Distributed Query Processing Using Partitioned Inverted Files

Claudine Badue[1]
Berthier Ribeiro-Neto[3]

Ricardo Baeza-Yates[2]
Nivio Ziviani[4]

[1,3,4]Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, Brazil
{claudine,berthier,nivio}@dcc.ufmg.br

[2]Computer Science Department
University of Chile
Santiago, Chile
rbaeza@dcc.uchile.cl

Abstract

In this paper, we study query processing in a distributed text database. The novelty is a real distributed architecture implementation that offers concurrent query service. The distributed system adopts a network of workstations model and the client-server paradigm. The document collection is indexed with an inverted file. We adopt two distinct strategies of index partitioning in the distributed system, namely local index partitioning and global index partitioning. In both strategies, documents are ranked using the vector space model along with a document filtering technique for fast ranking. We evaluate and compare the impact of the two index partitioning strategies on query processing performance. Experimental results on retrieval efficiency show that, within our framework, the global index partitioning outperforms the local index partitioning.

1. Introduction

The number and size of text databases has grown at explosive rates. At the same time, there is a rapid increase in the number of users and consequently, in the number of queries submitted to information retrieval systems. As text collections grows larger, they become more expensive to be managed by an information retrieval system. Furthermore, as the number of queries increases, it becomes even more important to provide high query processing rates.

A cost-effective solution for this problem is distributed computing, which consists of the application of multiple computers connected by a network to solve a single problem. Some of the advantages of the network of workstations model are as follows [1]. First, network of workstations has become extraordinarily powerful and offer a better price-performance than parallel computers. Second, most networks of workstations have a huge amount of memory and very fast processors, both of which sit idle most of the time. Third, switched networks allow bandwidth to scale with the number of processors and low overhead communication protocols have made it possible to do very fast communication among workstations.

Two approaches have been proposed in the work presented in [14] to distribute the index of text collections among various computers. Nevertheless, none of these approaches have been implemented in related work on a real case framework aiming at providing to the users concurrent access to the documents in the collection.

In this paper, we study concurrent query processing in a distributed text database by a real case implementation. The distributed system uses a network of workstations model and the client-server paradigm. We adopt two distinct types of inverted file partitions for indexing the text database, namely local index partitioning and global index partitioning. In the local index partitioning, the documents in the text database are distributed among the processors, and each processor generates an inverted file for its documents. In the global index partitioning, an inverted file is generated for all the documents in the text database and the inverted lists are distributed among processors. In both index partitioning strategies, documents are ranked using the vector space model along with a document filtering technique proposed in [10] and adapted to the distributed processing in [5]. This distributed filtering technique preserves retrieval effectiveness with reduction in ranking costs, according to the results shown in [3].

We evaluate and compare the impact of the two index partitioning strategies on query processing performance. Experimental results on retrieval efficiency show that, within our framework, the global index partitioning outperforms the local index partitioning specially when the number of processors exceeds the average number of terms in query. The processing time with the global index partitioning might be twice smaller as that with the local index partitioning.

To the best of our knowledge, this is the first work that presents experimental results on the performance of a concurrent query processing system implemented on a real case distributed framework.

This paper is organized as follows. Section 2 covers the related work. Section 3 presents the distributed text database, describing the system architecture, the index structure, the vector space model as ranking strategy and the query processing. Section 4 explains the implementation aspects of the system. Section 5 shows the experimental results, and Section 6 presents the conclusions and future work.

2. Related work

The work in [14] proposes the two basic and distinct options for storing the inverted lists, that we already mentioned. With the local index organization, the documents are evenly partitioned into sets, one for each disk; in each partition, inverted lists are built for the documents that reside there. In the global index organization, the full lists are evenly spread across all the disks in the system. The architecture is that of a LAN, where the number of CPUs (each CPU has its own local memory), the number of I/O controllers per CPU, and the number of disks per controller are varied. The adopted query type is the "boolean and". The data used are synthetic documents and queries. Experiments are based on a simulation model.

Our work differs from that presented in [14] in the following aspects. First, while they adopted the boolean model, we adopt the vector space model. Second, while they model the documents and the queries, we base our experimental results on the documents and queries in the TREC-3 collection [6]. Third, we implement and thoroughly evaluate distributed query processing performance on a real case framework, while they derive experimental results from a simulation model, that hardly foresee all the factors that influence system performance. Fourth, while their simulator considers only a sequential query service, we address a concurrent query service, that provides a higher performance than the poor and unrealistic sequential query service. Fifth, while they conclude that within their framework the local index organization is a preferable choice, our results show that the global index organization is the best.

The work in [8] considers the two different index partitioning schemes proposed in [14] for a shared-everything multiprocessor machine with multiple disks. The query type used is the boolean, the database and query models are synthetic, and experimental results are derived by simulation. Our work differs from that presented in [8] in the following aspects. First, instead of adopting a shared-everything model, we use a shared-nothing architecture. Second, instead of adopting the boolean model, we use the vector space model as ranking strategy. Third, instead of modeling the database and the queries, we use the TREC-3 collection. Fourth, instead of deriving results from a simulation model, we implement and thoroughly evaluate distributed query processing performance on a real case framework. Fifth, instead of considering only a sequential query service, we address a concurrent query service.

The work in [11] adopts the two index organizations proposed in [14]. The architecture is that of a network of workstations, where each machine has its own local memory and disk (shared-nothing). They adopt the vector space model as ranking strategy, use the documents and queries in the TREC-3 collection [6], address a concurrent query service, and derive experimental results from an analytical model coupled with a simulator. Our work differs from that presented in [11] in the following way. Instead of studying query performance using a simulation model, we implement and thoroughly evaluate concurrent query processing performance on a real case framework. Our results show that the global index organization overcomes the local index organization, confirming their simulation model results.

The work in [9] investigates the two types of index partitions proposed in [14]. The search topology is a shared-nothing master/slave topology. Documents are searched using the probabilistic model. The data used in experiments are part of the documents and queries in the TREC-7 collection [7]. Experiments are based on a real case implementation. Nevertheless, they address only a sequential query service. Our work differs from that presented in [9] in the following aspects. First, we adopt the vector space model for ranking documents, while they adopt the probabilistic model. Second, we implement a concurrent query service, while they address only a sequential query service. Third, our results show that the global index partitioning is the best, while they conclude that within their framework the local index partitioning is a superior choice.

3. Distributed text database

In this section, we present the distributed text database. First, we characterize the system architecture. Second, we discuss the index structure and the two strategies to partition it across the network. Third, we explain the technique for ranking documents using the vector space model along with the document filtering technique. Finally, we detail query processing in the distributed text database, presenting the differences between the two index partitioning.

3.1. System architecture

The distributed system uses a network of workstations model. The workstations are tightly coupled by fast network switching technology. Each workstation has its own local memory and local disk. The advantages of this shared nothing model are that all communication between processors is done through messages, which eliminates interference from operating system memory control processes, and that disks are directly accessed by processors without going through the network. Figure 1 illustrates the network of workstations model.

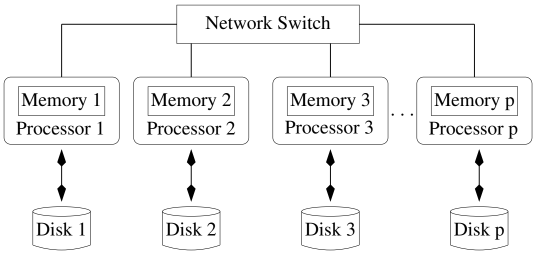

Figure 1. Network of workstations model.

The retrieval system adopts the client-server paradigm that consists of a set of server processes and a designated broker process, responsible for accepting client queries, distributing the queries to the servers, collecting intermediate results from the servers, combining the intermediate results into the final result and sending the final result to the client. Each of the server processes and the broker process runs on a separate processor. Figure 2 illustrates the client-server paradigm.

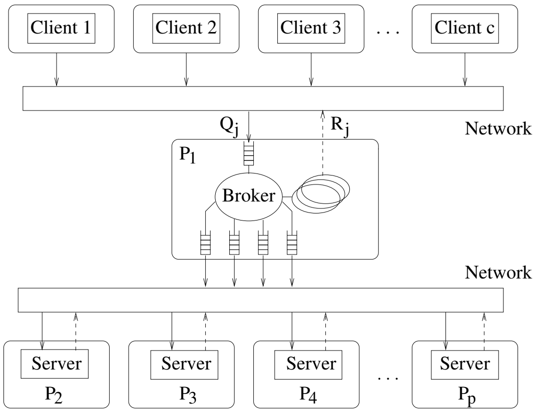

Figure 2. Client-server paradigm.

3.2. Index structure

The text database is indexed using the inverted file technique [4]. The main advantages of the inverted file are the relatively low cost for building and maintaining it, a searching strategy based mostly on the vocabulary which usually fits in main memory, and a good retrieval performance.

An inverted file is an indexing structure composed of two elements: the *vocabulary* and a set of *inverted lists*. The vocabulary contains each term t in the text document collection; the terms are sorted in lexicographical order. There is one inverted list for each term t, consisting of the identifiers of the documents containing the term and, with each identifier d, the frequency $f_{d,t}$ of t in d. Thus, inverted lists consist of term entries, that is, pairs of $< d, f_{d,t} >$ values.

3.3. Index partitioning

We consider two strategies to partition the inverted file across the network: local index and global index. Next, we describe both strategies.

3.3.1. Local index

One possible alternative to partition the text database index is to have a local inverted file for each subcollection. In this case, documents are evenly distributed among processors and each processor generates an inverted file for its documents. The size sc (in bytes) of the local subcollections is approximately given by Equation(1):

$$sc = \frac{N}{p} \tag{1}$$

where N is the size (in bytes) of the whole text database collection and p is the number of processors. In other words, each processor holds in its local disk a subcollection whose size is approximately given by sc. The value of sc is approximated, because we cannot split a document in the text database collection. Figure 3 illustrates the local index partitioning, considering a network composed by 4 processors.

In the local index partitioning, information on the global occurrence of terms in the text database is not available. The absence of this information slacks the estimates for the inverse document frequency (*idf*) weights used by the vector space model to rank documents in the text database collection. A solution to this problem is to compute the idf for all index terms and distribute this information to all processors.

3.3.2. Global index

The other alternative to partition the index is to have a global inverted file for the whole text database. In this case, an inverted file is generated for documents in the text

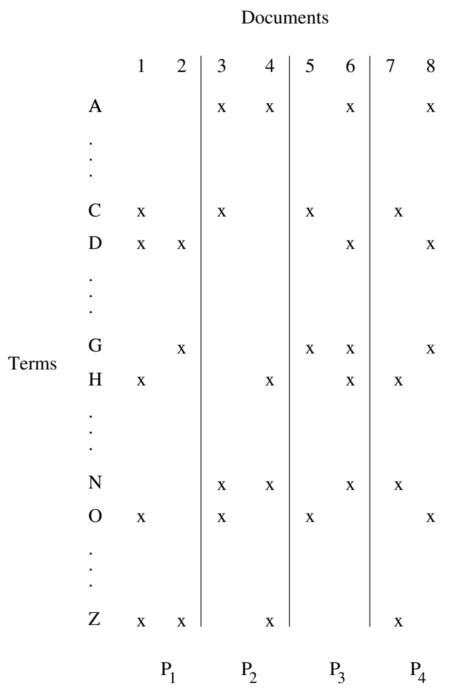

Figure 3. Local index partitioning.

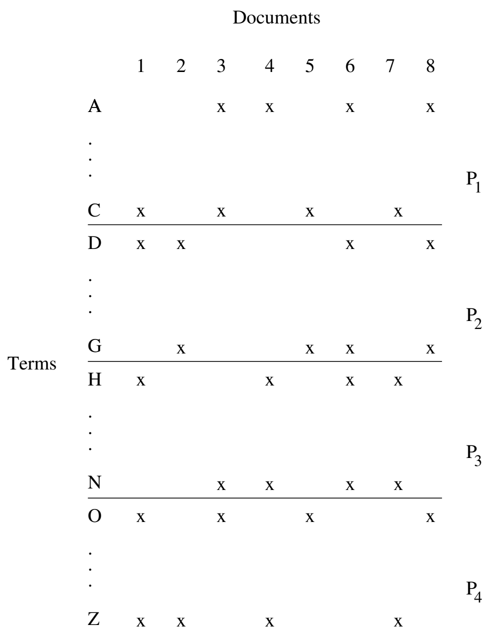

Figure 4. Global index partitioning.

database and the inverted lists are evenly distributed among processors. The size sl (in bytes) of the local subset of inverted lists is approximately given by Equation(2):

$$sl = \frac{L}{p} \tag{2}$$

where L is the size of the set of inverted lists in the text database and p is the number of processors. In other words, each processor holds in its local disk a subset whose size is approximately given by sl. The value of sl is approximated, because we cannot split an inverted list of a term in the text database collection.

We consider that inverted lists are distributed among processors in lexicographical order. According to this strategy, one possible partitioning for the global index might be one in which processor 1 holds the inverted lists for all the terms that start with the letters A, B and C; processor 2 holds the inverted lists for all the terms that start with the letters D, E, F and G; and so on, such that each processor holds a portion of the global index whose size is approximately sl. Figure 4 illustrates the global index partitioning, considering a network composed by 4 processors.

3.4. Ranking with the vector model

The documents in the text database collection are ranked using the vector space model. The main advantages of the vector space model are its term-weighting scheme that improves retrieval performance, its partial matching strategy which allows retrieval of documents that approximate the query conditions, and its cosine ranking formula that sorts the documents according to their degree of similarity to the query. A large variety of alternative ranking methods have been compared to the vector space model but the consensus seems to be that, in general, the vector space model is either superior or almost as good as other alternatives. Furthermore, it is simple and fast. For these reasons, the vector space model is a popular retrieval model nowadays [4].

In the vector space model, documents and user queries are represented as vectors of the weight of terms. A document vector is defined as $\vec{d} = (w_{d,1}, w_{d,2}, \ldots, w_{d,v})$, where v is the total number of index terms in the collection and $w_{d,i}$ is the weight of term i for the document d. A query is seen as a small document. We assign the weight to a term in a document or a query using the *tf-idf* scheme [12]. The vector space model proposes to evaluate the degree of similarity of the document d with regard to the query q as the correlation between the vectors $\vec{d}$ and $\vec{q}$, that can be quantified by the cosine of the angle between these two vectors. The standard algorithm for ranking documents using the vector space model uses a set of accumulators, one accumulator for each document in a collection, and a set of inverted lists. For each query term t, the contribution made by the term t to the degree of similarity between the query q and each document d in the inverted list is added to the document d's accumulator's value. The final result is composed by the documents with the highest accumulator values.

For a large document database, the ranking evaluation cost - volume of main memory, disk traffic and CPU processing time - can be prohibitively high, because it assigns a similarity value to every document containing any of the query terms. The work in [10] proposes a technique for filtering documents during ranking which allows a significant reduction of ranking evaluation costs without degradation in retrieval effectiveness. The filtering method considers as candidate answers only the documents with high within-document frequency. The memory usage is reduced because having fewer candidates means that fewer accumulators are required to store information about these candidates. Disk traffic and CPU processing time are also reduced because, by ordering inverted lists by decreasing within-document frequency, only the first portion of each list containing high frequencies will be processed, and the rest can be ignored.

Unfortunately, this filtering technique, as it states originally, does not work very well for distributed processing. The reason is that its efficiency is influenced by thresholds that are determined as a function of the accumulated partial similarity of the currently most relevant document S_{max}, whose growth in the distributed algorithm during ranking evaluation differs from that in the sequential algorithm.

In the local index partitioning, if one of the processors holds only a few high weighted documents, the rising of S_{max} is low; in the global index partitioning, when the processors receive only a few terms, the value of S_{max} is a fraction of that in the sequential algorithm. Consequently, the amount of pruned resources in the distributed algorithm is smaller than in the sequential algorithm, which might deteriorate the performance of the former, making it even worse than the latter.

The work in [5] proposes a solution to this problem that previews the rising of the S_{max} value before query processing. By adopting this adaptation of the filtering technique to our system, we obtain approximately the same effectiveness as the standard algorithm of the vector space model, for both the local and global index partitioning strategies, upon significant reductions in ranking evaluation cost - queries are processed in only 2% of the memory of the standard algorithm and only 10% of all term entries in the inverted lists are required. More details on how we implemented the distributed filtering technique in our system and on the retrieval effectiveness results we obtained may be found in the work presented in [3].

3.5. Distributed query processing

Our concurrent distributed query system consists of a set of server processes and a designated broker process, each running on a separate processor, as presented in Section 3.1. The broker process is responsible for scheduling the queries to the server processes, receiving the intermediate results returned by each one of the server processes and combining the intermediate results into the final result.

We do not study how the performance is affected by the query arrival rate. Instead, we assume that the arrival rate of queries in the system is high enough to fill a query processing queue. Hence, we do not compute the actual user response time for a query, but the system time. Next, we describe the query processing algorithms implemented in the broker, which differ according to the index partitioning strategy.

3.5.1. Local index

In the local index partitioning, an individual query is processed as follows. The broker process sends the query to all server processes. Each server retrieves the documents related to that query in the local subcollection and ranks them, using the vector space model along with the document filtering technique; selects a number of documents from the top of the ranking; and returns them to the broker as the local answer set. The broker uses a multiway merge [15] to fuse the local answer sets and produce the final ranked answer set.

Regarding the selection of a number of documents to be returned to the broker, consider that answer precision is evaluated through the first r documents in the top of the ranking. In the worst case, the broker will select the first r documents from only one of the local answer sets. This implies that each server needs to send to the broker at most the top r documents of its ranking, in order of guaranteeing that the final answer precision is not diminished.

3.5.2. Global index

In the global index partitioning, an individual query is processed as follows. The broker process determines which server processes hold inverted lists relative to the query terms, breaks the query into subqueries and sends them to the respective servers. Each subquery is composed by the terms which are stored in the server it is sent to. Once a server has received a subquery, it retrieves the documents related to its subquery and ranks them, using the vector space model along with the document filtering technique; selects a number of documents from the top of the ranking; and returns them to the broker as the local answer set. The broker adds the weights of the documents which are present in more than one local answer set and do a sort to produce the final ranked answer set.

Regarding the merging of the local answer sets, the broker cannot use the local rankings generated by individual servers because such rankings are based in partial information present in the subqueries. In other words, the local answer sets returned by the servers contain partial similarities between each document and each term present in the subquery; it is necessary to sum the partial similarities into the global similarity, which expresses the measure of relevance between each document and the query.

The fact that the local rankings are based in partial information complicates the cutting strategy, that consists of the selection of a number of documents to be sent to the broker. The work in [11] suggests a cutoff factor that depends on the number p of servers. The cutoff factor is given by $c \times p \times r$, where c is a constant and r is the number of documents in the final answer set. Using such factor for $c = 6$, we observed no significant variation in the final answer precision, as shown in the results of the work presented in [3].

3.5.3. Comparison of strategies

The local index partitioning and global index partitioning are compared in the following aspects, as presented in Table 1.

LI	GI
High parallelism	High concurrency
More disk seeks	Less disk seeks
Better load balance	Worse load balance
Smaller inverted lists	Larger inverted lists
Smaller local answer sets	Larger local answer sets
Top r documents are sent to the broker	Top $(c \cdot p \cdot r)$ documents are sent to the broker

Table 1. Comparison between the local and global index partitioning strategies.

In the local index partitioning, all processors are devoted to the execution of a single query. Therefore, the local index partitioning always provides high parallelism. On the other hand, in the global index partitioning, not all processors might be involved with the processing of a single query. A scenario that confirms this statement is when the number of processors is larger than the number of query terms. Another scenario is when many query terms are stored in a single processor releasing the others. Therefore, the global index partitioning might allow high concurrency.

In the local index partitioning, retrievals require more disk seeking operations, because the processors receive all query terms. On the other hand, in the global index partitioning retrievals require less disk seeking operations, because the processors do not necessarily receive all query terms.

In the local index partitioning, the load balance level is better than in the global index partitioning. The reason is that in the global index partitioning, the terms in a query are sent only to the processors which store their inverted lists. This implies that the processor that holds the most frequent terms in query is heavily loaded, while the processor that holds the least frequent query terms stays relatively idle. On the other hand, in the local index partitioning, all terms of a query are sent to all processors. Consequently, a good load balance level is always provided.

In the local index partitioning, inverted lists are smaller, because they contain only the documents from the subcollection assigned to the processor. On the other hand, in the global index partitioning inverted lists are larger, because they contain documents from the whole text database collection.

In the local index partitioning, local answer sets are smaller than in the global index partitioning. The reason is that in the global index partitioning each processor does not have the information on the documents inserted in the set of accumulators of the others. This implies that the selection of documents as answer candidates must be less severe, in order of avoiding the elimination of a relevant document that, however, has a low value in the accumulator of a determinate processor.

In the local index partitioning, the local rankings consider the global information related to the query, which allows the number of documents to be sent to the broker being equal to the number of documents in the final answer. In the global index partitioning, the local rankings consider only partial information related to the subquery, which implies that the number of documents to be sent to the broker must be larger than the number of documents in the final answer.

In the sequel, we investigate how these differences, which are determinant in query processing performance, can favor one of the index partitioning strategies in detriment of the other.

4. Implementation aspects

In this section, we describe some details of the system implementation with focus on the issues regarding the concurrent query service.

We adopt the client/server paradigm. In this scheme, client processes request services from a server process. A server process normally listens at a known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time, the server process "wakes up" and services the client, performing whatever appropriate requested actions.

According to these properties, the server process is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked. In our model, the service is the processing of a query. Clients request service to a central server, called broker. In its turn, the broker requests service to the other servers in the distributed architecture. When the broker requests service to a server, it plays the role of a client.

The broker process is constituted by an insertion thread, a merging thread and different scheduling threads for each server process in the network. All these threads run in parallel in the broker process. The main data structures shared by the threads are the scheduling queues, one for each server process, and the buffer of intermediate results. The scheduling queues contain queries, if the index partitioning is the local one, or subqueries, if the index partitioning is the global one. The buffer of intermediate results temporarily contains local answer sets waiting for being merged into final answer sets.

The insertion thread is responsible for inserting a query (or subquery) in the scheduling queues of the servers that must execute that query (or subquery). Each of the scheduling threads is responsible for taking a query (or subquery) out of its queue, sending the query (or subquery) to its server, receiving the local answer set, and storing the local answer set into the buffer of intermediate results. The merging thread is responsible for fusing local answer sets into final answer sets, as soon as all the local answer sets related to a query (or subquery) are available in the buffer of intermediate results.

In this way, these different threads run in parallel and in asynchronous mode to dispatch the several queries to the different servers, receive the intermediate results, and merge the intermediate results into the final results. This scheduling scheme increases the system throughput by allowing the simultaneous processing of more than one query and by avoiding to the utmost the idleness of processors in the network.

The interprocess communication between broker and server processes is socket-based. The data transmission mechanism is stream-based, which provides sequenced, reliable, two-way and connection-based byte streams. The synchronization of the access to shared memory segments is done with semaphores. The algorithms are implemented with the C programming language and compiled by the GCC 2.91.66 compiler. We use the C programming language because of its efficiency and its easy integration with operating systems in general.

5. Experimental results

In this section, we present the experimental results on the real case implementation. We compare the performance impact on query processing of both the local and the global index partitioning strategies.

5.1. Experimental setup

The network of workstations we used in the experiments is composed by 5 PCs with the same configuration. Each PC is an AMD-K6-2 with a 500MHz processor, 256Mbyte of main memory, 30Gbyte IDE hard disk, and running Linux kernel 2.2.14. The workstations are connected by a 100Mbps fast Ethernet with a 16 port switch.

The data we have used in the experiments comprise the disks 1 and 2 of the TREC-3 collection [6]. Each of the disks is about 1 gigabyte in size. We used two sets of queries, namely a TREC query set and an artificial query set, that mimics Web-like queries. The TREC query set is based on topics 151 to 200 of the ad-hoc task, totalizing 50 queries in all. The terms were automatically extracted from the topic descriptions, after eliminating SGML tags and stop words. The average number of terms per query is 21. In the artificial query set, composed by 2000 queries, the terms were randomly chosen from the collection vocabulary, but avoiding stop words [2]. The number of terms per query is 2 or 3.

5.2. Retrieval efficiency

In this section, we compare retrieval efficiency between the global and local index partitioning. We discuss the results for the 50 TREC queries, which are longer and force parallelism, and the results for the 2000 artificial queries, which are shorter and allow concurrency in our system. The metrics used are: (i) processing time, given by the elapsed time in seconds to process a batch of queries using p processors; (ii) speedup, given by the ratio between the processing time for one processor and the processing time with p processors; and (iii) load imbalance, given by the ratio between the maximum processing time and the average processing time of the processors. It follows the results and corresponding interpretations.

5.2.1. TREC queries

Figure 5 shows the time to process the 50 TREC queries as a function of the number of processors in the network, for the local and global index partitioning. As it can be seen, the local index partitioning outperformed the global index partitioning with a network composed by 2 processors, but the global index partitioning outperformed the local index partitioning with a network composed by 3 and 4 processors. The interpretation for this result is as follows.

In the global index partitioning, with a network composed by 3 and 4 processors, the number of seeks performed locally dropped to the point of counterbalancing the ranking and communication costs, which are higher than in the local index partitioning. However, with a network composed

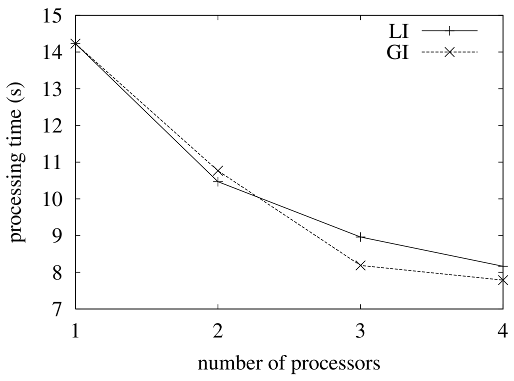

Figure 5. Processing time for the 50 TREC queries.

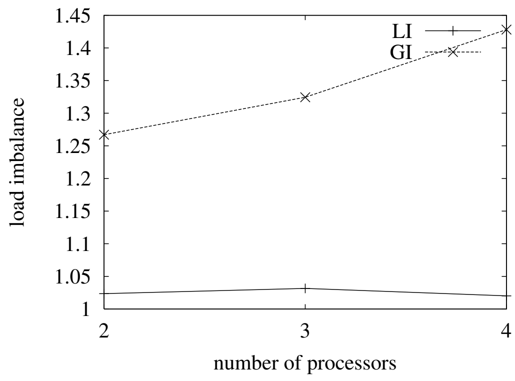

Figure 7. Load imbalance for the 50 TREC queries.

by only 2 processors, the number of seeks performed locally did not reduce enough for offsetting those prejudicial effects.

Figure 6 shows the speedup while processing the 50 TREC queries. We observe that speedup in the global index partitioning is not that much superior than in the local index partitioning, as a result of the parallelism constrained by the length of TREC queries.

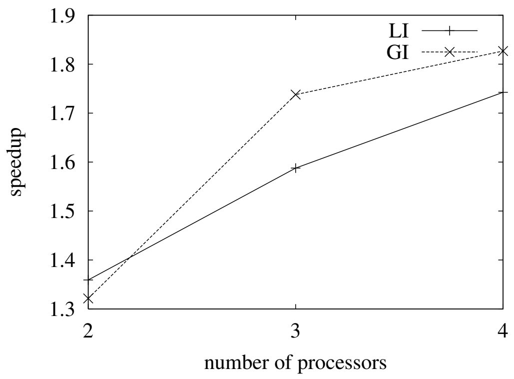

Figure 6. Speedup for the 50 TREC queries.

Figure 7 shows the load imbalance while processing the 50 TREC queries. In the local index partitioning, load imbalance is not an issue as for any network configuration it was found to be just over 1. However, it is perceptibly worse in the global index partitioning. The interpretation for these results is as follows.

In the global index partitioning, the query terms are routed to the processors which hold the respective inverted lists. So, if some terms are more frequently requested in a query, the processor that stores those terms is heavily loaded; on the contrary, the processor that stores the least frequent query terms stays relatively idle. Otherwise, in the local index partitioning, all query terms are sent to all processors. This implies that all processors are involved with the execution of all queries. Consequently, a good level of load balance is always provided. A modest load imbalance might occur if a processor holds documents that are more relevant to the query than other processors. In this scenario, the cost for reading inverted lists, accumulating document weights and ranking will be higher in the processors which hold the most relevant documents.

It is important to note that if the load balance were uniform in our system, the global index partitioning would have a better performance than the local index partitioning, no matter the number of processors in the network, as it can be seen in Figure 8 and Figure 9 that show the processing time and speedup respectively. Also, the relative performance improvement would increase with the number of processors, as shown in Table 2. For simulating the load balanced scenario, we simply averaged by processor the time taken by the broker to collect the local answer sets, instead of considering the maximum time associated with the slowest processor.

Number of processors	GI as percentage of LI (%)
2	80.94
3	74.39
4	73.36

Table 2. Processing time in the load balanced scenario for the 50 TREC queries: GI as percentage of LI.

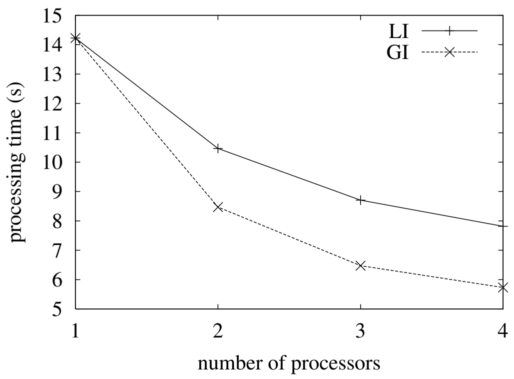

Figure 8. Processing time in the load balanced scenario for the 50 TREC queries.

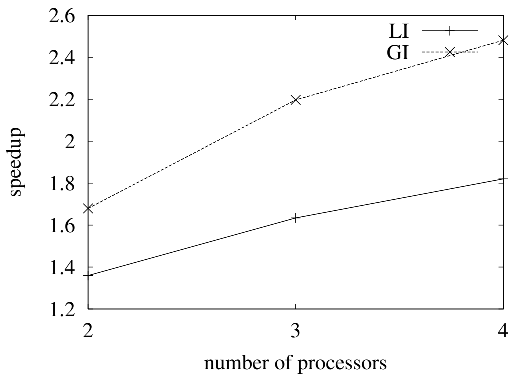

Figure 9. Speedup in the load balanced scenario for the 50 TREC queries.

5.2.2. Artificial queries

Figure 10 shows the time to process the 2000 artificial queries as a function of the number of processors in the network, for the local and global index partitioning. As we can observe, the global index partitioning consistently outperformed the local index partitioning. In addition, the relative performance improvement increases with the number of processors, as shown in Table 3. As it can be seen, the global index partitioning might be twice as faster than the local index partitioning. The reason is as follows.

In the local index partitioning, all the processors are forced to process the 2 terms (on average) of each query. Otherwise, in the global index partitioning, 2 processors at most are involved with the execution of a single query, as a result of one of the following events (or a combination of them): i) the query terms are held by a single processor, releasing the others to execute another query; or ii) the number of processors are larger than the number of query terms.

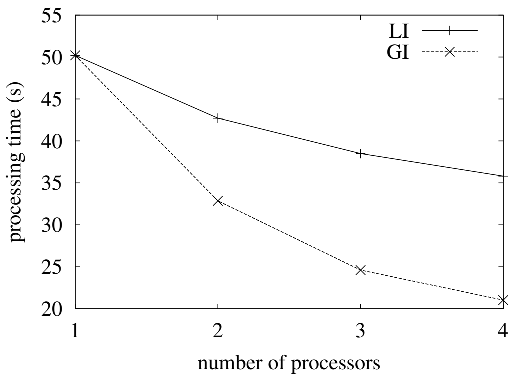

Figure 10. Processing time for the 2000 artificial queries.

Number of processors	GI as percentage of LI (%)
2	76.93
3	63.94
4	58.75

Table 3. Processing time for the 2000 artificial queries: GI as percentage of LI.

Figure 11 shows the speedup while processing the 2000 artificial queries. As it can be seen, the global index partitioning presented a much superior speedup than the local index partitioning, as a result of the higher concurrent query service provided by the first index organization.

Figure 12 shows the load imbalance while processing the 2000 artificial queries. For the local index partitioning, load imbalance is also found to be just over 1, like we discussed for the TREC query set. In the global index partitioning, load imbalance was not that much superior than in the local index partitioning. This result is due to the method used to generate the artificial queries, by which terms were randomly chosen from the collection vocabulary. In this way, the probability distribution of terms in the artificial queries tends to be uniform, which provides a better load balance.

6. Conclusions and future work

In this paper, we study concurrent query processing in a distributed text database. We have implemented a real distributed architecture and compared the impact of two different types of inverted file partitions on system performance. Documents are ranked using the vector space model along with a document filtering technique for fast ranking.

Experimental results on retrieval efficiency show that,

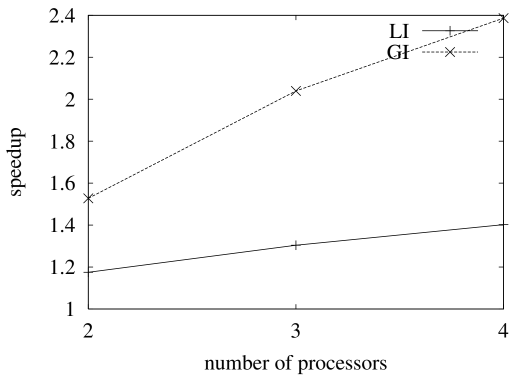

Figure 11. Speedup for the 2000 artificial queries.

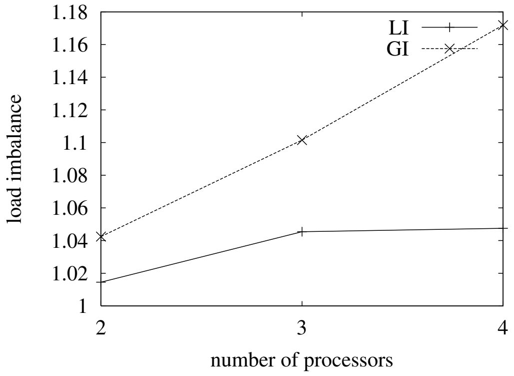

Figure 12. Load imbalance for the 2000 artificial queries.

within our framework, the global index partitioning outperforms the local index partitioning specially when the number of processors exceeds the average number of terms in query. The processing time with the global index partitioning might be twice smaller as that with the local index partitioning. The main reason is that the global index partitioning allows the parallelization of the most time consuming phase of the algorithm - disk seeking. Further, the global index partitioning provides a high concurrent query service, which is particularly evidenced when the number of processors exceeds the average number of terms in query.

In future work, we are interested in adding more processors to the network. Also, we intend to implement two types of brokers, one for query scheduling and another for merging of intermediate results; the merging broker can dynamically distribute its task with other processors when the workload is high. Further, we are interested in making use of multiprogramming in the server and evaluate the system performance while varying the multiprogramming level (number of simultaneous queries per server).

Other future direction of research is to evaluate the behavior of our system while processing Web data. A typical Web workload comprises very large collections and very short queries, and we are interested in specifying a query arrival distribution.

Another direction for future research is to study new strategies to generate the global index by exploiting usage statistics and other measures, in order of achieving better speedup, load balance and retrieval effectiveness. Also, for decreasing the index accessing time, we intend to investigate the global index structured in two levels; the first level is an index for the most frequent queries stored in main memory, and the second an index for the remaining of the queries stored in secondary memory.

Finally, we intend to study how the caching of query results and inverted lists proposed in [13] can improve the performance of our system or favor one of the index partitioning strategies.

References

[1] T. E. Anderson, D. E. Culler, D. A. Patterson, and the Now Team. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] M. D. Araujo, G. Navarro, and N. Ziviani. Large text searching allowing erros. In R. Baeza-Yates, editor, *Proceedings of the Fourth South American Workshop on String Processing*, pages 2–20, Valparaiso, Chile, November 1997. Carleton University Press.

[3] C. S. Badue. Distributed query processing using partitioned inverted files. Master's thesis, Federal University of Minas Gerais, Belo Horizonte, Minas Gerais, Brazil, March 2001.

[4] R. Baeza-Yates and B. Ribeiro-Neto, editors. *Modern Information Retrieval*. ACM Press New York, Addison Wesley, 1999.

[5] R. A. Barbosa. Desempenho de consultas em bibliotecas digitais fortemente acopladas. Master's thesis, Federal University of Minas Gerais, Belo Horizonte, Minas Gerais, Brazil, May 1998. In Portuguese.

[6] D. Harman. Overview of the third text retrieval conference. In D. Harman, editor, *Proceedings of the Third Text REtrieval Conference (TREC-3)*, pages 1–19, Gaithersburg, Maryland, U.S.A., 1994. NIST Special Publication 500-207.

[7] D. Hawking, N. Craswell, and P. Thistlewaite. Overview of TREC-7 very large collection track. In E. Voorhess and D.K.Harman, editors, *Proceedings of the Seventh Text Retrieval Conference*, pages 257–268, Gaithersburg, Maryland, U.S.A., November 1998. NIST Special Publication 500-242.

[8] B.-S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, February 1995.

[9] A. MacFarlane, J. McCann, and S. Robertson. Parallel search using partitioned inverted files. In *Proceedings of*

the 7th International Symposium on String Processing and Information Retrieval, pages 209–220, La Coruna, Spain, September 2000. IEEE Computer Society.

[10] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[11] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM Conference on Digital Libraries*, pages 182–190, 1998.

[12] G. Salton and C. Buckley. Term-weighting approaches in automatic retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[13] P. C. Saraiva, E. S. Moura, N. Ziviani, R. Fonseca, W. Meira, C. Murta, and B. Ribeiro-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th ACM SIGIR Conference*, New Orleans, Louisiana, U.S.A., September 2001 (to appear).

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

[15] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes - Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Inc., 2ª edition, 1999.