Sync/Async parallel search for the efficient design and construction of web search engines

Mauricio Marina\textsuperscript{a,b,*}, Veronica Gil-Costa\textsuperscript{a,c}, Carolina Bonacic\textsuperscript{d}, Ricardo Baeza-Yates\textsuperscript{a}, Isaac D. Scherson\textsuperscript{e}

\textsuperscript{a}Yahoo! Research Latin America, Santiago, Chile
\textsuperscript{b}Informatic Engineering Department, University of Santiago of Chile, Chile
\textsuperscript{c}Informatic Department, National University of San Luis, Argentina
\textsuperscript{d}Computer Architecture Department, Complutense University of Madrid, Spain
\textsuperscript{e}Department of Computer Science, University of California, Irvine, CA 92697, United States

\textbf{Article info}

\textbf{Article history:}
Received 13 August 2009
Accepted 2 February 2010
Available online 21 February 2010

\textbf{Keywords:}
Distributed indexing and parallel query processing
Information retrieval
Web search engines
Parallel and distributed computing

\textbf{Abstract}

A parallel query processing method is proposed for the design and construction of web search engines to efficiently deal with dynamic variations in query traffic. The method allows for the efficient use of different distributed indexing and query processing strategies in server clusters consisting of multiple computational/storage nodes. It also enables a better utilization of local and distributed hardware resources as it automatically re-organizes parallel computations to benefit from the advantages of two mixed modes of operation, namely: a newly proposed synchronous mode and the standard asynchronous computing mode. Switching between modes is facilitated by a round-robin strategy devised to grant each query a fair share of the hardware resources and properly predict query throughput. Performance is evaluated by experimental methods and two case studies serve to show how to develop efficient parallel query processing algorithms for large-scale search engines based on the proposed paradigm.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

One of the most challenging problems facing modern Web Server Clusters is the wide range of search traffic they are subject to over daily operations. Web Clusters are normally collections of modern computing nodes consisting of multicore processors for computing and a memory hierarchy for storage. Dynamic user-generated query traffic can range from a few hundred queries per second to perhaps tens of thousands of queries per second at peak times. Most frequent queries may be answered promptly and efficiently by keeping track of repetitive query results in special purpose caches. Results not found in cache are directly resolved by a selected set of processors (nodes) forming a sub-cluster. In normal operation, the aim is to determine the top-\(R\) results per query as quickly as possible and, from these results, construct a ranked answer web page to be presented to the user. For high query traffic, and given the huge volume of data associated with the web samples kept at each node, this task can involve the use of significant amounts of local and distributed resources such as processor utilization, disk and network bandwidth. Current search engines deal with peaks in traffic by including hardware redundancy, typically large enough so that at normal traffic the processors’ utilization is below 40\% [3].

* Corresponding author. Address: Yahoo! Research Latin America, Santiago, Chile.
E-mail addresses: mmarin@yahoo-inc.com (M. Marin), isaac@ics.uci.edu (I.D. Scherson).

0167-8191/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2010.02.001
Normally, arriving cluster queries are received by a broker machine that distributes them onto the processors. Search engines then use the standard asynchronous multiple master/slave paradigm to process the queries arriving from the broker. Each query is serviced by a master thread that is in charge of producing the results. This master thread can, in turn, contact slave threads located at other processors to obtain data for ranking of results. This scheme is prone to significant overheads and imbalance when the query traffic is high since multiple query threads compete for computing and/or storage resources locally or over the network. Experimental observations show that these inefficiencies can be reduced considerably by using query processing strategies that take advantage of the economy of scale present in situations where a large number of queries are resolved concurrently. Thus, peaks in traffic can be dealt with efficiently, enabling a reduction in hardware redundancy. The main challenge is how to manage the active queries across nodes of the cluster. A novel paradigm is introduced herein. The basic idea is to use the widely accepted standard asynchronous thread management strategy for light query loads and revert to a novel synchronous mode in heavy traffic situations. The synchronous mode is introduced here as a management strategy where one single thread is allowed to run in each cluster node and synchronization between node threads allows better control and minimizes access conflicts to local and/or networked resources to prevent imbalance.

The proposed design method for Web search engines is able to automatically switch between synchronous and asynchronous modes of operation depending on traffic load. The synchronous mode is configured to minimize performance degradation due to intervals in which the query traffic ranges from moderate to high. The asynchronous mode, the standard mode employed by conventional search engines, is better suited for cases with low query traffic. The switching between the two modes of operation is facilitated by properly organizing overall query processing so that software threads can be efficiently re-arranged to serve either mode. This is achieved by using a strategy properly dubbed and based on round-robin query processing in combination with an adaptive control algorithm devised to switch between modes.

To verify that the proposed Sync/Async Round-Robin query processing method indeed behaves as claimed, a detailed realization of it on two related example applications is presented. The first one is query processing upon distributed inverted files [2]. The standard practice in conventional Web search engines is local indexing in the sense that the text collection is evenly distributed onto the processors and an inverted file is constructed in each processor. Conjunctive (AND) queries can be solved efficiently in this scheme, namely queries requiring all terms contained in the documents that form part of the top-$R$ results. Global indexing, namely the case in which the index is constructed considering the whole text collection and evenly distributed onto the processors, is more useful for disjunctive (OR) queries such as the ones required in photo sharing search systems based on tags and small user texts.

The conventional approach to document ranking on global and local inverted indexing is what in this paper is called local ranking. It represents a case in which the ranking of a given query is performed in the processors that contain sections of the index that are pertinent to the query terms. As a complementary proposal that gets the best performance from the proposed method, this paper also introduces global ranking which is a less intuitive case in which the involved sections of the index are sent by fetcher processors to designated ranker processors where they are merged and passed through their document ranking routines. Each processor is ranker of a different subset of the queries being resolved and for each ranker the other processors act as fetcher processors that feed it with data. Experimental results show that global ranking is efficient for early-termination ranking methods as termination takes place quite before than in the local ranking counterpart and thereby less hardware is employed to solve each query on the average. The same is observed in the second example studied in the paper.

The second application example is a metric-space index which is useful to retrieve the terms that are most similar to a given query term. This kind of index can be used for spelling operations in search engines and for other related applications such as image retrieval. Like in the inverted file case, the paper illustrates how this index and respective query regime can be accommodated into the proposed Sync/Async Round-Robin query processing method by using (a) either global or local indexing and (b) either global or local ranking.

The remaining of this paper is organized as follows: we begin by describing the basic Sync/Async method in Section 2; Section 2.1 presents the Sync/Async Round-Robin query processing method proposed in this paper along with a strategy to automatically switch between the two modes of operation; Section 3 describes the data sets and machine we used to obtain the experimental results shown in the next sections; Sections 4.1 and 4.2 illustrate the application of the method in the context of example applications. Section 5 presents our concluding remarks. Related work is discussed in Sections 4.1 and 4.2. Comparison to related experimental work is interspersed in the paper wherever appropriate.

2. The Sync/Async Round-Robin method

Fig. 1 is a flowchart illustrating the proposed method. At 102, queries to the search engine are received by a broker machine. At 104, the queries are provided to the search engine processing 106. At 108, based on the received queries (more precisely, based on the arrival rate of the received queries, which reflects recent history measured from received queries) and the available resources to process received queries, an indication of the operation of the search engine is predicted. At 110, based on the predicted operation of the search engine, the mode of operation is determined which can be either synchronous or asynchronous. The selected mode 112 is provided to the search engine processing 106 so that it switches itself to the proper mode.

This abridged view of the method is given to show how simple the basic idea is. It is clear however that the complexity of keeping the traffic history and the decision on when to switch modes becomes much more complex as the details of the
query processing system is analyzed ("the devil is in the details!"). Focus is hence given to the query processing details in the next section.

2.1. Query processing

From the above, we assume a parallel processing architecture in which the receptionist broker machine receives queries from users and evenly distributes their processing onto the processors or nodes of a cluster of $P$ computers. The processors/nodes work cooperatively to produce the query answers and pass the results back to the broker. Each processor keeps a $1/P$ fraction of an index data structure that is used to speed up the processing of queries. This index is built from the collection of objects forming the database.

We describe with reference to Fig. 2, an example of the query processing method to distribute the workload onto $P$ processors. Queries 202 arrive to the broker machine 204. Each processor 206(1) to 206($P$) has a dual role at all times: fetcher and ranker. In general, fetchers can simply gather data (e.g., from secondary memory) and can also perform some pre-processing on them. Rankers are in charge of receiving the data/partial-results from the fetchers and calculate the final answer to queries 214.

In Fig. 2, this dual role is shown by the dashed line 208. In this example, in the fetching role, each processor 206 operates to fetch data from its index for a particular query. In the ranking role, each processor 206 operates to merge data to compute the ranking associated with a query. The broker is in charge of assigning, in a balanced manner, a processor to be the ranker for each query. The ranker then sends the query 201 to the correct $T$ fetchers 212 in the cluster of processors 206, where $T$
depends on the query and how the index is partitioned. In accordance with traffic, rankers may handle several active queries in parallel in a given period of time.

Every query is then processed using two major steps: the first step, on each fetcher, includes sending a $S$-sized piece of data involved in the received query to its ranker processor. The size $S$ is chosen such that the ranker processor gets about $K$ objects in total for each object forming the query. In the second step, the ranker processor performs the actual ranking of objects and, if necessary, requests additional $S$-sized pieces of data from the fetchers in order to produce the best $R$ ranked objects, which are then passed to the broker as the query results (e.g., typically $R = K/2$). The value of $S$ depends on the partitioning of the index. In fact the value of $K$ dictates the amount of round-robin quantum of each type granted to each query.

An important feature of the scheme ranker-fetchers is that it promotes a sort of centralized ranking for each query. This is very convenient since, as is shown in the next sections, it was found that in this type of application of parallel computing, ranking and indexing based on global information can be indeed very efficient and scalable. Local ranking methods can also be accommodated into the scheme by letting fetchers perform ranking themselves on their local data and communicate their partial-results to the ranker for merging and determination of the top-$R$ results. Hybrids between the two are straightforward as described below.

Notice that global ranking appears to be much more demanding in communication than local ranking and thereby less efficient, but it is not for two reasons. Firstly, user query terms tend to be highly skewed to a reduced set of dynamically-changing popular terms and the cost of the ranking process is much larger than the cost of communication in current cluster technology. The number of communicated and ranked items is roughly the same. Thus, unlike local ranking, for a given set of active queries under processing upon global indexing, the scheme of ranker/fetcher processors allows the broker machine to effectively balance parallel computations by evenly distributing query ranking onto the processors.

Secondly, the global ranking approach is useful for ranking methods based on early-termination where the ranking process tries to traverse only a small fraction of the involved parts of the index in order to quickly obtain the top-$R$ results. As shown below for two application examples, global ranking finishes ranking earlier than its local ranking counterpart either for local or global indexing, since it decides ranking at all time using data from all of the processors, and thereby it reduces the amount of hardware resources assigned to solve each active query.

2.1.1. Sync mode round-robin

Without loss of generality, for the synchronous mode of operation (Sync) the bulk-synchronous model of parallel computing (BSP) [25] can be used. In BSP the computation is organized as a sequence of supersteps. During a superstep, 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 underlying communication library ensures that all messages are available at their destinations before starting the next superstep. Notice that in practice the requirement of global synchronization of processors can be relaxed by simply letting each processor wait to receive one message from each processor to declare itself at the start of the next superstep.

The round-robin principle for sharing processors, disks and communication bandwidth among $Q$ active queries takes place as follows in its bulk-synchronous realization. Query processing can be seen as divided into "atoms" of size $K$, and the atoms are scheduled in a round-robin manner across supersteps and processors. Each of the tasks are given $K$ sized quantum of processor time, communication network and disk accesses.

Each query makes use of one $K$-sized quantum in each superstep and if this is not enough to produce the answer, another quantum is granted in the next superstep. As all atoms are equally sized, the net effect is that no particular task will restrain other tasks from using the resources. Once $Q$ new queries are evenly injected onto the $P$ processors, their processing is started in iterations as described above. At the end of the next or subsequent superstep, some queries, say $n$ queries, will finish their processing and thereby at the next superstep $n$ new queries can start their processing.

Fig. 3 illustrates, for $P = 2$ and $T = 2$, the iterative processing of seven queries labeled A, B, C, D, E, F and G. The processing is distributed across two processors and eight supersteps where there is exclusive use of supersteps for fetching and ranking operations. In the figure, queries are processed sequentially in each processor. For example, the thread on processor 0 first processes completely the quantum granted to query A and then the quantum granted to query B.

This basic scheme admits several alternative realizations where each of them makes sense depending on the relative trade-off of the comparative real time associated with the quanta in processing time, secondary memory accesses and communication. In one case, the supersteps can be overlapped when query traffic is high enough to work with two streams of $Q/2$ active queries each, displaced by one superstep from each other. While one stream is performing ranking, the second one is performing fetching. This is feasible when the time spent in secondary memory is comparatively similar to the time spent performing ranking and other CPU time consuming operations.

Another feasible approach to overlapping costs is to keep in each processor a slave thread in charge of servicing secondary memory requests. This is an asynchronous thread that communicates with the main BSP thread via asynchronous message passing. The service request messages are injected directly in the input message queue of the BSP thread by the slave thread, and the master BSP thread places asynchronously secondary memory request messages in the message queue of the slave thread. This is the approach used in the remainder of the paper.
In addition, facilities for light multi-threading available from multi-core processors can be easily exploited by letting a team of \( n \) openMP threads work in parallel making the ranking of \( n \) queries at a time in each processor during supersteps. For instance, in Fig. 3 the ranking of queries A and B can be made by using two openMP threads that work in parallel.

Regarding concurrency control in situations in which the data is updated concurrently with query processing, the Sync mode provides a relevant advantage over the Async mode. A first point to note is that the semantics of supersteps in BSP tells that all messages are in their target processors at the start of each superstep. If the broker assigns a correlative time-stamp to every query and document that it sends to the processors, then it suffices to process sequentially all operations embedded in messages in timestamp order in each processor to avoid R/W conflicts.

### 2.1.2. Async mode round-robin

The same round-robin scheme is also used when the search engine is operating in the asynchronous mode (Async). This simplifies the switching between the two modes of operation since the logic of the query processing task does not change between modes. The current states of all queries are kept at their ranker processors indexed by id_query.

The Async mode keeps a set of ranker and fetcher threads alive in each processor. For each ranker thread we have \( P \) fetcher threads and we use one ranker per active query. Thus each query is serviced by a unique ranker thread that asynchronously communicates via message passing with its \( P \) fetcher threads. Each fetcher is hosted by a different processor. In the asynchronous version of the Fig. 3, fetcher threads would be represented by circles whereas rankers by rectangles and no synchronization barriers among processors would be imposed, and thereby random displacements between circles and rectangles should be observed. On the average, for low query traffic and steady state, this asynchronous scheme still tends to provide a fairly similar share of the computational resources to the queries, though the exact sequence is unpredictable.

### 2.2. Trade-off between sync and async modes

In the Async mode, for high enough query traffic, the overheads associated with the scheduling of a large number of threads and the communication of many small messages among those threads can become significant. In the Sync mode, in each processor, all messages going to the same destination are packed together into a single message and sent at the end of supersteps. Also one thread per processor is used for ranking and fetching, and alternatively an additional thread for secondary memory. It was observed that this scheme reduces overheads significantly. Also as the processing is divided in \( K \)-sized quanta, all processors perform fairly the same work achieving good load balance. However, in a situation of low query traffic there is not a sufficient number of queries to sustain this convenient scenario and imbalance degrades performance significantly.

The trade-off between the two modes of operation can be observed in Fig. 4(a) for the inverted files application and in Fig. 4(b) for the metric-space application. Both cases are results from actual implementations on the actual data sets and machines described in Section 3. These figures show the performance for different traffic conditions, when the search engine is operating in either mode separately, and using either global or local indexing. Performance differences between Sync and Async can be indeed significant.
2.3. Switching between modes of operation

When the Sync mode comes into operation, all Async threads are put to sleep so that one thread per processor takes control to start Sync computation. Active queries can continue execution in the Sync threads since their states are kept at processor level. On the other hand, when the Async mode comes into operation, each active query can be taken by a different thread from the point that the next superstep of the Sync mode would have been.

Actually both modes can be run at the same time since the Sync thread in each processor is a normal thread with large computing granularity which signals the start of the next superstep when it receives one message from each processor including itself (in absence of messages, processors send each other null messages to synchronize).

Thus a practical realization of the proposal of this paper is to use the Sync threads to deal efficiently with sudden peaks in the query traffic. The hardware is dimensioned so that at normal traffic the search engine is operating in the Async mode. As soon as the broker detects in its input message queue a sufficiently large number of queries waiting to receive service, it sends Q of them to the Sync threads and it keeps performing Sync round-robin on those queries whilst the queries being processed by the Async mode start to finish. The broker also tries to keep injecting new queries to the Sync threads as they finish their current queries.

In such a case the value of Q for the Sync mode must be large enough to ensure good load balance but small enough to ensure that the response time of individual queries does not exceed a pre-defined limit. All this must be determined by benchmarking the system with actual logs containing past queries. Thus an average number of active queries below a certain threshold, observed during a certain number of supersteps, is a clear indication that the query traffic is so low that it is more efficient to switch the search engine to the Async mode. At this point, the broker starts to feed the Async ranker threads with queries.

The values of Q are different depending on the context, namely different values for the Sync and Async modes and they can change along time. In general, the broker must determine the average number of active queries from time to time. This average depends on the arrival rate of queries which is an users-driven value that dynamically changes in time. The average can change along time. In general, the broker must determine the average number of active queries from time to time. This determines the mode of operation Sync/Async of the search engine.

Define the weighted average number of active queries during the Async and Sync modes as \( Q_A \) and \( Q_S \) respectively. The Sync mode starts to be efficient from \( Q_{min} \) active queries upwards per unit time (this value can be obtained by benchmarking or by a minimal load balance requirement). Below this value \( Q_{min} \), it is more efficient to use Async query processing. In addition, the Sync mode can only handle up to \( Q_{max} \) active queries per unit time. Beyond this value the system cannot ensure a pre-defined upper bound for the response time of individual queries.

The broker can switch the system from Async to Sync when it has sampled, say, \( Q_A > 1.3 \cdot Q_{min} \) during one or more intervals of \( \Delta \) units of time. Sudden peaks in query traffic are reflected in the length \( L_q > 2 \cdot Q_{min} \) of the input message queue of the broker, which contains the pending queries waiting to be processed. In this case, the broker can aggressively send to Sync processing \( Q_{min} \) queries from this queue to try to sustain along time \( Q_{min} \) active queries in the Sync round-robin fashion. During this transitory phase both modes of operation coexist in the search engine. The exclusive Sync mode is set if in the next two or more intervals of \( \Delta \) units of time the total average number of active queries satisfies \( Q_A + Q_S > 1.3 \cdot Q_{min} \).

Notice that having the system fully operating at \( Q_{max} \) active queries all the time, produces a peak query throughput of \( X_{max} \) completely processed queries per unit time. This is only possible under a query traffic high enough so that it has a query
arrival rate $\lambda \geq X_{\text{max}}$. The broker can adjust the total number of queries $Q$ allowed to be active in the Sync mode by searching the steady state $Q$ as follows

$$Q_{\text{new}} = Q_{\text{current}} \cdot \left[(1 - \alpha) + \alpha \cdot \min\{X_{\text{max}}, \lambda(\Delta)\}/X(\Delta)\right],$$

where $\alpha$ is a smoothing factor determined experimentally (we use $\alpha = 0.4$), and $\lambda(\Delta)$ and $X(\Delta)$ are the query arrival rate and the Sync query throughput observed during the period of $\Delta$ units of time respectively.

Finally, when the observed $Q_S$ is below $0.7 \cdot Q_{\text{min}}$ the search engine is switched to the Async mode. To simplify implementation, the current queries being processed in either mode are allowed to finish in the mode they are being solved. We have observed that this transient effect between modes does not degrade performance significantly. We have also observed that the values 0.7 and 1.3 used in the control algorithm represent a good trade-off between efficient steady state operation and reaction time to Sync/Async changes.

Fig. 5 shows the main steps followed by the control algorithm in order to dynamically switch processors between the Sync and Async modes of operation.

**3. Experimental setting**

In the next sections of this paper we present results from actual executions of programs implementing the proposed strategies. The results were obtained on a NEC cluster composed of 200 dual processors (Intel Xeon EM64T CPU’s 3.2 GHz) connected by an Infiniband 1000 MB/s network. Each node has 1 GB of main memory. In this machine we had exclusive access to 32 and 64 processors to run our experiments, and limited access to the full 400 processors capacity to perform a few more experiments in which scalability is the metric of interest. Communication is via LAM MPI-2 and BSPonMPI (bspompi.sourceforge.net) and we use the front-end computer as a broker machine to route queries onto processors.

A 1.5 TB sample of the UK Web was used to perform two types of experiments on plain text. The first one is the usual indexing of documents where one is interested in retrieving the most relevant documents to a given query. Actual queries were executed against distributed inverted files. They were taken from a one year log containing queries submitted by actual users to http://www.yahoo.co.uk. On this collection an average query throughput of 1,800 queries per second using 32 processors was obtained. Also the log has information of the arrival time of queries. In Fig. 6 a normalized sample of 1% of query traffic submitted into the UK domain during 31 days is shown.

**Broker Event:** A new period of $\Delta$ units of time has expired.

```plaintext
if ( current query traffic state is set to LOW ) then
  if ( $L_q > 2 \cdot Q_{\text{min}}$ ) then
    Send $Q_{\text{min}}/P$ different queries from the input query queue to each Sync thread.
    Set $Q_S \leftarrow Q_S + Q_{\text{min}}$.
  endif
  if ( $Q_A + Q_S > 1.3 \cdot Q_{\text{min}}$ ) then
    Set current query traffic state to HIGH.
    Switch the $P$ processors to the SYNC mode by broadcasting a SYNC message.
  endif
else  // current query traffic state is set to HIGH
  if ( $Q_A + Q_S < 0.7 \cdot Q_{\text{min}}$ ) then
    Set current query traffic state to LOW.
    Switch the $P$ processors to the ASYNC mode by broadcasting an ASYNC message.
  else
    $Q_S \leftarrow Q_S \cdot \left[(1 - \alpha) + \alpha \cdot \min\{X_{\text{max}}, \lambda(\Delta)\}/X(\Delta)\right]$.
  endif
endif
```

Fig. 5. Controlling the mode of operation of $P$ processors. $L_q$ = average size of the input query queue. $Q_{\text{min}}$ = threshold value to decide between Sync/Async. $Q_A$ = Number of queries being processed in the Async mode. $Q_S$ = Number of queries being processed in the SYNC mode. $\alpha$ = Smoothing factor. $X(\Delta)$ = Number of queries resolved by unit time. $X_{\text{max}}$ = maximum $X(\Delta)$ achieved by the processors. $\lambda(\Delta)$ = Number of new queries per unit time arriving to the broker.
In the second type of experiments, focus centered on a similarity search on the metric-space index by extracting from the UK collection all terms (sequences of chars) and indexing 26 million of them. The query log was run on this index to retrieve the terms which are most similar to the query ones. The distance function used to determine the similarity between two terms is the edit distance function [2]. Here the motivating application can be query spelling.

We emphasize that in the execution of the parallel programs we injected the same total number of queries QT in each processor. That is, the total number of queries processed in each experiment reported below is \( QT \times P \) where \( P \) is the number of processors. Thereby running times are expected to grow with \( P \) since the communication hardware has at least \( \log P \) scalability. Thus in figures shown below (and Fig. 4 above), curves for, say, 16 processors are higher in running time than the ones for four processors. We have found this setting useful to see the efficiency of the different strategies in the sense of how well they support the inclusion of more processors/queries to work on a data set of a fixed size \( N \). This clearly exposes the ability of a given query processing strategy to involve more and more hardware efficiently. We executed experiments for \( R = 128 \) where \( R \) is the number of results, and \( QT = 10,000 \). We also show values normalized to one for each data set to better illustrate the percentage difference in terms of overall query throughput.

3.1. Experiments

In the following we present results that show the efficiency of the proposed method for switching between modes of operation. In the first experiment we start injecting queries at a rate that is convenient for the Async mode. Then suddenly the query traffic increases drastically and is hold for a certain period. At the end of this period the intensity of traffic is restored to its initial value. Namely, we simulated one of the peaks of Fig. 6 by using the query log data to get roughly the same dynamic traffic. We tested three cases: (1) the Async mode used exclusively during the whole experiment, (2) the Sync mode used also exclusively, and (3) the Sync/Async mode using the control algorithm proposed in the previous Section 2.3 to switch between modes in accordance with the intensity of query traffic.

The results are presented in Fig. 7(a), (b) and (c), and they show that the case 3 (Sync-Async) is able to complete its queries quite before the time required by the cases 1 (Async) and 2 (Sync). Fig. 7(a) shows results for the observed query throughput, Fig. 7(b) shows the average total number of active queries between the intervals in which the broker measures and acts, and Fig. 7(c) shows the average response time of individual queries.

The results in Fig. 7(d) are for a different experiment. In this case the query arrival rate is maintained at three values where all of them are sufficiently high to keep the search engine operating in the Sync mode. The Figure shows the search for a steady state operation made by the control algorithm. The traffic is sustained during a small period of time. The algorithm quickly gets close to the optimal number of active queries necessary to achieve the best throughput in each traffic condition.

4. Application examples

4.1. Application example 1: distributed inverted files

The inverted file [2] is a well-known index data structure for supporting fast query processing on very large text collections. An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of distinct relevant terms found in the collection. Each of these terms is associated with a posting list that contains the
document identifiers where the term appears in the collection along with additional data used for ranking purposes. The posting lists are used to quickly get the documents associated with the query terms to then compute their ranking and select the top $R$ documents.

A number of papers have been published on parallel query processing upon distributed inverted files [1,12–14,19,23,26]. The two dominant approaches to distributing an inverted index on $P$ processors are (a) the document partitioning strategy, in which the documents are evenly distributed onto the processors and an inverted index is constructed in each processor using the respective subset of documents, and (b) the term partitioning strategy, in which a single inverted index is built from the whole text collection to then evenly distribute the terms and their respective posting lists onto the processors. In the following we describe the document and term partitioned inverted files in the context of the Sync mode of operation.

The ranking of documents is performed using the list pruning method described in [21]. This method generates a workload onto processors that is representative of other alternative methods [2].

4.1.1. Document partitioned index (local indexing)

In the document partitioned inverted file, the ranker performs a broadcast of each query to all processors. All queries are broadcast to all processors in two steps. For a given set of $Q = qP$ queries, in the first step the broker sends $q$ different queries to each processor, and in the second step all processors send a copy of their $q$ queries to all others (in conventional systems, all queries are broadcast from the broker, which is less efficient and scalable when query traffic is high). Then all processors send about $S = K/P$ pairs (doc_id, term_frequency) of their posting lists to the ranker, which performs the ranking of documents. In the case of one or more query terms passing the posting list pruning filters [21], the ranker sends messages to all processors asking for an additional $S = K/P$ pairs (doc_id, term_frequency) of the respective posting lists.

Fig. 8 shows a single iteration for a given query 100 with two terms 101, 102 on a system with $P = 2$ and $K = 2$. At super-step $n$, the query 100 already sent by the broker is received in the processor $i$. In accordance with our approach this processor becomes the ranker for this query 100. At this point the first step of the broadcast is completed for all queries and the second step consists on sending a copy of the query 100 to all processors. Upon reception of the copies of query 100, each processor fetches the first $K/P$ items of the term-frequency sorted posting lists 201, 202 associated with each term 101, 102 of the query 100. These pieces of posting lists 301, 302 are then sent to the ranker processor where they are merged into a single

![Fig. 7. Performance of exclusive Sync, exclusive Async, and the Sync/Async combination ruled by the control algorithm of Section 2.3. A temporal peak in the arrival rate of queries is applied for $P = 32$. Each experiment ends after a fixed total number of queries have been completely processed. We obtained identical results both for the inverted files and the metric-space index (Sections 4.1 and 4.2). The type of workloads were fairly similar. Figures: (a) Values for the overall query throughput, (b) number of active queries and (c) the average response time of individual queries in the period. Figure (d) shows how the algorithm finds the optimal number of active queries for three traffic conditions in which the Sync mode is efficient.](image-url)
list which is then passed to the ranking and pruning routine 300. At this point, the ranker determines that it needs additional pieces of posting lists to decide the final top-R documents and as a result it sends messages to all processors asking for the next \( K/P \)-sized pieces of posting lists in decreasing frequency order 401, 402 to continue with the ranking 400 at superstep \( n+4 \). This is applied to all active queries in the search engine.

Notice that an interesting optimization to this (and the following) scheme is for the broker to select the ranker processors for queries taking into consideration the pieces of posting lists that have already been sent to processors in the recent past. This requires the design of a special purpose distributed cache strategy.

4.1.2. Term partitioned index (global indexing)

In the term partitioning approach, we distribute the terms and their posting lists in an uniformly and random manner onto the processors (we use the rule \( id_{term} \mod P \) to determine in which processor is located a given term). In this scheme, for a given query, the ranker processor sends messages to the fetchers associated to every term present in the query asking for the first \( S \equiv K \) pairs (doc_id, term_frequency). The same is repeated if one or more terms pass the filters in the ranking operation.

In Fig. 9 we show a single iteration for a given query 100. We assume that the posting lists associated with the terms 101 and 102 are stored in two different processors 201, 202. In this case messages requesting the first \( K \)-sized posting lists for each term are sent to the processors containing the query terms. These processors reply with the respective pieces of posting
lists 301, 302 to the ranker, and the ranker merges 300 these lists in descending frequency order to perform the ranking and pruning. At this point this process is independent of the type of indexing being implemented in the search engine. The results are equivalent and therefore for this example, like in the document partitioned index case, the ranker decides that it is necessary a new iteration 401, 402, to perform a new ranking 400 at superstep \( n + 4 \).

4.1.3. Performance analysis
The total running time 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 quantities: \( w \), \( hG \) and \( L \), where \( w \) is the maximum of the computations performed by each processor, \( h \) is the maximum of the messages sent/received by each processor with each word costing \( G \) units of running time, and \( L \) is the cost of the barrier synchronizing the processors. The effect of the computer architecture is included by the parameters \( G \) and \( L \), which are increasing functions of \( P \). The average cost of each access to disk is represented by \( D \).

In a situation of high traffic, we can assume that the broker distribute \( Q = qP \) queries in every superstep so that \( q \) new queries arrive at each processor in each superstep. In this case we have that the order of the cost to process every iteration of a single query is

\[
C = \text{Comm}(\cdot) + L + \text{Fetch}(S)(D + G) + L + \text{Rank}(K).
\]

In the document partitioning technique, \( \text{Comm}(\cdot) \) requires \( G \) time for each of the \( P \) processors, while in the term partitioning is the same time but less processors are involved (the number of terms in the query, which is in practice a constant). \( \text{Fetch}(S) \) requires linear time in \( S \) as it needs to read data from disk and send them back to the ranker. Again the difference is the number of processors that need to do that for each query. The term partitioned index performs \( \text{Fetch}(S = K) \) for each query term in the processors containing them, whereas the document partitioned index performs \( \text{Fetch}(S = K/P) \) for each query term in each processor. Overall, \( qP \) of these operations per processor are performed by the document partitioned index while the term partitioned index performs \( q \) of them.

Assuming reasonable load balance with OR or disjunctive queries requiring on the average \( r \) iterations and few terms per query, the average BSP cost of the document partitioning approach can be represented asymptotically by

\[
C_D = qrKD + qr(P + K)G + qr\text{Rank}(K) + L,
\]

whereas the cost of the term partitioned approach for random distribution of terms is given by

\[
C_T = qrKD + qr(1 + K)G + qr\text{Rank}(K) + L.
\]

This shows that the term partitioning approach has optimal scalability and better performance than the document partitioned approach, since \( C_D - C_T = O(P) \). Larger \( q \) (traffic) improves load balance and larger posting lists tends to increase \( r \) and by construction these values are independent of the index partitioning method. In both cases the ranker processors are selected in a circular manner, so the load balance of this part is also independent of partitioning.

The assumption of identical cost \( qrKD \) is optimistic for the document partitioned index because it does not consider the overheads of dealing with \( P \) chunks of size \( K/P \) instead of one of size \( K \) in the term partitioned index. For example the retrieval of a total of \( qK \) posting lists items belonging to \( qP \) queries comes at \( qP \) disk seeks as opposed to the \( q \) disk seeks in the term partitioning approach. Notice that in a fully asynchronous thread-based search engine we have a similar situation, namely there is an increased cost coming from the overheads associated with the scheduling of a comparatively larger number of threads per query.

If we consider the response time of individual queries, as the processors work in parallel, some authors claim that the document partitioning technique is faster. In fact, in our cost model, the parallel response time per query in this case can be modeled by

\[
t_D = r(K/P)D + r(P + K)G + r\text{Rank}(K) + L,
\]

while in the term partitioning case is

\[
t_T = rKD + r(1 + K)G + r\text{Rank}(K) + L.
\]

However this may be only valid for very low query traffic. For high traffic such an argument for \( t_D \) is misleading because at the same time the processor is being occupied by the activity generated by all other queries under processing. In the document partitioned index each query generates a \( P \)-fold activity in all other processors which in practice means that each query is delayed \( P \) times in completing its \( K/P \)-sized fetch operations making \( t_D = t_T \).

The situation is quite different when we consider ranking methods requiring posting lists intersection, namely AND or conjunctive queries requiring all terms be contained in the documents selected as the top-R ones. In the document partitioned index this operation can be done for free in terms of communication. Let us evaluate the overall cost of this operation by considering the trade-off between \( G \) and \( D \). If we consider that on average intersecting 2-terms queries demands the retrieval of a total of \( PB \) blocks of disk (this is for the full length of posting lists), then the cost of intersecting two posting lists in the document partitioned index is \( O(qPBD) \). This because even though every processor retrieves \( B \) blocks per query they have to deal with \( qP \) queries each.

For the term partitioned index we have two cases. In the first one the query terms happen to be co-resident in the processor, in which case the cost is also \( O(qPBD) \) because the processor maintains the \( PB \) blocks of each posting list...
and each of them only deals with \( q \) queries. Notice that by doing smart term distribution onto processors we can increase the co-residence probability. In the second case they are not co-resident. Here, for the sake of simple solutions, one of the processors has to retrieve from disk the pattern blocks and send them to the other one. Thus the cost is \( O(qPB(D + \frac{1}{2}G)) \) and this cost should be about twice the cost incurred by the document partitioned index (though current cluster technology indicates \( G < D \)).

4.1.4. Comparison with related work

Discussing previous work in contrast with ours is quite involved since almost each research group uses different methods for document ranking and query processing upon the term and document partitioned inverted files [1,12,13,23,19,26]. Some use either intersection (AND) or union (OR) queries [5,10,28]. Others perform exhaustive traversal of posting lists while others do list pruning which can be made on posting lists sorted by term-frequency or document id. In consequence in this section we enumerate what we believe are the main principles upon which scalable performance can be achieved and describe previous strategies in light of how well they attach to those principles.

First a clarification on our query processing regime on the document partitioned inverted file which we claim is by far more efficient and scalable than the standard method tested in the literature. In the standard method, the broker machine sends each query to all processors which then proceed to compute their proposal for the top-\( R \) results. Once processors finish these computations they send their local top-\( R \) results to the broker, which then performs a merge operation on them to produce the global top-\( R \) results.

In modern cluster hardware the communication network is very fast as compared to the cost of performing document ranking operations on posting lists. Sending a \( K \)-sized posting list from one processor to another is by far less time consuming than running a document ranking operation on the same piece of posting list. Thus consider any pruning method which uses a global barrier to stop the processing of posting lists associated with a given query. In the standard document partitioned index the best case scenario is that at the instant in which the local document ranking stops by hitting the barrier, all processors have scanned the same \( K/P \) of posting lists. But notice that this barrier is only updated using local information which not necessarily ensures \( K/P \). In fact, our experiments show that it is quite similar to \( K \) which potentially increases the overall cost of ranking \( P \) times.

Thus in our BSP parlance and considering an average rate of \( qP \) active queries and ignoring disk cost, the query throughput can be potentially degraded \( P \)-times since the standard method has cost \( q \cdot P \cdot \) [Rank\((r \cdot K) + R \cdot G\)] where \( r \) is a factor that indicates the portion of posting list that is scanned during ranking, and in our method this cost is \( q \cdot \) [Rank\((r \cdot K) + (rP + r \cdot K + R) \cdot G\)]. Both intuition and empirical evidence show that \( r^* > r/P \) and in fact one should expect \( r^* \approx r \) because barriers based on local posting list information are known to be less effective than global based ones. This is evident because the outliers that update the global barrier are now uniformly distributed on the \( P \) processors.

Regarding the term partitioned index all previous work perform partial document ranking on the processors that contain query terms. This potentially leads to a serious and unpredictable imbalance since term distribution is highly skewed and dynamic as it is user driven. Apart from the advantages of global ranking, this is another important reason to perform centralized ranking. Considering all active queries, the broker can easily distribute these expensive ranking operations onto the processors. List fetching imbalance can be solved with cache memory and communication imbalance can be tolerated as its cost is much less relevant than the cost of document ranking.

The two most recent proposals of query processing on term partitioned indexes are examples of the above drawbacks. In the strategy proposed in [1], the query is sent to the processors that contain the query terms. In each processor the same ranking method used by us in our experimentation is employed but not in a centralized manner. Instead, an estimate of the global barrier is calculated locally considering the posting list for the term in the processor, and partial document scores are calculated using that posting list. Because these scores are calculated using the contribution of local term frequencies a second round of score calculations is performed at the broker machine. For this purpose each processor sends to the broker their top \( c \cdot R \cdot P \) documents with \( c = 6 \) and the broker in turn re-scores those documents by considering the contribution of all query terms that are present in each document. Because documents are partially scored the method is compelled to communicate more documents than in our centralized method. The partial scoring causes imbalance as explained above.

When the partial scoring is allowed to take place one after another by traversing sequentially the processors containing query terms we get to the strategy proposed in [19]. For the same reasons this is also prone to the imbalance caused by partial scoring and overall communication is the same but distributed in another way along the real time. If we sum up after the processing of a large set of queries we find that both strategies are equivalent in terms of imbalance and communication. They just use the cluster resources for the same queries at different instants but the final outcome is the same.

In Fig. 10 we show results for OR queries that confirm the claims made in this section on comparative performance with previous approaches. We call global ranking (GR) to our query processing scheme based on global barriers and local ranking (LR) to that based on local barriers for pruned ranking on posting lists. Also LI and GI stand for document (local indexing) and term (global indexing) partitioning, and GP is pipelined query processing using the same ranking method of [1]. The results indicate that the strategy we propose in Section 4.1.2 (e.g., Fig. 9), which is based on global indexing and global ranking (GIGR) is more efficient when OR queries are feasible. For AND queries the method of choice should be the strategy we propose in Section 4.1.1 (e.g., Fig. 8), which is based on local indexing and global ranking (LIGR). Nevertheless, in [16] we show that both local and global indexing methods may achieve similar performance for AND queries.
4.1.5. The many variants of the same approach

Upon the proposed Sync/Async Round-Robin query processing method it can be implemented different strategies for indexing and ranking. Most of these strategies have been described in the previous sections of the paper. In addition, two alternatives for the term and document partitioned indexes are what in the following we call clustered global and local indexes respectively (cf. [10,28,22,11]). In the first case, the terms occurring very frequently in the same queries are placed in the same processors to support efficiently AND queries. A clustering method can be employed to determine which terms are co-resident. In the second case, when approximated answers to queries are acceptable, the documents that occur frequently in the top-R results can be clustered in \( M \ll P \) processors. We employ the prefixes CGI and CLI for these two variants respectively.

Table 1 shows the naming conventions for a number of practical strategies for ranking and indexing. Table 2 describes how these strategies can work upon the Sync/Async Round-Robin query processing method by using \( K \)-sized quanta in either modes of operation. Boldface letters in the expressions stand for the primitive operations Broadcast (B), Fetch (F), Rank (R), Send (S) and Merge (E). Also \( M \) and we use “\(+\)” to indicate one or more repetitions of the same sequence of operations where each repetition takes place in a consecutive superstep, and “\( x \)” indicates parallelism on \( x \ll P \) processors.

**Table 1**

<table>
<thead>
<tr>
<th>Code</th>
<th>Query processing strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>GILR</td>
<td>Global indexing with local ranking</td>
</tr>
<tr>
<td>CGIHR</td>
<td>Clustered global indexing with local ranking</td>
</tr>
<tr>
<td>LILR</td>
<td>Local indexing with local ranking</td>
</tr>
<tr>
<td>CLIHR</td>
<td>Clustered local indexing with local ranking</td>
</tr>
<tr>
<td>GIGR</td>
<td>Global indexing with global ranking</td>
</tr>
<tr>
<td>CGIGR</td>
<td>Clustered global indexing with global ranking</td>
</tr>
<tr>
<td>LIGR</td>
<td>Local indexing with global ranking</td>
</tr>
<tr>
<td>CLIGR</td>
<td>Clustered local indexing with global ranking</td>
</tr>
</tbody>
</table>

**Table 2**

<table>
<thead>
<tr>
<th>Code</th>
<th>Most-likely sequence of primitive operations for two-terms queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>GILR</td>
<td>( F(K) [2 \rightarrow R(K)] [2 \rightarrow S(cR)] \rightarrow E(2cR) ) (just one term in ( p_1 ))</td>
</tr>
<tr>
<td>CGIHR</td>
<td>( 2F(K) \rightarrow R(2K) ) (both terms are in ( p_1 ) with high probability)</td>
</tr>
<tr>
<td>LILR</td>
<td>( B(2r) \rightarrow (2F(K/P) \rightarrow P \rightarrow R(2K/P)) \rightarrow S(K) \rightarrow P \rightarrow E(PR) )</td>
</tr>
<tr>
<td>CLIHR</td>
<td>( B(2r)^M \rightarrow (2F(K/M) \rightarrow P \rightarrow R(2K/M)) \rightarrow S(K/M) \rightarrow P \rightarrow E(MR) )</td>
</tr>
<tr>
<td>GIGR</td>
<td>( F(K) [2 \rightarrow S(K)] \rightarrow R(2K) ) (one term in ( p_1 ), probability ( 1 - 1/P ))</td>
</tr>
<tr>
<td>CGIHR</td>
<td>( 2F(K) \rightarrow R(2K) ) (both terms are in ( p_1 ) with high probability)</td>
</tr>
<tr>
<td>LIGR</td>
<td>( B(2r)^P \rightarrow (2F(K/P) \rightarrow P \rightarrow 2S(K/P) \rightarrow P \rightarrow R(2K)) )</td>
</tr>
<tr>
<td>CLIGR</td>
<td>( B(2r)^M \rightarrow (2F(K/M) \rightarrow M \rightarrow 2S(K/M) \rightarrow M \rightarrow R(2K)) )</td>
</tr>
</tbody>
</table>
Notice that a practical realization of early-termination query processing for a system with very large posting lists, executed on a relatively inefficient communication hardware, is a combination of global and local ranking. For instance, for document partitioning, a combination LILR–LIGR can work by first performing \( r \) iterations using LIGR and ending with the broadcast of the global pruning barrier. Then the process continues by performing quanta of ranking in each processor as in LILR but periodically broadcasting among processors the local barriers values obtained by each processor to aggressively prune the posting list traversals.

4.2. Application example 2: metric-space indexing

This example further illustrates the trade-off between ranking and indexing. Metric spaces are useful to model complex data objects such as images or audio [4]. Search queries are represented by an object of the same type to those in the database wherein, for example, one is interested in retrieving the top-\( k \) objects that are most similar to the query. Good surveys on metric-spaces can be found in [7,24,27]. In this Section we use the so-called List of Clusters (LC) [6] as an example of index metric-space data structure. Parallelization of query processing on metric-spaces has been discussed in [20,8,15,17,18]. The parallel realization of the LC presented in [9,15] has been found to be more efficient than other metric-space data structures, including those based on trees.

The LC data structure is formed from a set of centers which are chosen from the database by using a particular heuristic. Each database object \( x \) is attached to the cluster whose center \( c \) is the closest one to \( x \). There exists a distance function \( d(x,c) \) that determines the distance between two objects \( x \) and \( c \). In order to support round-robin we build clusters of fixed size \( K \). Thus the extent of a cluster with center \( c \) is given by its radius \( r_c \), which is the distance between the center \( c \) and its \( K \)-nearest neighbor. A typical query is the range query \( (q,r) \) which consists on retrieving all objects that are within a distance \( r \) from a query object \( q \). The \( k \) nearest neighbors queries are implemented upon range queries.

During the processing of a range query \( q \) with radius \( r \), the idea is that if the first center is \( c \) and its radius is \( r_c \), we evaluate \( d(q,c) \) and add \( c \) to the result set if it is appropriate. Then, we scan exhaustively the cluster \( I \) only if the query ball \( (q,r) \) intersects the center ball \( (c,r_c) \). Next, we continue with the remaining set of clusters \( E \) recursively. However, because the LC is constructed in an asymmetric manner [6], we can stop the search before traversing the whole list of clusters: If the query ball \( (q,r) \) is totally and strictly contained in the center ball \( (c,r_c) \), we do not need to traverse \( E \) since the construction process ensures that all the elements that are inside the query ball \( (q,r) \) have been inserted in \( I \).

We define the plan of a query \( q \) as a sequence of \( n_q \) tuples \( \langle c_i, d(q,i), p_i, s_i \rangle \) with \( i = 1 \ldots n_q \), where \( n_q \) is the total number of clusters to be visited by the query \( q \). (b) \( c_i \) is the center/cluster id, (c) \( d(q,i) \) is the distance between \( q \) and the center \( i \), (d) \( p_i \) is the processor in which the cluster is hosted, and (e) \( s_i \) is the superstep at which the cluster \( c_i \) must be visited. The tuple elements \( c_i, p_i \) and \( d(q,i) \) are determined by the type of indexing (either local or global), and \( s_i \) is used to support round-robin by allowing each active query to process one visit to a cluster per superstep. The query plan can be calculated in parallel by using all of the processors and it consists on comparing the query against all centers by using the distance function.

In local indexing the objects are evenly distributed on the \( P \) processors and the centers are selected in each processor by considering the co-resident objects. This is equivalent to having \( P \) sequential LC indexes and each query is broadcast to all processors to be solved in parallel. We call this strategy local index local centers (LL). An step forward is to select the centers by considering the whole set of objects distributed onto the processors. In this case globally good outliers are selected as

![Fig. 11. Illustrating the impact of global indexing and global ranking in a metric-space index for a term spelling application. (a) LL indicates local indexing with local centers, LG local indexing with global centers, and GG global indexing with global centers. (b) L stands for local centers and G for global centers.](image-url)
centers and each processor holds a copy of the centers and builds up its local clusters using the same centers. We call this strategy local indexing global centers (LG).

Finally, in pure global indexing the global centers calculated for LG are evenly distributed at random onto the processors and the LG clusters are merged and stored together in the processor where their respective centers are located. We call this strategy global index global centers (GG).

Fig. 11(a) shows the effects of such arrangements. The impact in performance is significant and it comes from two sources. First, the Fig. 11(b) shows that the global centers (G) are more effective than the local centers (L) in reducing the number of objects that must be compared against the query (the index aim is to reduce distance computations between objects since this process is costly). This is equivalent to selecting those objects based on either global or local information which is used to prune the search space. Secondly, the global indexing approach (GG) has the ability to reduce the number of processors that are hit by the queries and this has a significant effect in increasing the overall query throughput. In addition, the previous Fig. 4(b) (Section 2.2) shows that the query throughput is also greatly influenced by the Sync/Async modes of operation which depends on the observed query traffic. Overall these results validate the claims made for the different alternatives for ranking and indexing in inverted files and illustrate the basic principles leading to efficient and scalable performance.

5. Concluding remarks

In this paper we have presented a technique to efficiently organize parallel query processing on search engines. We have validated the effectiveness of our proposal with two example applications which we believe are pertinent to modern requirements for large-scale systems. The experimentation was made by running actual implementations using actual data sets upon a high-performance machine.

Overall, this paper promotes as highly scalable and efficient strategies, the ones that perform query processing using global ranking and global indexing upon Sync/Async Round-Robin Search Engines. The principles, their relationship and justification, upon which we support this claim are the following:

5.1. Sync/Async modes of operation

Large search engines can be subjected to significant variations in the intensity of the query traffic arriving to the broker machine. In Section 2.1 we have proposed that it is more efficient to deal with peaks and/or sustainably high query traffic by switching the search engine into what we call the Sync mode of operation. The gain in performance comes from optimizations that are only possible when processing all active queries in bulk. When the traffic is low, the mode of operation is the conventional Async mode used by current search engines. In Section 2.3 we have proposed an adaptive algorithm which is able to automatically switch the search engine between the two modes of operation in accordance with the observed query traffic. The experimental results show that the algorithm is able to find by itself the point at which the search engine achieves a near optimal query throughput.

5.2. Round-robin

We have found convenient to force partitioning the different stages involved in the processing of queries into quanta of work of fixed size in order to grant each query the same chance of using the hardware resources. Certainly this improves the response time of individual queries and prevents from undesirable situations such as saturation of communication buffers. Notice that the round-robin scheme used in combination with the Sync mode allowed us to precisely model the cost of algorithms. We assume a steady state situation with Q active queries and a new one is injected as soon as a current one is finished. Also this combination enhanced with the scheme of ranker and fetcher processors leads to an almost perfect load balance when query traffic is heavy enough which is critical on large-scale high-performance hardware.

5.3. Global vs. local indexing

We have shown how to apply global and local indexing upon the Sync/Async Round-Robin approach, and scheme of rankers and fetchers. Both the experimental and BSP cost model based results show that scalable performance can only be achieved when one avoids using all hardware resources for each single query. Ideally each active query should use as little as possible of hardware. Global indexing provides this possibility. In both the inverted files and the metric-space indexes we have observed the same fact (Sections 4.1 and 4.2). For inverted files we propose avoiding the danger of imbalance in global indexing by preventing fetchers from performing costly computations and transferring them to rankers. The reason is that the broker is only able to properly load balance the computations of rankers onto the processors. It has little control on balancing the fetchers since this depends on the query terms themselves and the static mapping of data onto processors.
5.4. Global vs. local ranking

The scheme of rankers and fetchers is designed to encourage the use of what we called global ranking, namely ranking of results under a situation in which at any moment this process is performed considering global information (Section 2.1). In the metric-space indexes this can be seen very clearly. When the centers of clusters are selected considering the whole set of objects distributed onto the processors, the computations associated with the determination of the top-k results is significantly reduced. Intuitively this works well because all good outliers are included in the set of global centers. When ranking is based on local information, the outliers are distributed uniformly onto the processors, and therefore in each processor only a fraction 1/P participates in the local ranking. The experimental results have shown that local ranking achieves a very poor performance in this case. In the inverted files, when the ranking is performed using list pruning techniques the same also becomes evident. That is, when the ranking uses a global barrier at the ranker side to stop computations the performance improves significantly. In conventional search engines, each processor uses a local barrier to compute their best results to then send them to the ranker for merging. As the barriers use local information, the fraction of posting lists scanned is noticeably larger than in the case of global ranking. This degrades overall query throughput since queries require more processor time and disk bandwidth to be solved.

Acknowledgment

This work has been partially funded by FONDECYT 1060776.

References