky values are bad for BMW: If two two spikes are in two separate blocks, we probably have to decode both blocks, but if they are in one block we may only need to decode and access that one block. have to decode the two blocks. We call the new algorithm *N-layer BMW* where N is the number of layers. The disadvantage of doing this is that we will have a larger number of terms for each query. To minimize this disadvantage, we only split each list into 2 layers, a *fancy* layer and a *normal* layer, and each list is split only when it has more than α postings. We put the top-scoring β postings in each list into the fancy layer. After some experiments, we set $\alpha = 50K$ and $\beta = 2\%$, which seems to work well (getting the best possible parameters is left for future work).

Thus, in our *2-layer BMW*, we just treat the layers from one list as separate lists, and the different layers from the same list do not know the existence of each other. We also design a version where the layers from the same list know the existence of the others. The observation is that one document ID can only exist in one, if any, of the layers from the same list. In this case, in the "*pivoting*" phase of the BMW algorithm, we can do better by choosing the maximum score of the layers from the same list (instead of the sum of the scores). We experimented with this idea and found that it moderately decreases pointer movements and decoded integers, but not the actual running time, due to the overhead of tracking layers belonging to the same term during pivoting. For space reason we omit the results of these experiments, though future work may lead to more practical variants based on this idea.

Table 5 shows the query processing performance for 2-*layer BMW* without and with docID reassignment. As we can see, 2-*layer BMW* obtains improved running times over basic BMW. We also

before reassignment	time (ms)	evaluated docs	decoded ints
BMW	27.9	21921	2642752
2-layer BMW	22.9	7435	1731264
after reassignment	time (ms)	evaluated docs	decoded ints
BMW	8.89	9308	1181760
2-layer BMW	7.4	4196	790464

Table 5: Query processing performance after combining layered index and BMW, before and after docID reassignment, on the TREC 2006 query log. All numbers are averaged per query.

before reassignment	avg	2	3	4	5	> 5
BMW	27.9	4.07	11.52	33.6	54.5	114.2
2-layer BMW	22.9	2.9	10	30.8	46.3	98.2
after reassignment	avg	2	3	4	5	> 5
BMW	8.89	1.4	3.6	10.2	16.9	37.8
2-layer BMW	7.4	1.1	3.3	8.5	15.0	31.4

Table 6: Average query processing time in ms for different number of query terms for BMW and 2-layer BMW, before and after docID reassignment, on the TREC 2006 query log.

show the query processing time for queries with different numbers of terms in Table 6.

7.2 Increasing Top-k

As mentioned, one commonly used ranking technique in current web search engines is based on a two-phase approach, where we first get the, say, top-1000 documents according to a simple ranking function such as BM25, and then compute the exact score according to a more complicated ranking function only for these 1000. An example appears in [34] for a ranking function that uses position information in addition to BM25. Of course, in these scenarios we usually need much larger values of k than $k = 10$, e.g., a few hundred or few thousand results.

In Figure 7 we show the performance as we increase k, with reassigned docIDs. We find that the performance for the naive *exhaustive OR algorithm* is quite stable as it decodes and evaluates all the postings anyway. For other algorithms, the query processing time increases. For SC, we observe a huge performance degradation. This is mainly because the performance of the temporal data structure degrades more and more as we store more candidates. We also see that *BMW* and *2-layer BMW* perform quite well even when k is equal to 1000, and that the increase in time is fairly moderate.

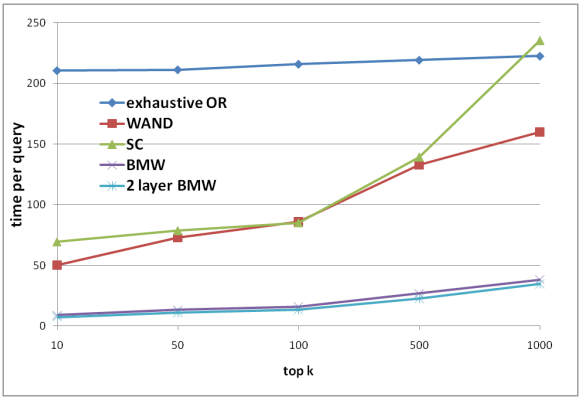

Figure 7: Query processing times for different techniques with docID reassignment, on the TREC 2006 query log. The X-axis is the value for k, while the Y-axis is the average processing time.

7.3 Block-Max AND

One advantage of BMW is that the idea can also be applied to *conjunctive* query processing using standard DAAT index traversal. For conjunctive query processing, *standard AND* starts from the shortest list, and then tries to find the corresponding docID in the longer lists. It is a natural extension to integrate the Block-Max Index structure and add shallow pointers in DAAT for conjunctive query processing. The improved DAAT algorithm is shown in Algorithm 4; it is called *Block-Max AND* (BMA).

```
Sort the lists from shortest to longest;
d = 0;
repeat
    d = NextGEQ(list[0], d);
    for i = 1 . . . n do
        NextShallow(d, list(i));
    end
    flag = CheckBlockMax(θ, n);
    if flag == true then
        search d in the rest lists, if found evaluate, otherwise
        d = d + 1
    end
    else
        d = GetNewCandidate();
        continue;
    end
until Stop;
```

Algorithm 4: Block-Max AND for conjunctive queries with n terms.

As we see we only have to slightly adapt the standard AND algorithm to get the BMA algorithm. In particular, we use three routines from BMW – *NextShallow(), CheckBlockMax()* and *GetNewCandidate()*. The processing cost for BMA is shown in Table 7. We observe that the BMA works better for queries with smaller numbers of terms (otherwise the shallow pointer movements will become expensive). So we also propose one *hybrid* algorithm: Apply BMA when the number of terms in the query is less than T; otherwise use exhaustive AND. We use $T = 4$ in this paper.

Table 8 shows the performance using BMA and the *hybrid* BMA, before and after docID reassignment. We can see significant improvements over an exhaustive AND. Note that *BMW* and *BMA* algorithms use the same index structure; thus the Block-Max Index can support both types of queries.

8. OPEN QUESTIONS

Our results in this paper raise several interesting open questions.

A Cleaner Algorithm and Analysis: While the described methods already achieve large benefits, we are not yet convinced that we have really found *the optimal algorithm*. It would also be interesting to provide some analysis, say of the optimal number of pointer movements and document evaluations under our approach.

Other Applications of Block-Max Indexes: The basic idea behind our augmented index structure could also be applied to other scenarios. For example, it would be natural to try to integrate local maximum scores into the two-level index structure in [1].

before reassignment	avg	2	3	4	5	> 5
exhaustive AND	11.4	10.8	10	12.5	11.57	9.94
BMA	9.89	3.96	7.92	13.2	14.08	14.63
after reassignment	avg	2	3	4	5	> 5
exhaustive AND	6.56	6.93	6.11	7.06	6.84	5.57
BMA	5.12	1.69	4.02	7.03	7.33	9.06

Table 7: Average query processing times in ms for different numbers of query terms, using exhaustive AND and BMA, before and after docID reassignment, on TREC 2006.

before reassignment	time	evaluated docs	decoded ints
exhaustive AND	11.4	20026	1939584
BMA	9.89	5725	1460992
Hybrid	9.4	6594	1568704
after reassignment	time	evaluated docs	decoded ints
exhaustive AND	6.56	20026	951744
BMA	5.12	3108	607680
Hybrid	4.53	3673	641344

Table 8: Average query processing times in ms, numbers of evaluated docIDs per query and average decoded integers per query for conjunctive queries, before and after docID reassignment, on TREC 2006.

	no reassignment	with reassignment
BMW	27.9	8.89
Clairvoyant BMW	23.0	7.2

Table 9: Average query processing times in ms for BMW versus Clairvoyant BMW, for top-10 results, on the Trec 2006 query log.

Estimating Top-k Thresholds: Our methods could be further improved if we somehow had a good a-priori estimate of what score is needed to make it into the top k results. Currently, we start with a threshold of zero, and then update the value as results are discovered. Thus, the algorithm starts slow and then speeds up. This motivates the following algorithmic problem that also has other applications, for example in distributed IR: Given an inverted index, a query, and a number k, how do we quickly estimate the score of the k-th best result (possibly using some small auxiliary structures), or conversely, given a threshold t how do we estimate the number of results with score higher than t.

For motivation, we show in Table 9 that a clairvoyant algorithm that knows the score of the k-th best result would get a circa 20% reduction in query processing costs.

Query Processing with Score Approximations: Another interesting more general question is how to best approximate the impact scores in inverted lists, and how to best use such approximations during query processing. Consider the scenario in Figure 3, where we have a long, sparse, array of impact values that is upper-bounded by some block-wise approximation. What is the best approximation for a given array of values? Does it make sense to have a multi-level structure (similar to wavelet trees) that provides upper bounds for progressively smaller block sizes as we descend to lower levels? Are there statistical measures other than the maximum impact, say the skew of the values, that are useful?

9. CONCLUSION

In this paper, we have described and evaluated improved safe early termination for disjunctive queries. This was achieved by an augmented index structure called a Block-Max Index, which stores maximum impacts for blocks of postings. We then showed how to integrate this structure into the WAND approach. Finally, we extended it to the impact-layered indexes, indexes with reassigned docIDs, and conjunctive queries, leading to additional improvements. Our results also lead to a number of interesting opportunities for future research, as discussed in Section 8.

Acknowledgments

This research was supported by NSF Grant IIS-0803605, "Efficient and Effective Search Services over Archival Webs", and by a grant from Google.

10. REFERENCES

[1] Ismail Sengor Altingovde, Engin Demir, Fazli Can, and Ozgur Ulusoy. Incremental cluster-based retrieval using compressed cluster-skipping inverted files. *ACM Transactions on Information Systems*, 26(3):1–36, 2008.

[2] V. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 226–233, 2005.

[3] V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 372–379, 2006.

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

[5] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proceedings of the 9th String Processing and Information Retrieval Symposium*, 2002.

[6] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addision Wesley, 1999.

[7] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

[8] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard 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] Roi Blanco and Álvaro Barreiro. TSP and cluster-based solutions to the reassignment of document identifiers. *Information Retrieval*, 9(4):499–517, 2006.

[10] Roi Blanco and Alvaro Barreiro. Probabilistic static pruning of inverted files. *ACM Transactions on Information Systems*, 28(1), January 2010.

[11] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 12th ACM Conference on Information and Knowledge Management*, 2003.

[12] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of the 18th Annual International Conference on Data Engineering*, 2002.

[13] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1985.

[14] Stefan Buttcher and Charles L. A. Clarke. Index compression is good, especially for random access. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, 2007.

[15] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. Interval-based pruning for top-k processing over compressed lists. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, 2011.

[16] J. Cho and A. Ntoulas. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

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

[18] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[19] 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 International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2001.

[20] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the ACM Symp. on Principles of Database Systems*, 2001.

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

[22] R. Lempel and S. Moran. Optimizing result prefetching in web search engines with segmented indices. In *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.

[23] 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.

[24] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proceedings of the 10th International Conference on World Wide Web*, 2000.

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

[26] W. Shieh, T. Chen, J. Shann, and C. Chung. Inverted file compression through document identifier reassignment. *Information Processing and Management*, 39(1):117–131, 2003.

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

[28] Fabrizio Silvestri. Sorting out the document identifier assignment problem. In *Proceedings of 29th European Conference on IR Research*, pages 101–112, 2007.

[29] Fabrizio Silvestri and Domenico LaForenza. Query-driven document partitioning and collection selection. In *Proceedings of the First International Conference on Scalable Information Systems*, 2006.

[30] T. Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

[31] Trevor Strohman, Howard Turtle, and Bruce W. Croft. Optimization strategies for complex queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2005.

[32] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, November 1995.

[33] HW. Wong and D. Lee. Implementations of partial document ranking using inverted files. *Information Processing and Management*, 29(5):647–669, 1993.

[34] Hao Yan, Shuai Ding, and Torsten Suel. Compressing term positions in web indexes. In *Proceedings of the 32th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2009.

[35] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*, 2009.

[36] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web*, 2008.

[37] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.