Efficient Parallel Block-Max WAND Algorithm

Oscar Rojas[1], Veronica Gil-Costa[2,3], and Mauricio Marin[1,2]

[1] DIINF, University of Santiago, Chile
[2] Yahoo! Labs Santiago, Chile
[3] CONICET, University of San Luis, Argentina

Abstract. Large Web search engines are complex systems that solve thousands of user queries per second on clusters of dedicated distributed memory processors. Processing each query involves executing a number of operations to get the answer presented to the user. The most expensive operation in running time is the calculation of the top-k documents that best match each query. In this paper we propose the parallelization of a state of the art document ranking algorithm called Block-Max WAND. We propose a 2-steps parallelization of the WAND algorithm in order to reduce inter-processor communication and running time cost. Multi-threading tailored to Block-Max WAND is also proposed to exploit multi-core parallelism in each processor. The experimental results show that the proposed parallelization reduces execution time significantly as compared against current approaches used in search engines.

1 Introduction

Large-scale Web search engines are built as a collection of services hosted by the respective data center. Each service is deployed on a set of processing nodes (processors) of a high-performance cluster of computers. Services are software components such as (a) calculation of the top-k documents that best match a query; (b) routing queries to the appropriate services and blending of results coming from them; (c) construction of the result Web page for queries; (d) advertising related to query terms; (e) query suggestions, among many other operations. The service relevant to this paper is the top-k calculation service.

The top-k calculation nodes are assumed to perform document ranking for queries by using the WAND algorithm [4]. This algorithm is been currently used by a number of commercial vertical search engines. The concept is that the document collection is evenly distributed on P processing nodes or partitions, and for each partition an inverted index is constructed from the respective documents. The inverted indexes enable the fast determination of the documents that contain the terms of the query under processing. They contain additional static and dynamic data to enable the WAND algorithm to reduce the number of documents that are fully evaluated during the ranking process. The static value, called *upper-bound* value, is calculated for each index term at construction time whereas the dynamic value, called *threshold* value, starts in zero for each new query and is updated during the WAND computations across the inverted file.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 394–405, 2013.

To answer a query, a broker machine sends the query to all of the P partitions. Then for each query, the top-k nodes locally compute the most relevant documents and send them to a broker machine. Later, the broker merges those $P \times k$ document results to obtain the global top-k document results. Hence, our aim is to reduce the total number of full document evaluations (score calculations) performed during the document ranking process, which in turn leads to a reduction in the running time required by the WAND algorithm to finish the ranking process. Also our aim is to reduce the total number of documents communicated among the top-k calculation nodes and the broker machine. The big picture is that by doing this one can be able to reduce the total number of top-k calculation nodes deployed in production.

In this paper we propose to make the top-k calculation service efficient in the sense of significantly reducing the average amount of computation and communication per query executed by the processing nodes hosting the service (namely, the top-k calculation nodes). We propose a 2-steps algorithm which determines the number of document results that must be sent to the broker by each top-k calculation node. The algorithm does not lose precision of results. We also propose a multi-threading strategy to schedule query processing in each of the top-k calculation nodes by using the Block-Max WAND (BMW) algorithm proposed in [6]. We analyze the effect of running queries with both high and low computational costs over a cluster of processors supporting multi-threading.

The remaining of this paper is organized as follows. Section 2 reviews related work and Section 2.1 describes the WAND algorithm. Section 3 describes our proposal. Section 4 presents a performance evaluation study considering different metrics and Section 5 presents conclusions.

2 Background and Related Work

To speed up query processing, the top-k calculation service relies on the use of an inverted index or inverted file) which is a data structure used by all well-known Web Search Engines. This index enables the fast determination of the documents that contain the query terms and contains data to calculate document scores for ranking. The index is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the document collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with data used assign a score to the document. To solve a query, it is necessary to get from the posting lists the set of documents associated with the query terms and then to perform a ranking of these documents in order to select the top-k documents as the query answer.

One important bottleneck in query ranking is the length of the inverted file, which is usually kept in compressed format. To avoid processing the entire posting lists or reducing the amount of expensive computations performed in each posting list item like the WAND [4]. Several optimizations have been proposed in the technical literature for the WAND algorithm. The aim of [8] and [10] is to

improve performance by computing tighter upper bound scores for each term. The work in [1] and [2] propose to maintain the posting lists ordered by score instead of document identifier (docID) in order to retrieve and evaluate first the documents with the highest scores. But in these cases, compression tends to be less effective. An optimization of the WAND algorithm named the Block-Max WAND [6] has been devised to avoid decompressing the entire posting lists.

A number of papers have been published on parallel query processing upon distributed inverted files (for recent work see [9]). Many different methods for distributing the inverted file onto P partitions and their respective query processing strategies have been proposed in the literature [4]. The different ways of doing this splitting are mainly variations of two basic dual approaches: document partition and term partition. In the former, documents are evenly distributed on P partitions and an independent inverted index is constructed for each of the P sets of documents. In the last one, a single inverted index is constructed from the whole document collection to then distribute evenly the terms with their respective posting lists onto the P partitions.

Also query throughput is further increased by using application caches (for recent work see [12]). But, to the best of our knowledge, the work presented in this paper is the first attempt to evaluate and determine the relevant features that have to be taken into consideration when evaluating the WAND algorithm in a distributed cluster of processors supporting multi-threading.

2.1 The WAND Query Evaluation Process

The method proposed by Broder et al. [4] assumes a single threaded processor containing an inverted index. As usual, each query is evaluated by looking for query terms in the inverted index and retrieving each posting list. Documents referenced from the intersection of the posting lists allow to answer conjunctive queries (AND bag of word query) and documents retrieved at least from one posting list allow to answer disjunctive queries (OR bag of word query).

The ranking is used to compute the similarity between documents and the query. Then this function returns the top-k documents. There are several ranking algorithms such as BM25 or the vector model [4]. Ranking algorithms should be able to quickly process large inverted lists. But the size of these lists tends to grow rapidly with the increasing size of the Web. Therefore, in practice, these algorithms use early termination techniques avoiding processing complete lists [3]. In some early termination techniques the posting lists are sorted so that most relevant documents are found first. Other ingenious techniques have been proposed when the posting lists are sorted by docIDs. They reduce running time avoiding computing the scores of all documents of the posting lists by skipping document score computations. This is the case of the WAND method proposed by [4].

The WAND approach uses a standard docID sorted index. It is a query processing method based on two levels. In the fist level, some potential documents are selected as results using an approximate evaluation. Then, in the second level those potential documents are fully evaluated to obtain their scores. This

two-steps process uses a heap to keep the current top-k documents where in the root is located the document with least score. The root score provides a threshold value which is used to decide the full score evaluation of the remaining documents in the posting lists associated with the query terms. To this end the algorithm iterates through posting lists to evaluate them quickly using a pointer movement strategy based on pivoting. In other words, pivot terms and pivot documents are selected to move forward in the posting lists which allows skipping many documents that would have been evaluated by an exhaustive algorithm.

In Figure 1.(a) we show how the WAND algorithm works for a query with three terms "*tree, cat* and *house*". First, posting lists of the query terms are sorted by docIDs upper bounds (UBs) from top to bottom. Then we add the upper bounds of the terms until we get a value greater or equal to the threshold. In Figure 1.(a) by adding the UBs of the first two terms we get $(2+4 \geq 6)$. Thus *cat* is selected as the pivot term. We assume that the current document in this posting list is "503". Therefore, this document becomes the pivot document. If the first two posting lists do not contain the document 503, we proceed to select the next pivot. Otherwise we compute the score of the document. If the score is greater or equal to the threshold we update the heap by removing the root document and adding the new document. This iterative algorithm is repeated until there are no documents to process or until it is no longer possible for the sum of the upper bounds to exceed the current threshold.

The work presented in [6] proposes using compressed posting lists organized in blocks (see Figure 1.(b)). Each block stores the upper bound (Block max) for the documents inside that block in uncompressed form, thus enabling to skip large parts of the posting lists by skipping blocks. This drastically reduces the cost of the WAND algorithm but does not guarantee correctness because some relevant documents could be lost. To solve this problem, the authors propose a new algorithm that moves forward and backwards in the posting lists to ensure that no documents are missed. Independently, the same idea was presented in [5]. In this later work, authors presented an algorithm for disjunctive queries that first performs pre-processing to split blocks into intervals with aligned boundaries and to discard intervals that cannot contain any document capable of making into the top-k results. Multi-threading algorithms for ranking methods different to WAND have been studies in [11] and [7].

3 Two-Level Ranking on a Distributed Search Engine

In this work we consider a Web search engine in which there is one broker and P top-k calculation nodes, where P indicates the level of document partitioning considered in the distribution of the document collection. Each top-k calculation node has its own inverted index, where the posting lists refer to local documents. When a new query arrives, the broker sends the query for evaluation to each node. Then, the nodes work on their inverted indexes to produce query answers and pass the results back to the broker. Conventionally, search engines use the standard asynchronous multiple master/slave paradigm to process queries.

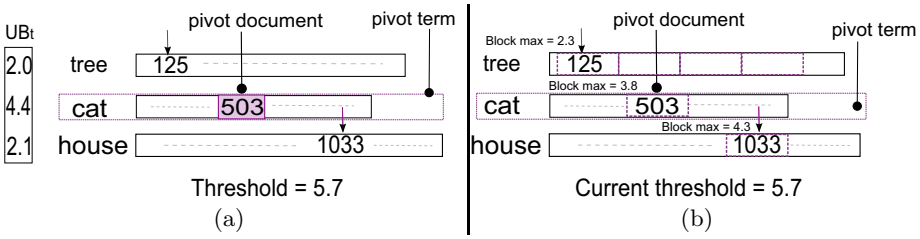

Fig. 1. (a) Executing WAND algorithm. (b) The Block-Max WAND (BMW): each block stores the upper bound for the documents inside the block.

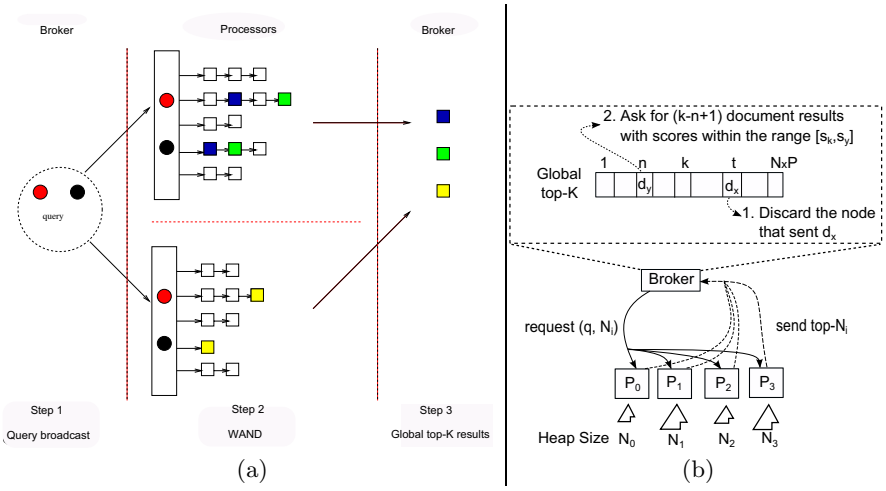

Fig. 2. (a) A distributed WAND query evaluation process. (b) Proposed algorithm.

A WAND-based search engine in this context considers parallel term iteration over each top-k calculation node. If a document satisfies the threshold condition, then it is inserted in the heap of the top-k document results. Notice that document identifiers are kept sorted in posting lists allowing iteration in linear time proportional to the posting list length. Finally, a full document score evaluation is executed over the set of candidate documents determined by the WAND algorithm, and the global top-k results are determined. Figure 2.(a) illustrates this process.

Let us shortly discuss some considerations for the query evaluation process. There exist two alternatives for WAND scores estimation. As the search engine is document-partitioned, upper bounds for each term can be calculated considering each sub-collection, i.e. each processor determines its own term upper bounds. In this case upper bounds are stored in local inverted indexes and eventually each term can register P different upper bounds. Another alternative is to calculate the upper bound for each term considering the full collection, thus upper bounds correspond to global scores. In this case it is also possible to store these values in the local inverted indexes, replicating upper bounds across the processors. We consider that the second alternative is more recommendable because it allows to skip more documents in the WAND iteration process.

3.1 Distributed 2-Steps Algorithm

In this section we detail our proposed algorithm which aims to obtain accurate results while reducing the number of operations (scores calculations) performed when executing the WAND algorithm, the memory allocated to the heap data structure and the communication among processors. The algorithm works as follows: (1) in the first step, the top-k calculation nodes send $N = (k/P+\alpha) < k$ document results to the broker machine, (2) the broker machine detects and discards the nodes that do not have more relevant documents, (3) the broker requests more documents to the remaining nodes. Our hypothesis is that k/P is the average number of document results retrieved from each top-k calculation node as the document collection tends to be evenly distributed among the P partitions. Thus we propose to emulate ranking as a distributed priority queue.

Formally, given a set of top-k nodes $P = \{p_1, p_2, \ldots, p_p\}$ and for each query q, each node $p_i \in P : 1 \leq i \leq p$, sends to the broker machine a set of document results $\{< d_1, sc_1 >< d_2, sc_2 >, \ldots, < d_N, sc_N >\}$, where d_j is the document identifier, sc_j is the score associated with the document and N is the heap size (i.e. the number of document results initially requested). Each top-k node computes a disjoint set of document results.

After the broker machine performs the merge of $N \times P$ document results, it gets a list of tuples: $[< d_{x,1}, sc_{x,1}, i >, \ldots, < d_{w,N\times P}, sc_{w,N\times P}, s >]$. The first component of each tuple represents the document identifier and the position of the document within the global document result set. For example, $d_{x,1}$ represents the most relevant document (i.e. the top-1 document) identified by x. The second component represents the score of the document using the same nomenclature. The third component represents the identifier of the node which sent the document. Then, the proposed algorithm determines whether to request more documents results to the top-k nodes as follows (see Figure 2.(b)):

1. If $\exists < d_{x,t}, sc_{x,t} > \in p_i$ with $sc_{x,t} > sc_{s,k}$ (i.e. the global position of document x is $t > k$) then remove p_i from the list. In other words, no more document results are requested to p_i.
2. If $\exists \ < d_{y,n}, sc_{y,n} > \in p_i$ with $sc_{y,n} < sc_{s,k}$ and $sc_{y,n}$ is the least relevant document sent by p_i, then request to p_i the next $k-n+1$ document results. $sc_{s,k}$ is the score of the k-th document in the global set of results. The upper-bound for those requested documents is given by $sc_{y,n}$ and the lower-bound is given by $sc_{s,k}$. In other words, the broker machine requests to p_i documents with scores in the range of $[sc_{s,k}, sc_{y,n}]$. The number of requested documents is given by $k-n+1$, because $k-n$ is the number of documents required to get the k-th position and $+1$ is used to support documents with the same score.

If we have a set of document results $\{< d_{a,1}, sc_{a,1}, 1 >, < d_{b,2}, sc_{b,2}, 1 >, < d_{c,3}, sc_{c,3}, 2 >, < d_{d,4}, sc_{d,4}, 3 >, < d_{e,5}, sc_{e,5}, 2 >, < d_{f,6}, sc_{f,6}, 3 >\}$ for $P = \{p_1, p_2, p_3\}$ where $|P| = 3$. In this example $k = 4$ and $\alpha = 1$, therefore each processor sends $N = 4/4 + 1 = 2$ documents results to the broker machine. In this case, p_2 and p_3 are discarded and no more documents are required from

them because they have sent documents that are located after the k-th global position. On the contrary, it is necessary to ask to p_1 a total of $k-2+1=3$ more documents, with scores is in the range $[sc_{d,4}, sc_{b,2}]$. Recall that the formula uses $+1$ to include documents with the same score. As in the first step the broker sets $N=2$, the size of the heap used by the WAND algorithm to store the most relevant documents at the nodes side is set to $N=2$. But in the next step, the broker machine asks to p_1 a total of $N=3$ documents results. Therefore the size of the heap is set to 3. To avoid re-computing all document results from the beginning, the broker also sends the range $[sc_{d,4}, sc_{b,2}]$ to p_1 to discard documents with scores outside this range.

The α parameter used to request document results in the first step is dynamically set for each query and for each top-k node as follows. For a period of time Δ_i we warm up the system by running an oracle algorithm which predicts the optimum value of α. In other words, queries are solved in the first step of the proposed algorithm. Also, every time a query is processed, we store the query terms (t_i), the posting list size of each term (L_i) and the optimal α parameter for each top-k node ($q_a =< t_1, t_2, L_1, L_2, \alpha_{p_1}, \ldots \alpha_{p_P} >$). The optimal α parameter determines the additional number of documents each node has to send to the broker machine. This information is kept in memory for just one interval of time. Then, in the following intervals of times $[\Delta_{i+1}, \ldots \Delta_{j-1}, \Delta_j]$ where Δ_j is the current period of time, we estimate the α_{p_i} values for a query q_a by applying the rules:

1. If q_a was processed in Δ_{j-1} we use the value α_{p_i} stored in the previous interval for $i=1 \ldots P$.
2. We define the set $X = q_a.\text{terms} \cap q_b.\text{terms}$, $\forall q_b$ processed in Δ_{j-1}. Then if $X <> \emptyset$, we select the query q_b which maximize $|X|$ (i.e. the query with the greatest amount of terms in X). If more than one query has the same maximum amount of terms in X, then we select the one which contains the term with the largest posting list.
3. Otherwise, for each top-k node p_i we use the average α_{p_i} value registered in Δ_{j-1}.

3.2 Multi-threading Algorithms

In this section we describe our multi-threading algorithms. Our first algorithm uses a local heap (**LH**) data structure for each thread. The posting lists are evenly distributed among threads. In other words, each thread holds a portion of the inverted file. Then each thread process incoming queries with its own local inverted file and using a local heap of size k, where k is the top-k documents retrieved to the user. At the end, the documents identifiers stored in the local heaps have to be merged. A synchronization barrier is implemented before computing the merge operation.

Our second approach, named shared heap (**SH**) inverted files are distributed among threads as in the LH approach. But query processing is performed using a global heap of size k. Then, no merge operation is performed at the end of the query process, but additional *locks* have to be implemented to guarantee exclusive access to the threads when updating the heap contents. Our hypothesis

is that for large collections where queries are more expensive to compute, it is better to use a single global heap, because the threshold of the heap can be quickly updated. Achieving the optimal threshold value quickly has several advantages: (1) we can reduce the number of scores computations and (2) fewer heap update operations are preformed (reducing the number of locks).

4 Evaluation

4.1 Data Preparation and Experiment Settings

We experiment with a query log of 16,900,873 queries submitted to the AOL Search service between March 1 and May 31, 2006. We set the term weights according to the frequency of the term in the query log. Then, we applied these queries to a sample (1.5 TB) of the UK Web obtained in 2005 by Yahoo!, over a collection compounded by 26,000,000-terms and a 56,153,990-document inverted index. We consider a Web search engine computing top-100 and top-1000 document results. In the case of the top-100 results we considered a sample of 1,000,000 queries.

We divide the following experiments into two groups. First, we evaluate our proposed 2-steps algorithm in a distributed cluster of 16 processors. We measure the number of scores computations, number of heap updates, running time and communication cost. Second, we compute the number of decompressed blocks performed by the Block-Max WAND (BMW) [6] using the multi-threading LH and SH approaches. The Block-Max WAND algorithm groups the posting lists into blocks of fixed size. Each block stores 100 documents in compress form. The optimal threshold value is quickly reached because each block has its own UB_t. We also measure the speed-up achieved by both multi-threading algorithms. The results were obtained on a cluster of 16 processors with 32-core AMD Opteron 2.1GHz Magny Cours processors, sharing 32 GB of memory. All experiments were run using an inverted file of 27Gb in main memory.

4.2 Distributed Algorithm Evaluation

We compare the performance achieved by our proposal against the art baseline algorithm which always request k document results per query to each node partition. Figure 3 shows normalized running times and Figures 4 shows communication cost. We show results normalized to 1 in order to better illustrate the comparative performance. To this end, we divide all quantities by the observed maximum in each case.

For a small top-k, our proposal algorithm significantly improves the baseline performance by 40%. This means that the proposal can reduce the total number of scores computation performed by the WAND algorithm at the top-k nodes side and the number of document results communicated to the broker machine. This claim is confirmed in Figure 4.(a) which shows that the proposal reduces communication cost logarithmically. Also, Figure 5.(a) shows that the proposal reduces the average number of score computation by 50% with 16 processors

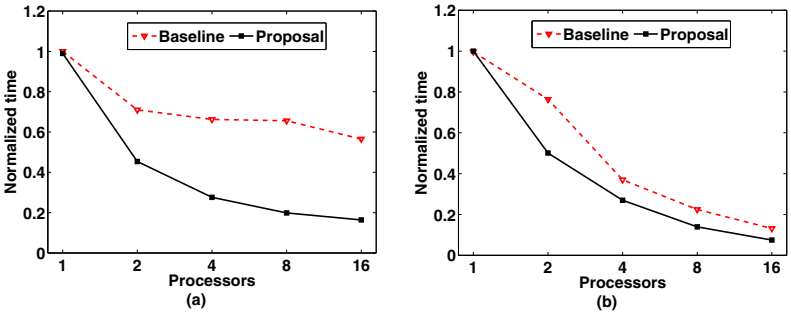

Fig. 3. Running time reported for (a) top-100 and (b) top-1000 document results

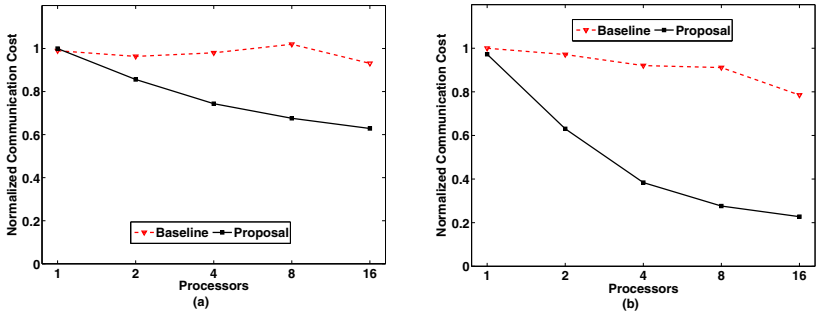

Fig. 4. Communication cost for (a) top-100 and (b) top-1000 document results

and Figure 5.(c) shows that for the same number of processors the number of heap updates is reduced by almost 60%.

With a larger number of document results (Figure 3.(b)) the gain achieved by the proposal algorithm (in terms of running time) is reduced to 10% in average. Again, these results are validated in Figure 5.(b) and Figure 5.(d) for the number of score computation and the number of heap updates respectively. In this case, the proposal algorithm reports 3% less number of score computations than the baseline and 25% less heap updates for a total of 16 processors. However, communication cost is improved by the proposal due to less than k document results are requested per query per top-k node.

To understand the effect of using a larger value of k we have to remember that posting lists follow the Zip law [4]. In other words, there are few documents with high score and many documents with low score. Moreover, posting lists are in compress form organized in blocks and each block has an upper bound named block max for the documents store inside the block. Therefore, with a larger k the threshold value tends to be small and more score computations are preformed due to less blocks are discarded when comparing the block max and the threshold.

4.3 Multi-threading Algorithms Evaluation

The total cost of solving a query is determined by three main stages: (1) the load of parameters, (2) the search algorithm and (3) the merge and sort of

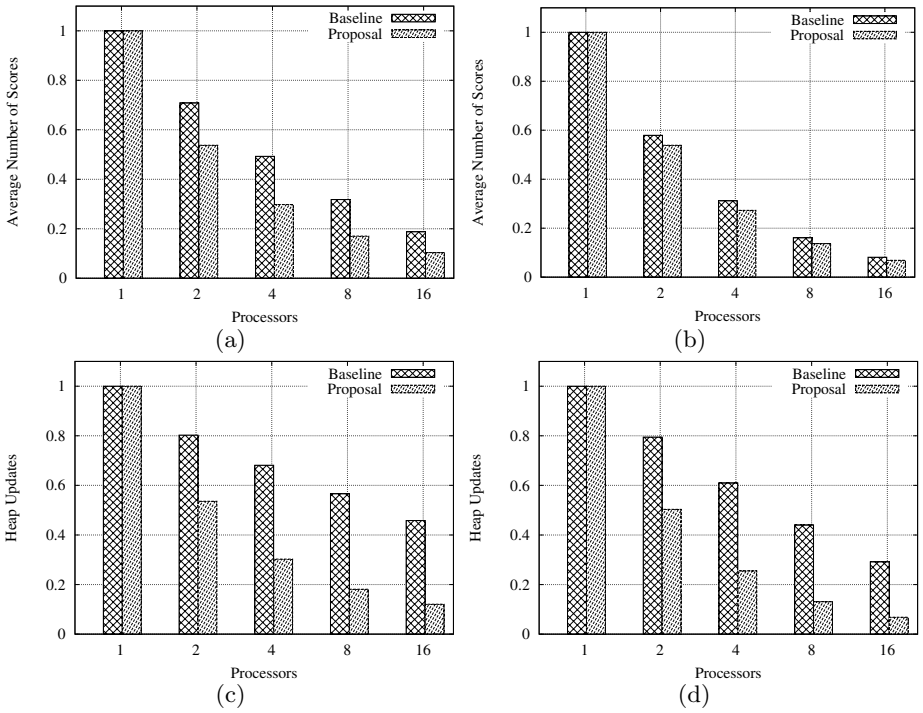

Fig. 5. Score computation and heap update operations reported for (a)(c) top-100 and (b)(d) top-1000 document results

results. The first stage involves the computation of the upper bounds UB for each block of the posting lists and also the multiplication of each term weight with the UB. This product is used at running time. In the second stage we apply the WAND to find the top-k documents for the query. In the last stage we perform the merge of local heaps and sort the final results. Usually the higher computational cost is performed by the last two stages. The algorithm that computes the top-k documents and merge tasks and sort algorithms. Therefore, we focus on the number of scores, the number of heap updates and number of blocks decompressed using the Max-Block WAND.

Figure 6 shows the results regarding the number of decompressed blocks. First we show the average number of blocks required by query per thread. Then we show the average number of decompressed blocks using both multi-threading approaches and its relation to the number of heap updates.With a small k value, the algorithms decompress only the blocks containing the documents that will be added to the heap. These results show that the SH approach drastically reduces the number of decompressed blocks which is an essential issue in search engines where fast access to the information and resource usage optimization is essential.

This because queries are evaluated in parallel and threshold values are accessed and updated jointly by all threads. The threshold value is the decisive factor in the speed of the WAND algorithm. It is used to decide whether to

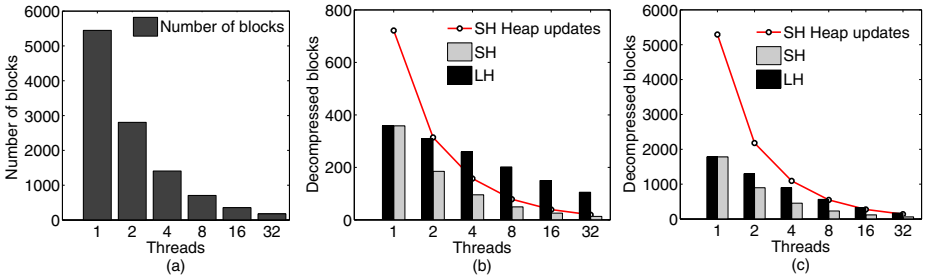

Fig. 6. Number of decompressed blocks achieved by both the LH and SH approaches

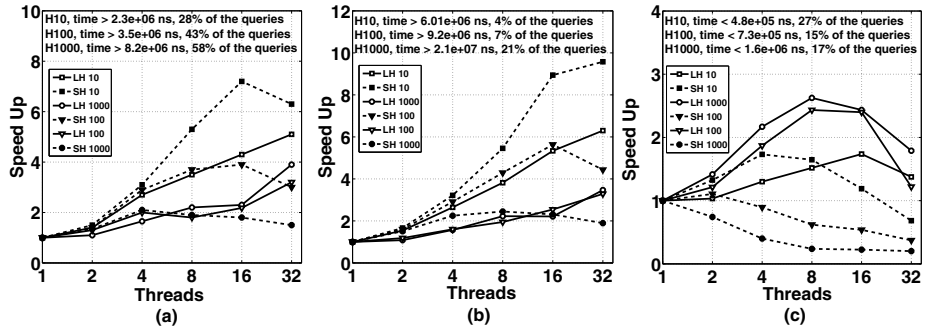

Fig. 7. (a)Speed-up for expensive queries requiring at least a running time of {2ms,3ms,8ms}) for a heap size of $k = \{10, 100, 1000\}$ (b)Speed-up for expensive queries requiring at least a running time of {6ms,19ms,20ms}) for a heap size of $k = \{10, 100, 1000\}$ (c) Speed-up for queries that have a low computational cost requiring a running time less than {0.4ms,0.7ms,1.6ms} for a heap size of $k = \{10, 100, 1000\}$

compute a score and whether to access a compress block of the posting list. Also, regardless of multi-threaded version, this figure shows that the WAND algorithm is very efficient when unzipping the posting lists. By using a heap of size $k = 100$, the worst decompression level is reached with a single thread.

Figures 7.(a) and 7.(b) show the speed-up obtained for queries that are expensive to compute. Typically, these queries tend to compute a larger number of scores and make a greater number of updates on the heap. Figure 7.(a) shows results for a heap of size $k = 10$ and executing 28% of the queries log. The sequential average time to run these queries is $2.3e + 06ns$. For a heap of size $k = 100$ we executed 43% of the query log and for a heap size of $k = 100$ we executed 58% of the query log. Figure 7.(b) shows the same experiment but for queries more expensive to process, requiring at least a sequential processing time of $6.01e + 06ns$ for a heap size of $k = 10$, $9.2e + 06$ for a heap size of $k = 100$ and $2.1e + 07$ for a heap size of $k = 1000$. With this experiment we show that the performance of the SH approach begins to increase when computing queries with higher computational cost. Finally, Figure 7.(c) shows results obtained for queries requiring a low computational cost. In other words, queries that are

quickly solved with a low number of scores calculations and few heap updates. The LH approach presents the best performance.

5 Conclusions

We have presented a 2-steps algorithm which aims to reduce computation and communication cost between top-k nodes and the broker machine. In the first step, it uses an adjustment parameter which is dynamically set for each query to request results from each processor. Our experiments show that the proposed algorithm significantly outperforms the baseline strategy.

We also evaluated the WAND algorithm using two multi-threading strategies. We performed experiments using queries with both high and low computational costs. The shared heap (SH) multi-threading approach is less efficient than the local heaps (LH) approach for low cost queries. In general, SH reduces the number of decompressed blocks, the number of heap updates and the number of scores computed to retrieve the top-k documents. However, the gain in these metrics is not reflected in the execution time for low cost queries. Thus a combination of both approaches should be used based on a prediction of the running time cost of queries. In practice, the running time tends to be proportional to the size of the involved posting lists.

References

1. Anh, V.N., Moffat, A.: Pruned query evaluation using pre-computed impacts. In: SIGIR, pp. 372–379 (2006)
2. Bast, H., Majumdar, D., Schenkel, R., Theobald, M., Weikum, G.: Io-top-k: Index-access optimized top-k query processing. In: VLDB, pp. 475–486 (2006)
3. Blanco, R., Barreiro, A.: Probabilistic static pruning of inverted files. ACM Trans. Inf. Syst. 28(1) (2010)
4. Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.Y.: Efficient query evaluation using a two-level retrieval process. In: CIKM, pp. 426–434 (2003)
5. Chakrabarti, K., Chaudhuri, S., Ganti, V.: Interval-based pruning for top-k processing over compressed lists. In: ICDE, pp. 709–720 (2011)
6. Ding, S., Suel, T.: Faster top-k document retrieval using block-max indexes. In: SIGIR, pp. 993–1002 (2011)
7. Jonassen, S., Bratsberg, S.E.: Intra-query Concurrent Pipelined Processing for Distributed Full-Text Retrieval. In: Baeza-Yates, R., de Vries, A.P., Zaragoza, H., Cambazoglu, B.B., Murdock, V., Lempel, R., Silvestri, F. (eds.) ECIR 2012. LNCS, vol. 7224, pp. 413–425. Springer, Heidelberg (2012)
8. Long, X., Suel, T.: Optimized query execution in large search engines with global page ordering. In: VLDB, pp. 129–140 (2003)
9. Marin, M., Costa, V.G.: Sync/async parallel search for the efficient design and construction of web search engines. PARCO 36(4), 153–168 (2010)
10. Strohman, T., Turtle, H.R., Croft, W.B.: Optimization strategies for complex queries. In: SIGIR, pp. 219–225 (2005)
11. Tatikonda, S., Cambazoglu, B., Junqueira, F.: Posting list intersection on multicore architectures. In: SIGIR (2011)
12. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: WWW, pp. 401–410 (2009)