Inf Retrieval J (2017) 20:292–316
DOI 10.1007/s10791-017-9298-6

INFORMATION RETRIEVAL EFFICIENCY

Waves: a fast multi-tier top-*k* query processing algorithm

Caio Moura Daoud[1] · Edleno Silva de Moura[1] · David Fernandes[1] · Altigran Soares da Silva[1] · Cristian Rossi[1] · Andre Carvalho[1]

Received: 30 May 2016 / Accepted: 28 February 2017 / Published online: 13 March 2017

Abstract In this paper, we present Waves, a novel document-at-a-time algorithm for fast computing of top-*k* query results in search systems. The Waves algorithm uses multi-tier indexes for processing queries. It performs successive tentative evaluations of results which we call waves. Each wave traverses the index, starting from a specific tier level *i*. Each wave *i* may insert only those documents that occur in that tier level into the answer. After processing a wave, the algorithm checks whether the answer achieved might be changed by successive waves or not. A new wave is started only if it has a chance of changing the top-*k* scores. We show through experiments that such lazy query processing strategy results in smaller query processing times when compared to previous approaches proposed in the literature. We present experiments to compare Waves' performance to the state-of-the-art document-at-a-time query processing methods that preserve top-*k* results and show scenarios where the method can be a good alternative algorithm for computing top-*k* results.

Keywords Information retrieval · Query processing · Search system

1 Introduction

In recent decades search systems have been one of the main forms of locating and retrieving information in digital environments. They are present in a large number of applications, such as web search engines and e-commerce systems. Users of these systems

✉ Caio Moura Daoud
caiodaoudalti@icomp.ufam.edu.br

Edleno Silva de Moura
edlenoalti@icomp.ufam.edu.br

David Fernandes
davidalti@icomp.ufam.edu.br

[1] Institute of Computing, Federal University of Amazonas, Manaus, AM, Brazil

more often than not have very specific information needs, only being satisfied with a few, highly relevant results. Due to this behavior, part of the recent research effort related to search systems aims to reduce computational costs to compute the top results of queries, which are the ones usually presented to most users.

While 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 the titles of the pages, URL content, and link structure, among others. These sources of relevance evidence are combined using a myriad of approaches, such as the adoption of learning to rank techniques, in order to produce the final ranked results. In such cases, the initial process of computing the ranking consists of applying a basic IR model, such as BM25 (Robertson and Walker 1994) or the Vector Space Model (Salton et al. 1974), to compute an initial rank of top results (Carvalho et al. 2012). Usually, only these top results are considered in the more sophisticated learning to rank stage. Thus, much of the research aiming to reduce computational costs in query processing is related to reducing the number of documents considered when the initial rank is created.

In this paper we propose and evaluate Waves, a multi-tier query processing algorithm that performs multiple traversals, which we call waves, in the index tiers. Each traversal starts from a tier i and computes the scores of only those documents which occur in i and that have a chance of being present in the final result. To do so, each wave accesses document entries not only for tier i, but for every tier $j \geq i$, thus each wave traverses the index from the current tier to all remaining tiers. We experimented our method in distinct scenarios using a benchmark collection also adopted in previous articles related to this topic. The results indicate that our new method is competitive in all scenarios when compared to the state-of-the-art document-at-a-time query processing methods we found in the literature.

The rest of this article is structured as follows: Sect. 2 presents the background and related research necessary to understand the proposed methods better. Section 3 presents our proposed method. Section 4 presents the experimental results. Finally, Sect. 5 presents the conclusion and prospective future research.

2 Background and related work

Search systems usually perform a prior indexing of the data in order to reduce processing times and cope with the usually large volumes of data that they must process. The basic data structure adopted by most search systems is known as *inverted file* (Baeza-Yates and Ribeiro-Neto 2011). An inverted file contains, for each term t, where a term is usually a word or some variation, the list of documents where it occurs and an *impact factor*, or weight, which indicates the importance of t within each document. This list of pairs of document and term impacts is called the *inverted list* of t and is used to measure the relative importance of t in the stored documents. Each document is represented in these lists by a value named *document ID*, referred to as *docId* in this article.

Queries are usually processed by traversing the inverted files either in a Term-At-A-Time (TAAT) or a Document-At-A-Time (DAAT) query processing. In the TAAT strategy, the inverted lists are sorted by term impact in non-increasing order and the query results are obtained by sequentially traversing one inverted list at a time. In the DAAT strategy, the inverted lists are sorted by *docIds*, which allows the algorithms to traverse all the inverted lists related to a query in parallel.

Several authors proposed methods to discard partial scores, thus reducing the amount of memory required to process queries in the TAAT mode (Strohman and Croft 2007; Akbarinia et al. 2007; Anh et al. 2001; Anh and Moffat 2006). Anh and Moffat (2006) studied the application of dynamic pruning over inverted indexes where the entries are sorted by impact, thus adopting a TAAT approach. Strohman and Croft (2007) proposed a new method for efficient query processing when documents are stored in the main memory, which modifies the method presented by Anh and Moffat (2006). In their proposal, 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.

In the DAAT strategy, the final scores of documents can be computed while traversing the lists in parallel. Since the lists are ordered by *docIds*, it is possible to skip documents that are unlikely to be relevant, so the memory requirements are fairly smaller than in TAAT. 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 proposed algorithms and data structures to accelerate query processing in the DAAT approach. For instance, data structures to allow fast skipping in the inverted lists, named skiplists (Moffat and Zobel 1996), are adopted to accelerate 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 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. Here, we review in-depth those algorithms which are close to our research, focusing on DAAT, which is the approach adopted by our algorithm.

Broder et al. (2003) proposed a successful strategy for query processing, known as WAND, that processes the queries using the DAAT approach. In WAND, a heap of scores is created to maintain the top-k documents with larger scores at each step of the query processing. The smallest score present 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*.

WAND has a two level evaluation approach. In the first level, the documents have their *maximum possible score* evaluated using the information of *maximum score* of each list where they may occur. If the *maximum possible score* of a document is greater than the current *discarding threshold*, a second level evaluation is started, and only in these cases the actual score of a document is evaluated; otherwise, the document is discarded.

In another effort to improve query processing performance, Ding and Suel (2011) 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 skipped without any decompression of their content. Each block contains information about the maximum impact among the entries found in it, the *block max score*. BMW introduces a third level in the evaluation of documents: after checking the *maximum score* of the terms, if it is greater than the current *discarding threshold*, an additional evaluation is performed using the block max score. Whenever the maximum impact is not relevant to change the results for a given query, the whole block is discarded, which avoids important query processing costs. A document

entry is decompressed and the score is completely evaluated only when it passes the *max score* and *block max score* tests.

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 (Ding and Suel 2011).

The authors present experiments to show that BMW achieves significant reductions in query processing times compared to previous studies, comparing their method not only to document-at-a-time query processing strategies, such as WAND, but also to TAAT query processing methods, such as the one proposed by Strohman and Croft (2007). The authors concluded that their method was the fastest query processing method at the time it was proposed.

Special optimizations to the DAAT methods, including BMW, were proposed by Dimopoulos et al. (2013). Authors propose the use of a set of block-max and posting-bitmap structures that can be used to accelerate any DAAT index traversal algorithm transparently. They also propose an optimized implementation of their mechanisms that exploits SIMD instructions of modern CPUs and restricts the critical data structures to L1 cache. As our method is also a DAAT algorithm, it can also take advantage of these ideas, although the use of machines that support SIMD instructions restricts the application of the software.

In a parallel development, Chakrabarti et al. (2011) also developed document-at-a-time algorithms for computing top-k query results using block max scores to improve the pruning strategies. The authors formally study the optimality characteristics of the proposed algorithms and present a discussion about the costs related to query processing and how to set block sizes. We choose to adopt BMW as one of the baselines in our experiments, since BMW was developed considering in memory query processing and since it has a multi-tier version which results in performance gains.

Shan et al. (2012) show that the performance of BMW is degraded when static scores, such as Pagerank, are added to the ranking function. A problem that arises while combining query-independent features, such as PageRank, and query-dependent features, such as BM25 to produce the final ranking is that the maximum score of documents may become extremely high in cases where there are documents containing both a high BM25 score and a high Pagerank in the inverted lists of a term. The authors modified the BMW algorithm to deal with this scenario, selecting the pivot by using the local block max information, instead of using max score. The same adaptation of using block max instead of max score could also be applied to our algorithm, since it also uses local block max information.

Fontoura et al. (2011) studied and compared the performance of several query evaluation algorithms for searching and processing queries in memory. The experiments were conducted using two advertising datasets which are not publicly available. The datasets adopted differ from the ones typically used in large search systems, containing very small posting lists, with an average size varying from 3 entries per posting list in the smaller collection to 6 entries per list in the larger collections. As the average size of inverted lists in the queries are quite small, pruning strategies are not so effective in the experimented scenarios.

The most successful alternative studied by Fontoura et al. (2011) is *Hybrid*, a hybrid approach for query processing which combines the DAAT and TAAT approaches. The algorithm starts by first splitting the query terms into two groups: short postings list terms and long posting list terms. The sub query with small postings list is then evaluated, obtaining partial scores for all documents present. The k-th higher score of this first processing is then considered as a lower bound threshold for processing the subquery containing the terms with long postings list. In the end, the results of short and long term

subqueries are merged to produce the final result. The idea is that, as the longer lists are processed with a high threshold, the strategy may allow speedup of query processing.

Considering their application scenario, Hybrid achieved an improvement of up to 19% in performance compared to the WAND method. They have not considered block max strategies in their study, which are now state-of-the-art strategies for top-k query processing methods. We performed experiments with the Hybrid method in several scenarios, and it was outperformed in all cases by the block max based strategies studied here. Hybrid presents a serious disadvantage when compared to Waves, as in Hybrid, the partial union of the smaller lists should be stored, thus requiring extra memory for storage. Another problem is that the authors do not present a strategy for classifying query terms within the sets of smaller lists and of larger lists, leaving this issue to be studied as future work. Given these limitations, we decided to not use Hybrid as a baseline in our study.

2.1 Multi-tier query processing

Previous work proposed the splitting of the inverted index in more than one tier in an attempt to accelerate query processing. In these architectures, query processing starts in a small tier and it only proceeds to larger tiers if necessary. These tiered architectures can be considered hierarchical query processing systems where the smaller tier is on the top of the hierarchy and the larger one is on the bottom. Risvik et al. (2003) divided the index into three tiers with the goal of achieving better scaling when distributing the index. According to static and dynamic sources of relevance evidence, documents considered more relevant are selected to compose the smaller tier. The query processing starts in this tier and only if the result set is not satisfactory, according to an evaluating algorithm, does the query processing proceed to the next tier. Performance gains are achieved when the algorithm 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.

Ntoulas and Cho (2007) 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 the top results being kept unchanged, it does not access the second tier, basing its results solely on the first tier. Otherwise, the query is processed 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 while processing the first tier, assigning the maximum possible score that could be achieved while processing the full index for entries that are not present in the first tier. The authors demonstrate 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 achieve good results for first-tier index sizes varying from 10 to 20% of the full index.

Skobeltsyn et al. (2008) evaluate the impact of including a cache in the system while using the two-tier method presented by Ntoulas and Cho (2007) and show the query distribution workload is affected by the system cache. While 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 we consider a system that adopts caching of results.

The authors of BMW also presented a multi-tier version of their algorithm, called *MBMW* (Ding and Suel 2011). The idea is to split the inverted lists, putting the documents with higher scores in smaller layers and the ones with smaller scores in larger layers. The query is then processed taking all the posting lists into account and using the BMW algorithm. As the smaller posting lists contain higher scores, it is expected that they quickly increase the pruning thresholds, causing larger skips in the blocks of larger lists. As a result, the algorithm may allow a speedup in the query processing. Experiments performed by the authors indicate that *MBMW* outperformed the previously proposed query processing methods found in the literature, including the BMW using single tier. For this reason, it is adopted as one of our baselines.

Rossi et al. (2013) have also proposed a multi-tier variation of BMW, named *Block Max WAND with Candidate Selection*, or BMW-CS. It uses the first tier to accelerate the query processing in the second tier (i.e., the larger index) while computing the final ranking. As in MBMW, the two tiers are disjoint, that is, the high-impact entries in the first tier are not present in the second one. The query is processed initially using only the first tier to select candidate documents to be included in the final answer. The algorithm computes partial scores for each document entry in this first step. While selecting candidates, the algorithm estimates whether each document has chance of achieving a score to become part of the final top-k results or not. The second tier is used to complete the score of the candidate documents, thus obtaining the final ranking. A first consequence of this strategy is that BMW-CS cannot deal with more than two tiers. Further, it requires extra memory to keep candidate lists in memory. Depending on the term weight scheme adopted and on the distribution of such term weights, the number of candidates stored by the method may significantly increase the memory requirements for processing queries.

Daoud et al. (2016) proposed the method BMW-CSP, which extends BMW-CS to preserve the top-k results by adding a third phase to the method. The authors show that BMW-CSP achieves performance similar to BMW-CS without the disadvantage of removing entries from the top-k results. The method presents a performance, better than the multi-tier version of BMW with the disadvantage of requiring extra space to store the candidate documents while processing queries. We included BMW-CSP as one of the baselines in our experiments.

Daoud et al. also proposed an optimization in BMW and MBMW to accelerate query processing by adding an initial threshold for processing queries based on the maximum scores of terms found in the second tier. This optimization resulted in a small improvement in the performance of both methods. We adopted the optimized versions as the baselines in our experiments, naming them BMWT and MBMWT.

3 Waves

In this section we present Waves, our method for fast query processing. It is based on multi-tiered index organization, in which the entries of a term in tier i have greater impact than the entries of this term in all tiers $j > i$. The tiers are disjoint, meaning that the entries in a tier are not present in others. Our method processes the queries by performing multiple traversals, or waves, in the index tiers. Each wave starts from a tier i and computes the scores of documents that occur in i and only that have a chance of being present in the final result. To do so, it accesses document entries not only for tier i, but also for every tier $j > i$,

thus each wave traverses the index from the current tier to all remaining tiers. Whenever a wave starting from tier i finishes, our algorithm checks whether there is a guarantee that top-k results are already computed or not. If it is not there, the algorithm executes a new wave starting from tier $i + 1$.

At each moment while processing a query, we keep the top-k documents with the highest scores obtained till that moment. The minimum score of this set is used as a *discarding threshold* adopted while checking whether a new document has a chance of being part of the final result or not. Notice that this procedure guarantees that the top-k results will be preserved, since the algorithm discards only documents that have no chance of reaching a final score greater than the current minimum score found among the partial top-k higher results and also evaluates the full score of all documents that are not discarded. While the algorithm starts a new wave, the top-k results found by previous waves are used to set the discarding threshold and speed up the traversal by discarding more documents.

A new wave starts from tier i whenever the sum of maximum scores of the query terms in tier i is higher than the *discarding threshold*, since in such cases there might be documents that have not yet been included in the answer and that have a chance of being included in the top-k results. Our assumption is that the strategy of postponing the decision of evaluating scores for documents that do not occur in the current tier may reduce the number of evaluated entries. The tradeoff is that multiple traversals might be necessary to assure that the top-k results are correct.

While at a first glance performing multiple traversals in the index tiers seems to be expensive, our experiments show that they, in fact, may indeed reduce average query processing times, since the number of waves necessary for processing most of the queries tends to be small.

Listing 1 Waves Algorithm

```
1  Waves (Q,k, I[1..m])
2   Let Q be the set of query terms
3   Let I[1..m] be the index divided into m tiers
4   Let k be the number of elements in top answer
5   Let H be a heap to store top–k results
6   Let maxS(t,j) be the maximum possible score of term t in tier j
7   Let MAX(S) be the max value among elements of set S
8
9   H ← ∅;
10  θ ← MAX({k_th_score(t) |t ∈ Q});
11  for i = 1 to m do
12      H ← waveExecution(Q, H,i,I[1..m],θ)
13      θ ← H_1.score;
14      if((i < m) and (θ ≥ ∑_{∀t∈Q} maxS(t,i + 1)))
15          break;
16      end if
17  end for
18  sort H by score;
19  return H;
```

Listing 1 presents an overview of our algorithm. Each wave traverses the index from the current to the last tier, but only those documents that appear at least once in the current tier are evaluated. Waves starts by computing an initial discarding threshold θ, that is set as the highest value among the k-th higher score of each query term (line 10). These scores are

computed at indexing times and value zero is assigned in cases where the term contains less than k entries.

After setting the initial threshold, the algorithm executes the waves, making successive calls to function *waveExecution* (line 12). Given a tier i, function *waveExecution* checks for those documents which occur at least once in i and includes them in the current top-k answer list ($\mathcal{H}$). We detail this function in Listing 2.

After finishing a wave execution, the algorithm checks whether any document not yet taken into account could eventually be included in the answer or not. This is performed from lines 14 to 16 of the algorithm. If the sum of maximum possible scores (or max scores) of query terms in the next tier is higher than the current discarding threshold, then there is a chance of including new documents in the answer, and a new wave is started. Otherwise, the algorithm finishes and the final top-k results are sorted to produce the final answer. This checking is the key step to assure that no document with a chance of being in the top-k results will be discarded, thus also assuring that Waves preserves the top-k ranking results.

Listing 2 presents details about how we process each wave in our algorithm. While processing a wave, the algorithm keeps a pointer to the current position for each of the inverted lists of the query terms in the current tier i. The algorithm starts by moving all the lists pointers to their corresponding first document (line 11). Then it repeats the document selection step until the pointers reach the end of the lists in the current tier.

Listing 2 WaveExecution Algorithm

```
1  WaveExecution (Q, H, i, I[1..m], θ)
2   Let Q be the set of query terms
3   Let H be the heap with current top–k results
4   Let i be the current tier
5   Let I[1..m] be the multi–tier index
6   Let θ be the current discarding threshold
7   Let D(t) be the current pointed document id in tier i of index I for a given term t
8   Let maxS(t,j) be the maximum possible score of term t in tier j
9   Let maxBl(d,t,j) be the block max score of d for term t in tier j
10  Let MIN(S) be the smallest id of the given set S
11  move list pointers to the beginning of tier i ;
12  while (have documents in the tier to inspect)
13    pivot ← MIN({D(t)|t ∈ Q ∧ Σ_{∀t'∈Q|D(t')≤D(t)} maxS(t',i)+
14                          Σ_{∀t'∈Q|D(t')>D(t)} maxS(t',i+1) >θ});
15
16    estimate ← (Σ_{∀t'∈Q|D(t')≤pivot} maxBl(pivot,t',i)+
17                Σ_{∀t'∈Q|D(t')>pivot} maxS(t',i+1));
18    if (estimate > θ)
19      estimate ← (Σ_{∀t'∈Q|D(t')≤pivot} maxBl(pivot,t',i)+
20                  Σ_{∀t'∈Q|D(t')>pivot} maxBl(pivot,t',i+1));
21      if (estimate > θ)
22          score ← compute the score of pivot taking their frequencies from the index;
23          if( score > θ)
24              update H with the pivot and its score;
25              θ ← H_1.score; // Smallest score in H
26          endif
27       endif
28    endif
29    move lists pointers forward;
30  end while
31  return H;
```

The selection of documents is performed using a multiple evaluation step. First, we use only the maximum possible score (max score) of documents in the current tier and in the

next tier to select a pivot document, the candidate needs to be included in the top-k results. We select as a pivot document, the candidate with smallest id currently pointed to in the lists whose sum of its max scores is higher than the discarding threshold θ (lines 13 and 14). The max score of a document d is computed by adding its max scores in all query terms. For the terms whose currently pointed document is smaller or equal to d, there is a chance of finding d in the entries of such terms for the current tier i, and thus the max score is taken from tier i. For the remaining terms, we already know that d does not occur in the current tier i, and thus we take the max score from tier $i + 1$. Given an index with m tiers, we assign max score values as zero after tier m.

For the pivot candidate selected in the first evaluation step, we compute an estimated score (line 16), but now by taking the block max score of the pivot while getting information from tier i and taking the max score while getting information from tier $i + 1$ (lines 16 to 18). As in BMW method, we store, for each block of documents, the maximum possible score of any entry in the block, or block max score. It is a tighter estimation of the maximum score that a document may have if it occurs in the list. We also store the block boundaries, which indicate the interval of document ids that may occur in the block. To read the block max information for a document d in a term, we search for the block that has boundaries containing d. Again, for the terms that do not contain the pivot in tier i, we take the max score from tier $i + 1$.

After the second evaluation step, we compute a tighter estimated score by taking the block max information for the pivot candidate in all terms (lines 19 to 21). For the terms that do not contain the pivot in tier i, we take the block max from tier $i + 1$.

If a candidate is approved in all the previous evaluation steps, it is considered a potential candidate to be included in the answer and we perform a final evaluation by computing its full score. We show a simplified version of this score computation. The best way to score is to iteratively remove the current upper bound and add the true score contribution per term. Once the score drops below theta, scoring can stop. This optimization is included in the source code we make available for allowing the reproduction of results.

The frequency information is taken from the current tier i if pivot occurs in it. For the terms where it does not occur in i, we access the following tiers until we find a block where the pivot occurs or until last tier is inspected. Note that, in the worst case we may access one block per tier to read a frequency of a document in a given a term. After evaluating the score of the candidate pivot, the algorithm updates the heap of results in case it surpasses the document with the smallest score among the top-k currently computed answers (lines 23 to 26).

Finally, the algorithm moves forward the pointers of all inverted lists (line 29) and repeats the multilevel evaluation process for another set of documents. The move forward procedure moves the pointers of all the lists to at least the next entry after the pivot. Block boundary information can be used to make further aggressive movements in special cases, using ideas similar to the ones detailed in Ding and Suel (2011).

When the pivot candidate is fully evaluated, the pointers are just moved to the next entry higher than the pivot in each list. For the remaining situations, the pointers are moved to the next entry d, such that

$$d \leftarrow MIN(\{Boundary(pivot, t, i) | D(t) \leq pivot \wedge t \in Q\} \cup \{D(t) | D(t) \geq pivot \wedge t \in Q\}) \quad (1)$$

where $Boundary(x, t, i)$ is a function that gets the id of the first document after the block where document x occur in tier i of term t and MIN, Q, D(t) and pivot are the defined as in Listing 2.

To further illustrate the wave execution algorithm, we give an example in Fig. 1, where the index is divided into only two tiers. Lets consider the current wave is processing tier 1, and the current documents in the lists are 69, 466 and 900, respectively for the query terms 'Very', 'Large' and 'Data'. In this example, document 69 occurs in the list of the term 'Very' in the first tier. To check whether it has potential to become part of the top-k results, we would add its max score, 1.8 to 1.7. The value 1.7 comes from the sum of $1.1 + 0.6$, and represents the maximum score it would receive if occurring in the second tier in lists of terms 'Large' and ' Data'. Notice that as the lists of 'Large' and 'Data' are pointing to document ids higher than 69, we take the max score from the second tier, since this means document 69 does not occur in tier 1 for those lists.

The final score obtained would be 3.5, which is the maximum potential score for document 69 in the collection. If this score is higher than the discarding threshold θ, Waves accesses the first tier to get the block max score value associated to document 69, using the information to compute a tighter upper bound score. The second evaluation level will thus result in $0.5 + 1.1 + 0.6 = 2.2$. If this tighter upper bound also surpasses the threshold θ, Waves computes a third estimation of score for document 69, taking its block max scores for terms 'Large' and 'Data' in the second tier, and getting an even tighter estimation. If this third estimation is again higher than θ, its full score is evaluated.

As mentioned in Listing 1, while processing a tier i, documents that occur only in the following tiers of the lists are not taken into account, but in some cases they might be a part of the top-k results. This situation occurs when the sum of max scores of the query terms in tier $i + 1$ is higher than the discarding threshold after processing the query in tier i.

In the example of Fig. 1, the sum of max scores of the query terms in tier 2 is 2.1. If the discarding threshold θ after processing tier 1 were smaller than 2.1, the result could be incorrect and new documents could be inserted in the top-k results. In cases such as these, the next wave is started, traversing the following tier to ensure the final top-k results are correct. Since the second tier is the last tier in the example, after processing it the top-k results will be properly computed. With this reinforcement step, documents that occur only in second tier, such as document 5 in Fig. 1, are evaluated, if necessary, to check whether they should be included in the top-k results or not.

Thus, after finishing each wave, a new wave may be started to guarantee that the top-k results of the ranking are preserved. To avoid repetition of results, each new wave updates the top-k entries with documents whose entries are not considered in the current top-k answers. Since the results computed in the previous waves are used in the current one, and given that the scores in the currently processed tier are usually lower than the ones

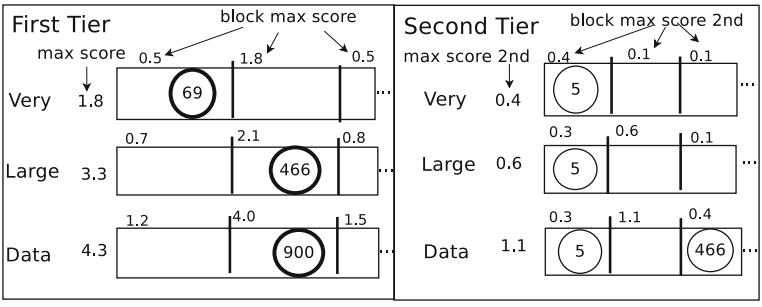

Fig. 1 Inverted lists when processing query 'Very Large Data' and using lists divided into two tiers

found in previous tiers, performing new waves does not represent a prohibitive time overhead.

4 Experimental evaluation

We used the TREC GOV2 collection to compare the performance of our method with previous proposals presented in the literature. 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 postscript documents. The total number of index entries is about 6 billion and the vocabulary has about 45 million distinct terms. Our indexes use the frequency of the terms to compute the score of documents while processing queries.

We selected two subsets of the TREC 2006 efficiency track for the experiments. One containing 1000 queries is used in initial experiments performed to set the best tier division for the multi-tier methods evaluated, and will be referred to as *small query set*. The second set contains 10,000 queries and is used in final experiments to compare the performance of the methods with the best parameters chosen, and is referred to as *large query set*. These setup options were chosen because they are similar to those adopted in previous studies (Rossi et al. 2013; Ding and Suel 2011; Strohman and Croft 2007), including the baselines we adopt here. We ran the experiments in a Intel(R) Xeon(R), with X5680 Processor, 3.33GHz and 64 GB of memory. All the queries were processed in a single core and the term lists were loaded into the main memory. The time for processing queries was recorded using wall clock time and does not include time for loading the inverted lists from the disk.

We used Okapi BM25 (Robertson and Walker 1994) as the score function, which is an extremely popular ranking function adopted in search systems, as well as it is the one adopted in most of the previous work on computing top-k results in search systems, but our method can be adopted to compute other ranking functions as well. The generated skiplists have one entry for each block of 128 documents, as was done in previous work (Ding and Suel 2011; Rossi et al. 2013). Each skiplist entry keeps the first *docId* in the block, and the *maximum block score*, or *block max score*, registered in the block.

We performed experiments with the index without any compression. Several compression methods could have been adopted, such as pforDelta (Zukowski et al. 2006) or the more sophisticated Elias–Fano indexes (Ottaviano and Venturini 2014). Notice however that compression methods only affect the time to access blocks in the methods, making them slightly slower. Thus the impact of using compression techniques can be inferred by comparing the number of block accesses made by each method. This parameter is studied in the experiments, where we compare not only the time performance of the methods, but also the number of block accesses necessary to process queries. If considering fast decoding methods, such as pforDelta, the results almost do not chance, since they cause just an small increase in the time to access blocks. We have performed experiments with pforDelta and the performance differences between the methods did not change, since the overall time execution for the methods also did not change much. Time for processing queries with Waves in the experiments increased less than 2% in the experiments. As the impact of compression may vary according to the method adopted, we however decided to not report experiments with compression in the article.

To select the entries for each tier, we adopted the same global selection strategy proposed in Carmel et al. (2001) and adopted in several other researchers in the literature (Moura et al. 2008; Rossi et al. 2013; Daoud et al. 2016). We compute global *splitting thresholds* to select entries for each tier, adjusting the thresholds to the desired tier sizes. For each term, we sort its inverted list entries in decreasing order of BM25 score. The BM25 score of each inverted list entry of a term is computed while processing a query containing only the term.

For the first tier, we select entries with scores above its splitting threshold, adopting a minimum size of 1000 entries in each inverted list to prevent any individual list from becoming too small. Thus, lists with sizes smaller than 1000 will be all fully stored in the first tier. The remaining tiers have their entries selected according to their respective splitting thresholds, each tier getting the entries higher than the threshold among the ones not taken in the previous tiers. We have also considered the use of a local strategy, where we compute local *splitting thresholds* for each inverted list to select entries for each tier, but this strategy resulted in poor performance when compared to the global *splitting thresholds* selection.

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. The scenario retrieving top-1000 results was included given the common use of the top results as input to more sophisticated ranking methods, such as machine learning techniques.

We evaluated the methods in terms of query response time and the amount of processed entries from the inverted lists, computing only the entries that were used to compute the score of documents.

We implemented the methods BMWT, MBMWT (Ding and Suel 2011; Daoud et al. 2016), and BMW-CSP (Daoud et al. 2016). To avoid a bias due to optimization of any version, all the codes were developed using the same basic BMW code, which was provided by one of the authors of BMW and MBMW (Ding and Suel 2011). BMW-CSP is the same code adopted in the original article. The main step of Waves is very similar to BMW, but includes the control to determine whether new waves should be started or not, thus we created the Waves source code by using the BMW code as the starting point, without any special optimization. The access to the inverted lists was completely implemented by us, and was similar in all experimented methods. Finally, we notice that the final performance is compatible with the ones reported by the authors of the methods.

These baselines represent some of the most recent and most successful, document-at-a-time methods found in the literature and the comparison to them allows an indirect comparison to several other methods proposed for computing top-k results also. MBMW (Ding and Suel 2011) was chosen as baseline because their authors show that it outperforms the main baseline alternatives we found in the literature, including the most efficient TAAT and DAAT algorithms. We also performed experiments with Hybrid, another in-memory method proposed recently that has not previously been compared to BMW. Hybrid requires extra space for storing partial results (accumulators) and achieved fair worse performance in our experiments. Therefore, we do not report its results, since they would add noise to the comparison with the best baselines we found. All the baselines and source codes adopted are available online in a git repository (https://github.com/CaioDaoud/Waves)[1] to allow reproduction of results and future comparison.

[1] The function for Waves will be available only in case acceptance, since the repository is public and can be reached by other people, but still the other codes maybe useful for the reviewers in case of doubts about the implementation of the methods.

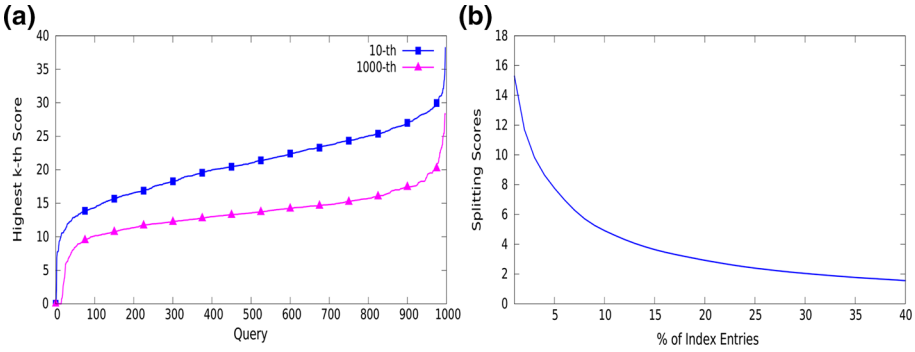

Fig. 2 Values of the highest k-th result for all queries found in the smaller query set and the variation in the split threshold for distinct tier sizes

Our main goals with the experiments are: (1) check whether the tier size parameters selected for the smaller query set produces competitive results when processing the large query set. The selection of tier sizes and number of tiers is an important step while using multi-tier indexes, since these parameters may affect the final performance of the methods. (2) Verify whether the Waves strategy is effective. While considering the cost of each individual traversal in the lists performed by Waves, the reader may have the misguided impression that its cost is higher than that of baselines, which perform a single traversal in the posting lists. We want to show that, in practice, each wave traversal amortizes the costs for the next wave, since it increases the discarding threshold, reducing the number of elements checked in the following waves. Further, the cost of each wave itself is amortized by the restriction of evaluating only those documents which occur in the current tier processed by the wave and that have potential to surpass the discarding threshold. The algorithm may even discard waves, especially while using three or more tiers in the index, which may further reduce its costs.

4.1 Results

An important question while using multi-tier indexes is deciding the best index tier sizes for each collection. We first study the expected relation between k-th document score of each query and the possible *splitting threshold* values adopted to divide the collection into tiers. In Fig. 2a, we present the highest k-th score among the query terms found in the smaller query set for GOV2 collection, showing the values for top-10 and top-1000 results. These values can be used as lower bound estimations to the k-th score of each query. The actual values may be higher in all cases. However, this estimation is useful to give us an insight about why the Waves basic strategy works.

We can see in Fig. 2a [no (a) or (b) in the image] that the scores are higher than 10 in most of the queries in both cases, 10th and 1000th higher scores. When we compare these scores to the splitting thresholds results presented in Fig. 2b, we see that the global splitting thresholds obtained while dividing the index above 10% become significantly smaller than the k-th values for most queries. For instance, with a threshold that separates 20% of entries, the splitting threshold is about 2, while the k-th result is above 10 for most queries. That means the maximum score of any query terms on any tier above 20% would be at most 20% of the k-th score for most queries, thus the necessity to start a new wave at that point is a rare situation. We remind the reader that the sum of the maximum scores in

the tier for all query terms is used as the threshold to decide whether a new wave should start at that tier or not. If the k-th higher score is greater than such threshold, the query processing stops.

4.1.1 Adjusting parameters with the smaller query set

The best choice of tier size depends on several factors, including the size of top-k results that need to be computed and the algorithm chosen for processing the queries. Consequently, we performed experiments to choose the best distribution of tier sizes for Waves and MBMWT by using the smaller query set.

Results for the division into two tiers are presented in Fig. 3, that shows a comparison of times achieved by Waves and MBMWT. As we can see, while computing the top-10 results, the best performance of Waves and BMW-CSP, was achieved with first tier size of 25%, while MBMWT achieved its better results with first tier size of 5%. For the top-1000 results, the best result for Waves and BMW-CSP was achieved with 40%. MBMWT achieved its better results for top-1000 with first tier size of 10%.

We have also divided the index into three tiers for methods Waves and MBMWT, varying the size of the three tiers to find the best result for each algorithm. Results are shown in Figs. 4 and 5. When looked at the results achieved by MBMWT for three tiers, our conclusion is that, dividing the index into more tiers is not very advantageous while using MBMWT. While computing top-10 results, it resulted in no gain when compared to the best option with 2 tiers. While computing top-1000 results, the gain was modest. Authors of MBMWT already described this phenomenon, and it is a consequence of the strategy adopted by the method. It divides the lists into tiers as they were distinct terms. As a consequence, the number of terms processed increases linearly with the number of tiers adopted, which increases the cost for selecting pivots at each step.

As demonstrated in Fig. 5, this phenomenon does not occur while using Waves, and we were able to achieve further improvements in query processing times when we use Waves with three tiers. Times were reduced for both of the experimented scenarios. For instance, times achieved by Waves dropped from 16.55 to 13.27 ms per query while computing top-10 results and from 61.50 to 53.71 ms while computing top-1000 results.

Table 1 presents a summary with the best performance parameters we achieved for the smaller query set. With this summary we can see that Waves performed better both while computing top-10 and top-1000 results. For MBMWT, we divided the index into 2 tiers,

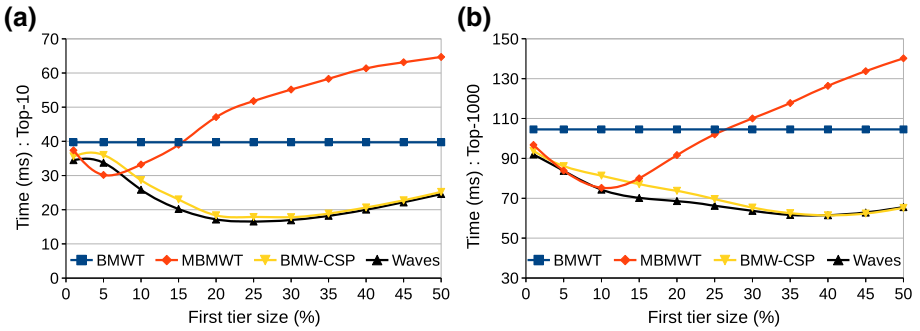

Fig. 3 Average time achieved by Waves, BMW-CSP and MBMWT when dividing the index into two tiers and varying the size of first tier for computing the top-10 results (**a**) and the top-1000 results (**b**) for collection GOV2

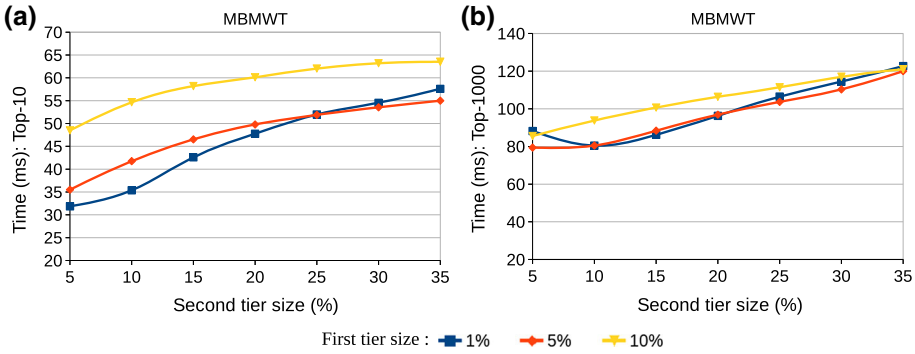

Fig. 4 Average time achieved by MBMWT when dividing the index into three tiers and varying the size of first and second tier for computing the top-10 results (**a**) and the top-1000 results (**b**)

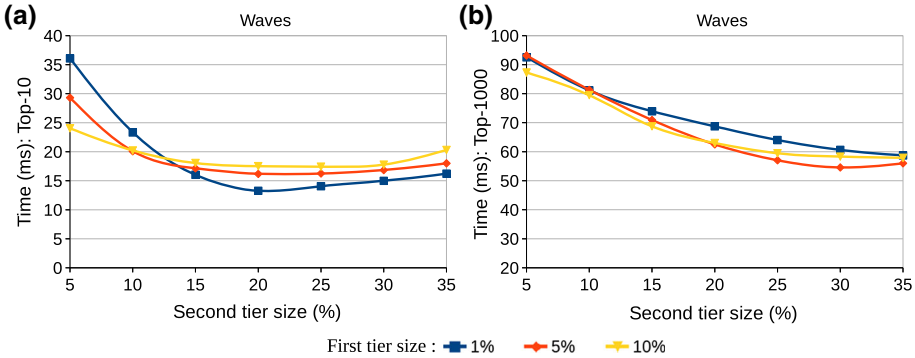

Fig. 5 Average time achieved by Waves when dividing the index into three tiers and varying the size of the first and second tier for computing the top-10 results (**a**) and the top-1000 results (**b**)

using a first tier of 5% while computing top-10 results and 10% when we compute top-1000 results. For BMW-CSP, we divided the index into 2 tiers, using a first tier of 30% while computing top-10 results and 40% for top-1000 results. For Waves, we divided the index into 3 tiers, using sizes 1%, 20% and 79% while computing top-10 results, and sizes 5%, 30% and 65% for computing top-1000 results.

The table compares the time performance and number of basic operations performed by all the baselines. We can see that Waves achieved considerable improvements in time performance both while computing top-10 and top-1000 results for the smaller query set with the best parameters. Compared to MBMWT, it achieved a reduction of 54% in time while computing top-10 results (from 29.03 to 13.27 ms). While computing top-1000 results, the reduction was close to 25% (from 71.96 to 53.71 ms). We can also see that this reduction reflects the reduction in the number of basic operations performed by the methods.

Again while comparing Waves and MBMWT, we find that Waves was able to reduce the number of selected pivots by about 42% and the number of accessed blocks by about 52% in the top-10 results. For the top-1000, there is also a reduction both in the number of processed blocks, 16% of reduction, and in the number of selected pivots, about 41% of

Table 1 Average number of processed blocks, average number of evaluated pivots and average time performance when using Wand, BMW, MBMWT and Waves in their best configuration obtained when processing the smaller query set

	#Blocks	#Pivots	Time (ms)	Size of each tier (%)
Top-10				
BMWT	34,712	544,980	40.05 ± 1.90	100
MBMWT	22,931	368,601	29.03 ± 1.71	5–95
BMW-CSP	11,758	245,260	16.20 ± 1.17	30–70
Waves	12,306	178,027	13.27 ± 0.73	1–20–79
Top-1000				
BMWT	74,503	1,472,891	95.04 ± 3.61	100
MBMWT	54,575	935,599	71.96 ± 3.15	10–90
BMW-CSP	43,337	701,898	57.52 ± 2.81	40–70
Waves	46,076	542,064	53.71 ± 1.86	5–30–65

We adopted the best parameter found for each method in the remaining experiments of the article

reduction. When compared to BMW-CSP, Waves achieved a reduction from 16.20 to 13.27 ms when computing the top-10 results. We notice that the difference is larger than the confidence intervals and represents a reduction of about 18% in the time for processing queries. When computing the top-1000 results, times were reduced from 57.52 to 53.71, again with a difference greater than the confidence interval. This number represents a reduction of 7% in the time for processing queries.

4.1.2 Results with the larger query set

After checking the best parameters for each method in the smaller query set, we also performed experiments to compare the methods' performance in a larger query set. With this comparison, we verify whether the good performance found in the smaller query set, which was adopted for studying parameters, is repeated in the larger one. In these experiments we adopted the best configuration achieved for each method in the experiments with the smaller query set.

A question that would arise at this point is whether the parameters selected for the smaller query set are also the best for the larger query set. We subsequently varied the parameters in the larger query set, and realized that all the choices are either optimal or very close to optimal parameters both for Waves and for the baselines. This indicates that parameters might be tuned by using sample queries, as we did here. The experiments varying the parameters with the larger query set are not depicted to avoid showing redundant information in the paper.

Table 2 presents a summary of the performance of the compared methods in terms of time, number of processed blocks and number of selected pivots. We can see that, while there is a natural variation in the average number of operations and time for processing each query, the improvements in performance achieved by Waves when compared to the baselines is close to the ones achieved with the smaller query set. For instance, compared to MBMWT, Waves again achieved a reduction of 54% in time while computing top-10 results (from 35.27 to 16.08 ms) when we process the large query set. While looking to the top-1000 results, the gain in the large set was 31% (from 82.66 to 57.40 ms). Comparing

Table 2 Average number of evaluated pivots (entries where the actual frequency was accessed) and average time(ms) while computing top-10 and top-1000 results and processing the large query set

	#Blocks	#Pivots	Time (ms)
Top-10			
BMWT	38,455	613,341	46.21 ± 2.09
MBMWT	25,958	422,986	35.27 ± 1.91
BMW-CSP	11,028	313,153	19.64 ± 1.35
Waves	14,577	176,709	16.08 ± 1.03
Top-1000			
BMWT	79,329	1,580,775	108.28 ± 3.69
MBMWT	58,053	1,021,467	82.66 ± 3.14
BMW-CSP	37,241	910,131	66.35 ± 2.92
Waves	49,363	579,925	57.40 ± 2.55

Confidence interval for times were computed at confidence level 95%

Waves and BMW-CSP, the gain in time was close to 18% for top-10 scenario and close to 13% for the top-1000 scenario.

Figure 6 presents an overview about the time performance of the methods Waves and BMW-CSP while processing all the queries in the dataset. The queries are sorted by the time required to process them while using Waves, as this order allows a clear distinction between Waves performance and the performance of BMW-CSP. Points above the line indicate queries where Waves was better than the baseline. Results achieved while processing top-10 are presented in Fig. 6a and top-1000 are presented in Fig. 6b. To allow a better view of the points in the graphics, we present separate graphs for queries that take less than 0.5 s, Fig. 6a, b, and for queries that take more than 0.5 s, Fig. 6c, d. Queries that

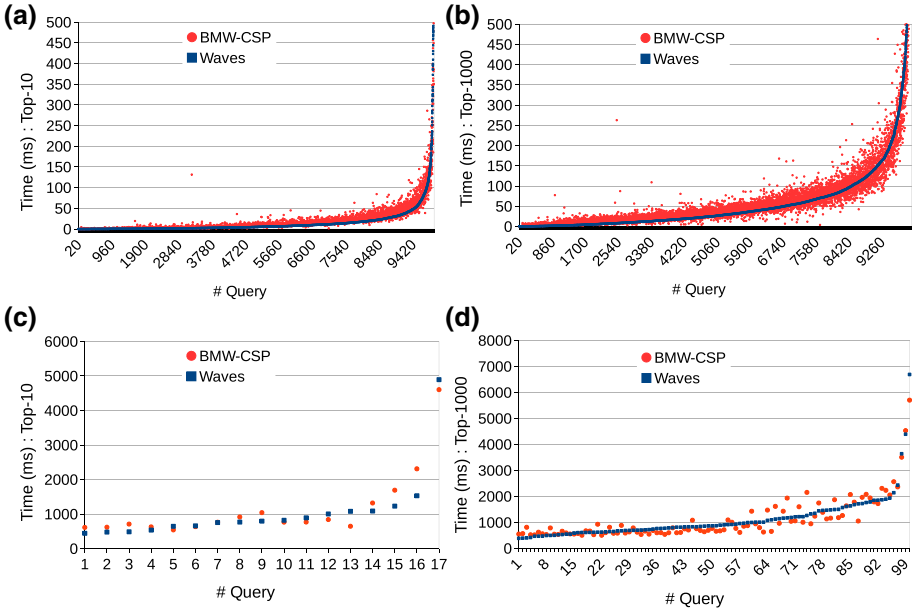

Fig. 6 Time achieved by Waves and by the BMW-CSP for each query while computing the top-10 results (**a**) and the top-1000 results (**b**). For visualization purposes, more expensive queries that take more than 0.5 s are presented in separate, for top-10 (**c**) with 16 queries and for top-1000 (**d**) with 99 queries

take more than 0.5 s represent a small set in both scenarios, which constitutes exactly 16 queries(0.16%) in the top-10 scenario, and 99 queries (0.99%) in the top-1000 scenario. The Figures are useful to show that Waves was clearly faster than BMW-CSP while processing most of the queries for top-10. When looking at the top-1000 scenario, still Waves is faster for most of the queries, although the difference between the methods is smaller.

Figure 7 presents a similar comparison, but now comparing Waves to MBMWT. MBMWT is slower than Waves in both top-10 and top-1000 scenarios. There is also a greater difference between the times achieved by MBMWT than by Waves, so in this graph we see scattered points, making it easy to realize that Waves is faster for most queries.

Table 3 describes the number of queries that Waves is slower, faster or has almost the same time than the baselines MBMWT and BMW-CSP. We consider a tie when the difference is less than 0.01 ms. While computing Top-10, Waves was faster than the BMW-CSP in about 91% of queries and is faster than MBMWT in about 98% of queries. For top-1000, Waves was faster than BMW-CSP in 85.2% of the queries and is faster than MBMWT in about 91.0% of the queries.

Table 4 depicts the performance of the methods while varying the number of terms per query. We can see that the comparative advantage of Waves reduces as the query sizes increase when compared to the baselines. As we increase the number of terms in a query, the expected number of waves required to process the query also increases, as the chance of finding a document in one of the tiers for a query term, and not finding the same document in the same tier for other query terms, increases. This phenomenon is illustrated in Fig. 5, where we can see the evolution of the number of required waves while varying

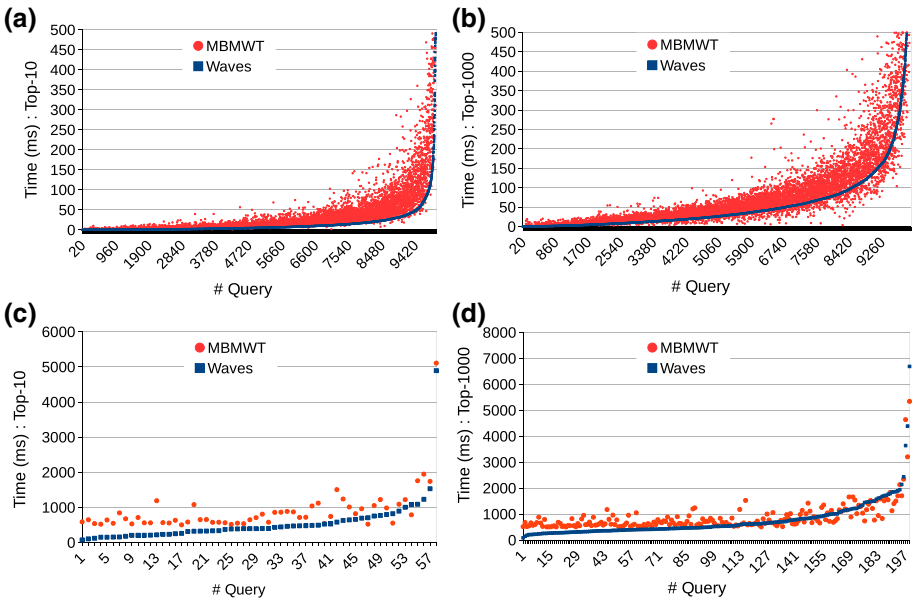

Fig. 7 Time achieved by Waves and by the MBMWT for each Query while computing the top-10 results (**a**) and the top-1000 results (**b**). For visualization purposes, more expensive queries that take more than 0.5 s are presented in separate, for top-10 (**c**) with 57 queries and for top-1000 (**d**) with 198 queries

Table 3 Number of queries that Waves is faster or slower than the baselines MBMWT and BMW-CSP

#Baseline	Top 10	Top 1000
Waves is faster		
BMW-CSP	9088	8521
MBMWT	9747	9101
Waves is slower		
BMW-CSP	749	1403
MBMWT	159	827
Tie (difference <0.01 ms)		
BMW-CSP	163	76
MBMWT	94	72

Almost same: when the difference is less than 0.01 ms

the query size. We can see that query with more terms are likely to require more waves to be processed, which affects the performance of Waves while processing longer queries.

Anyway, we also see improvements in performance for all query sizes while comparing Waves and MBMWT. The large difference in times while comparing MBMWT and BMW-CSP to Waves for 1 term queries can be explained by the selection of best parameters for tier sizes in each method. The first tier in Waves was fairly smaller than the first tier in BMW-CSP and MBMWT in their best chosen parameter. Thus the expected time for processing queries with Waves became smaller for 1 term queries. We stress that 1 term queries represent a tiny portion of the total queries in this dataset, which explains why such huge difference in times did not affect the final average query processing times. While still the Waves is better than the baselines for all query sizes experimented, the results presented in Table 4 indicate that the gains achieved by Waves are smaller for larger queries.

Another question that could arise is how long is the time obtained by Waves while processing queries that require more waves to be processed. To answer this question, we grouped the queries according to the number of waves required by our method with its best configuration and compared its performance to the ones achieved by the baselines. Results are depicted in Table 6. We can see that few queries required three waves to be processed, namely 28 while computing top-10, and 21 while computing top-1000. Those queries represent a hard case for waves, but the extra cost for processing them was not prohibitive when compared to the ones achieved by the baselines. For instance, BMW-CSP was the best method when computing the 28 queries that require three waves in the top-10 scenario. Its time was however quite close to the ones achieved by Waves, being 823.79 ms, while Waves required 827.64 ms for processing these queries. For the top-1000 scenario, there were 21 queries that required three waves. In this case, BMWT and MBMWT were better than Waves (Table 5).

A comment on the queries that go to the third wave is that they are all quite long, varying from 6 to 18 terms and the average time shown in Table 6 is dominated by the longer queries. Further, as we see in the experiments, they are rare, with less than 0.3% of the queries requiring the third wave in the experiments. We notice that even in these queries, which represent the worse situation for Waves, the algorithm still achieved competitive time performance when compared to all considered baselines, but it was not the best option for such long queries.

Table 4 Average time when processing queries with distinct sizes and computing top-10 and top-1000 results

Algorithm	Time (ms)					
	1	2	3	4	5	+5
#Queries	195	1484	2498	2388	1599	1836
Top-10						
BMWT	0.41 ± 0.16	3.35 ± 0.29	13.77 ± 0.71	28.61 ± 1.22	51.42 ± 2.43	149.15 ± 9.62
MBMWT	0.37 ± 0.09	2.44 ± 0.21	10.58 ± 0.64	21.30 ± 1.03	37.68 ± 2.14	115.81 ± 9.22
BMW-CSP	0.19 ± 0.06	1.80 ± 0.14	6.53 ± 0.30	12.56 ± 0.50	20.58 ± 0.91	62.65 ± 6.93
Waves	0.08 ± 0.01	1.02 ± 0.09	4.21 ± 0.21	8.85 ± 0.39	15.31 ± 0.65	56.00 ± 5.80
Top-1000						
BMWT	1.64 ± 0.31	9.51 ± 0.62	37.00 ± 1.46	72.59 ± 2.34	124.87 ± 4.49	330.52 ± 15.34
MBMWT	1.52 ± 0.41	7.50 ± 0.49	28.24 ± 1.22	55.36 ± 1.89	94.32 ± 3.66	252.94 ± 13.85
BMW-CSP	1.38 ± 0.27	9.43 ± 0.54	25.77 ± 0.79	44.83 ± 1.26	70.10 ± 2.21	200.59 ± 14.01
Waves	0.42 ± 0.10	4.73 ± 0.31	18.56 ± 0.53	36.23 ± 0.98	61.47 ± 1.93	182.89 ± 13.98

Confidence interval for times were computed at confidence level 95%

Table 5 Distribution of queries according to the number of required waves when processing queries with distinct sizes and computing top-10 and top-1000 results

# of required waves	# of queries					
	1	2	3	4	5	+5
Top-10						
1	195 (100%)	1173 (79%)	890 (36%)	399 (17%)	128 (8%)	42 (2%)
2	0 (0%)	311 (21%)	1608 (64%)	1989 (83%)	1471 (91%)	1766 (96%)
3	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	28 (2%)
Top-1000						
1	195 (100%)	875 (58%)	377 (15%)	133 (5%)	27 (2%)	3 (0%)
2	0 (0%)	609 (42%)	2121 (85%)	2255 (95%)	1572 (98%)	1812 (98%)
3	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	21 (2%)

Table 6 Time achieved by the experimented methods when processing queries grouped according to the number of waves required by the method Waves with 3 tiers to process them

#Waves	Time (ms)		
	1	2	3
Top-10			
#Queries	2827	7145	28
BMWT	7.34 ± 0.95	57.59 ± 2.04	1125.03 ± 317,87
MBMWT	6.03 ± 0.90	42.95 ± 1.72	1063.86 ± 340,09
BMW-CSP	3.09 ± 0.35	23.11 ± 0.77	823.79 ± 329,60
Waves	1.59 ± 0.18	18.58 ± 0.80	827.64 ± 321,93
Top-1000			
#Queries	1610	8369	21
BMWT	12.15 ± 0.95	123.02 ± 3.31	1788.44 ± 211.02
MBMWT	9.57 ± 0.75	92.99 ± 2.57	1688.39 ± 221.60
BMW-CSP	8.19 ± 0.56	72.88 ± 1.77	2042.86 ± 242.66
Waves	4.40 ± 0.36	62.99 ± 1.94	1892.60 ± 248.20

4.1.3 Impact of the collection size

Table 7 presents the impact of varying the collection sizes while processing queries with the experimented methods. The goal of this experiment was to check whether variations in the collection size affect the differences in performance between the methods. For this experiment, we have selected chunks of size 100K, 1M, 10M and the full collection GOV2. The results indicate that the differences between Waves and BMW-CSP increase with the collection size, specially for the top-1000 scenario. For the smaller collections, the cost for maintaining the candidates in BMW-CSP is reduced, since the size of the initial tiers becomes smaller. Further, for any given k (being 10 or 1000) the proportion between top-k results and the total list sizes is higher for the smaller collections and we see from the other experiments that as the size of k gets larger, the times achieved by the two methods gets closer. The results indicate that the relative difference in performance between the methods Waves and BMW-CSP tend to increase for larger collections and become smaller for small collections. While comparing Waves to other baselines, we see that the

Table 7 Average time(ms) while computing top-10 and top-1000 results for subsets of GOV2

#Terms	# Gov100k 1,539,109	Gov1m 6,113,818	Gov10m 26,329,422	GovFull 45,664,906
Top-10				
BMWT	0.88 ± 0.03	3.83 ± 0.14	22.30 ± 0.91	46.21 ± 2.09
MBMWT	0.77 ± 0.02	3.06 ± 0.12	16.68 ± 0.83	35.27 ± 1.91
BMW-CSP	0.30 ± 0.01	1.63 ± 0.09	9.34 ± 0.62	19.64 ± 1.35
Waves	0.29 ± 0.01	1.51 ± 0.09	8.31 ± 0.55	16.08 ± 1.03
Top-1000				
BMWT	1.87 ± 0.04	8.58 ± 0.24	52.14 ± 1.65	108.28 ± 3.69
MBMWT	1.71 ± 0.04	7.18 ± 0.21	39.82 ± 1.42	82.66 ± 3.14
BMW-CSP	1.23 ± 0.04	6.06 ± 0.20	32.42 ± 1.25	66.35 ± 2.92
Waves	1.21 ± 0.04	5.85 ± 0.20	31.31 ± 1.20	57.40 ± 2.55

Confidence interval for times were computed at confidence level 95%

differences are slightly larger while query processing is done in the smaller portion of GOV2 collection. It decreases up to the collection of 1M and then stabilizes.

4.1.4 Applying document reassignment techniques

An important strategy to further optimize query processing methods is to apply document id reassignment techniques, which are usually clustering strategies in an attempt to assign similar ids to similar documents. While most document id reassignment techniques are based on computationally expensive clustering strategies, a simple and low cost technique was proposed by Silvestri (2007) for the specific case of web collections. It sorts the documents by the lexicographical order of their URLs and uses the obtained ordering to reassign document ids.

Performing experiments with document id reassignment optimization is important to avoid doubts about the possible differences in performance between BMW and the multi-tier methods studied by us, including the Waves method proposed here. A reader might have doubts about the possibility of improvements achieved by the multi-tier methods just because of the fact that they modify the ordering of documents while dividing the lists into distinct tiers. In this case, the application of document reassignment techniques could give similar improvements to the ones achieved by methods such as MBMW or Waves when applied to BMW.

To avoid such doubt, we applied the Silvestri's document reassignment technique to GOV2 collection. The time results are presented in Fig. 8. In our experiments with document id reassignment, the times achieved by BMW dropped from 46.21 to 16.55 ms while computing top-10 results, and from 95.04 to 44.58 ms while computing top-1000 results. This first result confirms the importance of document id reassignment as a technique to reduce query processing times.

For the Waves method, the time dropped from 16.08 to 9.24 ms and from 53.71 to 31.57 ms while computing top-10 and top-1000 results respectively. When compared to the time achieved by BMW, the Waves method reduced the query processing times by about 44.2 and 29.2% while computing top-10 and top-1000 results using the document reassignment technique adopted in the experiments. The results show that, while

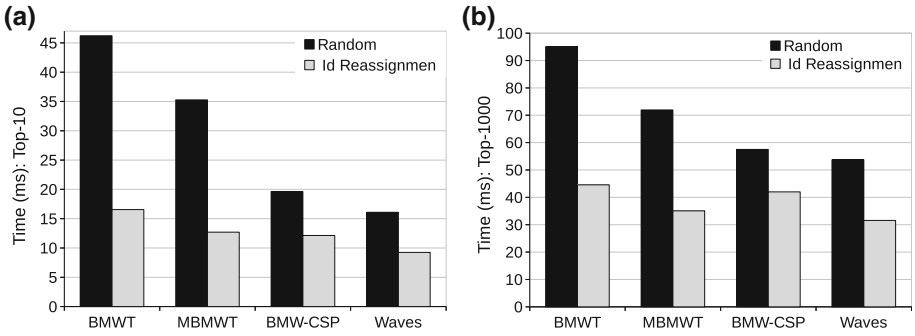

Fig. 8 Average times achieved by BMWT, MBMWT, BMW-CSP and Waves while using a document id reassignment technique to index the GOV2 collection (Id Reassignment), compared to the times achieved while using a random document id assignment (Random Assignment)

reassignment reduces the differences in performance between the methods, applying Waves produces further gains in performance even in this scenario.

4.2 Experiments with CLUEWEB09

A further question that would appear is whether the differences in performance between the methods would also occur while performing experiments in another collection, specially for bigger document collections. In this section, we present experiments with a subset of CLUEWEB09 collection, the Category B subset (English 1), which consists of about 50 million pages of English Web. We have processed the first 1000 queries of TREC 2009 Million Query 40k (MQ09#40K) and adopted the same optimal distributions of tier sizes for each method chosen with GOV2 small query set to separate the index into tiers. The collection was sorted by URLs to determine the document ids. This experiment is useful to show the performance of the methods in a second collection. The results are presented in Table 8. The differences in performance while comparing the methods is roughly the same that is obtained while performing experiments with GOV2 collection.

5 Conclusion and future work

The experimental results indicate that Waves is quite a competitive algorithm for in-memory query processing when compared to the previous state-of-the-art document-at-a-time query processing methods. We achieved considerable gains in time performance, reducing the time to compute top-10 results to about half the time attained by MBMWT and achieving a reduction of 31% while computing top-1000 results in the experiments with GOV2 using the larger query set. These two scenarios are quite common in practical search applications. When compared to MBMWT, Waves better explored the possibility of dividing the index into tiers. The lazy strategy of trying to compose the answer by taking only the results of the top tiers into account first, and taking other possibilities only when this strategy fails, has resulted in significant gains in performance.

When compared to BMW-CSP, Waves was superior while processing queries in the full GOV2 collection and achieved quite similar performance while processing queries in smaller subsets of GOV2. It is important to notice however that BMW-CSP requires the

Table 8 Average number of evaluated pivots (entries where the actual frequency was accessed) and average time(ms) while computing top-10 and top-1000 results and processing queries of CLUEWEB09

	#Blocks	#Pivots	Time (ms)
Top-10			
BMWT	15,482	181,891	13.06 ± 2.41
MBMWT	13,719	122,501	10.97 ± 1.92
BMW-CSP	5715	95,688	7.85 ± 0.75
Waves	3779	76,468	4.27 ± 0.59
Top-1000			
BMWT	42,994	683,119	44.62 ± 5.32
MBMWT	32,472	418,113	33.36 ± 3.96
BMW-CSP	14,883	480,486	30.27 ± 2.19
Waves	14,580	188,081	18.22 ± 1.61

Confidence interval for times were computed at confidence level 95%

storage of candidate results while processing the first tier, which is an extra memory requirement that does not appear in Waves. Finally, in the experiments, Waves performed slightly worse than the baselines for a few very long queries that required three waves to be fully processed by the method. The result indicates that Waves might not be a good option for processing queries in environment where the workload is dominated by long queries. Waves is substantially different from BMW-CSP and from MBMWT. Waves has important properties, such as the possibility of achieving gains even when we use more tiers. BMW-CSP does not allow more tiers and MBMWT becomes worse with 3 tiers, as also is reported by its authors. Waves also has the advantage of not requiring extra accumulators while processing queries. BMW-CSP, our previous method requires accumulators.

As for the open questions we plan to address in the future, we include a study on how to estimate the best number of tiers for Waves given a collection without performing test experiments and how to efficiently set the best sizes of each tier. While the cost of performing tests to adjust parameters is not prohibitive, still this is an interesting topic to address. We also plan to study the possible impact of using our method in systems where the complete index does not fit into memory; checking whether the multi-tier strategy proposed here would be useful in this scenario. When we have multiple memory levels in hierarchy, the multi-tier strategies could compete with, or complement, the performance of cache systems. Going in a similar direction, it would be interesting to further study the distribution of documents into tiers and how it could affect performance.

Another opportunity that came out of our experiments is that a combination of Waves and MBMWT could further improve the time performance of the methods. We could detect queries that require more waves in advance, and thus decide to process those queries using MBMWT, instead of Waves. This is quite possible, since the index format adopted by the two methods is compatible. Thus a future research direction would be to investigate methods for classifying queries according to their estimated number of required waves. We could also investigate the possibility to automatically detect the user goals while processing queries (Herrera et al. 2010) and then change the pruning thresholds according to the user goals to further improve the performance of the methods.

Acknowledgements Authors thank E-vox/FAPEAM project and CNPq fellowship grants (Edleno S. de Moura and Altigran S. da Silva) for the financial support.

References

Akbarinia, R., Pacitti, E., & Valduriez, P. (2007). Best position algorithms for top-k queries. In *Proceedings of the 33rd international conference on very large data bases, VLDB '07* (pp. 495–506). VLDB Endowment. http://dl.acm.org/citation.cfm?id=1325851.1325909.

Anh, V., & Moffat, A. (2006). Pruned query evaluation using pre-computed impacts. In: *ACM SIGIR* (pp. 372–379).

Anh, V. N., de Kretser, O., & Moffat, A. (2001). Vector-space ranking with effective early termination. In *ACM SIGIR* (pp. 35–42).

Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern information retrieval* (2nd ed.). Reading: Addison-Wesley Publishing Company.

Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., & Zien, J. (2003). Efficient query evaluation using a two-level retrieval process. In *ACM CIKM* (pp. 426–434).

Carmel, D., Cohen, D., Fagin, R., Farchi, E., Herscovici, M., Maarek, Y. S., et al. (2001). Static index pruning for information retrieval systems. In *ACM SIGIR* (pp. 43–50).

Carvalho, A., Rossi, C., de Moura, E. S., Fernandes, D., & da Silva, A. S. (2012). LePrEF: Learn to pre-compute evidence fusion for efficient query evaluation. *JASIST*, *55*(92), 1–28.

Chakrabarti, K., Chaudhuri, S., & Ganti, V. (2011). Interval-based pruning for top-k processing over compressed lists. In *Proceedings of the 2011 IEEE 27th international conference on data engineering, ICDE '11* (pp. 709–720). IEEE Computer Society, Washington, DC, USA. doi:10.1109/ICDE.2011.5767855.

Daoud, C., de Moura, E. S., Fernandes, D., da Silva, A. S., Carvalho, A. L., & Rossi, C. (2016). Fast top-k preserving query processing using two-tier indexes. *Information Processing & Management*, *52*(5), 855–872.

Dimopoulos, C., Nepomnyachiy, S., & Suel, T. (2013). A candidate filtering mechanism for fast top-k query processing on modern cpus. In *ACM SIGIR* (pp. 723–732).

Ding, S., & Suel, T. (2011). Faster top-k document retrieval using block-max indexes. In *ACM SIGIR* (pp. 993–1002).

Fontoura, M., Josifovski, V., Liu, J., Venkatesan, S., Zhu, X., & Zien, J. Y. (2011). Evaluation strategies for top-k queries over memory-resident inverted indexes. *PVLDB*, *4*(12), 1213–1224.

Herrera, M. R., de Moura, E. S., Cristo, M., Silva, T. P., & da Silva, A. S. (2010). Exploring features for the automatic identification of user goals in web search. *Information Processing & Management*, *46*(2), 131–142.

Moffat, A., & Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM TOIS*, *14*(4), 349–379. doi:10.1145/237496.237497.

Moura, E Sd, Santos, C Fd, Araujo, Bd S, Silva, A Sd, Calado, P., Nascimento, M. A., et al. (2008). Locality-based pruning methods for web search. *ACM Transactions on Information Systems (TOIS)*, *26*(2), 9.

Ntoulas, A., & Cho, J. (2007). Pruning policies for two-tiered inverted index with correctness guarantee. In *ACM SIGIR* (pp. 191–198).

Ottaviano, G., & Venturini, R. (2014). Partitioned Elias–Fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval* (pp. 273–282). ACM.

Risvik, K., Aasheim, Y., & Lidal, M. (2003). Multi-tier architecture for web search engines. In *First Latin American web congress* (pp. 132–143).

Robertson, S. E., & Walker, S. (1994). Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *ACM SIGIR* (pp. 232–241).

Rossi, C., de Moura, E. S., Carvalho, A. L., & da Silva, A. S. (2013). Fast document-at-a-time query processing using two-tier indexes. In *ACM SIGIR* (pp. 183–192).

Salton, G., Wong, A., & Yang, C. S. (1974). *A vector space model for automatic indexing*. Tech. Rep., Ithaca, NY.

Shan, D., Ding, S., He, J., Yan, H., & Li, X. (2012). Optimized top-k processing with global page scores on block-max indexes. In *WSDM* (pp. 423–432).

Silvestri, F. (2007). Sorting out the document identifier assignment problem. In *European conference on information retrieval* (pp. 101–112). Springer.

Skobeltsyn, G., Junqueira, F., Plachouras, V., & Baeza-Yates, R. (2008). ResIn: A combination of results caching and index pruning for high-performance web search engines. In *ACM SIGIR* (pp. 131–138).

Strohman, T., & Croft, W. B. (2007). Efficient document retrieval in main memory. In *ACM SIGIR* (pp. 175–182).

Zukowski, M., Heman, S., Nes, N., & Boncz, P. (2006). Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd international conference on data engineering, ICDE'06* pp. 59. IEEE Computer Society, Washington, DC, USA.