Comparative Study of Concurrency Control on Bulk-Synchronous Parallel Search Engines

Carolina Bonacic[1] **and Mauricio Marin**[2]

[1] ArTeCS, Complutense University of Madrid, Spain
CEQUA, University of Magallanes, Chile
cbonacic@fis.ucm.es

[2] Yahoo! Research, Santiago, Chile
mmarin@yahoo-inc.com

In this paper we propose and evaluate the performance of concurrency control strategies for 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 operations 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.

1 Introduction

Web Search Engines use the inverted file data structure to index the text collection and speed up query processing. 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[1–5]. 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 ids 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.

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.

The concurrency control algorithms we propose in this paper are designed upon a particular way of performing query processing. In section 2 we describe our method for read-only queries and in section 3 we extend it to support concurrency control under read/write operations. Section 4 presents an evaluation of the algorithms and section 5 presents concluding remarks.

2 Query Processing

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. 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 processors and receiving the respective answers. For each query the method produces the top-K documents that form the answer.

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.

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.

The search engine is implemented on top of the BSP model of parallel computing[6] 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 one message per destination 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).

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, $h\,G$ 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 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 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 the document partitioned approach the broker performs a broadcast of every query to all processors. First and exactly as in the term partitioned approach, the broker sends one copy of the query to their respective ranker processors. Secondly, the ranker sends a copy of every query to all other processors. Next, all processors send K/P pairs (doc_id, frequency) of their posting lists to the ranker which performs the documents ranking. In the case of one or more query terms requiring another iteration, the ranker sends messages to all processors asking for additional K/P pairs of the respective posting lists.

In the term partitioned approach we distribute the terms and their posting lists in an uniformly at 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 broker sends the query to its ranker processor which upon reception sends messages to the other processors holding query terms, to get from them the first K pairs (doc_id, frequency) of every term present in the query. After the ranking of these pieces of posting lists, new iterations (if any) are processed in the same way. Similar to the document partitioned index, the ranker is selected in a circular manner and without paying attention in which processors are located the query terms. Notice that the effect of imbalance due to frequent terms in queries is minimized because the ranking computations (which is most costly component in the running time) are well balanced and disk accesses for frequent terms are reduced by LRU disk caching of the posting list segments being required by iterations.

3 Concurrency Control Strategies

Timestamp protocol. 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 correlative timestamp to every query and document that it sends to the processors, then it suffices to process all messages in timestamp order in each processor to avoid R/W conflicts. To this end, every processor maintains its input message queue organized as a priority queue with keys given by id_query integer values (timestamps). The broker selects the processors to send documents and queries in a circular manner. Upon reception of a document, the respective processor parses it to extract all the 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 broadcast terms belonging to queries

placed in the same superstep, which can have timestamps smaller than the one associated with the current document insertion operation. To this end, 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 the respective processor holding the posting list. However, insertion of a new document is expected to be comparatively less frequent than queries in real-life settings.

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 one associated with the pair. 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. Garbage collection can be made periodically every certain number of supersteps by performing a parallel prefix operation in order to determine the largest timestamp among the fully completed query timestamps at the given superstep.

Exclusive write supersteps. Another strategy is to let the broker control the order in which the queries and documents are sent to the cluster processors. Write operations associated to each new document are granted exclusive use of the cluster. That is, upon reception of a new document, the broker waits for the current queries in the cluster to be completed and queues up all new arriving queries and documents. Then it sends the new document to the cluster where it is indexed and the respective posting lists are updated. Once these write operations have finished the broker let the next set of queued read operations to be processed in the cluster until the next document and so on.

Two-phases locks protocol. Concurrent access to terms and their respective posting lists can be protected by using read and write locks[3]. We employ a distributed version of the locks protocol in which every processor is in charge of administering a part of the lock and unlock requests associated with the processing of queries and documents. To this end, we assume terms evenly distributed onto the processor using the rule id_term % P. To insert a new document it is necessary to request a write lock for every relevant term. This is effected by sending to the respective processors messages with the lock requests. Once that these locks are granted and modifications to the inverted file are finished the locks are released by sending unlock messages. Similar procedure is applied for read operations required by queries but in this case one or more read locks can be granted for each term. Write locks are exclusive.

Optimal protocol. For comparison purposes we use this hope-for-the-best protocol. This is a case in which we assume that queries and documents are submitted to the search engine in a way in which read/write conflicts never takes place. To this end, we process the queries and documents without including extra-code and messages to do any concurrency control. This represents the best a protocol can do in terms of total running time.

3.1 Tradeoff between the term and document partitioned indexes

To motivate the comparative study presented in the next section we describe the BSP cost of the timestamp protocol. This to expose the tradeoff of using either of these distributed

inverted files.

Let us assume that at the beginning of a superstep i there is a certain average number Q of new queries per processor per superstep and t_q terms per query on the average. The average length of posting lists is ℓ. Also an average of D new documents arrive per processor per superstep containing on average t_d relevant terms. The steps followed by the term and document partitioned indexes are the following.

[*Term partitioned index*]

Superstep i Each processor obtains from its input message queue the arriving messages in timestamp order where messages can be queries or documents.

- For each document send all its terms to the processors holding the respective posting lists. Cost $t_d\, D\,(1+G)+L$.
- For each query send its terms to the processors holding the respective posting lists. Cost $t_q\, Q\,(1+G)+L$.

Superstep $i+1$ Each processor obtains the new messages in timestamp order from its input queue.

- For each message containing the triplet (id_term, id_doc, freq) update the posting list associated with the term. Cost $t_d\, D$ updateListCost(ℓ).
- For each message containing a query term retrieve from disk the respective posting list to obtain the best K pairs (id_doc, freq) and send them to the ranker processor. Cost $t_q\, Q\,[\,\text{diskCost}(\ell)+K\,G\,]+L$.

Superstep $i+2$ In each processor and for each query started in the superstep i get the messages containing the K pairs (id_doc, freq), effect the final ranking (we assume queries with one iteration) and report to the broker the top-K documents. Cost Q rankingCost($t_q,\ K$) $+\,L$.

On the other hand, in the document partitioned index the timestamp strategy works as follows.

[*Document partitioned index*]

Superstep i Each processor obtains from its input message queue the arriving messages in timestamp order where messages can be queries or documents.

- Store each document and wait to the next superstep.
- Send each query to all the processors (broadcast). Cost $t_q\, Q\,(1+P\,G)+L$.

Superstep $i+1$ Each processor obtains the new messages in timestamp order from its input queue.

- For each message containing a document extract all its relevant terms and update the respective posting lists. Cost $t_d\, D$ updateListCost(ℓ/P).
- For each message containing a query obtain the posting lists associated with its terms and get the best K/P documents and send them to the ranker processor. Cost $t_q\, Q\, P\,[\,\text{diskCost}(\ell/P)+(K/P)\,G\,]+L$.

Superstep $i + 2$ Each processor and for each query started in the superstep i, obtain the P messages per query with K/P pairs (id_doc, freq) and perform the final ranking and report to the broker the top-K documents. Cost Q rankingCost(t_q, K) $+ L$.

Basically the difference between the two strategies is given by the tradeoff in communication arising in the first superstep. That is, the tradeoff is given by the factors $t_d\,D\,G$ and $t_q\,Q\,P\,G$ in the cost of the term and document partitioned indexes respectively.

4 Evaluation

For the performance evaluation we used a text database which is a 12GB sample of the Chilean Web taken from the `www.todocl.cl` search engine. Using this collection we generated an 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 query log contains 33,000 terms. 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. All the results were obtained using 4, 8, 16 and 32 processors as we found this to have a practical relation between the size of the inverted file and the number of processors. With 32 processors we observed significant imbalance in some cases. In addition, the number of queries available for experimentation makes 32 processors large enough. 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. To evaluate the effect of write operations on the inverted files we inserted documents taken at random from the same text sample. On average documents contain 200 relevant terms which are also in the query log. We observed a large number of read/write conflicts during execution of query and document operations.

In the experiments the first fact to become clear is the poor performance achieved by the locks protocol. We observed that its throughput was too small causing an increasingly large queue of pending operations. The assignment of a write lock to each relevant term of the document being inserted/updated restrains significantly the rate of user queries processed per unit time. Figure 1 shows the percentage of scheduled write/read operations that are completed in each superstep considering that in each superstep new queries and documents are injected using P= 4 processors. In this experiment the total number of document insertion operations is a 22% of the total number of operations (queries plus documents). The figure shows that on the average a 16% of the queries and a 8% of the document insertions are completed which lead to an average of 10% of unlock operations completed per superstep.

In figure 2 we show results for the timestamp (TS) and exclusive supersteps (ES) protocols for both the term and document partitioned indexes. We also show results for the optimal protocol. Also the results are for different query traffic Q and different number K of documents presented to the user as the answer for each query, and number of processors P= 4, 8, 16 and 32. The curves are labeled “R” for the case in which 80,000 queries are distributed uniformly at random onto the P processors and no documents are inserted in the inverted file, and “W” for the case in which the 80,000 queries are randomly mixed with other 80,000 operations of insertion of new documents.

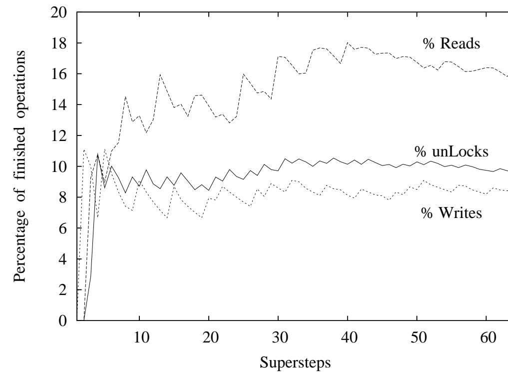

Figure 1. Locks protocol. The curves show the percentage of read, write and unlock operations finished.

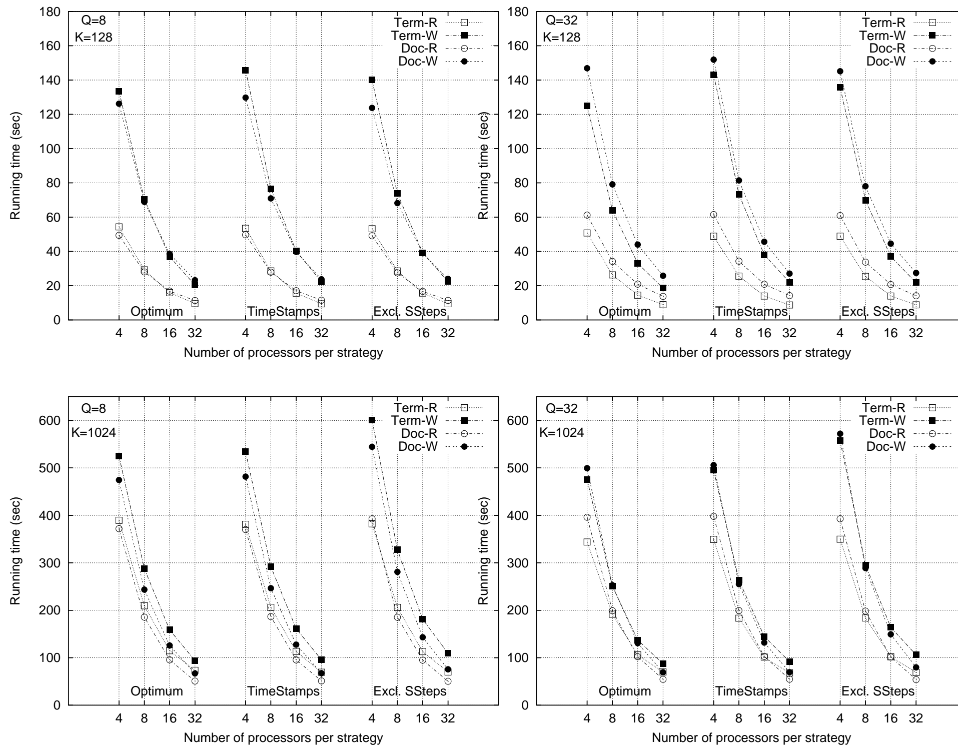

Figure 2. Running times for the timestamp and exclusive supersteps protocols.

The results in figure 2 show that the performance of both protocols is fairly similar to the performance of the optimal protocol. This indicates that overheads are small. The term

partitioned index performs slightly better than the document partitioned index. Even for the case with large number of write operations the term partitioned index performs efficiently. The ratio $t_d\,D\,G\,/\,t_q\,Q\,P\,G$ is not detrimental to the term partitioned index. This is because the cost of ranking queries can be more significant than the cost of indexing and updating the posting lists. This is seen in the figure 2 when one compares the difference between the results for K= 128 and K= 1024. Low traffic $Q = 8$ tends to produce higher running times as this causes imbalance across processors. For the case K= 1024 the ES protocol tends to be noticeably less efficient than the TS protocol. In this case imbalance is more significant for two reasons: each document is indexed in one processor during the write-only supersteps and posting lists are updated in parallel, and ranking of queries is split in several supersteps which reduces the average number of ranking operations per processor per superstep.

5 Conclusions

We have presented a comparison of different protocols to perform concurrency control in distributed inverted files. The particular manner in which we organize query processing and insertion of new documents allows a suitable bulk-synchronous organization of parallel computation. This provides a simple but very efficient timestamp protocol for controlling concurrency. The empirical results show a performance similar to the optimal protocol.

We have also found that the write exclusive supersteps protocol achieves competitive performance. In this case it is not necessary to barrier synchronize the processors as it only requires to detect when all current queries have been finished to send the document insertion operation and then detect when all associated write operations have finished to start a new set of queries and so on. This scheme can be implemented in message-passing asynchronous implementations of inverted files in which case operation termination must be handled using extra messages. Even in this case it is clear that this scheme is more efficient than the locks protocol since this protocol requires similar number of extra messages and our results show that it is not efficient because it achieves a very low throughput.

Acknowledgment: Partially funded by FONDECYT Grant 1060776.

References

1. A. Arusu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACM Trans.*, 1(1):2–43, 2001.
2. A. Barroso, J. Dean, and U. H Olzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2002.
3. A.A. MacFarlane, S.E. Robertson, and J.A. McCann. On concurrency control for inverted files. In *18th BCS IRSG Annual Colloquium on Information Retrieval Research*, pages 67–79. March, 1996.
4. W. Moffat, J. Webber, Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, Oct. 5 2006.
5. S. Orlando, R. Perego, and F. Silvestri. Design of a parallel and distributed web search engine. *In Proc. 2001 Parallel Computing Conf.*, pages 197–204, 2001.
6. L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.