Optimistic Concurrency Control for Inverted Files in Text Databases

Mauricio Marín *
Computing Department, University of Magallanes, Chile

ABSTRACT

Inverted files are frequently used as index data structures for very large text databases. Most applications of this data structure are for read-only query operations. However, the problem of introducing update operations has deserved little attention so far and yet it has important applications. In this paper we propose an optimistic concurrency control algorithm devised to handle mixes of update operations and read-only queries efficiently. We work on top of the BSP model of parallel computing and take advantage of a BSP Time Warp realization to formulate the proposed algorithm. We present a comparison with the traditional lock-based approach and the results show that optimism is particularly efficient in this case.

Keywords: Information Retrieval, Text Databases, Parallel Computing, Concurrency Control, BSP Computing, Synchronization Protocols.

1. Introduction

This paper is concerned with the synchronization of *text* database R/W query operations (transactions) running on a parallel distributed-memory environment which supports an architecture independent and cost predictable model of computation. The model of computation is the so-called *bulk-synchronous parallel* (BSP) model [14]. The text database is assumed to be distributed on P processors, and transactions are allowed to read/write data from/on any processor. Associated with the text database there is an "index" which is a data structure devised to speed-up the processing of queries. We use the classic inverted file [2] for this purpose. New text comes in the form of "documents" that are added to the database which also implies to update the index data structure. For the sake of simplicity we assume transactions composed of just one query operation be it a read-only or a write operation (actually this is most frequent type of transaction in text databases). Extension to two or more operations is straightforward.

We compare an optimistic event-synchronization protocol used in both application domains with the two-phases lock protocol which is commonly used in relational database management systems. We have adapted those protocols to the text database context. This is a quite extreme case since every time a document is added/removed a significant number of index's data items must be updated. Moreover, in applications based on natural language text it is expected that a fairly small subset of the index's data items are the ones that are most referenced (those related to the most popular words of the particular language).

Our results show the better comparative performance of the optimistic protocol and its clear potential for achieving scaleable performance on architectures ranging from expensive super-computers to clusters of PCs. Another contribution of this paper is the specific BSP realization of the synchronization protocols we use in our experiments. To the best of our knowledge, this particular topic, namely text databases transaction processing under the BSP model of computation, has received no attention in the literature and thereby our discussion on previous work and reference list is necessarily short. Related work on BSP and databases systems can be found in [10] and in the references there mentioned.

The bulk-synchronous parallel (BSP) model of computing [9, 14, 13, 12] has been proposed to enable the development of portable and cost-predictable software which achieves scalable performance across diverse parallel architectures. BSP is a distributed memory model with a well-defined structure that enables the prediction of running time. Unlike traditional models of parallel computing, the BSP model ensures portability at the very fundamental level by allowing algorithm design to be effected in a manner that is independent of the architecture of the parallel computer. Shared and distributed memory parallel computers are programmed in the same way as they are considered emulators of the more general bulk-synchronous parallel machine.

2. BSP server

We assume a server operating upon a set of P identical machines, each containing its own main and secondary memory. We treat secondary memory like the communication network. That is, we include an additional parameter D to represent the average cost of accessing the secondary memory. Its value can be easily determined by benchmark programs available on Unix systems. The textual database is evenly distributed over the P machines. If the whole database index is expected to fit on the P sized main memory, we just assume $D = 1$.

Clients request service to one or more broker machines, which in turn distribute them evenly onto the P machines implementing the server. Requests are queries that

*Partially supported by Fondecyt project 1030454. Author's E-mail: Mauricio.Marin@umag.cl

are solved by using an index data structure distributed on the P processors. We assume that the index is implemented using an inverted file which, as described in the next section, is composed of a vocabulary (set of terms) and a set of identifiers (inverted list) representing all the documents that contain at least one of the words that are members of the vocabulary. The inverted file data structure enables the efficient retrieval of all identifiers for which a given term appears in the respective documents.

We assume that under a situation of heavy traffic the server is able to process batches of $Q = q\,P$ queries. Every query is composed of one or more vocabulary terms for which it is necessary to retrieve all document identifiers associated with them. Only the identifiers of the most relevant documents are presented to the user, namely those which more closely match the user information need represented by the query terms. For this, it is necessary to perform a ranking of documents. A widely used strategy for this task is the so-called vector model [2], which provides a measure of how close is a given document to a certain user query. We assume that the reader is familiar with this method and overall terminology [2].

In order to better exploit the available parallelism we try to minimize the amount of sequential work performed by the broker machine. We restrict its functionality to receive user requests, distribute the queries onto the processors (uniformly at random), receive the best ranked documents (K in total) from the server, and pass them back to the user.

The two most basic operations related to providing answers to user queries are left to the parallel sever. That is, the retrieval of document identifiers and its respective ranking. Both operations are effected in parallel where the broker is responsible for scheduling those in a manner that keeps load balance of processors work as close to the optimal $1/P$ as possible [5].

3. Inverted Files

The inverted files (lists) [2] are a popular data structure for supporting query processing in textual databases. We have proposed an efficient parallelization of inverted lists in [5], which can be implemented using any communication library. Its design combines the best of previous approaches: it mixes the so-called local and global index approaches into a single inverted list parallel algorithm which is more robust and efficient. Previous work on parallelization of inverted files can be found in [1, 3, 6, 11],

For a collection of documents the inverted file strategy can be seen as a vocabulary table in which each entry contains a term (relevant word) found in the collection and a pointer to a list of document's identifiers (inverted list) that contains such term. Thus, for example, a query composed of the logical AND of terms 1 and 2 can be solved by computing the intersection between the inverted-lists associated with the terms 1 and 2. The resulting list of documents can be then ranked so that the user is presented with the most relevant documents first (the technical literature on this kind of topics is large and diverse, e.g., see [2]). Parallelization of this strategy has been tackled using two approaches.

In the local index approach the documents are assumed to be uniformly distributed onto the processors. A local inverted-lists index is constructed in each processor by considering only the documents there stored respectively. We thus have P individual inverted-lists structures so that a query consisting of, say, one term must be solved by simultaneously computing the local sub-lists of document identifiers in each processor, and then producing the final global list from these P local sub-lists.

The BSP realization of the local index approach is as follows. Once the broker machine routes by ways of the hashing function a term w belonging to a query u to processor i, this processor broadcasts the term w to all other processors in the current superstep. Every processor does the same for each term they receive. In the following superstep, all processors scan their local inverted lists to obtain the sub-list of document identifiers for each term they received in the previous broadcast. These sub-lists are then sent to the processors acting as rankers. Thus if processor k happens to be the ranker for query u, the sub-list associated with term w in processor i along with the ones located in all other processors for term w, are routed to processor k. The same is effected for all other terms belonging to the query u so the processor k can perform the final ranking in the following superstep. The size of these sublists is reduced by performing a pre-ranking before sending them to their rankers. Also if two terms of query u happens to be in the same processor a pre-merging is performed (see [1] for this kind of optimizations). The whole process of a query u takes 3 supersteps to complete and send back the final list of document identifiers to the broker.

The second approach is the so-called global index. Here the whole collection of documents is used to produce a single inverted lists index which is identical to the sequential one. Then the T terms that form the global term table are uniformly distributed onto the P processors along with their respective lists of document identifiers. This is done by ways of the same hashing function employed by the broker. Thus, after the mapping, every processor contains about T/P terms per processor. In the local index case, each processor contains the same T terms but the length of document identifier lists are closely a fraction $1/P$ of the global index ones.

The BSP realization of the global index is as follows. Like the previous strategy, each term is routed to one server processor by the broker. For each term w belonging to a query u the inverted lists associated with terms of u are retrieved in their respective processors. Then these lists are sent to the ranker processor defined for the query u to then proceed in the next superstep like the local inverted lists case. The whole process takes 2 supersteps to complete.

Practice has shown that the global and local approaches can outperform each other under different situ-

ations. Better combine the two, or even put things in an intermedia situation by thinking in terms of inverted lists partitioned in buckets which are evenly distributed onto the processors.

It is straightforward to combine into a single BSP algorithm the two above described inverted files strategies. Each processor maintains a hash table to keep information about which terms are kept as in the local index case and which as in the global one. The following pseudo-code shows the superstep executed by a BSP server using a composite inverted list index to solve user queries,

```
while(true)// Every BSP processor.
{
 -Receive new messages, put them in queue Q.
 -Foreach message msg in the queue Q do
   switch( msg.type )
   {
    case BROKER://new term from the broker.
      if ( IsLocal(msg.term) == true )
        Broadcast(msg.term);
      else
      {//retrieve and sub-rank document list.
        List= getInvertedList(msg.term);
        subList= preRanking(List);

       //buffer message for ranker processor.
        bufferMsg(msg.ranker,RANKING,subList);
      }
      break;
    case BROADCAST://broadcast term reception.
      List= getInvertedList(msg.term);
      subList= preRanking(List);
      bufferMsg(msg.ranker,RANKING,subList);
      break;
    case RANKING:
      if ( queueSize(msg.queryId) ==
                          msg.numTermsQry )
      {
        Set = dequeueAll(msg.queryId);
        List= CalculateFinalRanking( set );
        bufferMsg( broker, SERVER, List);
      }
      else//queue up to wait for more terms.
       enqueue(msg.queryId,msg);
   }
 -Send all buffered messages to their target
   processors and synchronize processors.
}
```

A term is treated as global or local depending on the size of its associated inverted list. We set the maximum size of a list to be the one which produces the same ratio of computation to communication than the global inverted list approach. List sizes below this maximum are treated as in the global index case whereas sizes above the maximum are treated as in the local index one.

This straightforward combination of the local and global approaches is a strategy which we have found to be practical, efficient and very simple to implement. Its efficiency comes from the fact that most queries containing relevant terms tends to have inverted lists of small sizes.

4. Synchronization protocols

In our setting, every processor of the BSP machine must execute R/W operations of a large number of transactions (queries). Every transaction is created in a given processor so that all processors maintain about the same number of them. When a transaction must perform a read or write operation on a data item located in another processor, it performs such task by sending messages to the target processor. Messages take one superstep to reach their destinations.

In the **two-phases protocol**, transactions first request locks on the subset of index-terms that are part of a read-only query or the relevant terms found in the document being added to the collection. After all of the locks have been granted, the associated operations are allowed to take place and the locks are released. Deadlocks are avoided by asking locks in lexicographic term order. A direct realization of this protocol on a BSP machine is to use one superstep for sending a message to request a single lock, then wait one superstep for the remote processor to make all arrangements for granting the lock, and finally at the next superstep receive the grant notification, then proceed with the same procedure for the next lock, and so on. If the required lock is being held by another transaction, it is necessary to wait until this transaction releases the lock. Read locks are answered with the data itself to be read. Write locks are requested by sending into the same message the new data to be written. That is, no additional message traffic is necessary for effecting the R/W operations. All messages releasing the granted locks can be sent in the same superstep.

A more aggressive strategy is to request all the required locks in the same superstep, and then wait during one or more supersteps to receive all the pending lock authorisations. However, this introduces deadlock problems since the above mentioned global order can be broken. A solution is to define priorities for the transactions, and use such values to determine which transaction should get the lock on a certain data item at a given superstep. Priorities must be unique across all transactions which leads to the problem of getting unique and increasing integer values in a distributed environment. In addition, read operations have to be treated as exclusive ones, like writes, since at all times only the transactions with the best global priorities should get the locks they request for. [At the time of writing this paper we have not been able to figure out a realization of this aggressive strategy which be significantly more efficient than the straightforward one described above. So we use the former one in our experiments. Similar strategy can be used in the protocol described below, but we exclude this possibility in order to compare both approaches under the same context.]

The **Time Warp protocol** is a popular event synchronization strategy for parallel discrete event simulation

[4]. This asynchronous and distributed memory algorithm is based on the optimistic assumption that no events will probably get into conflict with each other, and if that situation happens to occur a correction procedure is executed by moving backwards the computation, correcting the error, and then moving it forward again but this time taking into consideration the cause of the trouble. The same strategy can be applied to the parallel processing of transactions. That is, they are allowed to perform their R/W operations at will, but each time a data item is read or written a consistency check is executed to detect if it necessary to do a roll-back of all causally related transactions or let them continue forward.

Our realization of this strategy for the BSP setting is as follows. Transactions are created in any superstep and each of them is composed of R/W operations. A timestamp is assigned to each transaction. This is an increasing integer number. All operations of a given transaction receive the transaction timestamp and the protocol is in charge of ensuring that all operations on records are done in increasing timestamp order. Whenever an operation breaks this rule, all already-executed operations on the involved record that have timestamps greater than the new one are undone and re-executed on the record to obtain the right sequence. Only read operations are allowed to be done in different timestamp order as long as no write operation should have been executed in between. When an operation of a given transaction is undone, it is also necessary to undone all following operations of the same transaction which have already been executed on other records. Note that these records can be located in other processors. Then these operations must be re-executed again since each one in the sequence can depend on the previous one. All this process is call a *roll-back*. Efficiency depends heavily on the amount of roll-backs performed during the computation. Transactions are committed when all their operations become ones with timestamps less than the smallest timestamp of any operation waiting to be executed (this considering all processors).

In [4] we propose an efficient BSP algorithm for Time Warp on BSP Computers which can be easily adapted to support this strategy. This can be done as follows. Every processor keeps a priority queue where priorities values are the operations timestamps. No operations from different transactions can have the same priority value and operations belonging to the same transaction can be given a second priority value in accordance with their relative ordering within the transaction. To reduce the number of roll-backs we use the local priority queues in a way that emulates a global priority queue. Note that if all operations were taken from this global priority queue, no roll-back would ever occur. In each superstep, each processor executes the n operations with the highest priorities (least numerical values). Every time an error is detected, all the operations are returned to the local priority queues. Errors occur because messages containing read/write operations take one superstep to reach their destinations, and this causes that in a given superstep the processors do not contain the correct first n priorities. Some of them are due to arrive in the following supersteps. However, already-executed and new operations are treated identically for the purpose of determining when the upper limit n is reached. This is precisely what tends to emulate the global queue since a processor working at a high error rate will be kept restrain from advancing too far in the processing of operations with low priorities. The value of n can be calculated automatically during execution as we propose in [4].

5. Performance Analysis

In this section we empirically compare the two synchronization strategies above described by using a demanding workload. Our performance metrics are independent of the particular implementation of the protocols since we measure the amount of computation, communication and synchronization required to complete a given instance of the work-load.

We worked with a 2GB sample of the chilean web and a query log from www.todocl.cl. This gave us a realistic setting both on the set of terms that compose the text collection and the type of terms that typically are part of user queries. Transactions were generated at random by taking terms from the query log. We started with 60% of the text collection and increased it by including the remaining 40% divided in documents as part of the write transactions generated at random.

On a P processors BSP machine we initially create T transactions per processor. Every time a transaction finishes the execution of all its operations, a new one is created so that the total of $T \cdot P$ transactions is kept constant throughout the whole experiment. In each work-load instance, namely different values for P and T and other parameters described below, the experiment ends after a very large number of superteps is executed. When a new transaction is created, it is given a random number N of operations to be executed. All performance metrics are normalized to the amount of something done per superstep. Each instance of a experiment is executed several times, each with a different random number seed.

Our performance metrics are the following: (**i**) Average number of transactions processed per superstep S considering the effect of all the processors; (**ii**) Average load balance in computation C measured as the average maximum across supersteps of the amount of computation registered in each processor divided by the observed average. We count as 1 the execution of a single R/W operation. (**iii**) Average load balance in communication M which is similar to C but counting messages sent at the end of each superstep. In other words, we measure performance in terms of the costs of synchronization, communication and computation. In this case, an algorithm is more efficient than another if it has less activity in synchronization (supersteps), less communication, and computation is

well balanced across processors (the same is valid for communication).

The best performance is achieved in a situation in which no R/W conflicts ever take place so that all transactions are able to execute their operations as fast as possible with no waiting supersteps. For a remote read operation it is necessary to wait one superstep for the data to come over. Remote write operations require no waiting superstep. In addition, in this best case scenario, transactions can send all their read requests at the very first superstep in which they are created and start their execution. Thus each transaction requires at least three supersteps to complete its execution, where in the last one all the operations are executed in bulk and write messages are sent to remote processors (this assumes that at least one read is in every transaction). Then the average number of transactions processed per superstep S is given by $T \cdot P/3$. Load balance in computation and communication tends to the optimum $C = 1$ and $M = 1$ as T increases since this is equivalent to the average occupancy of the "balls thrown into a set of baskets" problem.

Note that synchronization strategies like the optimistic one have the potential for achieving the above optimum performance as the number of relevant terms per processor R is scaled up whilst the number of transactions per processor T is kept fixed and large, since this reduces the probability of two given transactions willing to access the same record during the same superstep. The aggressive two-phases protocol with priorities can also present the same behaviour. However, taking a more realistic approach we are going to assume that it is not possible to send all read messages in the same superstep in which the transaction is created. That is, we assume that reads must be executed one after the another so that, for example, if 3 consecutive reads are required, each in a different processor (our case with high probability $1-1/P$), the machine will use at least 6 supersteps to complete the reads. Of course, we are not interested in high performance at the individual transaction level but at the overall level since our goal is to efficiently process a large number of transactions during a long period of time wherein transactions are created at any point (but uniformly distributed) of this period. That is, transactions are created in any superstep of the computation. Note that our workload achieves this because it maintains a constant number of transactions per processor and the creation of new transactions is distributed along the time because the number of operations per transaction is a random variable. A new transaction is created as soon as an old one finishes the execution of all its operations.

The standard realization of the two-phases lock protocol will in the best case require a sequence of $2N$ supersteps to complete all lock requests and then proceed to execute all associated operations in bulk. Thus, on the average, the total number of transactions per superstep in this protocol is $T \cdot P/(2\,\mathrm{E}[N])$. Since N is a random number, the bulk of operations will tend to ocurr in different supersteps which reduces the total amount operations per superstep that are executed by a factor of $1/(2\,\mathrm{E}[N])$. Less

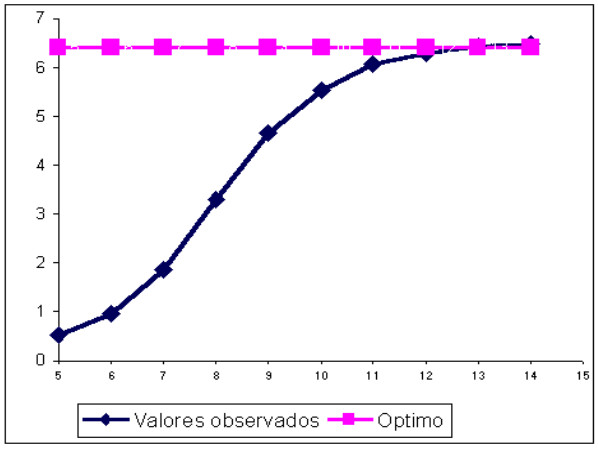

Figure 1. Number of processed transactions per superstep. The x-axis is in log_2 scale.

operations per superstep, leads to poorer load balance in computation and communication.

To observe how the lock protocol achieves the above optima we performed experiments with the work-load using $E[N] = 10$, $P = 16$, $T = 8$ and R= 32, 64, 128, 256, ..., 16384. The results are shown in figure 1. This figure shows that it is necessary a large number R of most-referenced relevant terms per processors to achieve the optimal number of transactions per superstep. For smaller values of R contention for the same records becomes intensive. Load balance in computation and communication is not good since typical C and M values were in the range 4 to 6. However, this problem has an easy solution. As we know what the optimal S should be, we can estimate the average number of operations executed in each superstep. (These values can be measured empirically for any system during execution as proposed in [4]). A factor of this estimate can be used as an upper limit to the number of operations executed per processor in each superstep. If in a given supersteps this maximum is reached and there are more operations to be executed, these are processed in the following superstep and so on. Using a factor 1/2 we observed that load balance improved significantly at the cost of increasing about three times the total number of supersteps. This reduces three times the values shown in figure 1.

Though the strategy used by the Time Warp algorithm is quite different to the one used by the lock protocol, the optimal value for the expected number of transactions per superstep S is the same value $T \cdot P/(2\,\mathrm{E}[N])$. In Time Warp a transaction sends messages containing R/W operations where reads take, in the best case, two supersteps to complete. If no roll-back takes place, then the optimal S is achieved. We performed similar experiments to that of figure 1. The results for S are very similar to those of figure 1 for large R. However, in the region of small R the Time Warp protocol achieves much quicker near optimal S val-

ues. But much more important is the fact that load balance is near optimal (on average $C = 1.2$ and $M = 1.6$). Similar load balance was achieved with the lock protocol but S had to be decreased three times. Roll-backs were below 5% in all cases considered.

6. Conclusions

We have presented and analysed two bulk-synchronous parallel algorithms for synchronizing concurrent text database transactions running on a distributed memory environment. Overall, we have observed that the BSP Time Warp algorithm we propose in this paper outperforms an efficient BSP realization of the traditional two-phases lock protocol by a wide margin. BSP Time Warp requires a near optimal number of supersteps to process a large number of transactions whilst at the same time it also achieves near optimal load balance.

Note that the amount of computation performed by both algorithms is similar since we have observed that the number of roll-backs in Time Warp is not significant whereas the overheads associated with locks administration are avoided completely. Thus what matters in the comparison is balance in computation and communication and the amount of synchronization of processors.

We are at present exploring ways to take advantage of the fact that the set of document-ids to be presented to the user as the answer to a read-only query must be ranked and shown in parts. Usually it is expected that user will only be interested in a few of the best ranked documents. The rest are discarded. Thus it is convenient to keep inverted lists associated with every term sorted by the respective relevance of the term in the document. During the ranking only a sub-set of every processor are considered which relaxes the conflict with the insertion in the inverted lists of document-ids for which the respective term is not very relevant in the associated document.

References

[1] C. Badue, R. Baeza-Yates, B. Ribeiro, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Eighth Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 10–20. (IEEE CS Press), Nov. 2001.

[2] R. Baeza and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley., 1999.

[3] S.H. Chung, H.C. Kwon, K.R. Ryu, H.K. Jang, J.H. Kim, and C.A. Choi. Parallel information retrieval on a sci-based pc-now. In *Workshop on Personal Computers based Networks of Workstations (PC-NOW 2000)*. (Springer-Verlag), May 2000.

[4] M. Marín. Time Warp on BSP Computers. In *12th European Simulation Multiconference*, June 1998.

[5] M. Marín. Parallel text query processing using Composite Inverted Lists. In *Second International Conference on Hybrid Intelligent Systems (Web Computing Session)*. IO Press, Feb. 2003.

[6] A.A. MacFarlane, J.A. McCann, and S.E. Robertson. Parallel search using partitioned inverted files. In *7th International Symposium on String Processing and Information Retrieval*, pages 209–220. (IEEE CS Press), 2000.

[7] W.F. McColl. General purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, pages 337–391. Cambridge University Press, 1993.

[8] B.A. Ribeiro-Neto and R.A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Third ACM Conference on Digital Libraries*, pages 182–190. (ACM Press), 1998.

[9] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. Technical Report PRG-TR-15-96, Computing Laboratory, Oxford University, 1996. Also in *Journal of Scientific Programming*, V.6 N.3, 1997.

[10] K.R. Sujithan. Towards a scalable parallel object database — the bulk-synchronous parallel approach. Technical Report PRG-TR-17-96, Computing Laboratory, Oxford University, 1996.

[11] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Second International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1993.

[12] URL. BSP and Worldwide Standard, http://www.bsp-worldwide.org/.

[13] URL. BSP PUB Library at Paderborn University, http://www.uni-paderborn.de/bsp.

[14] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.