Distributing efficiently the Block-Max WAND algorithm

Oscar Rojas, Veronica Gil-Costa, Mauricio Marin

Abstract

Large search engines are complex systems composed by several services. Each service is composed by a set of distributed processing nodes dedicated to execute a single operation required to user queries. One of these services is in charge of computing the top-$k$ document results for queries by means of a document ranking operation. This ranking service is a major bottleneck in efficient query processing as billions of documents has to be processed each day. To answer user queries within a fraction of a second, techniques such as the Block-Max WAND algorithm are used to avoid fully processing all documents related to a query. In this work, we propose to efficiently distributing the Block-Max WAND computation among the ranking service processing nodes. Our proposal is devised to reduce memory usage and computation cost by assuming that each one of the $P$ ranking processing nodes provide top-$K/P + \alpha$ documents results, where $\alpha$ is an estimation parameter which is dynamically set for each query. The experimental results show that the proposed approach significantly reduces execution time compared against current approaches used in search engines.

Keywords: efficient distributed query evaluation; WAND; inverted files.

1. Introduction

Current Web search engine query processing systems are designed with the goal of returning query results in a fraction of a second. These engines embody a number of optimization techniques and algorithms designed to let them efficiently cope with heavy and highly dynamic user query traffic [1]. Among these algorithms, we have the ranking algorithm used to calculate the most relevant documents results for user queries. Web search engines are composed of a large set of search nodes organized as a collection of services hosted by the respective data center. Each service is deployed on a set of processing nodes of a high-performance cluster of processors.

Services are software components such as (a) calculation of the top-$k$ document results 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, between others. The service relevant to this paper is the top-$k$ calculation service.

Corresponding author. Tel.: +54-2664-430823.
E-mail address: gvcosta@yahoo-inc.com.
The top-$k$ nodes are assumed to perform document ranking for queries. The ranking operation is based on an inverted index, which is a simple and efficient data structure that allows us to find documents that contain a particular term [2]. The concept (not the actual realization and construction process) is that the document collection is evenly distributed into $P$ partitions of the service and for each partition an inverted index is constructed considering the respective documents. The inverted indexes enable the fast determination of the documents that contain the terms of the query under processing. 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 text collection. Each of these terms is associated with a posting list (usually kept compressed) which contains the identifiers where the term appears in the collection along with additional data used for ranking purposes. 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.

There are many ranking functions such as BM25 or Cosine Similarity [2]. In particular, the WAND algorithm [3] is currently used in a number of commercial search engines. The WAND algorithm [4] keeps posting lists sorted by document identifier (docID). Therefore, the relevant or more promising documents are spread out throughout the inverted lists. To avoid fully evaluating the score of all documents in the posting list of each term belonging to a given query, a smart pointer movement technique is used to skip many documents that are not relevant for that query. The inverted index used by the WAND algorithm contains static and dynamic data in order 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 and is updated during the WAND computations upon the inverted file.

The query processing process is as follows. There is a broker machine that sends the query to all of the $P$ partitions of the top-$k$ document calculation service. Then for each query, the top-$k$ nodes in parallel 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 both the total number of full document evaluations (scores) performed during the document ranking process and the total number of documents communicated among the top-$k$ nodes and the broker machine. The idea is that by doing this one can be able to reduce the total number of top-$k$ nodes deployed in production.

In this paper we propose a distributed algorithm to significantly reduce the average amount of computation and communication per query executed by the processing nodes hosting the service (namely, the top-$k$ nodes). Our algorithm estimates the number of document results that must be evaluated in each of the top-$k$ nodes using the Block-Max WAND (BMW) algorithm and communicated to the broker machine. The Block-Max Index [4] was recently presented as an improvement to the WAND algorithm. Unlike the standard WAND strategy which stores the maximum upper bound for each term, the Block-Max Index divides the posting lists into blocks and stores maximum upper bound for individual blocks of terms.

The remaining of this paper is organized as follows. Section 2 briefly reviews related work and Section 2.1 describes the WAND and the Block-Max WAND algorithms. 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

As explained before, 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 [1]. 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, the ranking operation usually applies early termination algorithms like the WAND [3]. Several optimizations have been proposed in the technical literature for the WAND algorithm. The aim of [5] and [6] is to improve the pruning of posting lists by computing tighter upper bound scores for each term. The work in [7] and [8] proposed to maintain the posting lists in the score order instead of document identifier (docID) order to retrieve and evaluate first 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 [4] has been devised to avoid decompressing the entire posting lists.

There exists a number of methods to compress lists of integers (ie., Doc Ids). Among them, we have classical methods like Elias’ encodings [9] and Golomb/Rice’s encoding [10], as well as the recent VByte [11], Simple
9 [12], Interpolative [13] and PForDelta [14] encodings. All these methods benefit from sequences of integers whose distribution has many small numbers. Frequencies and term positions can be usually compressed using the same (or similar) techniques. There are, however, some previous studies aiming at compressing the term frequencies [15, Section 5] and term positions [16]. Compression can be further improved by re-labeling document IDs [17, 18, 19, 20, 21] in order to generate smaller DGaps.

A number of papers have been published on parallel query processing upon distributed inverted files (for recent work see [22, 23]). Many different methods for distributing the inverted file onto $P$ partitions and their respective query processing strategies have been proposed in the literature [1]. 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 [24, 15]). But, to the best of our knowledge, this work 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 PCs supporting multi-threading.

2.1. The two-level evaluation process

The method proposed by Broder et al. [3] assumes a search engine constructed over a single processor with a term-based 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 [1]. 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 [25, 26, 27, 28]. In some early termination techniques the posting lists are sorted so that most relevant documents are found first [29]. Other ingenious techniques have been proposed when the posting lists are sorted by docID. They avoid computing the scores of all documents of the posting lists by skipping score computations. This is the case of the method proposed by [3].

The WAND approach, uses a standard document-sorted index, and can thus employ a Document-At-A-Time (DAAT) approach. It is a query processing method based on two levels. Some potential documents are selected as results in the first level using an approximate evaluation. Then, in the second level those potential documents are evaluated with a comprehensive and precise metric. This two-step process in which evaluates an upper bound as results in the first level using an approximate evaluation. Then, in the second level those potential documents are selected to move forward in the posting lists which allows to skip many documents that would be evaluated by an exhaustive algorithm. WAND achieves early termination by enabling skips over postings that cannot make into the top results.

Let's consider an additive scoring model for document ranking, i.e. \( \text{Score}(d, q) = \sum_{t \in q \cap d} w(t, d) \), where \( w(t, d) \) represents the score for \( t \) in \( d \). Each term is associated with an upper bound \( UB \), which correspond to its maximal contribution to any document score in the collection. The threshold is set dynamically based on the minimum score among the top-$k$ results found so far. The top-$k$ results are stored into a heap, which is initially empty. An admission of a new document in the top-$k$ heap results is conducted when the WAND operator is true, i.e. when the score of the new document is greater than the score of the document with the minimum score stored in the heap. If the heap is full, the document with the minimum score is replaced, updating the value of the threshold. Documents with a score smaller than the score of the document with the minimum score stored in the heap are skipped. A larger threshold allows to skip more documents, reducing computational costs involved in query evaluation. It is also true that if upper bounds are accurate, then final scores are no greater than preliminary upper bounds. As
a consequence all documents skipped would not be placed in the top-$k$ results. Initial values for the threshold depend on the type of query to be processed. Disjunctive queries consider an initial threshold equals to zero. Conjunctive queries consider an initial threshold equals to the sum of the upper bounds of the query terms.

A WAND search engine allows to conduct a two-level retrieval process. At the first level, using WAND thresholds it is possible to identify candidate documents; at the second level these documents are fully evaluated calculating their exacted scores and recommending them to users. A WAND iterator operates over query term posting lists in parallel at the first level of evaluation, allowing to skip a significant amount of documents, avoiding full evaluation. However, the WAND search engine has only approximate upper bounds for the contribution of each term. In addition, as the ranking function might involve query independent variables, so the tuning of the threshold is a critical factor for the query evaluation. The threshold setting process determines a trade-off between false positive and false negative document candidates.

The work presented in [4] proposed to use compressed posting lists organized in blocks. Each block stores the upper bound for the documents inside that block in uncompressed form, thus enabling to skip large parts of the posting lists. This is a simple idea which drastically reduces the cost of the WAND algorithm but does not guarantee correctness because some relevant documents can be lost. To solve this problem, the authors proposed a new algorithm that moves forward and backwards in the posting lists to ensure the top-$k$ document results.

Independently, the same basic idea has been presented in [30]. 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 top results.

Recently, the work presented in [31] performs experiments on the WAND and Block-Max Index with a larger set of blocks, and posting lists are ordered according to static scores. Results show better performance than previous algorithms.

3. Distributed Ranking Process

Web search engines are composed of a large set of processing nodes and one or more broker machines in charge of sending user queries to nodes for answer calculation and blending of results. In this work we consider a Web search engine in which there is one broker and $P$ top-$k$ nodes, where $P$ indicates the level of document partitioning considered in the distribution of the document collection. Each top-$k$ node has its own inverted index, where the posting lists point to local documents. When a new query arrives, the broker sends the query for evaluation to each node (process also known as query broadcast). Then, the nodes work to produce query answers and pass the results back to the broker.

A WAND-based search engine in this context considers parallel term iteration over each top-$k$ node. If a document satisfies the threshold condition, then it is retrieved and included into the heap of the top-$k$ document results. Notice that document identifiers need to be sorted allowing for posting lists iteration at linear time (i.e. proportional to the posting list length). Finally, a full evaluation process is conducted over the set of candidate documents determined by the WAND algorithm, and the global top-$k$ results are determined.

There are 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 could 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 local inverted indexes, replicating upper bounds across the nodes. We consider that the second alternative is more recommendable because allow to skip more documents in the WAND iteration process.

3.1. Proposed Distributed Algorithm

Now, we detail the proposed algorithm which aims to obtain accurate results while reducing the number of operations (scores calculations), and the communication among nodes when executing the distributed WAND computations. The algorithm works as follows: (1) we assume a document partition index, (2) in the first step, the top-$k$ nodes sends $N = (k/P + \alpha) < k$ documents results to the broker machine, (3) the broker machine detects and discards those nodes that do not have more relevant documents, (4) the broker requests for more documents to
the non discarded nodes. Our hypothesis is that $k/P$ is the average number of document results retrieved by each top-$k$ node as the document collection tends to be evenly distributed among the $P$ partitions.

Formally, given a set of top-$k$ nodes $P = \{p_1, p_2, \ldots, p_P\}$, 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: $\{ <d_{x,1}, sc_{x,1}, i >, \ldots, <d_{x,k}, sc_{x,k}, j >, \ldots, <d_{w,N \times P}, sc_{w,N \times P}, s >\}$. 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 $x$ and $d_{y,k}$ represents the k-th document result identified by $y$. 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 1):

1. If $\exists d_{x,t}, sc_{x,t} \in p_i$ with $sc_{x,t} > sc_{x,k}$ (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 $sc_{y,n} < sc_{y,k}$ 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. The upper-bound for those requested documents is given by $sc_{y,n}$ and the lower-bound is given by $sc_{y,k}$. In other words, the broker machine requests to $p_i$ documents with scores in the range $[sc_{y,k}, sc_{y,n}]$. 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 $|P| = 3$, and $k = 4$. 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 in the range $[sc_{b,2}, sc_{e,5}]$. Note that in the first step the broker machine ask to all nodes a total of $N = 2$ document results and therefore 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 ask 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 $[sc_{b,2}, sc_{d,4}]$ to $p_1$ to discard documents with scores outside this range.

The $\alpha$ 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 $\Delta$, we warm up the system by running an oracle algorithm which predicts the optimum value of $\alpha$. 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, the posting list size of each term and the $\alpha$
parameter \((q_a = \langle t_1, t_2, L_1, L_2, \alpha_{p_1}, \ldots, \alpha_{p_P} \rangle)\). 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 \(\Delta_j\) is the current period of time, we estimate the \(\alpha_{p_i}\) values for a query \(q_a\) by applying the rules:

1. If \(q_a\) was processed in \(\Delta_{j-1}\) we use the value \(\alpha_{p_i}\) 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 \(\Delta_{j-1}\). Then if \(X \neq \emptyset\), 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 and use its value of \(\alpha\).
3. Otherwise, for each top-\(k\) node \(p_i\) we use the average \(\alpha_{p_i}\) value registered in \(\Delta_{j-1}\).

4. Evaluation

4.1. Data preparation and Experiment Settings

We experiment with a query log of 36,389,567 queries submitted to the AOL Search service between March 1 and May 31, 2006. We pre-processed the query log following the rules applied in [32]. The resulting query trace has 16,900,873 query instances, where 6,614,176 are unique queries and the vocabulary consists of 1,069,700 distinct query terms. 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. Figure 2.(a) shows that the query frequencies follow the Zipf law. Figure 2.(b) also shows that the size of the posting lists follows the Zipf law.

We also observe in Figure 3.(a) that the total running time reported by the WAND algorithm required to compute the similarity among documents and a query is dominant over the time required to update the heap. Computing the similarity is less expensive, like 15% of the time required to update the heap. But, the number of heap updates is much lower than the number of similarity computations. Figure 3.(b) at left shows the average number of heap updates and similarity computations per query. Each query performs 0.1% heaps updates of the total number of operations with \(P = 32\) and top-10 document results. This percentage increases to 1% with \(P = 256\) because the size of the posting lists are smaller and the WAND algorithm avoids computing score for non-relevant documents.

Results of Figure 3.(b) at right show the variance of the number of score computations. The variance also decreases with more top-\(k\) nodes, because posting lists are smaller and the upper bound (\(UB\), which is the maximum frequency of the term in all documents) of each term of the query tends to be smaller. Thus, the top-\(k\) nodes perform fewer score computations.

To study how the proposed algorithm performs under different operating conditions, 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. For the top-1000 case a total of 6,000,000 queries were considered for evaluation.
We evaluate the proposed algorithm in a distributed cluster of 16 nodes with 32-core AMD Opteron 2.1GHz Magny Cours processors, sharing 32 GB of memory. We measure the number of score computations and running time. 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 $UB_t$.

All experiments were run using an inverted file of 27Gb in main memory.

4.2. Estimation of the Parameter $\alpha$

In this section we compare the performance achieved by the proposed algorithm against the state of the art baseline algorithm which always request $k$ document results per query to each top-$k$ node. We also, evaluate the proposal $\alpha$-estimation algorithm against three algorithms: (a) Maximum, (b) Average and (c) Minimum. To avoid keeping information about old queries, all $\alpha$-estimation algorithms evaluated in this section keep a cache memory with the information about the queries processed in the recent past intervals of time. The three algorithms are:

- **Maximum**: This algorithm records the maximum $\alpha$ parameter computed for each query and each top-$k$ node in the previous interval of time $\Delta_{i-1}$. The aim is to reduce the number of steps required to compute the global top-$k$ document results for each query. However, increasing the number of document results ($\alpha$ parameter) requested to each top-$k$ node has a negative impact on the Web search engine performance, when this number is larger than the local top-$k$ documents retrieved by each top-$k$ node. In other words, each node has to build a larger heap (more memory is required) and therefore the number of score computation will be increased as well.

- **Average**: This algorithm records the average $\alpha$ parameter computed for each query and each top-$k$ node in the previous interval of time $\Delta_{i-1}$. The aim of this algorithm is to reduce the impact of asking a large number of document results to the top-$k$ nodes.

- **Minimum**: This algorithm records the minimum $\alpha$ parameter computed for each query and each top-$k$ node in the previous interval of time $\Delta_{i-1}$. In this case, the aim is to minimize the impact of asking a large number of document results. However, this scheme tends to increase the probability of requiring a second step of the algorithm to obtain the global top-$k$ document results.

Figure 4 shows normalized number of score evaluations and heap updates. 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. Figures 4.(a) and 4.(c) show results for top-100. Comparing results with the baseline algorithm, the proposed $\alpha$-estimation algorithm reduces the number of scores by 50% in best case for $P = 16$. The number of heap updates is significantly reduced by 73%. The Maximum algorithm obtains results close to the baseline due to this algorithm tends to reduce the number of steps required by the distributed ranking algorithm but it over-estimates the size of the heap required by each top-$k$ node to compute the local document results (as
shown in Figure 4.(c)). Hence, more score computations are performed. The Average algorithm presents a trade-off between the number of steps and the heap memory size (and therefore the number of score computations). However, results show that the average $\alpha$ parameter tends to be close to the maximum.

Figures 4.(b) and 4.(d) present the number of scores and heap updates performed for top-1000. Results show similar trend as the top-100 results, but the gain obtained by our proposal is smaller. In this case, the proposal reduces the total number of scores computation by 16% with $P = 16$ and the number of heap updates keeps a good gain of 69%.

Therefore, results show that our $\alpha$-estimator algorithm can reduce the computational cost and memory usage in the top-$k$ nodes. The main reason is that our algorithm can adjust the value of the $\alpha$ parameter for each query and top-$k$ node based on the length of posting lists and historical information.

4.3. Performance Evaluation

Figure 5 shows normalized running time reported by the top-$k$ nodes. In this graphic we show the communication cost plus the computation cost in terms of running time for both the baseline and the proposed algorithms. For a small top-$k$, the proposal 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 5.(a) which shows that the proposal reduces communication cost logarithmically. Also, as we explain above, Figure 4.(a) shows that the proposal reduces the average number of score computation by 50% with 16 nodes and Figure 4.(c) shows that for the same number of nodes the number of heap updates is reduced by almost 73%.

With a larger number of document results (Figure 5.(b)) the gain achieved by the proposed algorithm (in terms of running time) is reduced 10% in average. Again, these results are validated in Figure 4.(b) and Figure 4.(d) for the number of score computations and the number of heap updates respectively. In this case, the proposed algorithm reports 16% less number of score computations than the baseline and 69% less heap updates for a total of 16 nodes. The communication cost is improved by the proposal due to less than $k$ document results are requested per query per top-$k$ node.
Fig. 5. Running time in terms of communication and computation costs reported at the top-k nodes side for (a) top-100 and (b) top-1000 document results.

Fig. 6. Running time reported at the broker side for (a) top-100 and (b) top-1000 document results.

With both large and small top-k values, the computation cost achieved by the distributed ranking algorithm is about 90% of the communication cost. Hence, these results denote that it is relevant to reduce the number of score computations preformed by the distributed top-k nodes.

To understand the effect of using a larger value of k we have to remember that posting lists follow the Zip law as shown in Figure 2.(b). 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 stored 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.

Finally, Figure 6 shows the running time reported at the broker side for top-100 and top-1000 document results. In the baseline approach, the broker machine performs the merge operation over a total of k * P document results. Therefore, the cost reported by the broker machine tends to be increased with P. On the other hand, the proposed algorithm requests ((k/P) + α) * P = k + α * P for each query. In practice, the α value tends to be small.

5. Conclusions

In this paper, we have described and evaluated a distributed version of the Block-Max WAND algorithm. The proposed algorithm uses a α parameter to estimate the number of document results retrieved by each top-k node. This parameter is dynamically set for each query and is based on information from the recent past. Results show that computation cost consumes like 90% of the total running time and therefore it is relevant to devise efficient algorithms to reduce the score calculation in each top-k node. Moreover, results show that the proposed algorithm is able to achieve this goal by significantly reducing computation and communication cost.
References