Fast Document-at-a-time Query Processing using Two-tier Indexes

Cristian Rossi
Univ. Federal do Amazonas
Manaus,AM, Brazil
cristian.infor@gmail.com

Edleno Silva de Moura
Univ. Federal do Amazonas
Manaus,AM, Brazil
edleno@icomp.ufam.edu.br

Andre Luiz Carvalho
Univ. Federal do Amazonas
Manaus,AM, Brazil
andre@icomp.ufam.edu.br

Altigran Soares da Silva
Univ. Federal do Amazonas
Manaus,AM, Brazil
alti@icomp.ufam.edu.br

ABSTRACT

In this paper we present two new algorithms designed to reduce the overall time required to process top-k queries. These algorithms are based on the document-at-a-time approach and modify the best baseline we found in the literature, Blockmax WAND (BMW), to take advantage of a two-tiered index, in which the first tier is a small index containing only the higher impact entries of each inverted list. This small index is used to pre-process the query before accessing a larger index in the second tier, resulting in considerable speeding up the whole process. The first algorithm we propose, named *BMW-CS*, achieves higher performance, but may result in small changes in the top results provided in the final ranking. The second algorithm, named *BMW-t*, preserves the top results and, while slower than BMW-CS, it is faster than BMW. In our experiments, BMW-CS was more than 40 times faster than BMW when computing top 10 results, and, while it does not guarantee preserving the top results, it preserved all ranking results evaluated at this level.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Top-k Query Processing, Efficiency, Two-tier indexes

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

1. INTRODUCTION

Computing ranking of results by using information retrieval (IR) models is one of the core tasks of search systems. While search systems often index a massive number of documents, usually their users are not interested in an in-depth list of results related to a query, but rather to a small list of highly relevant documents that will satisfy their informational needs. Thus, part of the recent research related to search systems is aimed at improving the quality of the top results presented to users, instead of the overall quality of the presented list. This focus on a narrower set of high quality results has led to the development of a number of technologies to improve the efficiency of methods to compute the top results in search systems.

When determining the best results for a given query, a search system usually deploys a number of different sources of relevance evidence. For instance, web search engines use information such as titles of the pages, URL tokenization, and link analysis, among others. These sources of relevance evidence may be combined using a myriad of approaches, such as the adoption of learning to rank techniques. Even in these cases, the initial process of computing the ranking consists of applying a basic IR model, such as BM25 [11] or the Vector Space Model [12], to compute an initial rank of top results, typically limited to the size of just about one thousand documents [5].

In this paper we propose two new algorithms which reduce the overall time required to compute the final query ranking. The algorithms modify the best baseline we found in the literature, the *Blockmax WAND* (*BMW*) [6], to take advantage of a two-tiered index. The first algorithm we proposed, named *BMW-CS*, from BMW using the first tier as a candidate selector, uses the first tier to select candidate documents that are taken into account to compute the final ranking when processing the second tier. It achieves higher performance, but may result in small changes in the top results provided in the final ranking. In BMW-CS, the entries of the first tier are not present in the second tier. The second algorithm, named *BMW-t*, from BMW using the first tier as a threshold selector, preserves the top results and, while slower compared to BMW-CS, it is faster than BMW. The first tier is used only to compute a safe initial threshold to be adopted when processing the second tier, thus allowing

a slightly faster query processing when compared to BMW. The price paid is that the second tier in BMW-t should contain the full index and the first tier is an extra index.

While there were previous approaches based on using a smaller index tier to improve efficiency, our proposed algorithms are designed to take advantage of dynamic pruning techniques, not to minimize the effort of processing answers from the first tier, but to reduce the time required to read and process entries from the second tier.

The remainder of this article is structured as follows. Section 2 presents the background and related research necessary to better understand the proposed methods. Section 3 presents our methods, BMW-CS and BMW-t. Section 4 presents the experimental results. Finally, Section 5 presents the conclusion and prospective future work.

2. BACKGROUND AND RELATED WORK

Usually, most of the data needed by a search system to process user queries is stored in data structures known as *inverted files* [3]. They contain, for each term t, the list of documents where it occurs and an *impact factor*, or weight, associated with each document. This list of pairs of document and term impacts is called the *inverted list* of t and it is used to measure the relative importance of terms in the stored documents. Each document is represented in these lists by a value named *document id*, referred to as *docId* in this article. The inverted files may become huge in some systems, thus they are usually stored in a compressed format.

2.1 TAAT approach

In a query processing approach named Term-At-A-Time (TAAT), the inverted lists are sorted by term impact in non-increasing order. The query results are obtained by sequentially traversing one inverted list at a time. As an advantage, we can mention the fact that a sequential access behavior to inverted lists may speed up the process.

As the main disadvantage, we can mention that this strategy requires the usage of large amounts of memory to store partial scores achieved by each document when traversing each inverted list. These partial scores should be stored to accumulate the score results obtained by each document when traversing each inverted list. The final ranking can be computed only when all the inverted lists are processed. Several authors proposed methods to discard partial scores, thus reducing the amount of memory required to process queries in the TAAT mode [15, 2, 1]. Despite the significant reduction of required memory for query processing, the space required to store the partial scores remains one of the drawbacks of the TAAT query processing approach.

Anh and Moffat [1] studied the application of dynamic pruning over inverted indexes where the entries are sorted by impact, thus adopting a TAAT approach. In their work, a first phase processes blocks containing entries with higher impact in a disjunctive mode. Once all documents that might be present in the top-k results, k being a parameter, are found, the method starts a second phase applying a conjunctive mode query processing where only documents included as results by the first phase are considered. They also present a modified version of their algorithm, named as method B, where the top results may be changed, thus giving approximate results. In method B, only a percentage of the results obtained in the first phase are taken into account in the second phase. This modified version results in even faster query processing, but does not guarantee top k results of the ranking will not be modified.

Strohman and Croft [15] proposed a new method for efficient query processing when documents are stored in main memory that modifies the method presented by Anh and Moffat [1]. In their proposal, a dynamic pruning is applied in each phase of query processing over the candidate documents with the goal of obtaining the final query results without requiring a full evaluation of all candidates. The query processing is performed over impact sorted inverted lists and the impact values are discretized in a small range of integer values.

2.2 DAAT approach

Another approach to process queries is to adopt a Document-At-A-Time (DAAT) query processing. In this alternative, the inverted lists are sorted by *docIds*, which allows the algorithms to traverse all the inverted lists related to a query in parallel. As a consequence, the final scores of documents can be computed while traversing the lists and the system may store only the top results required by the user, so the memory requirements are fairly smaller. On the other hand the fragmented access may slow down the query processing and, since the inverted lists are sorted by *docIds*, important entries are spread along the inverted lists, making the pruning of entries more complex in this query processing approach.

As in the TAAT approach, several authors have presented algorithms and data structures to accelerate the query processing in the DAAT approach. For instance, data structures to allow fast jumps in the inverted lists, named as skiplists [7], are adopted to accelerate the query processing. Skiplists divide the inverted lists into blocks of entries and provide pointers for fast access to such blocks, so that a scan in the skiplist determines in which block a document entry may occur, if it does, in the inverted list.

The problem of efficiently computing the ranking of results for a given user query has been largely addressed in the literature in several research articles. We here detail the ones closer to our research, focusing on DAAT, which is the approach adopted by our algorithms.

2.2.1 WAND

Broder et al [4] proposed a successful strategy for query processing, known as WAND, which allows fast query processing for both conjunctive and disjunctive queries, with the possibility of configuring the method for preserving the top k documents in the query answer or not. In the case where the top answers are not guaranteed to be preserved, the method leads to even faster query processing.

WAND processes the queries using DAAT approach, so the inverted lists of the query terms are traversed in parallel. A heap of scores is created to keep the top k documents with larger scores at each step of the query processing, k being the number of documents requested to the search system. The smaller score in the heap at each moment is taken as a *discarding threshold* to accelerate the query processing. A new document is evaluated and inserted in the heap only if it has a score higher than this *discarding threshold*.

The WAND has a two level evaluation approach. In the first level, a document pivot has its *maximum possible score* evaluated using the information of *maximum score* of each

list where the document may occur. If the *maximum possible score* is greater than the actual *discarding threshold*, the document has its actual score evaluated. Otherwise, the document is discarded and a new pivot is selected.

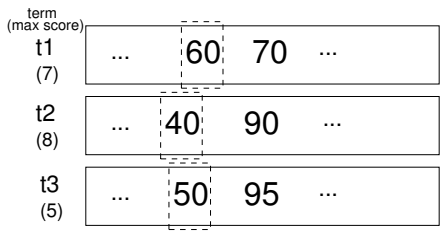

Figure 1: Inverted lists when processing query with terms t1, t2, and t3. Marked entries are the ones currently processed.

At each moment in the DAAT query processing, there is a pointer to the next document to be processed in each inverted list associated with the query. For instance, if we have a query with terms $t1$, $t2$ and $t3$, there will be a pointer to the next document processed in each of their lists, as illustrated in Figure 1. In the Figure, documents 60, 40 and 50 are currently pointed in lists $t1$, $t2$ and $t3$, respectively. The WAND algorithm assures that previous occurrences in each list were already examined and that each of the pointed documents represents the smaller *docId* in the list that was not yet processed [4]. Thus, since the lists are organized by *docIds*, at this point we know the smaller *docId* (40) occurs only in the list of term $t2$, while document 50 occurs in $t3$ and might occur in $t2$. Finally, document 60 occurs in the list of $t1$ and might occur in the other two lists.

At this point, knowing the maximum score a document may reach in each list, we can estimate the maximum scores each of the currently pointed documents can obtain when processing the query: 40 can reach *maximum possible score* equal to the maximum score a document may achieve in the list of $t2$; 50 can reach the sums of maximum scores of $t2$ and $t3$, and 60 can reach the sum of maximum scores of the three terms as its *maximum possible score*. Using this information, we may discard the documents which are not able to reach the *discarding threshold*, i.e., cannot achieve a score higher than the minimum score among the top k results already processed at this moment.

So, to discard documents, we check the current documents pointed in each inverted list associated with the query and estimate their *maximum possible score*. The entry with smaller *docId* among the ones that reach a maximum possible score higher than the current *discarding threshold* is then chosen as a candidate to be included in the answer. This entry is known as the pivot. We then move all posting lists so that they point to *docIds* of at least the same as the pivot. Notice only pointers of lists where the current *docIds* are smaller than the pivot require a movement. If one of the posting lists with *docId* smaller than the pivot does not have the pivot document, the document is discarded, a new pivot is selected and the process repeated. Otherwise, if all these lists contain the document, its actual score is then computed. If the actual score of the pivot is higher than the *discarding threshold*, it is included in the answer set and the *discarding threshold* is updated. After processing the pivot, we move all the lists to the next document with *docId* bigger than the pivot and start the process again.

An interesting feature of the WAND method is that it can be configured as a more or less aggressive pruning method by either applying a reduction in the max score values of each list or by increasing the pruning threshold so that fewer documents will be accepted in the top results. When the pruning threshold is increased, there is no guarantee of preserving the exact top result set. Further details about the WAND method can be found in the article where it was first proposed [4].

2.2.2 Blockmax WAND

Ding and Suel [6] have recently revisited the ideas presented in the WAND method, and proposed an even faster solution named the *Blockmax* WAND (BMW). In BMW, the entries of the inverted lists are grouped into compressed blocks that may be jumped without any decompression of their content. Each block contains information about the maximum impact among the entries found in it. Whenever such maximum impact is not relevant to change the results for a given query, the whole block is discarded, which avoids important query processing costs.

Authors present experiments which indicate BMW results in significant reduction of query processing times when compared to previous work, being currently the fastest query processing method found by us.

The BMW is based on the WAND algorithm and uses the same approach for selecting a document pivot during the query processing. In BMW, the pruning of entries is performed using two main pieces of information: (i) the maximum impact found inside an inverted list, which is also adopted in WAND; and (ii) The maximum impact found in each block pointed by the skiplist entries, known as the *block max score*, so that before accessing a block it is possible to predict the maximum impact of an entry among those found in such a block.

The basic idea of BMW is to take advantage of information (ii) to speedup query processing. Once a candidate document is selected to have its score computed, the algorithm uses the *block max score* information to accelerate the query processing by discarding documents with no chance of being in the top answers. The algorithm also allows skipping entire blocks of inverted list entries based on the *block max score* present on the skiplists, a procedure that they call shallow movement. Contrary to the regular movement present on the inverted lists, the shallow movement accesses only the skiplist entries, which avoids costs to decompress blocks of the inverted lists when processing queries.

The BMW algorithm starts with a pivoting phase, where a document is selected as a candidate to be inserted in the answer. Its pivoting phase is similar to the one performed in the WAND algorithm. Using the global max score of each list and the lists ordered by the value of the current *docId* pointed in each of them.

Before decompressing and evaluating the pivot document, BMW makes a shallow movement to align the inverted lists over the blocks that possibly have the document. After the alignment, the algorithm uses the information of *block max score*, stored in the skiplists, to estimate a local upper bound score of the candidate document. If this upper bound score is lower than a given pruning threshold, the document is discarded and one of the lists is advanced.

As in WAND, the pruning threshold is dynamically updated according to the score of the evaluated candidate. As the processing is DAAT, each evaluated candidate has its complete score calculated, since all inverted lists are processed in parallel. Further details about the BMW algorithm can be found in the article where it was proposed [6].

Shan et al [13] show that the performance of BMW is degraded when static scores, such as Pagerank, are added in the ranking function. They study efficient techniques for Top-k query processing in the case where a page's static score is given and propose a set of new algorithms based on BMW which outperform the existing ones when static scores are taken into account when computing the final ranking. Their study can also be applied to our proposal as a future work, being orthogonal to the study presented here.

2.3 Multi-tier indexes

Some authors proposed the splitting of the inverted index in more than one tier in an attempt to accelerate query processing. In these architectures, the query processing starts in a small tier and only if necessary proceeds to larger tiers. Risvik and Aasheim [10] divided the index into three tiers with the goal of achieving better scaling when distributing the index. According to static and dynamic sources of evidence, documents considered as more relevant are selected to compose the smaller tier. The query processing starts in the higher tier and only if the result set is not satisfactory, according to an evaluating algorithm, the query processing proceeds to the next tier. Performance gains are achieved when the processing does not visit the larger tiers. There is no guarantee that the result set is the exact set if compared to the exhaustive query processing.

In our proposed algorithms, we also kept the documents considered more important, in our case those with higher impact, in the smaller tier. However, in this paper we use each tier as part of the query processing. Our algorithms could, for instance, be applied to each tier proposed by Ravisk and Aesheim [10], being then ortoghonal to their proposal.

Ntoulas and Cho [8] presented a two-tiered query processing method that avoids any degradation of the quality of results, always guaranteeing the exact top-k results set. The first tier contains the documents considered the most important to the collection, selecting the entries by using static and dynamic pruning strategies that remove non-relevant documents and terms. The second tier contains the complete index. Their proposal uses the first tier as a cache level to accelerate query processing. Whenever the method detects that the results of the first tier assures that the top results will not be changed, it does not access the second tier, basing its results only on the first tier. Otherwise, they process the query using the second tier.

To guarantee that their method preserves the top answer results when compared to a system without pruning, the method evaluates the ranking function when processing the first tier, assigning the maximum possible score that could be achieved when processing the full index for entries that are not present in the first tier. The authors show how to determine the optimal size of a pruned index and experimentally evaluate their algorithms in a collection of 130 million Web pages. In their experiments, the presented method achieved good results for first tier index sizes varying from 10% to 20% of the full index. The two-tier strategy is also adopted in our article, but instead of using the first tier as a cache, we use it as a candidate selection layer as part of our algorithms to compute the top results of a given query. Another important difference is that we always process the queries using both tiers, and we do not guarantee the exact top-k results.

Skobeltsyn et al [14] evaluate the impact of including a cache in the system when using the two-tier method presented by Ntoulas et al [8] and shows the query distribution workload is affected by the cache system. When using the cache, queries with a few terms, which would be the ones that would benefit more from the two-tier strategies, are usually solved by the cache system. As a consequence, the queries with more terms, which are difficult to process with pruning strategies, become more important in the workload when considering a system that adopts a cache of results.

3. BMW-CS AND BMW-T

In this section, we present the details of the algorithms we proposed to accelerate the query processing when computing the ranking of results, which are named *BMW-CS*, from BWW with candidate selection, and *BMW-t*, from BMW with a threshold selection. These algorithms are based on a two-tiered index organization in which the first tier is a small index created using the entries with the highest impact from each term list, while the second tier is a larger index. The idea of relying on a high-quality tier is similar to the one adopted by Ntoulas et. al. [8].

The main objective of our algorithms is to use the first tier to accelerate the query processing in the second tier (i.e., the larger index) when computing the final ranking. In *BMW-CS*, the two tiers are disjointed, that is, the high-impact entries in the small index in first tier are not present in the second one. In *BMW-t*, the second tier contains the full index. Thus, even the high-impact entries in the first tier are also present in it.

In both cases, to select the entries for the first tier, we compute a global threshold to select entries so that the size of the first tier is about Δ% of the full index. The parameter Δ provides an estimation of the final size of the first tier index. We adopted a minimum size of 1000 entries in each inverted list to prevent any individual list from becoming too small.

In our index organization, we used skiplists in both tiers in order to accelerate the query processing. For each block of 128 document entries, a skiplist entry is created that keeps the information of the current $docId$ and the highest impact among the documents of the block. Also, for each term in the collection, the highest impact (max score) in the whole inverted list and the lowest impact found in this list in the first tier are computed and stored along with the term information. The lowest impact in the list at the first tier can be seen as an upper bound for the impact of the document entries that were not included in this list. Using this information, we can set the upper bound of contribution for the entries that are not present in the first tier, but appear in the inverted list in the second tier.

3.1 BMW-CS

Listing 1 presents our first algorithm called *BWM with candidate selection*, or *BMW-CS*. In its first phase, BMW-CS uses the first tier to select documents that are candidates to be present in the top results. Initially, the set of candidate documents, denoted by $\mathcal{A}$, is generated by the function

Listing 1: Algorithm BMW-CS

```
1  BMW-CS(queryTerms[1..q], k)
2    A ← SelectCandidates(queryTerms, k)
3
4    sort A by score
5    min_score ← A_k.score
6
7    //Remove the candidates with low upper_score
8    for (i = 0 to |A|)
9      if(A_i.upper_score < min_score) remove A_i
10   end for
11
12   R ← CalculateCompleteScore(A, queryTerms, k)
13
14   return R
```

SelectCandidates, which computes the candidates for the top k results of a query composed of a set of terms (Listing 2). Then, the algorithm trims this set of documents, removing all candidates that cannot be present in the top-k results. This trimming decreases the cost of the second phase. In the second phase, the second tier is processed to compute the final ranking of the top k results. This phase is performed by the function *CalculateCompleteScore* (Listing 6).

The main idea behind BMW-CS is to take advantage of the fact that the first tier has entries with higher impact in order to significantly reduce the amount of documents analyzed in the second tier. We show in the experiments that this approach yields a quite competitive query processing algorithm.

3.1.1 Candidates Selection

BMW-CS selects candidate documents from the first tier. However, the inverted lists in the first tier are not complete, which means, for instance, that a document which appears only in the list of one of the terms of a query in the first tier, may appear in lists of other terms of this query when considering the entries present in the second tier. Thus, during the candidate selection phase, the algorithm may discard a document that could have a high enough score when the full inverted lists are evaluated.

To reduce the possible negative impacts of this incompleteness of the lists in the first tier, we modified the BMW algorithm so that it considers the possibility of a missing pair $(term, docId)$ in the first tier to occur in the second tier. During the pivoting phase and the upper boundary score checking, a lower boundary score is added for each missing term of the document that might be absent in the first tier, thus avoiding the possibility of discarding high score candidates due to incompleteness in the first tier.

This lower boundary score represents the max score that a document can achieve after processing the inverted list in the second index. With these two values, we can adjust the *discarding threshold* used to prune documents in BMW. A minimum heap is used to store the top-k documents with a higher score. The smallest score of the heap is used as the *discarding threshold* for BMW to dynamically prune entries with no chance to be part of the final top-k results. All evaluated documents that have a score higher than the *discarding threshold* when processing the first tier are added to the set of candidate documents.

We can see the detailed algorithm for the candidate selection phase in Listing 2. The algorithm starts by selecting

Listing 2: Algorithm SelectCandidates

```
1  SelectCandidates (queryTerms[1..q], k)
2    Let H be the minimum heap to keep the top k results
3    Let A be the list of candidates
4    Let I_cand be the first tier index
5
6    lists ← I_cand(queryTerms);// Gets inverted lists
7    θ ← 0;
8    //Point to the first docId in each list
9    for each {0 ≤ i < |lists|} do Next(lists[i], 0);
10
11   repeat
12     sortByCurrentPointedDocId(lists);
13     p ← Pivoting(lists, θ);
14     if (p == −1) break; //No more candidates
15     d ← lists[p].curDoc;
16     if(d == MAXDOC) break; //End of the list
17
18     //Move only the skip pointers
19     for each {0 ≤ i ≤ p} do NextShallow(lists[i], d);
20
21     if( CheckBlockMax(θ, p) == TRUE)
22       if(lists[0].curDoc == d)
23         doc.docId ← d;
24         doc.score ← Σ_{i=0}^{p} BM25(lists[i]);
25         doc.upper_score ← doc.score +
26                           Σ_{i=p+1}^{|lists|} lists[i].min_score;
27
28         if (|H| < k) H ← H ∪ doc;
29         else if (H_0.score < doc.score)
30           remove H_0; // the one with smallest score
31           H ← H ∪ doc;
32           θ ← H_0.score; //Update the threshold
33         end if
34
35         //Insert only documents with possible score > θ
36         if (θ <= doc.upper_score)
37             doc.terms ← queryTerms[0..p];
38             A ← A ∪ doc;
39             if ({∃dLow ∈ A| dLow.upper_score < θ})
40                 A ← A − dLow;
41             endif
42         endif
43         //Advance all evaluated lists
44         for each {0 ≤ i ≤ p} do Next(lists[i], d+1);
45       else
46         j ← {x|lists[x].curDoc < d ^
47             |lists[x]| < |lists[y]|, ∀0 ≤ y < p};
48         Next(lists[j], d);
49       end if
50     else
51       d_next ← GetNewCandidate(lists[j], p);
52       j ← {x| |lists[x]| < |lists[y]|, ∀0 ≤ y ≤ p};
53       Next(lists[j], d_next);
54     end if
55   end repeat
56
57   return A
```

the inverted lists to be processed (Line 6), which are the lists that represent each query term. The *discarding threshold*, Θ, is initially set to 0(Line 7) and is updated to the minimum score stored in the heap $\mathcal{H}$ if it is full (Line 32). Line 9 makes each of the inverted lists point to their first document. The function $Next(l, d)$ searches in the skiplist associated with list l for the block where there is the first occurrence of a $docId$ equal or bigger than d, setting $l.current\_block$ to the found position. Then, it moves the pointer to the current document of the list, $l.curDoc$, to the smallest entry with value greater than d.

The lists shown in Listing 2 are represented by vector *lists* and each list has an internal pointer to the docId being processed at each moment, the current $docId$. In Line 12 we sort this vector into increasing order according to the current $docId$ pointed by each of these lists. We then compute in Line 13 the next document that has a chance to be present in the top results, performing the pivoting, which is

Listing 3: Algorithm Pivoting

```
1  Pivoting (lists, θ)
2    accum ← 0;
3    for each 0 ≤ i < |lists| do
4      accum ← accum + lists[i].max_score;
5      accum_min ← Σ_{j=i+1}^{|lists|} lists[j].min_score
6      if (accum + accum_min >= θ)
7          while(i+1<|lists| AND
8                lists[i+1].curDoc == lists[i].curDoc) do
9              i ← i + 1;
10         end while
11         return i
12     end if
13   end for
14   return −1;
```

Listing 4: Algorithm CheckBlockMax

```
1  CheckBlockMax (lists, p, θ)
2
3    //Sum the max score of each block, that d can appear
4    max ← Σ_{i=0}^{p} lists[i].getBlockMaxScore();
5
6    //Add the min score of the lists that d may appear in
         the full index
7    max ← max + Σ_{i=p+1}^{|lists|} lists[i].min_score;
8    if(max > θ) return true
9    return false
```

Listing 5: Algorithm GetNewCandidate

```
1  GetNewCandidate (lists, p)
2    mindoc ← MAXDOC
3
4    //Selects the lower docId between the blocks boundaries
5    // of the lists already checked
6    for each {0 ≤ i ≤ p} do
7      if(mindoc > lists[i].getDocBlockBoundary())
8        mindoc ← lists[i].getDocBlockBoundary();
9      end if
10   end for
11
12   //Select the lower docId between the lists not checked
13   for each {p + 1 ≤ i < |lists|} do
14     if(mindoc > lists[i].curDoc)
15       mindoc ← lists[i].curDoc;
16     end if
17   end for
18
19   return mindoc; //Return the smallest docId found
```

the main step of the BMW heuristic. Our pivoting, however, is computed taking into consideration the possibility of some of the entries of a document being not included in the first tier, which let us add this information to compute the *upper_score*, which is the maximum score when selecting the pivot. This procedure is described in Listing 3. Lines 15 to 17 test break conditions and set the current document to be analyzed by the algorithm.

Line 19 adopts the function NextShallow to move the current documents pointer in each list. The function NextShallow is the same presented in the original BMW proposal, and differs from function Next because it does not need to access the documents, accessing only the skiplists of the inverted lists to move their pointers and set a new current block in each inverted list. Using this function, we can skip entries without needing to access or decompress them. Line 21 calls function CheckBlockMax, detailed in Listing 4, which is also modified when compared to the original one proposed in BMW, since it also needs to deal with the incompleteness of the first tier.

The remaining algorithm checks whether a document has enough score to be included in the answer. Line 25 takes the incompleteness of the first tier into consideration when computing the *upper_score*. The score of each document is used to include it in heap $\mathcal{H}$, Lines 28 to 33. $\mathcal{H}$ is maintained to control the discarding threshold θ. The upper_score of each document is used in Line 36 to check whether a document should be included in the candidate documents set $\mathcal{A}$.

The threshold θ changes as more documents are processed, so, whenever we add a document to $\mathcal{A}$, we also check if there is at least one document in $\mathcal{A}$ with an upper_score value lower than the current value of θ. In such cases, we remove the document (Lines 39 to 40). This procedure avoids wasting memory by keeping elements in $\mathcal{A}$ which will be discarded at the end of the process. By the end of the candidate selection algorithm, the list of candidate documents $\mathcal{A}$ is returned, so that the final result can be obtained by processing the remainder of the index in the second tier.

3.1.2 Computing the Final Ranking

In function *CalculateCompleteScore* (Listing 6), the scores of candidates with missing terms are evaluated using the larger index in the second tier, which contains the index entries not present in the first tier. To avoid unnecessary costs with decompression, the shallow movement described in [6] is used to align all the term lists. Then, a second BlockMaxScore check is made to verify whether the document can be part of the top k results or not. Each document is evaluated only if it has a high enough score, otherwise it is discarded. As in the first phase, we keep a minimum heap with the documents with the greatest scores evaluated, and the minimum score of this set is used as a *discarding threshold* to prune candidates.

As the candidate set is small, and only documents with incomplete scores are evaluated, this phase is expected to be performed extremely fast, even considering that it processes a the larger tier.

In *BWM-CS*, the first tier may not contain enough information to assure all top documents are considered as candidate documents. Since only candidate documents can be included in the final results, it does not guarantee exact results in the final ranking. For instance, a document which contains entries for all three terms of a query, but whose entries are present only in the second tier, will not be included in the candidate selection. However, this document may achieve scores higher than the ones in the top documents found by the candidate selection, in cases where there are top results that do not contain all query terms. Notice however that such a situation tends to occur for documents that would be included in the final positions of the top results and, as we show in our experiments, is it does not affect the final results very much.

3.2 BMW-t

In our second algorithm, *BMW with threshold selection*, or *BMW-t*, we use just the first tier to set the initial discarding threshold adopted by methods WAND and BMW. In these methods, this initial discarding threshold is set to 0 at the beginning, and grows as the documents with higher scores

Listing 6: Algorithm CalculateCompleteScore

```
1  CalculateCompleteScore( A, queryTerms[1..q], k)
2    Let H be the minimum heap to hold the k most relevant
       candidates
3    Let I_second_tier be the second index
4    lists ← I_second_tier(queryTerms) //Select the terms lists
5    θ ← 0
6    H ← ∅
7    sort A by docId;
8
9    for each {0 ≤ i < |A|} do
10     if (A_i.score < A_i.upper_score)
11      local_upper_score ← A_i.score;
12
13      for each {0 ≤ j < |lists|} do
14       if (queryTerm[j] ∉ A_i.terms)
15        NextShallow(lists[j], A_i.docId);
16        local_upper_score ← local_upper_score +
17                              lists[j].getBlockMaxScore();
18       end if
19      end for
20
21      if (local_upper_score > θ)
22       for each {0 ≤ j < |lists|} do
23        if (queryTerm[j] ∉ A_i.terms)
24         Next(lists[j], A_i.docId);
25        end if
26       end for
27       //Complete the score with the missing lists
28       for each {0 ≤ x < |lists||lists[x].curDoc = A_i.docId} do
29         A_i.score ← A_i.score + BM25(lists[x]);
30       end for
31      end if
32     end if
33
34     if (θ < A_i.score)
35       if (|H| < k)
36         H ← H∪ A_i;
37       elseif (H_0.score < A_i.score)
38         remove H_0;
39         H ← H∪ A_i;
40         θ ← H_0.score;
41       end if
42     end if
43   end for
44
45   sort H by score;
46   return H;
47 end
```

are found and included in the answer. As a consequence, the query processing discards fewer documents at the beginning of the process, since the discarding threshold starts with a small value. We thus propose the usage of the first tier of the index to support a pre-processing stage just to compute an initial discarding threshold that is higher than 0. This simple strategy naturally may speed up the process if the gains when processing the full index are worth the cost of computing the initial discarding threshold when processing the first tier.

We then experiment with a variation of BMW, we named BMW-t, and a variation of WAND, we named WAND-t. These variations use the first tier to select an initial *discarding threshold* when processing the queries. This new usage of the two tier index presents the advantage of preserving the top k results, which does not happen in BMW-CS. The WAND-t performed worse than the BMW-t, thus we report only BMW-t in the experiments.

4. EXPERIMENTAL EVALUATION

We used the TREC GOV2 collection for the experiments in this paper. The collection has 25 million web pages crawled from the .gov domain in early 2004. It has 426 gigabytes of text, composed of HTML pages and the extracted content of pdf and postscripts documents. The full index has about 7 gigabytes of inverted lists and a vocabulary of about 4 million distinct terms. We applied the Porter Stemmer [9] when indexing the collection. Our indexes use the frequency of the terms as the impact information. To evaluate the quality of query results, we randomly selected a set of 1000 queries from the TREC 2006 efficiency queries set and removed the stop-words from these queries. During the query processing, the entire index is loaded to memory, to avoid any possible bias in the query processing time. | All these setup options were chosen for being similar to those adopted in the baseline [6] and previous studies [15]. We ran the experiments in a 24-cores Intel(R) Xeon(R), with X5680 Processor, 3.33GHz and 64 GB of memory. All the queries were processed in a single core.

We used Okapi BM25 as the rank function, but our method can be adopted to compute other ranking functions. The generated skiplists have one entry for each block of 128 documents. Each skiplist entry keeps the $docId$, to help the random decompression, and the *maximum impact* registered in the block. We also experimented with blocks of 64 entries, and the results and conclusions were about the same, thus we decided to report only results with 128. We varied the first tier $\Delta\%$ from 1 to 20 percent of the full index in the experiments.

Another parameter evaluated in the experiments was the size of the top results required by the algorithms. We evaluated the algorithms requesting 10 and 1000 results. Retrieving top 1000 results was included to simulate an environment where the top results are computed to feed a more sophisticated ranking method that performs a re-rank of results. The top 10 results was included to simulate a more common scenario where the user is interested in getting only a small list of results.

We evaluated the methods in terms of query response time, the amount of accumulators required to process the queries, amount of decoded entries from the inverted lists, and finally the mean reciprocal rank Distance (MRRD), presented in Equation 1, which was the measure adopted by Broder et al [4] to evaluate the distance between the results when preserving top results to rankings that do not preserve the top results. Using the MRRD distance, we are able to know how much an approximated rank differs from the one that preserves the top results. This measure returns a value between 0 and 1, where identical results provide MRRD=0, and completely distinct results provide MRRD=1.

$$MRRD(B,P) = \frac{\sum_{i=1,d_i \in B-P}^{k} 1/i}{\sum_{i=1}^{k} 1/i} \quad (1)$$

4.1 Baselines

One of the implemented baseline algorithms was BMW [6]. As one of our methods may not preserve the top ranking results, we also have considered including as a baseline the version of WAND that does not preserve the top results, proposed by Broder et al [4]. However, the results achieved by the approximate WAND were even slower than the BMW. We thus removed it from the baselines, and modified the method BMW to implement the same approximation strategy proposed to WAND. As a result, we transformed BMW to an approximated top-k query processing algorithm, which does not guarantee preserving top results, but may be faster

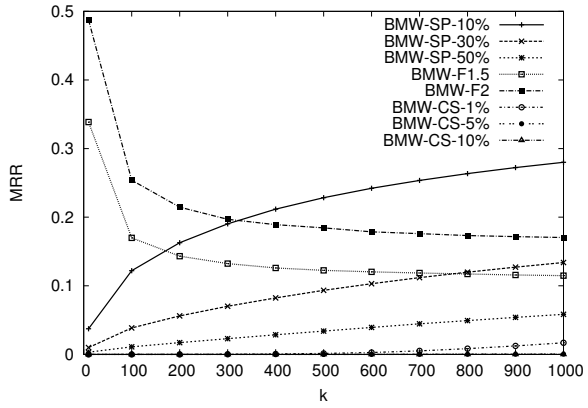

Figure 2: MRRD values compared to a exact rank for baselines BMW-SP and BMW-f

than original BMW. Basically, we artificially increase the pruning threshold during the query processing by multiplying this threshold for a pruning factor f. If f is 1, the exact top-k is guaranteed. When f is greater than 1, there is no guarantee of preserving the exact rank, but less entries will be evaluated, resulting in performance gains.

Finally, a reader could wonder if the results would also be good if we had adopted just the first tier for processing queries, considering the first tier as a statically pruned index. Thus, in order to avoid these doubts, we included a naive method using only the first tier for processing queries. We named it *BMW-SP* (applying BMW with static pruning).

4.2 Results

We begin the report of our experimental results by answering the possible doubts about the advantage of using the BMW-CS algorithm, which does not guarantee that all the top results are preserved, when compared to a simple static pruning strategy that adopts only the first tier to compute the ranking. Figure 2 presents the MRRD results when computing the top k results by using our method with the first tier of 1%, 5% and 10% compared to the usage of the static pruning approach, named BMW-SP. As can be seen in Figure 2, even when using BMW-SP with the first tier with 50% of the full index, the error level (MRRD) obtained by BMW-CS with 1% is still smaller than it. Even considering this observation, for the sake of completeness, we still report the time efficiency results obtained when using the BMW-SP with the first tier being 50%.

Figure 2 also presents the BMW-f MRRD results when varying parameter f. As can be seen in Figure 2, BMW-f achieves error levels worse than BMW-CS even when setting the factor f to the low value of 1.5. Lower factor values would slow down the performance of the method, thus we adopt this factor in our efficiency experiments, even considering that its error level is higher than those achieved by our method. We stress that both BMW-f and BMW-SP are included in the experiments to avoid doubts about these possible variations in the usage of BMW. These methods were not explicitly proposed in the literature.

Next we present the percentage of entries we included in the first tier of BMW-CS. This choice affects three main factors: the time for processing queries, the MRRD results of our methods and the amount of memory required to process queries. Variation of these parameters are illustrated in Figure 3. As can be seen, when computing the top 1000 results, the MRRD results decrease as the size of the first tier increases, being almost zero for first tier sizes higher than 8%. When computing the top 10 results, MRRD was always zero for all sizes of first tier experimented. Time tends to increase as the first tier increases in both cases. On the other hand, the number of accumulators presents more complex behavior. In the top 1000, it first increases as the size of the first tier increases. Then, at some point, it starts to decrease, since the candidate selection procedure starts to perform better pruning, thus reducing the set of candidate documents.

When looking to the general results presented in Figure 3, we conclude that a good size for the first tier in BMW-CS is 2% when tuning the method to compute the top 10 results and 10% when tuning the method to top 1000 results. These parameters provide a good combination of a low number of candidate documents, low query processing times and low MRRD. Notice however that even if choosing other first tier sizes among the ones presented in the experiments, still our method would be quite fast and competitive.

Figure 3 (c) and (d) also indicates how much memory our algorithm needs to store the candidate documents. We can see the requirement is not so high in both the top 10 and top 1000 scenarios, being limited to a few times greater than the size of top results required to be computed.

Finally, regarding the MRRD error level achieved by BMW-CS it is noteworthy that a user would almost not perceive the differences in top k results when using BMW-CS even when considering higher values of k. For instance, as $k = 1000$, the error of BMW-CS is still smaller than 0.0001 in terms of MRRD for the 10% first tier size. To better illustrate what this error level means, analyzing the results in detail, we perceived that BMW-CS with a 10% first tier resulted in no changes in the top 1000 results for 90% of the evaluated queries. Further, in all experimented first tier sizes, the top 10 results were preserved for all queries. Changes in the ranking, when they occur, are more common in the bottom results. For instance, we preserved the top 100 results for all queries, and preserved the top 200 results for 99.9% of the queries.

We also studied the MRR variation of method BMW-t, but do not present the variation due to space restrictions. Its performance is close to the best when using 1% for the first tier, becoming slower as the partition increases and not improving so much when it decreases. We report the results on the remaining experiments with our methods using 2% and 10% first tier sizes in case of BMW-CS, and 1% in case of BMW-t.

In Tables 1 and 2, we can observe the performance of the algorithms in terms of MRRD, decoded integers and query time when processing the top-10 and top-1000. The results of time are presented with confidence level of 95%. BMW-CS provides extremely low MRRD results, which means these answers are almost the same as the correct top k results. On the other hand, its performance is considerably better than BMW and BMW-f, which do not have a preprocessing phase during query evaluation, and thus are performed directly over the full index.

We can see the performance of BMW-t, which preserves the top results, presents an improvement of around 10% in

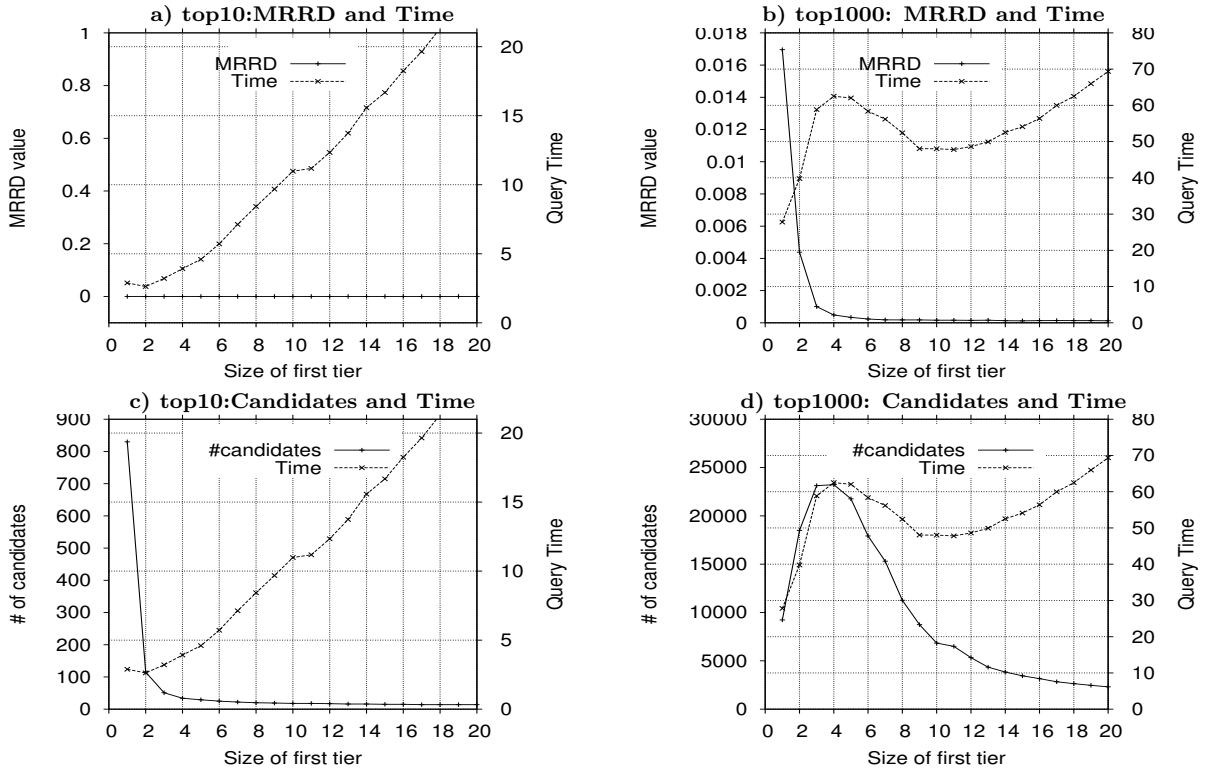

Figure 3: Variation in time and MRRD(a and b); and between time and number of candidate documents stored in (c and d) when processing first tier for method BMW-CS when using distinct sizes of first tier.

query processing times when compared to BMW. The approximated version BMW-f, using the factor f=1.5, processes query 20% faster than the exact BMW. However, BMW-CS is not only faster than BMW-f, but also presents lower MRRD values.

Still regarding Tables 1 and 2, we can see that BMW-CS preserves the result set more than the other approximated methods implemented and is about 40 times faster than BMW when computing the top 10 results (using a 2% tier), and about 4.75 times faster when computing the top 1000 results(using a 10% tier). These gains can also be seen when analyzing the number of decoded integers, which is one of the main costs when elements are stored in memory.

	top 10		
Algorithm	MRRD	Decoded	time
BMW	0	2353066	100.1 ± 9.9
BMW-f1.5	0.3386	1797451	78.5 ± 7.8
BMW-SP-50	0.0032	1700488	79.8 ± 8.0
BMW-t	0	2082782	89.5 ± 8.8
BMW-CS-2	0	48989	2.4 ± 0.4
BMW-CS-10	0	217627	10.4 ± 0.5

Table 1: MRRD, number of decoded entries, and time(ms) efficiency achieved by the experimented methods when computing top 10 results.

Table 3 presents the performance of the algorithms when processing distinct query sizes for both the top 10 and top 1000 results computation. BMW-CS was the fastest option for all query sizes. Although the gain is smaller for queries with more than 5 terms, our method still results in impressive gains when compared to all baselines even for long queries.

	top 1000		
Algorithm	MRRD	Decoded	time
BMW	0	5032834	226.0 ± 18.0
BMW-f1.5	0.1149	4463099	193.7 ± 15.4
BMW-SP-50	0.0584	3495885	174.8 ± 13.0
BMW-t	0	4799477	205.7 ± 16.9
BMW-CS-2	0.0043	1258859	39.8 ± 4.6
BMW-CS-10	0.0001	921687	47.6 ± 4.2

Table 2: MRRD, number of decoded entries, and time(ms) efficiency achieved by the experimented methods when computing top 1000 results.

One explanation for the smaller gain in long queries is that in the first phase of the process, as we have many terms in the query, the *upper_score* of a candidate will be higher because it sums the *minimum score* of the missing lists. Thus, as we have an index with only a small fraction of the full index, the number of documents in the first phase with a complete score will be lower according to the number of terms in the query, making the threshold values lower when compared to the estimated *upper_scores*. This configuration will lead to a less effective pruning during the candidate selection, increasing the costs of the whole process.

We observed differences in time results when comparing our experiments to the ones presented by Ding et al [6]. While we adopt exactly the same dataset, the query processing times we obtained are lower than the ones presented in their article. However we see that the number of decoded

Algorithm	Query time (ms)				
	2	3	4	5	>5
	top 10				
BMW	12.0	46.6	100.5	197.9	442.7
BMW-f1.5	7.8	34.0	77.3	155.5	367.7
BMW-SP-50	10.3	38.7	82.4	160.1	331.6
BMW-t	10.4	41.1	87.7	179.3	401.7
BMW-CS-2	0.58	1.42	2.42	3.74	9.96
BMW-CS-10	2.3	5.7	11.6	19.7	36.8
	top 1000				
BMW	37.9	122.3	243.6	437.9	848.6
BMW-f1.5	29.9	102.5	206.3	390.1	725.0
BMW-SP-50	32.6	102.1	192.0	333.5	610.3
BMW-t	29.8	104.6	220.9	406.5	803.5
BMW-CS-2	8.49	20.62	44.58	79.07	139.1
BMW-CS-10	14.6	29.3	51.8	82.2	160.8

Table 3: Time achieved by the experimented methods when processing queries with distinct sizes and computing top 10 and top 1000 results.

integers still similar. The final difference in times can be due to the choice of the queries, since we do not know the exact set of queries adopted by them, the architecture of the implemented system, and the machines used for the experiments. These differences do not affect the conclusions presented in our study because they affect all the experimented methods.

Acknowledgements

This work is partially supported by INWeb (MCT/CNPq grant 57.3871/2008-6), DOMAR (MCT/CNPq 476798/2011-6), TTDSW (FAPEAM), by CNPq fellowship grants to Edleno Moura (307861/2010-4) and Altigran Silva (308380/2010-0), and FAPEAM scholarship to Cristian Rossi.

5. CONCLUSION

The algorithms proposed and studied by us outperform the existent state-of-the-art algorithms for DAAT query processing. BMW-CS presents the advantage of being about 40 times faster than BMW when computing top 10 results and 4.75 times faster when computing top 1000 results. While it does not guarantee to preserve the top results, we show through experiments that the application of the algorithm does not change the results very much. The MRRD error level is quite small and the algorithm provides an impressive gain in performance. Thus, in situations where preserving the top results is not mandatory, the BMW-CS algorithm is an interesting alternative.

The price paid for this fast query processing is the necessity of more memory for processing queries. As we show, the number of candidate documents stored by the algorithm, which is the extra memory required by it, is not prohibitive, being, for instance, around 10 times the size of the final results in our experiments. Further, in practice a search system usually processes queries in multiple threads per machines, and the reduction in the query processing times cooperates to increase the throughput, thus compensating for the extra memory required by each thread. We intend to better study this question in future studies, since this was not the focus of this current study.

The second algorithm proposed by us, BMW-t, presents the advantage of preserving the top results. It delivers smaller, but significant gains when compared to the application of plain BMW, being about 10% faster.

Finally, in the future, we also want to study the combination of our algorithm BMW-CS to ranking strategies that take more information into account. In this regard, we plan to study how our algorithms can be adapted to strategies as those presented by Shan et al [13], where the authors study the impact of including external sources of relevance evidence into the performance of query processing algorithms, and such as [5], in which the authors show how to encode several features into a single impact value.

6. REFERENCES

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

[2] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *ACM SIGIR*, pages 35–42, 2001.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Publishing Company, USA, 2nd edition, 2011.

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

[5] A. Carvalho, C. Rossi, E. S. de Moura, D. Fernandes, and A. S. da Silva. LePrEF: Learn to Pre-compute Evidence Fusion for Efficient Query Evaluation. *JASIST*, 55(92):1–28, 2012.

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

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

[8] A. Ntoulas and J. Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *ACM SIGIR*, pages 191–198, 2007.

[9] M. Porter. An algorithm for suffix stripping. *Program: electronic library and information systems*, 40(3):211–218, 2006.

[10] K. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In *First Latin American Web Congress*, pages 132–143, 2003.

[11] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *ACM SIGIR*, pages 232–241, 1994.

[12] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. Technical report, Ithaca, NY, USA, 1974.

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

[14] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates. ResIn: a combination of results caching and index pruning for high-performance web search engines. In *ACM SIGIR*, pages 131–138, 2008.

[15] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *ACM SIGIR*, pages 175–182, 2007.