Performance Evaluation of Improved Web Search Algorithms

Esteban Feuerstein[1], Veronica Gil-Costa[2], Michel Mizrahi[1], Mauricio Marin[3]

[1] Universidad de Buenos Aires, Buenos Aires, Argentina
[2] Universidad Nacional de San Luis, San Luis, Argentina
[3] Yahoo! Research Latin America, Santiago of Chile

Abstract. In this paper we propose an evaluation method for parallel algorithms that can be used independently of the used parallel programming library and architecture. We propose to predict the execution costs using a simple but efficient framework that consists in modeling the strategies via a BSP architecture, and estimating the real costs using as input real query traces over real or stochastically generated data. In particular we apply this method on a 2D inverted file index used to resolve web search queries. We present results for OR queries, for which we compare different ranking and caching strategies, and show how our framework works. In addition, we present and evaluate intelligent ranking and caching algorithms for AND queries.

1 Introduction

Inverted files [1] are the most widely used index data structures to cope efficiently with high traffic of queries upon huge text collections. An inverted file consists of a vocabulary table which contains the set of relevant terms found in the text collection, and a set of posting lists that contain the document identifiers where each term appears in the collection along with additional data used for ranking purposes. To answer a query, in a Web search engine context, a broker machine sends the query to a set of search processors in order to get the set of documents associated with the query terms. These processors perform a ranking of these documents in order to select the top-K documents.

Many different methods for distributing the inverted file onto P processors or computers 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 (a.k.a local index) and term partition (a.k.a global index). Variants of these two basic schemes have been proposed in [5] which focus on optimizing for particular situations.

In a previous paper [2] we introduced a novel distributed architecture for indexing, named the 2D index, that consists in arranging a set of processors in a two-dimensional array, applying term-partitioning at row level and document-partitioning at column level. We showed in that paper that, for AND queries, choosing the adequate number of rows (R) and columns (C) given the available number of processors it is possible to obtain significant improvements in the performance against the basic architectures of term and document partitioning with the same number of processors.

In this paper we propose to use the bulk-synchronous model of parallel computing (BSP) [8] as the cornerstone of a simple but efficient framework in which we blend empiric analysis and theoretical tools to predict the cost of a real system executions, which we apply to variations of the 2D index. In that framework, we combine the usage of real input logs, average-cost analysis of certain operations and stochastic generated values to compute realistic estimations of the cost of the real system.

Our framework has two important features. The first is precisely the ability to scale the system architecture obtaining reliable results without the need to acquire additional resources, permitting for example to predict what happens when there are thousands of processors with billions of documents. This is an important issue as deploying and running algorithms on large clusters to determine their efficiency and performance is extremely expensive, and prohibitive in many cases, as it is sometimes impossible to access the required resources. Our methodology enables us to validate the feasibility and convenience of an architecture or algorithm more easily than constructing a real system or building complex simulations involving the use of queuing theory, real-time managing techniques and, not less involved, the real implementation of all the operations.

The second important characteristic is the possibility of applying this framework to any kind of system, meaning synchronous or asynchronous, with distributed or centralized ranking. Besides, in this way we can ensure a fair comparison between algorithms independently of the hardware used to run the codes and particular implementation details. The result is that our framework based on the BSP model may be seen as a unifying setting to fairly compare different algorithms and implementations, even other researchers' proposals.

Apart from BSP, several general-purpose parallel models have been presented in the literature, like LogP or QSM [8]. As it has been proved, the different models can be reciprocally simulated efficiently, and we base our evaluation framework on BSP, which has a well developed literature, libraries and other resources that provide an efficient way of evaluating algorithms for a wide variety of problems. We insist on the fact that we are not choosing one model over the others as a real platform over which the algorithms would actually run, but choosing one of them as the "computational model" that will be used to compare the behavior of different strategies or configurations.

The rest of this paper is organized as follows: Section 2 presents generalities about distributed search architectures and some improvements proposed to speedup query response time. Some of these improvements were introduced in previous papers, while those in Sections 2 and 2 are original contributions of this paper. Section 3 presents our methodology using the BSP model. Section 4 shows the features of the query log and the parameters used in the experiments, as well as the results of the experiments we conducted. Finally, Section 5 presents our conclusions.

2 Search Architecture

Queries are introduced to the system via a receptionist machine named the *broker*. The broker is in charge of routing the queries to the clusters *search processors* and receiving the respective answers. It decides to which processor a given query will be routed, normally by using a load balancing heuristic. The parallel processing of queries consists in a series of operations that will be executed in different processors, the results of which will be combined to get the final answer to the query. These are the primitive operations of broadcast or communication, list intersection, list merging, list ranking, etc. Each of these operations has a cost depending on its actual parameters. The combination of these costs for all the queries in an execution environment conforms the total cost of a sequence of queries.

Several features may be added to the basic setting just described to further speed up the search process or provide fault tolerance, as for example the already mentioned partitioning schemes, replication, particular policies for routing the queries among the processors, different ranking algorithms, and various types of caches. In the following we describe some of these standard features as well as some original ones that we introduce in this paper, and will be tested using our evaluation framework.

Partitioning and Replications. We already mentioned in the introduction the exploitation of parallelism over the text database that is achieved by different variations or evolutions of the two "extreme" settings of document and term partitioning, in particular the 2D scheme of [2]. Fault-tolerance is generally supported by means of replication. To this end, and assuming a P-processor parallel search cluster, D copies for each of the P processors are introduced. This can be seen as adding an extra dimension to whichever configuration was initially chosen. Then the system can be seen as a $P \times D$ matrix in which each processor in the D dimension maintains an identical copy of the respective partition of the index. Upon receipt of a query by the broker node in the cluster, the broker sends it to all the processors that must evaluate it, and for each of them one replica is selected. The processors hit by the query work on their part of the solution to then merge all results to produce the final answer.

The basic model of concurrency in here is the distribution of the query search among the distributed knowledge of the P principal nodes and also internally to each of the D replicas, which contain the same data (global knowledge). The question arises on how to utilize the available resources to best execute search algorithms in the $P \times D$ arrangement of computers. A naive approach would be to attempt a text data partitioning throughout the system and hope that the multiplicity of computational resources can provide a speedup by virtue of simultaneous computations. Another approach would be to share in a more subtle way the resources of the D replicas. We explore these cases in this paper.

Caching. A typical way of reducing the number of processors involved in each query processing is by using caching in several ways. First, the broker machine may have a results cache (RCache) of previous queries, to avoid repeatedly computing the most frequent ones. The state of the art strategy for the results cache is the SDC strategy [6], which keeps a static section to store the answer for queries that are frequent along large periods of time and a dynamic part with the objective of capturing queries that become very popular during short periods of time. Additionally, we can reduce the number of processors participating on each query by sending the query only to a selected group of them as in [7] or [2]. The latter works propose to keep a compact location cache (LCache) at the broker machine.

Processors may maintain also caches of posting lists to reduce secondary memory accesses. Examples of these can be found in [6, 4, 3]. Alternatively there is an intersection cache [4] which keeps in each search node the intersection of the posting lists associated with pairs of terms frequently occurring in queries. The intersection operation is useful to detect the documents that contain all of the query terms, which is a typical requirement for the top-K results in major search engines. Figure 1 shows the five caches defining a hierarchy which goes from the most specific pre-computed data for queries (results cache) to the most generic data required to solve queries (posting list cache). When a new query arrives to the system, if the broker finds the new query in its RCache, the query is answered with no further computation. If that is not the case, the broker searches the query in its LCache to reduce the number of processors that will be involved in its processing. Finally, the broker selects a processor to send the query to. After that, the query is processed according to the partitioned index approach and the ranking scheme.

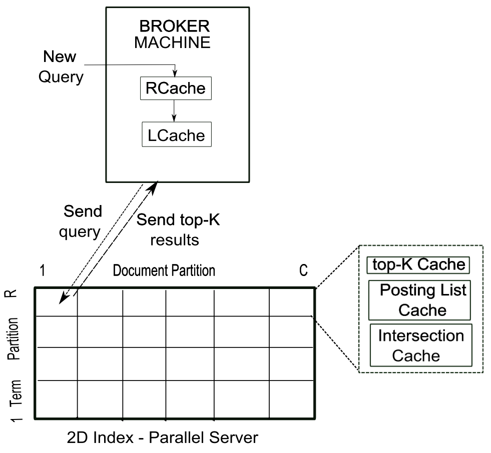

Fig. 1. Query processing using a 2D index and a five level cache hierarchy.

Distributed vs. Centralized Ranking. There are basically two types of rankings under distributed architectures: distributed or centralized ranking. For clarity we explain the ranking schemes without considering caching at any level. In the distributed ranking scheme, the broker sends the query to a *manager-merger* processor. Then, this processor sends the query to each other processor involved in the processing of the query (i.e. each processor owning terms of the query if the term partition scheme is used, or all processors when we use the document partition scheme). Then, these processors fetch the posting lists for each term and computes their union. In the last phase, they rank the local results and send the top-K documents identifier to the manager-merger processor which merges the partial results in order to build the global top-K results that are sent to the broker.

Using the centralized ranking strategy, there is one processor (named the manager-ranker processor) that globally ranks the pieces of posting lists that are sent to it by the processors involved in the processing of the query. In general it is enough that each involved processor sends a small list to the manager-ranker processor to allow him compute a barrier S_{MAX} that document frequencies for a certain term must surpass to qualify for the final result list. When posting lists are kept sorted by frequency, this allows to directly skip whole list-tails consisting of documents with frequencies smaller than the barrier.

To map these basic strategies to a 2D architecture we need first to describe the main computational path that each query will follow. At column-level a union (resp. intersection) is computed by sending the shortest list from the processor that owns it to the processor owning the second shortest list, who computes the union (resp. intersection) with its term (or terms) and sends the result to the following one and so on, until the last processor is reached. When using the distributed ranking scheme we perform a ranking over local data before sending partial results to the next processor. This allows reducing the communication cost. This mechanism requires that some processor, with the information of the lengths of the posting lists at that column, prepares the route the query will follow. Then the basic ranking strategies become:

Distributed Ranking for 2D index:

1. The broker machine sends the query to the manager-merger processor using a hash function over the query terms.
2. The manager-merger sends the query to a random processor at each column (which we call the manager-ranker).
3. The manager-ranker of each column prepares the route for the query and sends it to the first processor of that route.
4. Each processor that receives the query fetches the postings lists of its term(s), computes their union or intersection, makes the ranking and sends the result to the following processor in the routing list.
5. The last processor of the routing list sends the results to the manager-ranker of the column, which performs the final distributed ranking (in order to obtain the top-K postings of the column).
6. The manager-merger receives the results of all the columns and merges them to obtain the global top-K results.

Centralized Ranking for 2D index:

1. The broker machine sends the query to a manager-ranker processor.
2. The manager-ranker sends the query to a random processor at each column (which we call the column manager).
3. The column manager of each column prepares the routing list for the query and sends it to the first processor of that route.
4. Each processor that receives the query fetches the postings lists of its term(s), makes the union or intersection with the partial list it receives from previous processors from the same column, and sends the result to the following processor in the list.
5. The last processor sends the results to the column manager, who sends the results to the manager-ranker.
6. The manager-ranker receives the results of all the columns and ranks them.

Clairvoyant Distributed Ranking for AND queries. The main advantage of the centralized ranking approach with respect to the distributed one arises from the possibility of excluding documents that would not do it to the top-K results, by means of initially considering short prefixes of all the lists and then updating the global barrier as needed. The price to be paid for that gain is the extra communication and overhead due to the extra need of interaction among the processors. This suggests that it would be very valuable to know a-priory how many elements to consider of each list. In that case we could send exactly the required postings and do the minimum work. Considering that each successive intersection that is computed will further reduce the length of the resulting lists, just limiting the number of elements of the *first* list that must be sent would be useful to reduce the total computation and communication time. We can exploit the information of the query logs to *estimate* the number of elements of the shortest list that should be sent at the beginning of the computation to ensure that the appropriate number of results will be found. Of course we cannot be sure that that number will be effectively obtained, but we have two possible solutions to that. First, we could choose the number of postings to send so as the probability of having enough elements exceeds a certain threshold, admitting the (rare) possibility of having less results than needed. A second idea would be to choose that number and, in case of not having enough elements in the intersection, behave as in the centralized case, i.e. asking for extra results to the first processor and repeating the process. We implemented the first of these ideas and compared it with the Distributed and Centralized strategies above.

Clairvoyant Distributed Ranking:

1. The manager-merger of the query sends it to a random processor at each column, that will be the manager-ranker.
2. The manager-ranker of each column prepares the route for the query and sends it to the first processor of that route.
3. Each processor that receives the query fetches a certain number of postings of its term(s) according to the estimation, intersects them with the partial list it receives and sends the result to the following processor in the list.
4. The last processor sends the results to the manager-ranker of the column, which makes the ranking and sends the results to the manager-merger.

# of terms	# of postings sent
2	$(K/C + 1000) * 5$
3	$(K/C + 1000) * 22$
4	$(K/C + 400) * 50$

Table 1. # of postings of the shortest list sent to ensure 90% of the queries have enough results.

5. The manager-merger receives the results of all the columns and merges them.

We now explain how we did to compute the parameters of Clairvoyant Ranking, i.e. the number of postings of the shortest list that need to be sent to ensure that with high probability the top-K elements will be found. We used the same data sets and query logs described in Section 4 to compute the average length of the intersection for queries with different number of terms, and analyzed the results as a function of the length of the shortest posting list involved in the intersection. Figure 2 shows the resulting values for 2-term queries. We then computed

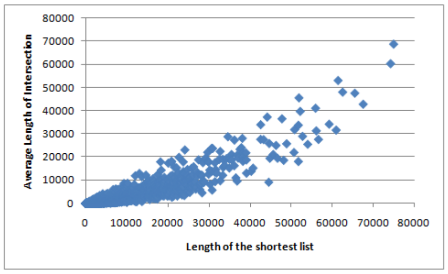

Fig. 2. Average length of the intersection vs length of the shortest list, two-term queries

min-square approximations, and another linear function that is a lower-bound to the average length for more than 90% of the queries. The inverse of the linear lower bound acts as an upper bound to the average length of the resulting list, so fetching that number of postings from the shortest list would, on average, be enough for more than 90% of the queries. We decided to fetch twice that number to cope with those intersections whose results are smaller than average. The values needed to have top-K results in at least 90% of the queries are those depicted in table 2. Those were the values used in the experiments described in section 4. For queries with 4 or more terms, we used the same values as for 4 term intersections.

Re-using Intersections. Although as we mentioned before replication is normally introduced to improve fault tolerance and throughput (due to the added processing capacity), the D replicas of a processor may contain intersections between pairs of terms that could be useful to solve new queries. It is then pertinent to try to exploit this advantage by first making a sort of tour along the replicas to "see" if there is something useful to solve a current query in the column. That way, the caches of the replicas behave in some way as a common shared resource. In the following we propose a strategy to achieve this goal.

Each processor will be responsible of a set of terms, given by a hashing function H. This function H must be such that it distributes terms among processors trying to cluster together terms frequently co-occurring in queries. H is extended from terms to queries, in some reasonable way (for example, defining $H(q) = H(t)$, where t is the term of q with the longest individual posting list among those forming q). Each processor p will have a cache of posting lists, divided in two parts: a static part where it will hold the most useful posting lists (according to the logs), among the terms t such that $H(t) = p$, and a dynamic part, both for terms t such that $H(t) = p$ and for terms t such that $H(t) <> p$. Besides, each processor will maintain a dynamic cache of intersections (pair intersections), which will contain only queries or partial queries q such that $H(q) = p$. Upon computing complex intersections, p will eventually cache pair-intersections in it's own cache and also may cache pair-intersections in other processor's intersection cache. Finally, each processor will have a results-cache, where it will hold results of previous queries assigned to it. The following algorithm shows the behavior of a processor upon receiving a query.

```
- Given a query $q = t_1, \ldots, t_r$ (abusing notation we denote by $q$ also the set of terms $\{t_1, \ldots, t_r\}$)
- $q$ is assigned to $p = H(q)$
- If $p$ has $q$ in its results cache then $p$ updates the validity of $q$ in cache and answers
- else
  If $p$ has $q$ in it's own intersection cache then $p$ updates the validity of $q$ in cache and answers
  - else
    - Let $T_Y$ be the set of terms of $q$ that appear already in an intersection with another term of $q$
    in the intersection cache of $p$
    - If $T_Y = q$ (i.e. $p$ has all the pairs needed to compute the answer in its own cache)
      - $p$ computes the answer, updates the validity of all the used pairs, caches $q$ in the results
      cache and answers
    - else
      - Let $T_N = q - T_Y$
      - Let $T_N^p = T_N \cap H^{-1}(p)$ (i.e. the set of terms that are not cached and must be looked
      for in $p$)
      - Let $\overline{T_N^p} = T_N \cap \overline{H^{-1}(p)}$ (i.e. the set of terms that are not cached and must be looked
      for in other processors)
      - $p$ asks processors in $H(\overline{T_N^p})$ for intersections or posting lists useful for computing $q$
      - Each processor answers, and blocks the contents of the cache for a superstep
      - Let $T$ be the set if terms in $\overline{T_N^p}$ such that NO useful intersections are kept in the caches.
      - $p$ couples together all the terms in $T_N^p$ (putting together pairs with maximum frequency or
      maximum frequency $\times$ length of the posting lists), computes and caches their intersections.
      - If $|T_N^p|$ is odd, the uncoupled term is coupled with one term in $T$, $p$ computes their
      intersection and caches the result where it corresponds according to $H$.
      - $p$ couples together the rest of the terms in $T$ (putting together pairs with maximum
      frequency or with maximum frequency $\times$ length of the posting lists), computes the
      intersection of each pair and caches each result where it corresponds according to $H$.
      - $p$ computes the query using the information already in the caches of the other processors
      and its own cache
      - $p$ communicates to the other processor which information it has used, the other processors
      update the validity of the information
      - $p$ caches $q$ in the results-cache and answers
```

We must also say what a processor does in case another processor makes a request: If I have a pair or a partial result useful for the query: (a) Answer to the asking processor, (b) Block it during one superstep, (c) When the processor tells me that it has used the provided pair or result, update the cache accordingly.

Finally, we must say what's the meaning of "caches the result where it corresponds according to H." When a processor computes a pair that does not belong to him accordingly, it will cache it in the owner's cache. This means that upon a request of caching, the other processor p' must: (a) Receive the list, (b) Make place in the cache according to the caching policy, and (c) Update the cache with the new list.

3 A Cost Estimation Methodology

Our cost estimation framework is based on the bulk-synchronous model of parallel computing (BSP) [8], where computation is organized as a sequence of supersteps, in each of which the processors may perform computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors.

The total running cost of a BSP program is the cumulative sum of the costs of its supersteps, and the cost of each superstep is the sum of three components: computation, communication and barrier synchronization. Computation cost is given by the maximum computation cost of a processor in the superstep, and will be denoted by a quantity w. This will include also disk accesses, whose unit cost will represented by a constant δ. Besides, we note that just for participating in the processing of a query, a processor incurs a certain overhead β, which must also be taken in consideration when accounting for the actual cost of the query. Communication cost is given by the maximum number of word-length messages sent or received by a processor during the superstep (denoted by h) times the cost of communicating one word (γ). Hence this will be denoted by the product $h * \gamma$. Finally, we will use L for the synchronization cost. Parameters γ, L, β and δ take into account the characteristics of the particular computer architecture that is emulated.

The impact of uneven load balance in the cost is taken into account by considering maximum cost among all the processors for each of the costs above. Moreover, we can compute the load work and communication efficiency using this model as follows: for any performance measure X, in each superstep we compute its value for each processor, and define BSP efficiency for X as the ratio average(X)/maximum(X) ≤ 1, over the P processors. This gives an idea of how well we distribute the workload among processors.

We end this section with the list of the primitive components that together conform the costs (computation and communication) that are charged to each query:

- $t_i(x, y)$: Expected time employed by a processor to compute the intersection of two lists of lengths x and y respectively.
- $t_m(x)$: Expected time employed by a processor to merge a set of lists of total length x.
- $t_r(x)$: Expected time employed by a processor to rank a list of length x.
- $I(x, y)$: Expected length of the intersection of two lists of length x and y.
- γ: time employed to transmit a unit of information from one processor to another.
- δ: cost of accessing a unit of information in disk.

– β: overhead due to the participation of a processor in a query.
– L: time for the barrier synchronization at the end of each superstep.

4 Experimental setting

We did our experiments to estimate the cost of our algorithm using P=512 processors with different combinations of $P = C \times R$. The number of rows ranged from 1 (local index) to 512(global index). Queries where selected from a query log with 36,389,567 queries submitted to the AOL Search service between March 1 and May 31, 2006. We preprocessed the query log following the rules applied in [3] by removing stopwords and completely removing any query consisting only of stopwords. We also erased duplicated terms and assumed that two queries are identical if they contain the same words no matter the order. The resulting query trace has 16,900,873 queries, where 6,614,176 are unique queries and the vocabulary consists of 1,069,700 distinct query terms. 75% of these queries occur only once, and 12% of them occur twice. The index was built using a 1.5TB sample of the UK's web obtained from Yahoo! searches.

We used caching in several ways as we mentioned before. For the experiments we set the overall cache sizes so as to maintain information of at most 10% of the processed queries. In all the cases the broker machine has a cache of results of previous queries (RCache), and the processors maintain caches of posting lists to reduce secondary memory accesses. In the experiments reported in Section 4 we considered an additional level of caching introduced in Section 2. For the algorithms using replication, to improve the throughput and fault tolerance (see for example [6]), we set the number of replications $D = 4$, so in these cases we have $P \times D$ processors available to compute queries.

We defined particular costs for the different primitive functions and constants, based on benchmarking runs we did on the same collection. The values are expressed relative to a base-line in terms of ranking time defined as $t_r(x) = x$. Merge operations require in average $t_m(x) = x/6$. An intersection between two lists of lengths x, y can be done in time proportional to $\min(x \log y, x + y)$. However, the constants discovered in the benchmark make us count $t_i(x, y) = \min(x \log y, x + y)/6$. For I, the random variable that models the expected length of the intersection of lists, we used a power-law distribution, with parameters varying according to the number of lists being intersected. The actual values of these parameters were determined via sampling from real queries. We also established that the communication cost for transmitting a query is, on average, 1/4 of the time required to transmit a K-word list. For $P = 512$, the communication time for one unit under a heavily loaded network was $\gamma \approx 0.453$, and the overhead constant $\beta \approx 1.5$. The cost for the synchronization was $L \approx 8$.

We give as example a succinct explanation of the costs charged to each processor in the Distributed Ranking case for AND queries, the other cases are analogous. For simplicity we omit the parameters of random variable I.

Distributed ranking costs:

1. Send the query to a processor at each column. Computation: β; Communication: $K/4 \times \gamma$

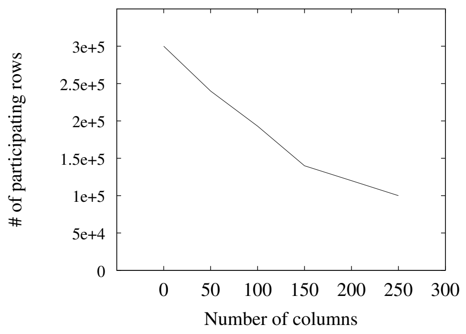

Fig. 3. Number of participating rows as a function of the number of columns

2. Send the query to the first processor in the route. Computation: β; Communication: $K/4 \times \gamma$
3. Fetch and intersect.(At each participating processor) Computation: $\delta \times K$ for disk access + $t_i(I)$ + β; Communication: $I \times \gamma$.
4. Rank. Computation: t_r+ β; Communication: $I \times \gamma$
5. Merge. Computation: t_m+ β; Communication: $K \times \gamma$

OR Queries with iterations. In this experiment we considered Centralized and Distributed ranking for OR queries, with an additional restriction imposed in order to balance communication, secondary memory recovery and computation costs among processors: we limit the number of operations of each type a processor may execute in each superstep. Operations not performed in one superstep due to that limitation, are delayed until the next one. Thus, the processing of a query is separated in successive *iterations*. The estimation of the costs is performed in each superstep, charging always the maximum cost for each operation. We used the RCache at the broker and top-K cache plus posting list cache at search processors' side.

As the number of columns augments, the probability that the query terms are located in fewer rows augments (see Figure 3), and therefore the communication costs tends to decrease, while the computation overhead increases because each query is processed by more columns. There must naturally exist a trade-off between the two extremes of the 2D index. Figure 4[Left] shows the cost for the centralized and distributed ranking algorithms when processing OR queries with an asynchronous system for different $P = R \times C$ combinations. Results show that when the index is working as a term partition (matrix size of 512×1) both ranking strategies tend to perform well, but a better configuration is 16×32. A document partition index has a higher cost due to the overhead of dealing with every query in all processors. The centralized ranking has a lower cost in each extreme of the matrix because it requires less iterations to finish the processing of a query when using Persin filters.

Figure 4[Right] shows the average efficiency measured as $\sum(w_{p_i}/Max_{wp})/P$ in each superstep, where w_{p_i} is the computation performed by processor p_i, and

Max_{wp} is the maximum computation performed in that superstep. Results show that using a large number of rows improves both ranking's efficiency, but when working with 512 columns and one row per column, the distributed ranking scheme reports a better efficiency than the centralized one.

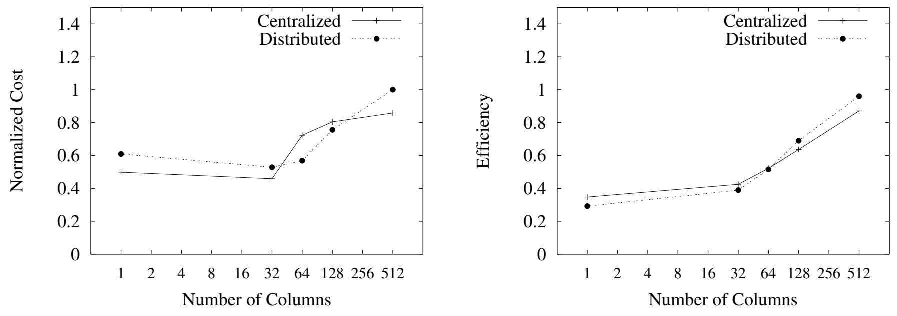

Fig. 4. OR queries [Left] Normalized costs. [Right] Computation efficiency.

OR Queries with complete lists. Figure 5 shows results for OR queries when we accessed to the whole posting lists of each term to perform the ranking operation, i.e. without the iterations defined in the previous experiment. The same cache settings were used. As we may see, in this case the term partition index with centralized ranking presents high costs due to the imbalance it has both in communication (because it has to transmit complete posting lists from the search processors to the manager) and in secondary memory access (when retrieving the posting lists for each term). On the other extreme, the document partition index is more balanced in both costs, but its overhead is clearly higher. The Distributed ranking scheme presents more balanced communication with all configurations (ranking is performed in each processor before sending results). But the term partitioned index presents some imbalance in secondary memory retrieval as ranking is performed over larger posting lists. Again, we see the trade-off communication/overhead, with the optimal configuration realized by a 128×4 arrangement of the processors.

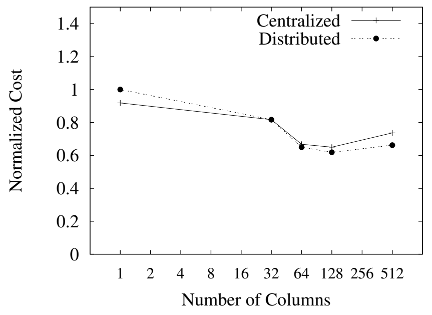

Fig. 5. Normalized costs for OR queries with complete lists.

Different ranking strategies for AND Queries. Figure 6 shows the performance of Distributed, Centralized and Clairvoyant strategies with different configurations. As we may see, the best configurations are those in which the 256 processors are arranged in 32 or 64 columns, depending on the ranking strategy, the optimal combination being the clairvoyant strategy with 64 columns. Again, as the number of columns augments, the probability that the query terms are located in fewer rows augments (see Figure 3), and therefore the communication costs tends to decrease, while the computation overhead increases because each query is processed by more columns. There is naturally a trade-off between the two extremes. Both the distributed and centralized approaches suffer from the inconvenient that each time a new group of postings must be considered for the intersection, they must be compared with *all* the previously fetched terms, with a cost that is almost quadratic in the required number of iterations. The clairvoyant strategy is free of that problem.

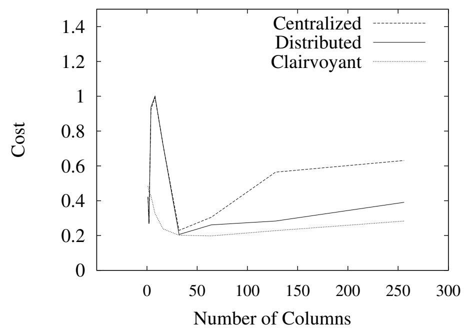

Fig. 6. Normalized costs vs. number of columns, for different ranking strategies

Intelligent Caching in a Replicated Local Index. In this section we report the results of an experiment in which we compare the effects of different caching protocols for AND queries in a $1 \times P \times D$ setting (i.e. a local index with replication, $D = 4, 8$). The base-line was the architecture having all the caching levels considered before, except the intersection cache (No Intersection). We then considered adding the intersection cache, but without any particular strategy for routing the queries and their corresponding caching decisions (Intersection). Finally, we considered the usage of intelligent distribution of the traffic among the rows, using the hashing function H as described in Section 4.2.1 (Intersection-h). Fig. 7(a) shows the throughput obtained by all three intersection algorithms. This graphics shows the improvement (almost 30% with K=128 and more notorious when with K=1024 is 60%) obtained when using a more intelligent algorithm to reassign the intersections operations. Fig. 7(b) shows the disk and CPU cost saving relative to an algorithm that uses no caching at all. As it may be seen, all three algorithms obtain visible differences in both aspects with respect to that base line. However, the use of intersection caches helps improve that in an extra 20%. The most significant savings are obtained for bigger values of D. In those cases, the number of disk accesses needed by the best algorithm is about 1/4 of the number needed without caching of intersections.

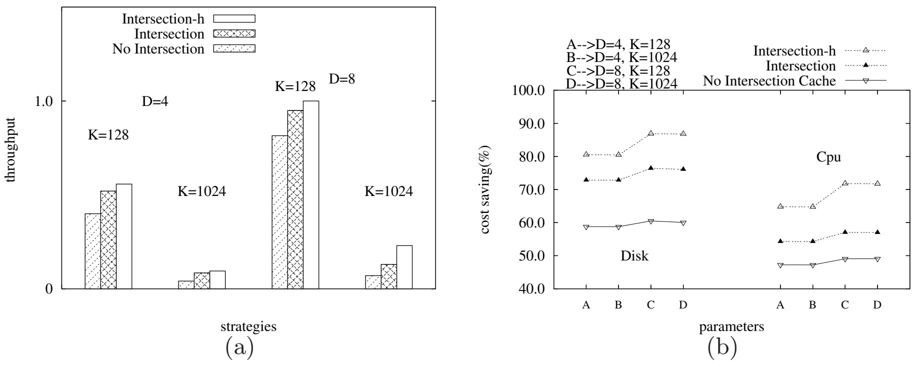

Fig. 7. Different caching strategies, AND (a) Throughput (b) Disk/CPU savings

5 Conclusions

We have presented a methodology for predicting the costs of different algorithms and architectures for distributed indexes of different sizes. We provided results for some interesting experiments regarding a novel two-dimensional architecture and some new ranking and caching strategies. The results obtained allow us to be optimistic on the further development and applications of the cost framework, as for example putting together the different features we have tested separately, like replication, caching, etc. The cost framework can be used for analyzing and comparing our work with other authors when evaluating the scalability of alternative strategies.

References

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley., 1999.
2. E. Feuerstein, M. Marín, M. Mizrahi, V. Gil Costa, and R. A. Baeza-Yates. Two-dimensional distributed inverted files. In *SPIRE 2009*, LNCS 5721.
3. Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW*, pages 431–440, 2009.
4. Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. *WWW*, 9(4):369–395, 2006.
5. M. Marin and G.V. Costa. High-performance distributed inverted files. In *CIKM 2007*, pages 935–938, ACM, 2007.
6. M. Marin, V. Gil-Costa, and C. Gomez-Pantoja. New caching techniques for web search engines. *ACM HPDC*, 2010.
7. Diego Puppin, Fabrizio Silvestri, Raffaele Perego, and Ricardo A. Baeza-Yates. Tuning the capacity of search engines: Load-driven routing and incremental caching to reduce and balance the load. To appear in *TOIS*, 2009.
8. V. Ramachandran, B. Grayson and M. Dahlin. Emulations between QSM, BSP and LogP: a framework for general-purpose parallel algorithm design, *J. Parallel Distrib. Comput.* 63 (2003) 11751192.