A Search Engine Accepting On-Line Updates

Mauricio Marin[1], Carolina Bonacic[2], Veronica Gil Costa[3], and Carlos Gomez[1]

[1] Yahoo! Research, Santiago, University of Chile
[2] ARTECS, Complutense University of Madrid, Spain
[3] DCC, University of San Luis, Argentina

Abstract. We describe and evaluate the performance of a parallel search engine that is able to cope efficiently with concurrent read/write operations. Read operations come in the usual form of queries submitted to the search engine and write ones come in the form of new documents added to the text collection in an on-line manner, namely the insertions are embedded into the main stream of user queries in an unpredictable arrival order but with query results respecting causality. The search engine is built upon distributed inverted files for which we propose generic strategies for load balance and concurrency control.

1 Introduction

The inverted file is a well-known index data structure for supporting fast searches on very large text collections. Web search engines use this strategy to index huge collections of text. A number of papers have been published reporting experiments and proposals for efficient parallel query processing upon inverted files which are distributed on a set of P processor-memory pairs [1,2,3,4,5,6]. It is clear that efficiency on clusters of computers is only achieved by using strategies devised to reduce communication among processors and maintain a reasonable balance of the amount of computation and communication performed by the processors to solve the search queries.

An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes. To solve a query, it is necessary to get the set of documents associated with the query terms and then perform a ranking of these documents so as to select the top K documents as the query answer.

The two dominant approaches to distributing the inverted file onto P processors are (**a**) the document partitioned strategy in which the documents are evenly distributed onto the processors and an inverted file is constructed in each processor using the respective subset of documents, and (**b**) the term partitioned strategy in which a single inverted file is constructed from the whole text collection to then distribute evenly the terms with their respective posting lists onto the processors. In addition, some strategies which are hybrids between these two schemes have been proposed to improve load balance. The way the inverted file is partitioned onto the processors dictates the way in which the parallel processing of queries is performed.

Query operations over parallel search engines are usually read-only requests upon the distributed inverted file. This means that one is not concerned with multiple users attempting to write information on the same text collection. All of them are serviced with no regards for consistency problems since no concurrent updates are performed over the data structure. Insertion of new documents is effected off-line in Web search engines. However, it is becoming relevant to consider mixes of read and write operations.

For example, for a large news service we want users to get very fresh texts as the answers to their queries. Certainly we cannot stop the server every time we add and index a few news into the text collection. It is more convenient to let write and read operations take place concurrently. This becomes critical when one think of a world-wide trade system in which users put business documents and others submit queries using words like in Web search engines. Notice that the pageRank based ranking and lack of support for on-line updates promoted by Web search engines do not apply in this context. Also in this case it is feasible to keep all posting lists in main memory.

In this paper we present strategies and performance results for a search engine devised for this purpose. The search engine can be built upon the document or term partitioned approaches, and variations. We focus on the two most important factors affecting its performance, namely the load balancing strategies used to achieve good parallelism in query processing, and the concurrency control strategy used to avoid R/W conflicts in the underlying distributed inverted file. The contribution of this paper is that in both cases we propose efficient strategies and evaluate their comparative performance against alternative strategies. Another contribution of this paper is the design of the search engine itself, a design that (**i**) decouples ranking from posting list treatment in order to achieve a high rate of queries per second and (**ii**) is able to accommodate an efficient concurrency control algorithm by ways of the bulk-synchronous organization of the ranking and posting lists fetching processes. To the best of our knowledge no search engine which is able to cope efficiently with concurrent read/write operations has been proposed in the literature.

The remainder of this paper is organized as follows. Section 2 describes our search engine. Section 3 discusses load balancing strategies and section 4 concurrency control algorithms. Section 5 presents conclusions.

2 Search Engine

The parallel processing of queries is basically composed of a phase in which it is necessary to fetch parts of all of the posting lists associated with each term present in the query, and perform a ranking of documents in order to produce the results. After this, additional processing is required to produce the answer to the user. This paper is concerned with the fetching+ranking part and we are interested in situations in which a high traffic of queries is arriving at the search

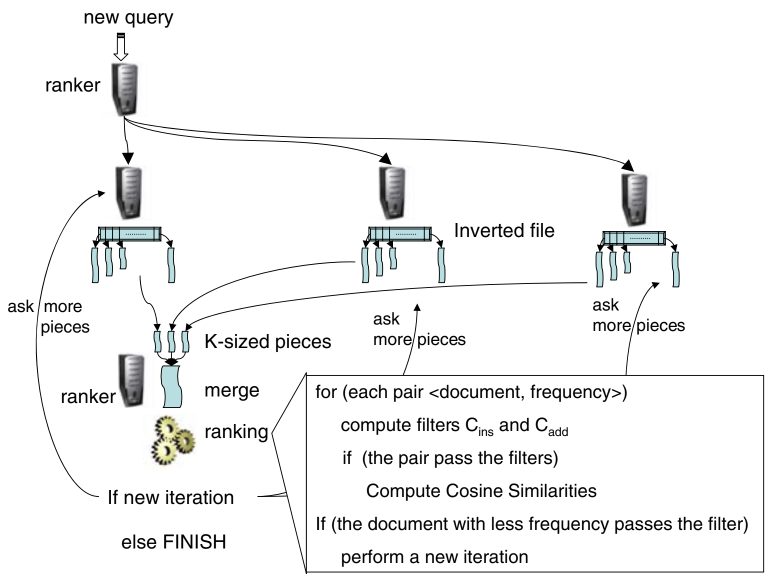
new query
ranker
Inverted file
ask more pieces
K-sized pieces
ask more pieces
ask more pieces
ranker
merge
ranking
If new iteration
else FINISH
for (each pair <document, frequency>)
compute filters C_{ins} and C_{add}
if (the pair pass the filters)
Compute Cosine Similarities
If (the document with less frequency passes the filter)
perform a new iteration

Fig. 1. Search engine: list fetching, ranking and iterations

engine and it is relevant to optimize the query throughput. At the parallel server side, queries arrive from a receptionist machine that we call the *broker*.

The broker machine is in charge of routing the queries to the cluster's processors and receiving the respective answers. It decides to which processor routing a given query by using a load balancing heuristic. The particular heuristic depends on the approach used to partition the inverted file. Overall the broker tends to evenly distribute the queries on all processors.

The processor in which a given query arrives is called the *ranker* for that query since it is in this processor where the associated document ranking is performed. Every query is processed using two major steps: the first one consists on fetching a K-sized piece of every posting list involved in the query and sending them to the ranker processor.

In the second step, the ranker performs the actual ranking of documents and, if necessary, it asks for additional K-sized pieces of the posting lists in order to produce the K best ranked documents that are passed to the broker as the query results. We call this *iterations*. Thus the ranking process can take one or more iterations to finish. In every iteration a new piece of K pairs (doc_id, frequency) from posting lists are sent to the ranker for every term involved in the query. See figure 1.

Under this scheme, at a given interval of time, the ranking of two or more queries can take place in parallel at different processors along with the fetching of K-sized pieces of posting lists associated with other queries. We assume a situation in which the query arrival rate in the broker is large enough to let the broker distribute $Q\,P$ queries onto the P processors.

We use the vectorial method for performing the ranking of documents along with the filtering technique proposed in [7]. Consequently, the posting lists are kept sorted by frequency in descending order. Once the ranker for a query receives all the required pieces of posting lists, they are merged into a single list and passed throughout the filters. If it happens that the document with the least frequency in one of the arrived pieces of posting lists passes the filter, then it is necessary to perform a new iteration for this term and all others in the same situation. We also provide support for performing the intersection of posting lists for boolean AND queries. In this case the ranking is performed over the documents that contain all the terms present in the query.

The search engine is implemented on top of the BSP model of parallel computing [8] as follows. 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. Thus at the beginning of each superstep the processors get into their input message queues both new queries placed there by the broker and messages with pieces of posting lists related to the processing of queries which arrived at previous supersteps. The processing of a given query can take two or more supersteps to be completed. All messages are sent at the end of every superstep and thereby they are sent to their destinations packed into single messages to reduce communication overheads. In addition, in the input message queues are requests to index new documents and merge them into the distributed inverted file the resulting pairs (id_doc,frequency).

2.1 Experiments

We use two text databases. The first one is a 2GB (and 12GB) sample of the Chilean Web taken from the `www.todocl.cl` search engine. The text is in Spanish. Using this collection we generated a 1.5GB index structure with 1,408,447 terms. Queries were selected at random from a set of 127,000 queries taken from the todocl log. The second collection is a set of crawled documents along with a query log from an experimental search engine at Yahoo! Labs. We took 300,000 random queries from the query log making sure that all terms are in the document collection. In practice we did not observe any significant differences in the results from both text collections and thereby in this paper we report results from the first one.

The experiments were performed on a cluster with dual processors (2.8 GHz) that use NFS mounted directories. This system has 2 racks of 6 shelves each with 10 blades to achieve 120 processors.

In every run we process 10,000 queries in *each* processor. That is the total number of queries processed in each experiment reported below is 10,000 P. For our collection the values of the filters C_{ins} and C_{add} were both set to 0.1 and we set K to 1020. On average, the processing of every query finished with

$0.6K$ results after 1.5 iterations. Before measuring running times and to avoid any interference with the file system, we load into main memory all the files associated with queries and the inverted file.

3 Load Balance

Key to the efficiency of search engines is the data structure used to support queries over the text collection. Inverted files are used for this purpose, in particular most search engines distribute the inverted file onto the processors using the document partitioned approach. The main advantages of this scheme are its suitability for boolean AND queries and simplicity of maintenance since inserting a new document requires updating the piece of inverted file stored in one processor only. Typically a certain number of documents are accumulated and inserted into a copy of the inverted file to then replace it for the new version.

However, the term partitioned approach has the advantage of being able to achieve a better performance and has deserved attention in the literature. Its main disadvantage lays in the problem of calculating the intersection of posting lists for AND queries. This because as terms are distributed onto the processors, the pairs (doc_id, frequency) tend to be in different processors with high probability, demanding as a result more communication to calculate intersections. How costly can be this increase in communication is matter of term distribution as we show in the following.

In figure 2 [left] we show a term distribution which makes more likely term co-residence for intersections using 32 processors. We obtained the normalized frequencies for each term distribution. From the query log, we obtain all possible pairs and compute a "hit" for each term doing as follows. If terms t_i and t_j are terms of a query q, we add 1 to the frequency of each term if t_i and t_j are in the same processor. The figure 2 [left] shows that the distribution of terms on processors labeled by v3 increases the probability of co-residence. In this case from the query log we detect the P^2 most frequent terms and distribute them circularly onto the processors.

For the rest of the terms we determine the processor in which they are stored by using the following strategy. From a query log we count the frequency in which pairs of terms (t_1, t_2) appear in queries (all permutations are considered for queries with three or more terms). We form a list sorted by decreasing frequency order. Then the pairs (t_1, t_2) are removed one by one from the front of the list and are placed on the processors using the following rules. If both t_1 and t_2 have not been assigned to a processor, place t_1 and t_2 in the processor with the least number of pairs. If one of the terms is already assigned to a processor, its companion goes to the same processor. This defines the initial allocation of terms to processors. We then use *tabu* search to improve on that. The method moves terms among processors by trying to reduce the BSP cost of executing the query log and using an optimization criteria to prune the search space. Distribution v1 applies this method without first distributing circularly the most frequent terms and distribution v2 distributes circularly the only first P most frequent terms.

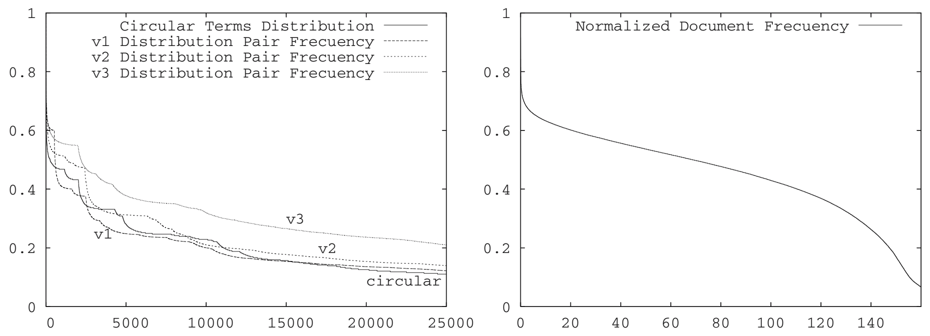

Fig. 2. Probability of processor co-residence for the most frequent query terms and documents containing terms in the query log. [left] the x-axis shows the normalized frequency of co-residence for the 25K most frequent pairs of terms in queries (0 is the most frequent one). [right] the x-axis shows documents sorted from the one containing most of the query terms to the one with least ones (values in thousands of docs).

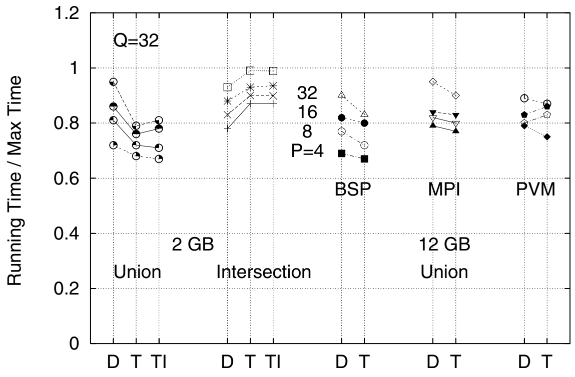

Fig. 3. Normalized running times for 4, 8, 16 and 32 processors under queries requiring union and intersection of posting lists. For the term partitioned index the intersection operation is performed under circular distribution of terms (T) and the v3 distribution of terms (TI). The document partitioned approach is denoted by D. The first part shows results for BSP over a 2GB text collection whereas the second part shows results for a 12GB text collection and inverted files implemented using the BSP, MPI and PVM communication libraries. Top curves are for P= 32 and bottom ones are for P= 4.

Figure 2 [right] is more useful in the context of the next section as it shows for each document the amount of relevant terms that are present in the query log. This indicates that there is a high probability that during the insertion of a new document in the collection some of its terms are likely to be identical to those coming from the user queries.

For the document partitioned approach (**D**) the load balancing heuristic is simple. The ranker is selected in a circular manner among the P processors. The broker performs a broadcast of every query to all the processors. But it does so in two steps. First and exactly as in the term partitioned approach, the broker

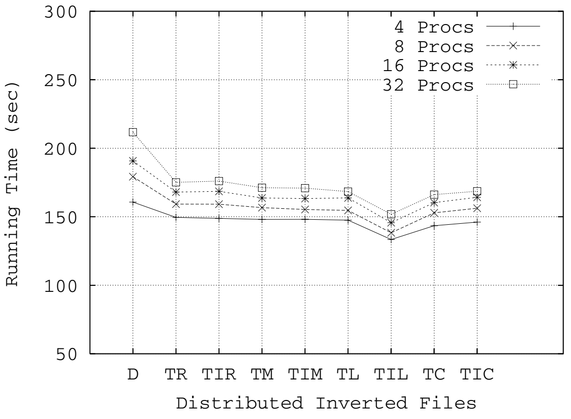

Fig. 4. Running times for 4, 8, 16 and 32 processors for different alternatives for load balancing computation and communication

sends one copy of each query to their respective ranker processors. Secondly, the ranker sends a copy of each query to all other processors. Next, all processors send K/P pairs (doc_id, frequency) of their posting lists to their rankers which perform the documents ranking. In the case of one or more query terms passing both filters, the ranker sends messages to all processors asking for additional K/P pairs (doc_id, frequency) of the respective posting lists.

In the term partitioned approach (**T**), for the case in which no intersection of posting lists is performed, we distribute the terms and their posting lists in an uniformly at random manner onto the processors (we actually 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 broker send the query to its ranker processor which upon reception sends messages to others asking for the first K pairs (doc_id, frequency) of every term present in the query. The same is repeated if one or more terms pass the filters.

In figure 3 we show results for the document and term partitioned approaches for distributed inverted files. We show running times for union and intersection of posting lists during ranking. For the term partitioned index if the terms are co-resident their posting lists are intersected locally. When the terms are not co-resident the complete smallest posting list is sent to the other processor. If the documents in the intersection pass the filters new pieces of posting lists are requested. In the document partitioned approach the intersections are always effected locally. We also present a comparison with inverted files implemented by using the asynchronous message passing approach to parallel computing upon MPI and PVM. Both indexes achieve competitive performance.

In figure 4 we show different strategies we devised to improve load balance in the term (T) partitioned approach and compare its running against the document (D) partitioned strategy. No intersections are performed. The letter R stands for ranker distributed circularly among the processors. Letter I indicates distribution v3 to improve locality for intersection operations. Letter M stands for upper limits to the number of ranking operations performed in every processor in each superstep. Letter L indicates limits only to the number of list fetches allowed in each processor and superstep. For both cases we used limit= $2.5\,Q$. Letter C indicates a case in which rankers keep in cache memory the pieces of posting lists arrived from other processors. The results show that the term partitioned strategy achieves good efficiency when we decouple ranking from list fetching. For ranking a good strategy is to distribute rankers circularly among the processors. The figure also shows that imposing upper limits to the number of list fetching performed in each superstep and processor can further improve running time. Performance also improves from the fact that in the strategies "I" we distribute circularly the most frequent terms onto the processors.

4 Concurrency Control

An important component of the proposed search engine is its support for concurrent write and read operations as new documents are expected to be arriving from the broker machine along with queries in an unpredictable manner. In the following we describe a very efficient strategy to support concurrent R/W operations over the document and term partitioned bulk-synchronous inverted files.

A first point to note is that the semantics of supersteps tell us that all messages are in their target processors at the start of each superstep. That is, no messages are in transit at that instant and all the processors are barrier synchronized. If the broker assigns a unique integer timestamp to every R/W operation that it sends to the processors, then it suffices to process all messages in timestamp order in each processor to avoid R/W conflicts. These timestamps are also used as the ids for the respective queries. To this end, every processor maintains its input message queue organized as a priority queue with keys given by id_query integer values. The broker selects the processors to send new documents in a circular manner. Upon reception, a document is parsed to extract all its relevant terms.

In the document partitioned inverted file the posting lists of all the parsed terms are updated locally in the same processor. However, this is not effected in the current superstep but in the next one in order to wait for the arrival of broadcasted terms which could have timestamps smaller than the one associated to the current document insertion operation. Instead the processor sends to itself a message in order to wait one superstep to proceed with the updating of the posting lists.

In the term partitioned inverted file the arrival of a new document to a processor is much more demanding in communication. This because once the document is parsed a pair (id_doc,frequency) for each term has to be sent to every processor holding the posting list of the respective terms. However, insertion of new document is expected to be comparatively less frequent than queries in real-life settings.

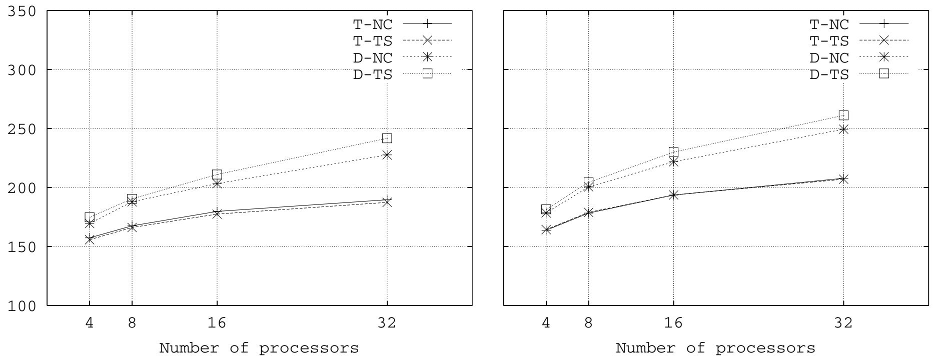

Fig. 5. Running times (sec) for processing 10, 000 queries in each processor by inserting Q= 32 new queries (read operations) in each superstep and processor respectively. With probability p a write operation is generated and new the document is formed with 100 words randomly generated from the *same* query log to increase R/W conflicts. D and T stand for document and term partitioned inverted files, and NC and TS stand for no-concurrency and timestamped-concurrency control respectively. Figure [left] shows results for a probability of document arrival per user query of p=0.25 whereas figure [right] shows results for probability p=0.5 (y-axis is the same to [left]).

A complication arises from the fact that the processing of a given query can take several iterations (supersteps). A given pair (id_doc,frequency) cannot be inserted in its posting list by a write operation with timestamp larger than the query being solved throughout several iterations. We solve this by keeping aside this pair for those queries and logically including the pair in the posting list for queries with timestamps larger than the pair one. In practice the pair is physically inserted in the posting list in frequency descending order but it is not considered by queries with smaller timestamps.

In figure 5 we show performance results for the concurrency control strategy. The figure shows the running time for both the document and term partitioned inverted files under a situation of doing nothing to prevent read/write conflicts and the case in which these conflicts are avoided using the bulk-synchronous timestamp based strategy. The results shows that the running time of the strategy is very similar to the running time of doing nothing, that is, a case in which R/W conflicts never take place and thereby it represents the "fastest" concurrency control algorithm.

5 Conclusion

We have presented the design of a search engine which provides an efficient support for the on-line arrival of new documents. The design can accommodate both the document and term partitioned approaches to distributing inverted files onto the processors. We have proposed efficient implementations of both approaches and found that even for intersection operations the term partitioned approach performs comparatively well whereas for cases in which this operation is not a requirement this strategy achieves a very high rate of queries per second.

It is not clear that for the type of application our search engine is intended for the intersection operations are a firm requirement for all queries. Moreover, an half-the-way case can be to perform the ranking of the K-sized pieces of posting lists without performing any intersection and once a sufficient amount of well ranked documents is reached (say $2K$), the ranker assigns the highest scores to the document that contain all the query terms. This is an issue we plan to explore in the near future.

Also, the adoption of the BSP model of computing allowed us to solve in a trivial manner the problem of controlling concurrency for read and write operations. The experimental results show that this strategy is very efficient.

Acknowledgment. Partially funded by Cyted Project 505PI0058.

References

1. Badue, C., Baeza-Yates, R., Ribeiro, B., Ziviani, N.: Distributed query processing using partitioned inverted files. Eighth Symposium on String Processing and Information Retrieval (SPIRE'01), pp. 10–20 (November 2001)
2. Buttcher, S., Clarke, C.: Indexing time vs. query time trade-offs in dynamic information retrieval systems. In: International Conference on Information and Knowledge Management, pp. 317–318 (2005)
3. MacFarlane, A., McCann, J., Robertson, S.: Parallel search using partitioned inverted files. In: 7th International Symposium on String Processing and Information Retrieval, pp. 209–220. IEEE Computer Society Press, Los Alamitos (2000)
4. Moffat, W., Webber, J., Zobel, Baeza-Yates, R.: A pipelined architecture for distributed text query evaluation. Information Retrieval (October 5, 2006)
5. Orlando, S., Perego, R., Silvestri, F.: Design of a parallel and distributed web search engine. In: Proc. 2001 Parallel Computing Conf., pp.197–204 (2001)
6. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys 38(2) (2006)
7. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. Journal of the American Society for Information Science 47(10), 749–764 (1996)
8. Valiant, L.G.: A bridging model for parallel computation. Comm. ACM 33, 103–111 (1990)