Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Processamento Distribuído de Consultas Usando Arquivos Invertidos Particionados

Claudine Santos Badue

Belo Horizonte
27 de Março de 2001

Claudine Santos Badue

Processamento Distribuído de Consultas Usando Arquivos Invertidos Particionados

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte
27 de Março de 2001

Resumo

O objetivo deste trabalho é estudar o processamento de consultas em um banco de dados textual distribuído. A principal contribuição é a implementação de uma arquitetura distribuída real que oferece um serviço concorrente de consultas. O sistema distribuído adota o modelo de rede de estações de trabalho e o paradigma cliente-servidor. A coleção de documentos é indexada por arquivos invertidos. Adotamos duas estratégias distintas de partição do índice no sistema distribuído, denominadas partição de índice local e partição de índice global. Na partição de índice local, os documentos da coleção do banco de dados textual são distribuídos entre os processadores, e cada processador gera um arquivo invertido para os seus documentos. Na partição de índice global, um arquivo invertido é gerado para todos os documentos da coleção do banco de dados textual e as listas invertidas são distribuídas entre os processadores. Em ambas as estratégias, os documentos são recuperados e ordenados através do modelo vetorial juntamente com uma técnica de filtragem de documentos, que permite uma redução significativa nos custos de ordenação sem degradar a eficácia da recuperação. Avaliamos e comparamos o impacto das duas estratégias de partição do índice no desempenho do processamento de consultas. Em relação a eficácia da recuperação, mostramos que obtemos aproximadamente a mesma eficácia do algoritmo seqüencial, para ambas as estratégias de partição de índice local e partição de índice global. Em relação a eficiência da recuperação, os resultados experimentais sobre o desempenho geral do processamento de consultas mostram que, dentro do nosso arcabouço, a partição de índice global supera a partição de índice local.

Claudine Santos Badue

Distributed Query Processing Using Partitioned Inverted Files

Thesis presented to the Graduate Course in Computer Science of the Federal University of Minas Gerais, as partial requirement to obtain the degree of Master of Science in Computer Science.

Belo Horizonte
March 27, 2001

Abstract

The objective of this work is to 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 by inverted files. We adopt two distinct strategies of index partitioning in the distributed system, 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 strategies, documents are retrieved and ranked using the vector space model along with a document filtering technique, that allows significant reduction in ranking costs without degradation in retrieval effectiveness. We evaluate and compare the impact of the two index partitioning strategies on query processing performance. Regarding retrieval effectiveness, we show that we obtain approximately the same effectiveness as the sequential algorithm, for both the local index partitioning and the global index partitioning. Regarding retrieval efficiency, experimental results on the overall query processing performance show that, within our framework, the global index partitioning outperforms the local index partitioning.

To Jesus Christ

"He [God] has rescued us from the dominion of darkness and brought us into the kingdom of the Son [Jesus Christ] he loves, in whom we have redemption, the forgiveness of sins. He [Jesus Christ] is the image of the invisible God, the firstborn over all creation. For by him all things were created: things in heaven and on earth, visible and invisible, whether thrones or powers or rulers or authorities; all things were created by him and for him. He is before all things, and in him all things hold together. And he is the head of the body, the church; he is the beginning and the firstborn from among the dead, so that in everything he might have the supremacy. For God was pleased to have all his fullness dwell in him, and through him to reconcile to himself all things, whether things on earth or things in heaven, by making peace through his blood, shed on the cross."

from the Bible (New International Version) in the book of Colossians 1:13-20.

Acknowledgments

To my Lord and Savior Jesus Christ for His great love towards me, that was shown in full extent when He laid down His life for me on the cross and which endures forever. My thanks goes with a grateful heart to Jesus for demonstrating His everlasting love to me during this Master of Science Course, while guiding me step by step, while renewing my strength day by day and while comforting me in times of trouble. To Him I give all thanks and honor and glory and praise and worship now and forever!

To my parents Celuta and Anuor and to my brothers Cassius and Christian for their very deep love and enthusiastic support in all the phases of this Master of Science Course. Their strong presence and prayers, despite the physical distance, have been healing for my soul and brought peace to my heart.

To Prof. Dr. Nivio Ziviani for his excellent advice and strong support over the entire Master of Science Course. His knowledge on Information Retrieval and experience in the research arena have been determinant in the accomplishment of this thesis. Also, his steadfast incentive helped me to persevere in this work until the very end.

To Prof. Dr. Berthier Ribeiro-Neto for presenting to me the idea of this thesis and for some discussions that helped outline directions on the development of this work.

To Prof. Dr. Ricardo Baeza-Yates for the valuable suggestions for future works.

To my friend Dr. Edleno Moura for his presence in technical meetings and the many informal discussions which enlightened some obscure issues in this thesis.

To my friend Rodrigo Barra for helping me to implement some of the algorithms, and to my friend Ramurti Barbosa for sharing with me some of his experience in performance analysis of distributed query processing systems.

To Dr. Wagner Meira Júnior for some valuable discussions in the area of parallelism, and to Prof. Dr. Antônio Loureiro for helping me to understand some concepts in the area of networking.

To my friend Daniela Seabra for exchanging valuable information with me in the courses which we participated in together and for her special friendship during the entire Master of Science Course.

To my friend Maria de Lourdes for patiently revising the entire text of this thesis.

To all my colleagues in the Laboratory for Treating Information - Charles, Cláudio, Cristiane, Maria de Lourdes, Pável, Tânia - for their friendship and company in the many exhausting hours of work.

To my roommates Fabiana and Marciana for participating with me in daily life and for sharing with me moments of happiness as well as moments of sadness.

To Karina, Adeílde and Maria Ferrari for being my closest friends in Belo Horizonte and for being able to count on them in the best and worst situations since I have moved to Belo Horizonte.

To Dave, Edie and Allison Lowe for receiving me into their home and for demonstrating a very special affection towards me during the days I lived in Pittsburgh. Those days were part of God's training for enabling me to accomplish this Master of Science Course.

To the Rev. Dr. Larry and Ida Selig, my "American parents", for keeping me in prayer and for helping me to stand firm in my faith by their powerful words of encouragement.

To all my brothers and sisters in faith from the Presbyterian Church of Anápolis for praying for me since the beginning of this Master of Science Course and for participating with me in the peaceful and restoring holidays in Anápolis.

To all my brothers and sisters in faith from the Communion Group of the Eighth Presbyterian Church of Belo Horizonte for the blessed and refreshing meetings on Saturday evenings. My thanks goes especially to Andréia and Marcus Vinícius, the leaders of the group, for offering their help whenever I have needed it and for entrusting me with the leadership of worship time in the meetings, which has drawn me nearer to God's presence where I find rest, healing and great joy.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 3

1.3 Contributions of the Thesis 5

1.4 Structure of the Thesis 8

2 Distributed Text Database **9**

2.1 Distributed System Framework 9

2.1.1 Distributed System Architecture 9

2.1.2 Index Structure 10

2.1.3 Index Partitioning 11

2.2 Distributed Query Processing 13

2.2.1 Local Index 14

2.2.2 Global Index 15

2.2.3 Comparison between the Local Index Partitioning and Global Index Partitioning Strategies 17

3 Ranking with the Vector Space Model **19**

3.1 Vector Space Model 19

3.2 Filtering Technique 21

3.3 Distributed Filtering Technique 24

4 Implementation Aspects **27**
4.1 Client/Server Model 27
4.1.1 Broker Process (*B-Process*) 28
4.1.2 Server Process (*S-Process*) 32

5 Experimental Results **35**
5.1 Experimental Setup 35
5.2 Metrics 36
5.3 Retrieval Effectiveness 38
5.3.1 Filtering Technique 38
5.3.2 Distributed Filtering Technique 40
5.4 Retrieval Efficiency 42
5.4.1 Cost Analysis 43
5.4.2 Overall Query Processing Performance 51

6 Conclusions and Future Work **57**
6.1 Conclusions 57
6.2 Future Work 58
6.2.1 Two Types of Brokers 58
6.2.2 Multiprogramming in the Server 58
6.2.3 Minimization of Network Traffic 58
6.2.4 Performance Evaluation with Web Data 59
6.2.5 New Strategies for the Global Index Partitioning 59
6.2.6 Global Index in Two Levels 61
6.2.7 Caching of Query Results and Inverted Lists 62

Bibliography **63**

List of Figures

2.1 Network of workstations model. 10
2.2 Client-server paradigm. 11
2.3 Local index partitioning. 12
2.4 Global index partitioning. 14

3.1 Basic algorithm for ranking using the vector space model. 21
3.2 Filtering algorithm for fast ranking using the vector space model. 23
3.3 Increasing of S_{max} in the sequential and distributed algorithm using local index partitioning with 4 processors to execute the TREC-3 query 193. . . 24
3.4 Increasing of S_{max} in the sequential and distributed algorithm using global index partitioning with 4 processors to execute the TREC-3 query 193. . . 25
3.5 Distributed filtering algorithm for fast ranking using the vector space model. 26

4.1 Insertion thread algorithm of the broker process in the local index partitioning. 29
4.2 Insertion thread algorithm of the broker process in the global index partitioning. 29
4.3 Scheduling thread algorithm of the broker process. 30
4.4 Merging thread algorithm of the broker process. 31
4.5 Server process algorithm. 33

5.1 Retrieval effectiveness for increasing values of c_{ins} ($c_{add} = 0$). 39
5.2 Retrieval effectiveness for increasing values of c_{add} ($c_{ins} = 1 \times 10^{-2}$). 40
5.3 Percentage of contribution of the costs averaged by processor to execute the 50 TREC queries in LI. 44
5.4 Percentage of contribution of the costs averaged by processor to execute the 50 TREC queries in GI. 45
5.5 Cost averaged by processor to execute the 50 TREC queries. 48
5.6 Cost of merging at the broker for the 50 TREC queries. 50

5.7 Processing time for the 50 TREC queries. 52
5.8 Speedup for the 50 TREC queries. 52
5.9 Load imbalance for the 50 TREC queries. 53
5.10 Processing time in the load balanced scenario for the 50 TREC queries. . . 54
5.11 Speedup in the load balanced scenario for the 50 TREC queries. 55
5.12 Processing time for the 2000 artificial queries. 55
5.13 Speedup for the 2000 artificial queries. 56
5.14 Load imbalance for the 2000 artificial queries. 56

6.1 Global index partitioning. 60
6.2 Global index in two levels. 62

List of Tables

2.1 Comparison between the local and global index partitioning. 17

5.1 TREC query set and artificial query set. 35
5.2 Percentage of term entries processed and retrieval effectiveness for increasing values of c_{add} ($c_{ins} = 1 \times 10^{-2}$) using the filtering algorithm. 39
5.3 Percentage of term entries processed and retrieval effectiveness for increasing values of c_{add} ($c_{ins} = 6 \times 10^{-3}$) using the distributed filtering algorithm. . . 40
5.4 Recall versus interpolated precision for the local index partitioning. 41
5.5 Recall versus interpolated precision for the global index partitioning. . . . 42
5.6 Cost by processor to execute the 50 TREC queries in LI. 46
5.7 Cost by processor to execute the 50 TREC queries in GI. 47
5.8 Processing time in the load balanced scenario for the 50 TREC queries: GI as percentage of LI. 54
5.9 Processing time for the 2000 artificial queries: GI as percentage of LI. . . . 54

Chapter 1

Introduction

1.1 Motivation

Traditional information retrieval systems usually adopt terms to index and retrieve documents. In its more general form, an index term is simply any word that appears in the text of a document in the collection. The user expresses his information needs through a query that, in its simpler form, is composed by index terms. The information system retrieves the documents that contain such terms and ranks them according to a degree of similarity to the user query. This model of information retrieval has been the most popular one along the years due to its efficiency and simplicity of implementation and use.

The appearance of large text databases, mainly the WWW, caused a sudden change in the setting of information technology. The new requirements of modern search environments have demanded a sophistication of this traditional information retrieval model. Architectures and algorithms that exploit parallel and distributed techniques offer a solution to this problem. The current technology trends benefit the network of workstations model, which motivates application of the distributed computing. Some of the reasons are [ACPtNT95]:

- Network of workstations has become extraordinarily powerful and offer a better price-performance than parallel computers;
- Most networks of workstations have a huge amount of memory and very fast processors, both of which sit idle most of the time;
- 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.

For efficient query processing, an indexing mechanism has to be used with the text database. The main indexing techniques for text collections are inverted files, suffix arrays and signature files [FBY92]. Inverted files have been the most popular indexing technique for text databases due to its simplicity and good performance.

For effective query processing, it is necessary to rank the retrieved documents according to some measure of relevance. The classic models to retrieve and rank text documents are called vector and probabilistic. The vector space model has been appreciated for yielding results either superior or almost as good as the known alternatives [BYRN99].

In the distributed system, in order to index the text database using an inverted file, it is necessary to apply a strategy to physically organize such index. The work presented in [TGM93] characterizes two basic and distinct index organizations. In the first one, each processor generates an inverted file for its local documents and stores this index locally. In the second one, a global inverted file for all the documents in the text database is generated and distributed among the various processors.

The objective of this work is to study query processing in a distributed text database. We implement and evaluate a real distributed architecture that offers concurrent query service. The distributed system adopts 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 retrieved and ranked using the vector space model along with a document filtering technique, that allows significant reduction in ranking costs without degradation in retrieval effectiveness.

We evaluate and compare the impact of the two index partitioning strategies on query processing performance. Regarding retrieval effectiveness, we show that we obtain approximately the same effectiveness as the sequential algorithm, for both the local index partitioning and the global index partitioning. Further, we show that the queries are processed in only 2% of the memory of the basic algorithm for ranking and that only 10% of all term entries in the inverted lists are required, which reduces disk traffic and CPU

processing time.

Regarding retrieval effectiveness, results on overall query processing performance 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 a query, as follows. First, the processing time with the global index partitioning might be twice smaller as that with the local index partitioning. Second, the speedup in the global index partitioning might be 1.7 times as that in the local index partitioning.

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

1.2 Related Work

The work in [TGM93] compares the performance impact on query processing of different physical organizations for inverted lists. It proposes two basic options for storing the inverted lists: disk index and system index. With the disk 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 system index organization, the full lists are evenly spread across all the disks in the system. The adopted query type is the "boolean and". The architecture is that of a LAN, where the number of CPUs, the number of I/O controllers per CPU, and the number of disks per controller are varied. The data used are synthetic documents and queries. Simulation experiments attempt to determine under what conditions each index organization is better, how each index organization scales up to large systems (more documents, more processors) and what is the impact of key parameters, such as seeking time of the storage device, load level, and number of keywords in a query. The experimental results indicate that the disk index organization is a good choice.

Our work differs from that presented in [TGM93] in the following aspects. First, while they adopted the boolean model, we adopt the vector space model that has the following advantages: i) its term-weighting scheme improves retrieval performance; ii) its partial matching strategy allows retrieval of documents that approximate the query conditions; and (iii) its cosine ranking formula sorts the documents according to their degree of similarity to the query [BYRN99]. Second, while they model the document collection and the queries, we base our results on the TREC-3 collection [Har94] and its set of real queries. 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 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 [Bar98, RNB98] studies the query performance for a distributed digital library in a tightly coupled environment. It adopts the two basic and distinct index organizations proposed in [TGM93], and renames them as local index organization and global index organization. In the local index organization, each machine generates an inverted file for its local documents and stores this index locally. In the global index organization, a global inverted file (for all the documents in the library) is generated and distributed among the various machines. The vector space model is adopted as ranking strategy. All estimates are based on the documents and queries in the TREC-3 collection [Har94]. The architecture is that of a network of workstations, where each machine has its own local memory and disk (shared-nothing). Experiments, based on an analytical model coupled with a small simulator, investigate how query performance is affected by the index organization, the network speed, and the disks transfer rate. The results indicate that a global index organization outperforms a local index organization consistently in the presence of fast communication channels.

Our work differs from that presented in [Bar98, RNB98] in the following way. Instead of studying query performance using an analytical model coupled with a simulator, we implement and thoroughly evaluate distributed query processing performance on a real case framework. Our experimental results show that the global index organization overcomes the local index organization, confirming their simulation model results.

The work in [MMR00] examines the search of partitioned inverted files with particular emphasis on issues that arise from different types of partitioning methods. The two types of index partitions proposed in [TGM93] are investigated and renamed as TermId and DocId. TermId partitioning is a type of partitioning which distributes unique word data to a single partition, while DocId partitioning distributes unique document data to a single partition. Documents are searched using the probabilistic model. The search topology is a master/slave topology with a top node and n leaf nodes (each with its own disk). The data used in experiments are part of the documents and queries in the TREC-7 collection [HCT98]. The results from runs on the two types of partitioning are compared

and contrasted. The results indicate that the DocId method is the best.

Our work differs from that presented in [MMR00] in the following aspects. First, we adopt the vector space model for ranking documents, while they adopt the probabilistic model, a less powerful model that does not take into account the frequency that each index term occurs inside a document [BYRN99]. Second, we implement a concurrent query service, while they address a sequential query service. They consider that users tend to submit smaller queries [SHMM99], which implies that in the sequential query service only one or two of the processors in the system will be servicing a query using TermId; the other processors will be doing no work at all. Using DocId, if a processor finishes the execution of a query before the others, then the faster processor stays idle waiting for all the others to finish their computation. So, the sequential query service dramatically deteriorates the system performance, penalizing specially the TermId partitioning. On the other hand, the concurrent query service aims at increasing the system throughput by avoiding to the utmost the idleness in the processors. As soon as a processor returns its local answer set for a query, the broker schedules to it the next query waiting for being serviced. Third, our work differs from that presented in [MMR00] from the fact that 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.

1.3 Contributions of the Thesis

In this work, we present a performance evaluation of a distributed query processing system using partitioned inverted files. Our study compares the impact on query processing performance of two strategies for partitioning inverted files across the network of workstations: local index partitioning and global index partitioning. The main contributions are as follows:

1. A real distributed architecture implementation. Related works base their results on simulation models that hardly foresee all the factors that cause impact on system performance.

2. Concurrent query service implementation. Related works do not evaluate concurrent query service, that provides a much superior performance than the poor and unrealistic sequential query service.

For supporting concurrent query service, the real case implementation adopts a network of workstations model and the client-server paradigm, as presented in Section 2.1. The network is composed by five workstations, each one having its own local memory and disk (shared-nothing). The client-server paradigm consists of four server processes and a designated broker process, that run separately in the different workstations.

The broker offers a concurrent service presented in Section 2.2 for the queries that arrive in the system. It maintains an insertion task, a merging task, and different scheduling tasks for each of the servers in the network. The insertion task inserts queries in the system. The scheduling tasks in parallel and in asynchronous mode dispatch several queries to their respective servers and receive the related responses. This scheduling scheme increases the system throughput by avoiding to the utmost the idleness of processors in the network. The merging task fuses the local answer sets, as soon as all the servers return them, and produces a final answer set for each query in the system.

In respect to the implementation aspects presented in Section 4.1, the interprocess communication between the 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 tasks, maintained by the broker, are implemented with threads that run in parallel. Semaphores are used to synchronize the access to shared memory segments.

3. Efficient ranking evaluation without degradation in retrieval effectiveness. The ranking method is based on the vector space model along with a technique presented in Section 3.3 for filtering documents during ranking. Experimental results presented in Section 5.3 on the sequential query processing algorithm show that the distributed filtering technique allows significant reduction in ranking costs without degradation in retrieval effectiveness. Queries are processed in only 2% of the memory of the basic algorithm for ranking. Disk traffic and CPU processing time are also reduced, as a result of the distributed filtering technique requiring only 10% of all term entries in the inverted lists. Experimental results on the distributed query processing algorithm show that both the local index partitioning and global index partitioning provide retrieval effectiveness comparable to the sequential algorithm.

4. Comparison of two different strategies for partitioning inverted files. Experimental

results presented in Section 5.4.2 show that the global index partitioning outperforms the local index partitioning specially when the number of processors exceeds the average number of terms in a query. In this situation, the global index partitioning takes half the processing time of the local index partitioning, and its speedup might be 1.7 times as that in 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 a query.

Experimental results on the overall query processing performance also show that load imbalance was found to be just over 1 in the local index partitioning, but perceptibly worse in the global index partitioning, because of the probability distribution of terms in a query. Therefore, if load balance was uniform, then the performance with the global index partitioning would be even better.

We also analyzed the costs involved in the main phases of the algorithm. According to the cost analysis presented in Section 5.4.1, both index partitioning strategies are compared in the aspects of disk seeking, reading and processing of inverted lists, ranking of the local answer set, network communication and merging of the local answer sets. In spite of some of these aspects performing better in the local index partitioning, the disk seeking is the dominant effect to make the global index partitioning the best.

5. Proposal of the following future work, as presented in Section 6.2:

   (a) In this work, there is only one broker responsible for scheduling queries to the different servers and merging intermediate results into final results. In future work, we are interested in implementing two types of brokers, one for query scheduling and another for merging of intermediate results, in order of relieving the bottleneck in the merging task.

   (b) In this work, we consider only one query per server at the same time. In future work, for increasing system throughput and decreasing response time, we intend to evaluate the system performance while varying the multiprogramming level in the server.

(c) As the number of processors increases in the network, the number of intermediate results to be sent to the broker enlarges and consequently, the network traffic becomes higher. In future work, in order of minimizing network traffic, we are interested in investigating alternatives for diminishing the amount of information sent to the broker.

(d) We intend to evaluate the behavior of our system while processing Web data, that comprises very large collections and very short queries. We suspect that the advantages of the global index partitioning over the local index partitioning might decrease as the size of the collection increases, not underestimating, however, the fact that the global index partitioning allows high concurrency, specially processing very small queries.

(e) In this work, for the global index partitioning, the inverted lists are evenly distributed by size between processors. In future work, we are interested in studying new strategies to generate the global index by exploiting usage statistics and other measures, for achieving better speedup and load balance.

(f) Also, in order of 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.

(g) Finally, we are interested in studying how the caching of query results and inverted lists proposed in [SMZ+ar] can improve the performance of our system or favor one of the index partitioning strategies.

1.4 Structure of the Thesis

The thesis is organized as follows. Chapter 2 presents the distributed text database, describing the system framework and query processing. Chapter 3 describes the technique for retrieving and ranking documents using the vector space model along with a document filtering technique. Chapter 4 explains the implementation aspects of the system. Chapter 5 shows the experimental results, and Chapter 6 presents the conclusions and future work.

Chapter 2

Distributed Text Database

In this chapter, we present the distributed text database. First, we describe the distributed system framework, explaining the system architecture, the index structure and the two strategies to partition it across the network, namely global index partitioning and global index partitioning. Second, we describe query processing in the distributed text database, presenting the differences between the local index partitioning and the global index partitioning.

2.1 Distributed System Framework

Next, we present the framework of our system, describing the features of the distributed architecture, the technique for indexing the text database and the strategies to partition the text database index across the network.

2.1.1 Distributed 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 2.1 illustrates the network of workstations model.

The retrieval system adopts the client-server paradigm that consists of a set of server

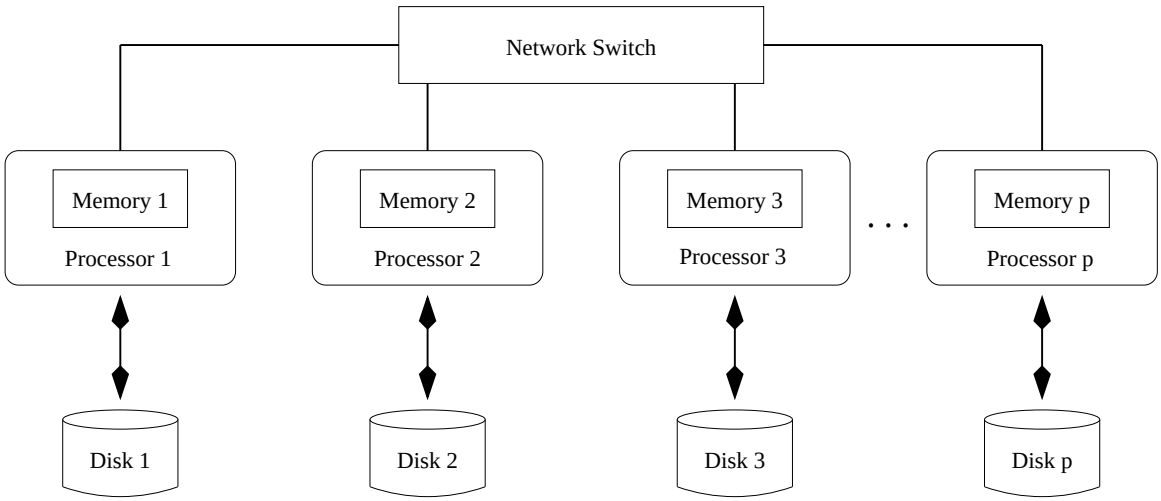

Figure 2.1: Network of workstations model.

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 2.2 illustrates the client-server paradigm. We do not implement the interactions with client applications that might request for service in varied rates. Instead, we consider that the query arrival rate is high enough to fill a query processing queue.

2.1.2 Index Structure

The text database is indexed using the inverted file technique [BYRN99]. 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.

As we adopt the vector space model along with a technique for filtering documents

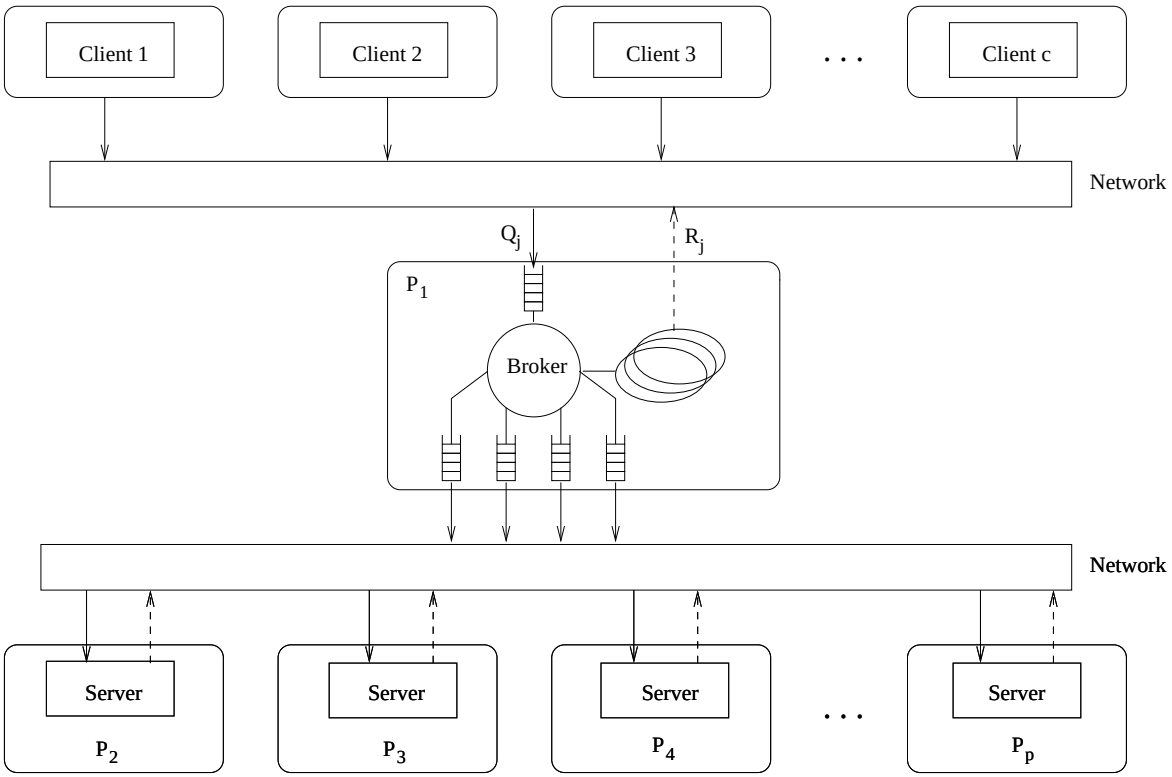

Figure 2.2: Client-server paradigm.

during ranking, the inverted lists are sorted by decreasing within-document frequency.

2.1.3 Index Partitioning

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

Local Index

One possible alternative to partition the text database index is to have a local inverted file for each subcollection. This strategy of index partitioning is denominated local index in the work presented in [Bar98, RNB98]. In the local index partitioning, documents are distributed among processors and each processor generates an inverted file for its documents.

Documents in the text database collection are evenly distributed across processors. The

size sc (in bytes) of the local subcollections is approximately given by Equation(2.1):

$$sc = \frac{N}{p} \tag{2.1}$$

where N is the size (in bytes) of the whole text database collection and p is the number of processors. In other words, considering that documents are evenly distributed, 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 2.3 illustrates the local index partitioning, considering a network composed by 4 processors.

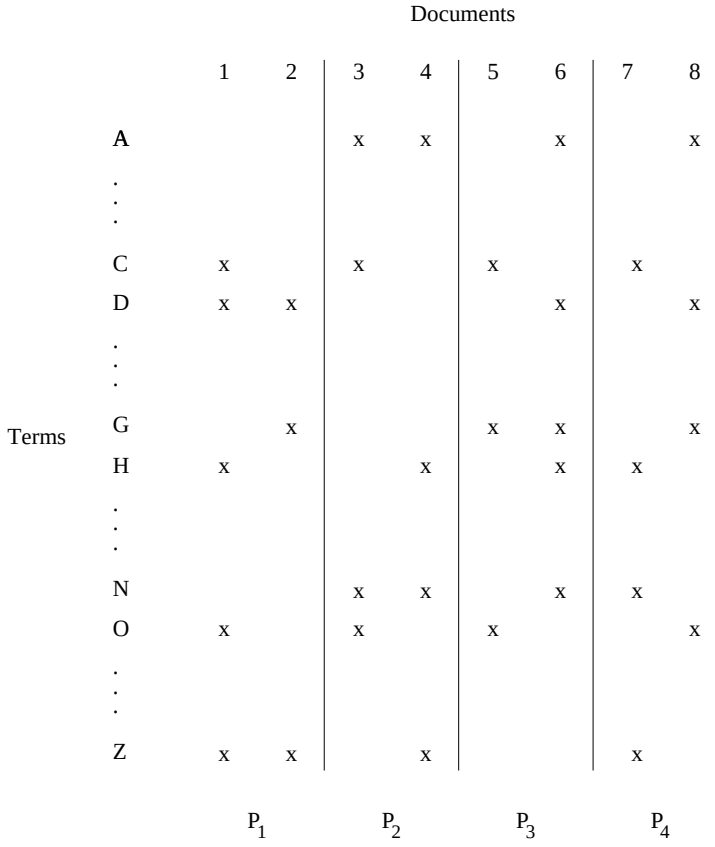

Figure 2.3: Local index partitioning.

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 retrieve and

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.

Global Index

The other alternative to partition the index is to have a global inverted file for the whole text database. This strategy of index partitioning is denominated global index in the work presented in [Bar98, RNB98]. In the global index partitioning, an inverted file is generated for documents in the text database and the inverted lists are distributed among processors.

The inverted lists of terms in the text database are evenly distributed among processors. The size sl (in bytes) of the local subset of inverted lists is approximately given by Equation(2.2):

$$sl = \frac{L}{p} \tag{2.2}$$

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, considering that the inverted lists are evenly distributed, 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 2.4 illustrates the global index partitioning, considering a network composed by 4 processors.

2.2 Distributed Query Processing

Our distributed query system consists of a set of server processes and a designated broker process, each running on a separate processor (see Section 2.1.1). 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 enough to fill a query processing

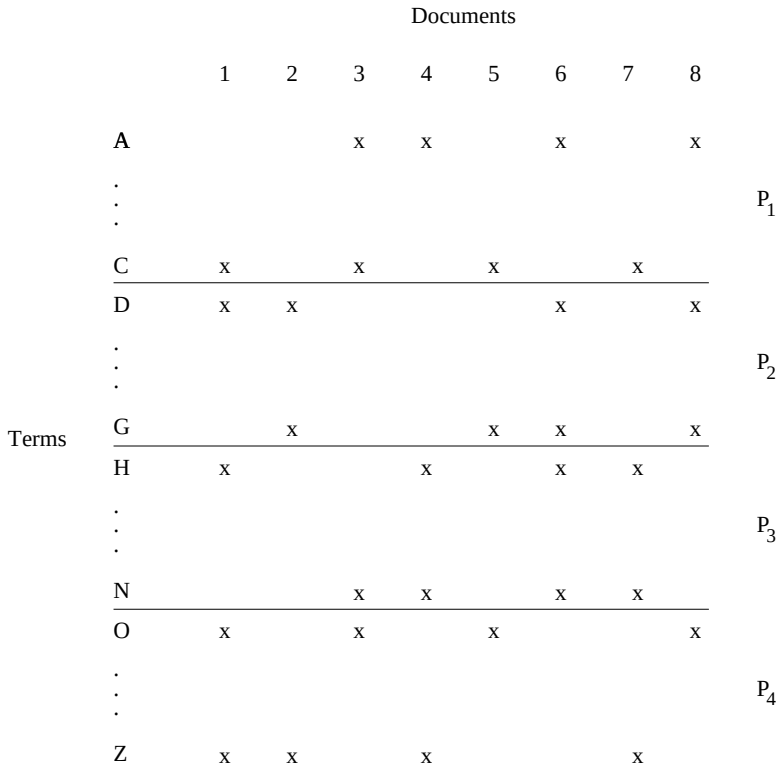

Figure 2.4: Global index partitioning.

queue. Hence, we do not compute the actual user response time for a query, but the system response time. Next, we describe the query processing algorithms implemented in the broker, which differ according to the index partitioning.

2.2.1 Local Index

In the local index partitioning, an individual query is processed in the following way. 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 [WMB99] to fuse the local answer sets and produce the final ranked answer set.

The broker offers a concurrent service for the queries that arrive in the system as follows. The broker maintains an insertion task, a merging task, and different scheduling tasks for

each of the servers in the network. The insertion task is responsible for inserting a query in the system, the scheduling tasks for scheduling queries to servers and receiving local answer sets, and the merging task for fusing local answer sets into final answer sets for each query in the system. All these tasks run in parallel and cooperate with each other in the following way.

The tasks share different scheduling queues, one for each server in the network. As queries arrives in the system, the insertion task inserts them in each of the scheduling queues. Each of the scheduling tasks takes a query out of the related queue, sends the query to the respective server and waits for the server to finish its computation before sending the next query. By this scheduling strategy, the faster server starts the processing of the next query even while the slower server is still processing the previous one. So, the faster server does not stay idle waiting for all the others to finish the execution of the current query. Therefore, this scheduling strategy allows some degree of concurrency with the local index partitioning, that intrinsically provides only a parallel query service, as discussed at Section 2.2.3.

The tasks also share the buffer of intermediate results, where the local answer sets are stored by the scheduling tasks. As soon as all the local answer sets are returned by the servers, the merging task fuses them to generate 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.

2.2.2 Global Index

In the global index partitioning, an individual query is processed in the following way. 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.

Analogous to the local index partitioning, the broker offers a concurrent service for the queries that arrive in the system by maintaining an insertion task, a merging task, and different scheduling tasks for each of the servers in the network. All these tasks run in parallel and cooperate with each other as follows.

The tasks share different scheduling queues, one for each server in the network. As queries arrives in the system, the insertion task breaks them in subqueries and each subquery is inserted in the scheduling queue related to the server that hold their terms. Each of the scheduling tasks takes a subquery out of the related queue, sends the subquery to the respective server and waits for the server to finish its computation before sending the next subquery. This scheduling scheme in parallel and in asynchronous mode dispatches the various subqueries, originated from several queries, to the respective servers and receives the related responses. In this way, more than one query might be processed simultaneously, which increases the system throughput and avoids to the utmost the idleness of processors.

Also analogous to the local index partitioning, the tasks share the buffer of intermediate results, where the local answer sets are stored by the scheduling tasks. As soon as all the local answer sets are returned by the servers, the merging task adds their partial weights and does a sort to generate 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 [Bar98, RNB98] suggests a cutoff factor that depends on the number p of processors. The cutoff factor is given by Equation 2.3:

$$cutoff\_factor = c \times p \times r \tag{2.3}$$

where c is a constant and r is the number of documents in the final answer set. Using such factor, they observed no significant variation in the final answer precision.

2.2.3 Comparison between the Local Index Partitioning and Global Index Partitioning Strategies

The local index partitioning and global index partitioning are compared in the following aspects, as presented in Table 2.1.

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

Table 2.1: Comparison between the local and global index partitioning.

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. 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 a 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, the local rankings consider the global information related to the query, which allows the selection of a set of documents, from the top of the ranking, to be sent to the broker. This quantity is 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 this work, 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. The results are presented in Chapter 5.

Chapter 3

Ranking with the Vector Space Model

In this chapter, we describe the technique for retrieving and ranking documents using the vector space model. We present a document filtering technique for fast ranking proposed in [Per94, PZSD96] and adapted to the distributed processing in [Bar98].

3.1 Vector Space Model

The documents in the text database collection are retrieved and 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 model is either superior or almost as good as the known alternatives. Furthermore, it is simple and fast. For these reasons, the vector space model is a popular retrieval model nowadays [BYRN99].

In the vector space model, documents and user queries are represented as vectors of the weight of terms. The document vector is defined as $\vec{d} = (w_{d,1}, w_{d,2}, \ldots, w_{d,v})$ and the query vector as $\vec{q} = (w_{q,1}, w_{q,2}, \ldots, w_{q,v})$, where v is the total number of index terms in the collection.

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 $\vec{q}$. This correlation

can be quantified, for instance, by the cosine of the angle between these two vectors, given by Equation 3.2:

$$sim(q,d) = \frac{\vec{q} \bullet \vec{d}}{|\vec{q}| \cdot |\vec{d}|} \quad (3.1)$$

$$= \frac{\sum_{t=1}^{v} w_{q,t} \cdot w_{d,t}}{\sqrt{\sum_{t=1}^{v} w_{q,t}^2} \cdot \sqrt{\sum_{t=1}^{v} w_{d,t}^2}} \quad (3.2)$$

where $|\vec{d}|$ is the norm of the document vector, $|\vec{q}|$ is the norm of the query vector, $w_{d,t}$ is the weight associated to the term t of document d, and $w_{q,t}$ is the weight associated to the term t of query q. The factor $|\vec{q}|$ does not affect the ranking, because it is the same for all documents. The factor $|\vec{d}|$ provides a normalization in the space of the documents.

Several term weighting systems have been proposed and explored. We assign the weight to a term in a document or a query using the inverse document frequency, given by Equation 3.3:

$$w_{x,t} = f_{x,t} \cdot log\frac{N}{f_t} \quad (3.3)$$

where $f_{x,t}$ is the number of occurrences of the term t in a document or query x, N is the total number of documents in the collection, and f_t is the number of documents containing t. This function assigns a high weight to terms which occur in only a small number of documents in a collection. It is supposed that rare terms have high discrimination value, and the presence of such a term in both a document and a query is a good sign that the document is relevant to the query.

A basic algorithm for retrieving and 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 $sim_{q,d,t}$, 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; this contribution, called partial similarity, is given by Equation 3.4:

$$sim_{q,d,t} = w_{d,t} \cdot w_{q,t} \quad (3.4)$$

The final result is composed by the documents with the highest accumulator values. A simple version of this algorithm is shown in Figure 3.1.

1. For each document d in the collection, set accumulator $A_d \leftarrow 0$.
2. For each term t in the query,
   (a) Retrieve the inverted list for t from disk.
   (b) For each term entry $< d, f_{d,t} >$ in the inverted list,
   set $A_d \leftarrow A_d + sim_{q,d,t}$.
3. Divide each non-zero accumulator A_d by the document norm $\mid \vec{d} \mid$.
4. Identify the r highest accumulator values (where r is the number of documents to be presented to the user) and retrieve the corresponding documents.

Figure 3.1: Basic algorithm for ranking using the vector space model.

3.2 Filtering Technique

For a large document database, the ranking evaluation cost - volume of main memory, CPU processing time and disk traffic - can be prohibitively high, because it assigns a similarity value to every document containing any of the query terms. The work in [Per94, PZSD96] proposes a technique for filtering documents during ranking, which allows a significant reduction in both the volume of main memory required and the time of query evaluation, without degradation in retrieval effectiveness. The approach uses early recognition of which documents are likely to be highly ranked to reduce costs and works as follows.

Query terms are sorted by decreasing f_t, so that important terms are processed first. Before each term t is processed, two thresholds are computed, an insertion threshold s_{ins} and an addition threshold s_{add}, where $s_{add} \leq s_{ins}$. As the inverted list for t is processed, the partial similarity $sim_{q,d,t}$ of query q and each document document d in the list is compared to the thresholds. If $sim_{q,d,t} \geq s_{ins}$, then document d is inserted in the set of candidate documents for the final answer and the $sim_{q,d,t}$ is added to the value of the accumulator for d. Otherwise, if $s_{add} \leq sim_{q,d,t} < s_{ins}$, although document d is not important enough to be one of the candidates, the $sim_{q,d,t}$ might affect the ranking; so if d has an accumulator, then $sim_{q,d,t}$ is added to its value. Otherwise, if $sim_{q,d,t} < s_{add}$, then this partial similarity is discarded and the processing goes to the next query term.

Inverted lists are sorted by decreasing $f_{d,t}$, so that the identifiers of the interesting documents are brought to the start of the list, yielding a reduction in disk traffic because

only part of each inverted list needs to be retrieved. In other words, once an $f_{d,t}$ value is encountered that is below s_{ins}, processing of the inverted list can stop. This implies that if the inverted list is longer than a disk block, then only one block of the list needs to be retrieved at a time.

The threshold s_{ins} allows us to ignore some documents, thus saving memory space. Using the threshold s_{add}, the inverted list entries that yield small partial similarities can be ignored, thus saving CPU processing time and disk traffic. The value of both thresholds are determined as a function of the accumulated partial similarity of the currently most relevant document S_{max}. This heuristic supposes that if the current most relevant document has a high weight, then we do not need to process a document that has a small value of similarity to a query, as it is unlikely to significantly change the final ranking or identify an important document that is not yet included in the candidate set.

The threshold s_{ins} is given by Equation 3.5 and the threshold s_{add} by Equation 3.6:

$$s_{ins} = c_{ins} \cdot S_{max} \tag{3.5}$$

$$s_{add} = c_{add} \cdot S_{max} \tag{3.6}$$

where $0 \leq c_{add} \leq c_{ins}$ are constants.

The term entry $< d, f_{d,t} >$ in the inverted list of t is processed only if the partial similarity $sim_{q,d,t}$ of document d and query q is greater than the current values of the threshold s_{ins} or s_{add}. This condition can be expressed by Equation 3.9:

$$s \leq sim_{q,d,t} \tag{3.7}$$

$$s \leq f_{d,t} \cdot f_t \cdot f_{q,t} \cdot f_t \tag{3.8}$$

$$\frac{s}{f_{q,t} \cdot f_t^2} \leq f_{d,t} \tag{3.9}$$

where s is either s_{ins} or s_{add}.

So the thresholds can be directly expressed by Equations 3.10 and 3.11:

$$f_{ins} = \frac{c_{ins} \cdot S_{max}}{f_{q,t} \cdot f_t^2} \tag{3.10}$$

$$f_{add} = \frac{c_{add} \cdot S_{max}}{f_{q,t} \cdot f_t^2} \tag{3.11}$$

For the first terms processed, the value of S_{max} is small and the value of f_t is large, so that most documents are considered. As S_{max} rises and f_t falls, the thresholds rise

until all $f_{d,t}$ values are less than f_{add}, so that processing an inverted list has no effect on accumulator values. The algorithm for the filtering technique is shown in Figure 3.2.

The constants c_{ins} and c_{add} are used to control the resources required by the algorithm. By increasing the constant c_{ins}, we reduce the number of documents that can be candidates for the final answer, and hence decrease memory usage; by increasing the constant c_{add}, we reduce the number of term entries processed and accumulated by the algorithm, and hence decrease CPU processing time and disk traffic. The value for these constants can be adjusted by running queries for several values of each constant and choosing the best values according to the distortion introduced into the answer set.

1. Create an empty structure of accumulators.
2. Sort the query terms by decreasing weight.
3. Set S_{max} to 0.
4. For each term t in the query,
   (a) Compute the values of the thresholds f_{ins} and f_{add}.
   (b) Retrieve the inverted list for t from the disk.
   (c) For each term entry $< d, f_{d,t} >$ in the inverted list,
      i. If $f_{d,t} \geq f_{ins}$, then create an accumulator for A_d if necessary, and set $A_d \leftarrow A_d + sim_{q,d,t}$.
      ii. Otherwise, if $f_{d,t} \geq f_{add}$ and A_d is present in the set of accumulators, then set $A_d \leftarrow A_d + sim_{q,d,t}$.
      iii. Set $S_{max} \leftarrow max(Smax, A_d)$.
5. Divide each non-zero accumulator A_d by the document norm $| \vec{d} |$.
6. Identify the r highest accumulator values (where r is the number of documents to be presented to the user) and retrieve the corresponding documents.

Figure 3.2: Filtering algorithm for fast ranking using the vector space model.

3.3 Distributed Filtering Technique

The efficiency of the filtering technique is influenced by the constants c_{ins} and c_{add}, and specially by the accumulated partial similarity of the currently most relevant document S_{max}. The value of S_{max} rises progressively as the value of accumulated similarity of documents in the set of answers grows.

The growth of the S_{max} value in the distributed algorithm, using both the local and global index partitioning, differs from that in the sequential algorithm, as illustrated in Figure 3.3 and Figure 3.4, for the local and global index partitioning, respectively, using a network composed by 4 processors to execute the query derived from the topic description 193 in TREC-3 [Har94].

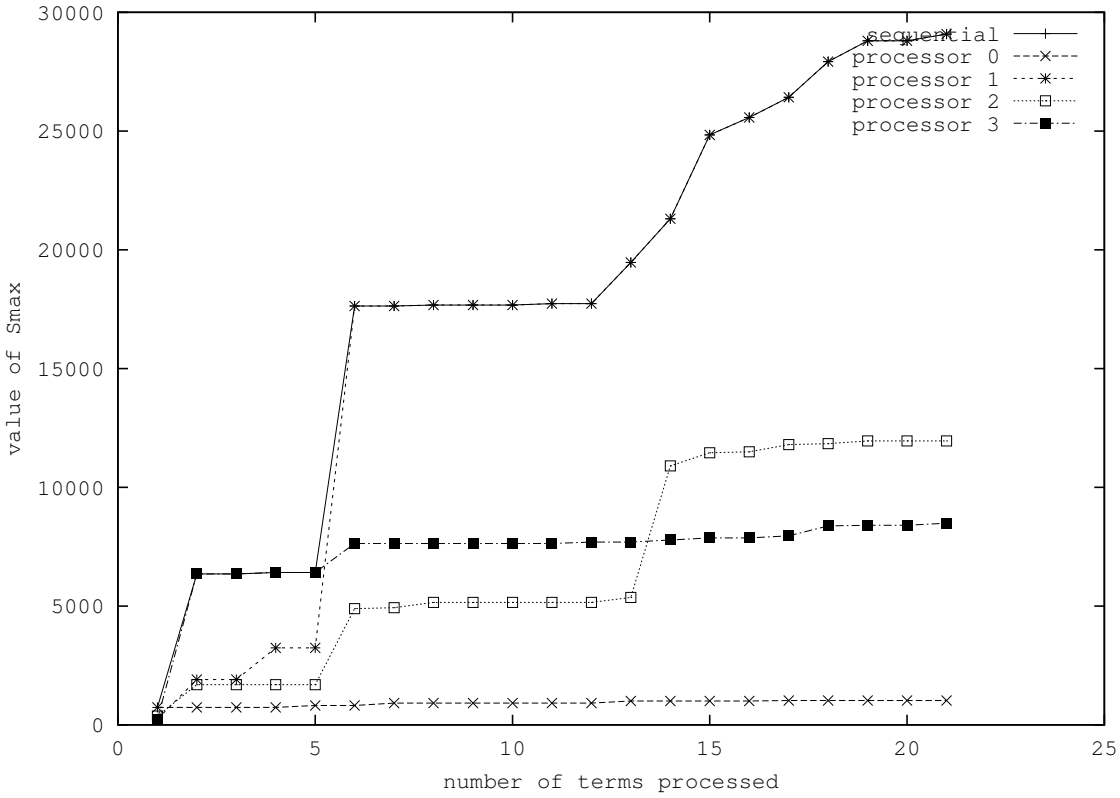

Figure 3.3: Increasing of S_{max} in the sequential and distributed algorithm using local index partitioning with 4 processors to execute the TREC-3 query 193.

In the local index partitioning, if one of the processors holds only a few high weighted documents, then the rising of S_{max} is low; consequently, the amount of pruned resources is smaller than in the sequential algorithm. This effect can deteriorate the performance of

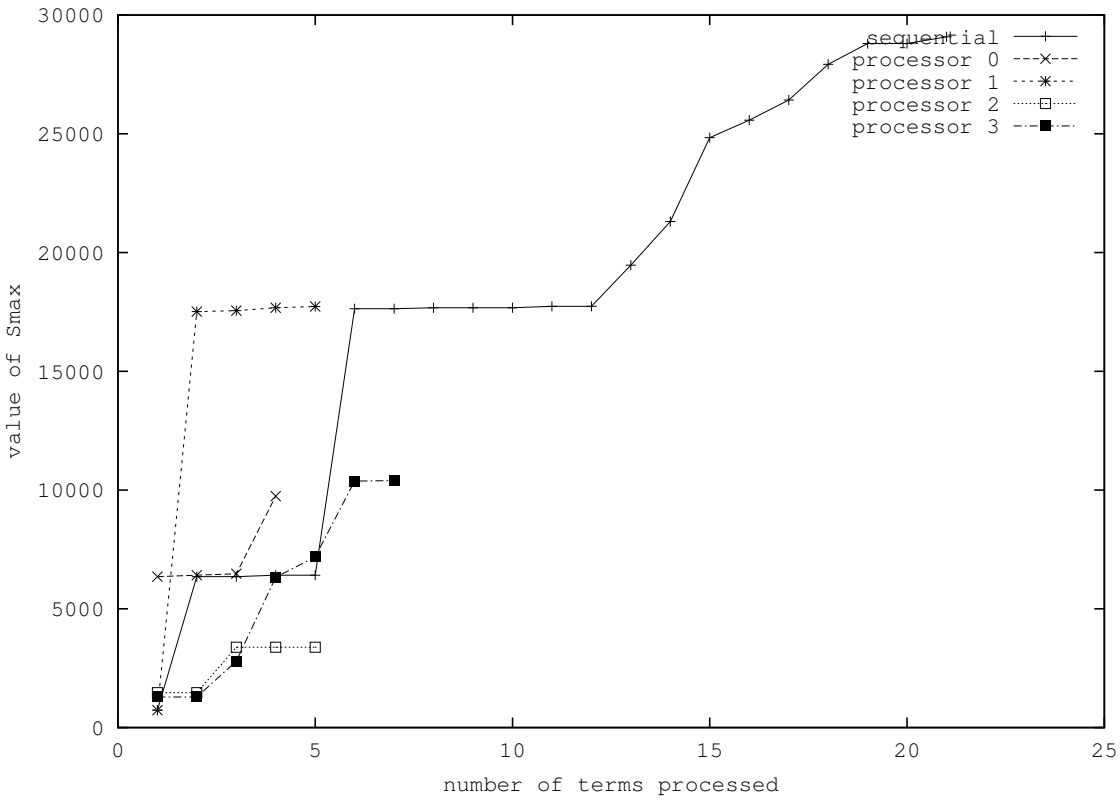

Figure 3.4: Increasing of S_{max} in the sequential and distributed algorithm using global index partitioning with 4 processors to execute the TREC-3 query 193.

the distributed algorithm, making it even worse than the sequential one. In the example shown in Figure 3.3, the execution time in processor 0 was higher than all the time taken by the sequential algorithm, because the rising of the S_{max} value was low, making poor the pruning of resources.

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. Again, the performance of the distributed algorithm might be seriously damaged.

The work in [Bar98] proposes a solution to this problem. They argued that it is necessary to preview the rising of the S_{max} value before query processing. They observed that, in fact, there is a pattern to the rising of S_{max}. It follows the growth of the accumulated partial similarity of the most relevant documents, which are often the documents that contain the most query terms. Based in this fact, they put forward the following hypothesis: for each query, there is at least one document that contains all the terms.

Turning back to Equation 3.3 and Equation 3.4, the partial similarity yielded by each

term can be calculated in a global way, if the values of N, f_t, $f_{d,t}$ and $f_{q,t}$ are known. The value of N is the number of documents in the collection, f_t is the number of documents containing t, and $f_{q,t}$ is the number of occurrences of term t in the query q. To estimate the value of $f_{d,t}$, they adopted the maximum within-document frequency f_{max_t} of term t. In this way, the points of growth of S_{max} can be previously calculated and distributed to the processors, along with the query. The pre-calculation of the points of growth of S_{max} can be given by Equation 3.12:

$$PGSmax_t = f_{q,t} \cdot log\frac{N}{f_t} \cdot f_{max_t} \cdot log\frac{N}{f_t} \tag{3.12}$$

Besides of adapting the filtering technique to the distributed processing, the work in [Bar98] obtained results even better than the original filtering technique proposed in [Per94, PZSD96] for the sequential algorithm. The distributed filtering algorithm is shown in Figure 3.5.

1. Create an empty structure of accumulators.
2. Sort the query terms by decreasing weight.
3. Set S_{max} to 0.
4. For each term t in the query,
   (a) Set $S_{max} \leftarrow S_{max} + PGS_{max}(t)$.
   (b) Compute the values of the thresholds f_{ins} and f_{add}.
   (c) Retrieve the inverted list for t from the disk.
   (d) For each term entry $< d, f_{d,t} >$ in the inverted list,
      i. If $f_{d,t} \geq f_{ins}$, then create an accumulator for A_d if necessary, and set $A_d \leftarrow A_d + sim_{q,d,t}$.
      ii. Otherwise, if $f_{d,t} \geq f_{add}$ and A_d is present in the set of accumulators, then set $A_d \leftarrow A_d + sim_{q,d,t}$.
5. Divide each non-zero accumulator A_d by the document norm $| \vec{d} |$.
6. Identify the r highest accumulator values (where r is the number of documents to be presented to the user) and retrieve the corresponding documents.

Figure 3.5: Distributed filtering algorithm for fast ranking using the vector space model.

Chapter 4

Implementation Aspects

In this chapter, we describe some details of the real case implementation of query processing in the distributed text database.

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. The operating system is the Linux kernel 2.2.14.

4.1 Client/Server Model

In this work, 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 appropriated 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 offers a concurrent service for the queries that arrive in the system. It is constituted by different tasks that in parallel and in asynchronous mode dispatch several queries to the different servers, receive the intermediate results and generate 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 servers is socket-based. The data transmission mechanism is stream-based, which provides sequenced, reliable, two-way and connection-based byte streams. The tasks, maintained by the broker, are implemented with threads that run in parallel. Semaphores are used to synchronize the access to shared memory segments.

Next, we describe more closely how we model the interactions between the broker process (*B-Process*) and server processes (*S-Processes*), thinking of the broker as a client, while it requests service to the other servers. Also, we describe how the threads of the broker process cooperate in order of carrying on their conjoined function of providing concurrent query service.

4.1.1 Broker Process (*B-Process*)

The function of the broker process (*B-Process*) is to insert queries in the system, schedule queries to the *S-Processes*, receive the local answer sets returned by the *S-Processes* and merge local answer sets into final answer sets. For carrying out its task and guaranteeing concurrent service, the *B-Process* runs an insertion thread (*I-Thread*), a merging thread (*M-Thread*), and different scheduling threads (*Sch-Threads*) for each *S-Process*. All these threads run in parallel in the *B-Process*. The *I-Thread* is responsible for inserting a query in the system, the *Sch-Threads* for scheduling queries to *S-Processes* and receiving local answer sets, and the *M-Thread* for fusing local answer sets into final answer sets for each query in the system.

The main data structures shared by the threads in the *B-Process* are the scheduling queues (*Sch-Queues*), the buffer of intermediate results (*R-Buffer*) and the merging queue (*M-Queue*). There is one *Sch-Queue* for each *Sch-Thread*. The *Sch-Queues* contain queries, if the index partitioning is the local one, or subqueries, if the index partitioning is the global one. The *R-Buffer* temporarily contains local answer sets waiting for being merged into final answer sets. The *M-Queue* contains identifiers of queries whose local answer sets are ready for being merged.

The main synchronization primitive is the merging semaphore (*M-Semaphore*), that synchronizes the access to the *M-Queue* shared by the *S-Thread* and the *M-Thread*. Next, we describe the implementation of the threads in details.

Insertion Thread (*I-Thread*)

The function of the insertion thread (*I-Thread*) is to insert a query in the system. A high level description of the implementation is presented in Figure 4.1 and Figure 4.2, for the local and global index partitioning, respectively.

1. Forever,
   (a) Receive a query;
   (b) Insert the query in all the scheduling queues;

Figure 4.1: Insertion thread algorithm of the broker process in the local index partitioning.

1. Forever,
   (a) Receive a query;
   (b) For each term t in the query,
      i. Determine the first character *char* of t;
      ii. Identify the processor p that holds terms initiating with *char*;
      iii. Insert t in the subquery to be sent to p;
   (c) Insert each subquery in the scheduling queue related to the processor that hold their terms;

Figure 4.2: Insertion thread algorithm of the broker process in the global index partitioning.

In the local index partitioning, the query to be serviced is inserted in each of the *Sch-Queues* related with the various *S-Processes*. In the global index partitioning, the query to be serviced is broken in subqueries and each subquery is inserted in the *Sch-Queue* related to the *S-Process* that hold their terms. The breaking of a query in subqueries is as follows. The inverted lists of terms are distributed among processors in lexicographical order. In this way, the sequence of terms designated to each processor can be easily determined by two characters of the alphabet. For example, the processor 1 might hold the sequence of

terms that begin with A to C. To break a query in subqueries, the *I-Thread* parses the query for identifying the terms. For each term t identified, the *I-Thread* determines the first character *char* of t; identifies the processor p that holds terms initiating with *char*; and inserts t in the subquery to be sent to p.

Scheduling Threads (*Sch-Threads*)

The function of the scheduling threads (*Sch-Threads*) is to schedule the queries to *S-Processes* and receive the local answer sets returned by the *S-Processes*. A high level description of the implementation is presented in Figure 4.3.

1. Forever,
   (a) Take a query (or subquery) out of the scheduling queue;
   (b) Request a connection to the server;
   (c) Send the query (or subquery) to the server;
   (d) Receive the local answer set from the server;
   (e) If all the local answer sets were returned by the servers, then insert the query identifier in the merging queue;

Figure 4.3: Scheduling thread algorithm of the broker process.

For each query (or subquery) in the *Sch-Queue*, the *Sch-Thread* requests service from the *S-Process* by creating a socket and initiating a connection to the *S-Process*'s socket. The connect() system call is used to initiate a connection.

With the connection established, data may begin to flow. The write() system call applied to the socket is used to send the query to the *S-Process*. To read the local answer set from the *S-Process*, the read() system call is used on the socket. The local answer set is stored in the *R-Buffer*.

After receiving the local answer set, the *Sch-Thread* checks if all the *S-Processes* returned their intermediate results for the query. This checking is done in the following way. To each query is associated a bit map that indicates if the *S-Processes* returned their results. The map contains a bit for each *S-Process* and initiates with zero. When a *S-Process* returns its result, the *Sch-Thread* updates the map, turning on the bit related with that

S-Process. The *Sch-Thread* deduces that all intermediate results have been returned if all the bits are turned on.

If all the *S-Processes* returned their intermediate results for the query, then the *Sch-Thread* inserts the query identifier into the *M-Queue* for being merged. Further, the *Sch-Thread* increments the *M-Semaphore*, in order of releasing the *M-Queue* to the *M-Thread*. The sem_op system call is used to increment the semaphore's current value.

Merging Thread (*M-Thread*)

The function of the merging thread (*M-Thread*) is to fuse the local answer sets returned by the *S-Processes* and produce a final answer set to each query in the system. A high level description of the implementation is presented in Figure 4.4.

1. Forever,
   (a) Suspend until the merging queue is not empty;
   (b) Take a query identifier out of the merging queue;
   (c) Merge the local answer sets related to the query identifier;

Figure 4.4: Merging thread algorithm of the broker process.

The *M-Thread* suspends until the *M-Queue* is not empty. Here, a semaphore (*M-Semaphore*) is used to synchronize the access to the *M-Queue*. The value of the *M-Semaphore*'s variable at any point in time is the number of elements available in the *M-Queue*. To wait until the *M-Semaphore*'s value becomes greater than zero, the *M-Thread* uses the sem_op system call. Then the value is decremented by one.

If the *M-Semaphore*'s value is greater than zero, then the *M-Thread* takes a query identifier out of the *M-Queue*. The query identifier indicates the positions in the *R-Buffer* that store the local answer sets related to the query. The merging of the local answer sets differs for the local and global index partitioning and is done as follows.

In the local index partitioning, we use a p-way merge [WMB99], where p is the number of processors and consequently the number of local ranked answer sets to be merged. A heap is employed to obtain the minimum of a slowly changing set of candidates to be maintained. The merge begins with the insertion of the first element of each local ranked

answer set into the heap. Next, an element is extracted from the heap and inserted into the final ranked answer set. Subsequently, the local ranked answer set that contains the element extracted from the heap inserts the next element of its ranking in the heap. The extraction of elements from the heap and the subsequent insertion into the heap of elements in the local ranked answer sets continues until all the elements passes through the heap, or until the final ranked answer set contains r elements. The time complexity of the p-way merge is $O(p \cdot log\ p)$.

In the global index partitioning, the merging procedure needs to add the partial weights of documents present in different local answer sets and do a sort to generate the final ranked answer set. A hash table is employed in the addition phase and the quicksort algorithm [Ziv93] is used in the sorting phase. In the addition phase, all the elements (pairs document-weight $< d, w >$) from all the local answer sets are inserted into the hash table. To insert the elements $< d, w >$, the algorithm uses d as the key and w as the value. When the algorithm tries to insert the element $< d, w >$ in the hash table, if d is already present in position pos, then the weight w is summed to the current value of the position pos. Otherwise, d is inserted in an empty position pos and the value of pos is initiated with w. As the size of each local answer set is $(c \cdot p \cdot r)$, where c is a constant, p is the number of processors (and consequently the number of local answer sets), and r is the number of documents in the final ranked answer set, and as the time complexity to insert an element into the hash table is $O(1)$, so the time complexity of the addition phase is $O(p^2 \cdot r)$.

Once all the elements are inserted into the hash table, the quicksort algorithm is used to sort them in decreasing order of weight. The first r elements of the sorted sequence composes the final ranked answer set. The time complexity for the sorting phase is $O(n \cdot log\ n)$, where n is the number of elements in the set to be sorted. In the worst case, where all the elements in the local answer sets represent distinct documents, the time complexity of the sorting phase is $O((p^2 \cdot r) \cdot log\ (p^2 \cdot r))$.

4.1.2 Server Process (*S-Process*)

The function of the server process (*S-Process*) is to service a query by retrieving and ranking documents in the local subcollection. A high level description of the implementation is presented in Figure 4.5.

The *S-Process*, willing to offer its query service, binds a socket to an address associated with the service and then passively listens on its socket. The bind() system call is used to bind the socket to the service address. After binding its socket, the *S-Process* must

1. Forever,
   (a) Block until a connection is requested by the broker;
   (b) Receive a query (or subquery) from the broker;
   (c) Process the query (or subquery);
   (d) Send the local answer set to the broker;

Figure 4.5: Server process algorithm.

indicate a willingness to listen for incoming connection requests, which can be done with the listen() system call. The listen() call also specifies the maximum number of outstanding connections that may be queued awaiting acceptance by the *S-Process*.

With the socket marked as listening, the *S-Process* may accept a connection with the *B-Process*, which can be done with the accept() system call. The accept() call will not return until a connection is available. Once the connection was established, the *S-Process* is ready to receive a query (or subquery) from the *B-Process*. The normal read() system call is used on the socket for receiving the query (or subquery).

The algorithm used to process the query (or subquery) is described in Figure 3.5. It uses the vector space model along with a document filtering technique for reduction in ranking costs proposed in [Per94, PZSD96] and adapted to the distributed processing in [Bar98]. More details on this algorithm may be obtained in Section 3.3.

The data structure of accumulators is a hash table. The key of the hash table is the document identifier, and the value is the accumulated similarity between the query and the documents. Before processing the query (or subquery), the positions in the hash table used to process the last query are initialized with zero.

For each term t in the query (or subquery) q, the values of the insertion and addition thresholds are calculated. Then, only one block of the inverted list of t is read from disk. For each term entry $< d, f_{d,t} >$ in the inverted list (where d is a document identifier and $f_{d,t}$ is the frequency of t in d), the partial similarity $sim_{q,d,t}$ between d and q is calculated. If $f_{d,t}$ is greater than or equal to the insertion threshold, then d is inserted into the hash table in the position *pos* and the value of *pos* is initiated as $sim_{q,d,t}$; if d is already present in position *pos*, then $sim_{q,d,t}$ is summed to the current value of the position *pos*. Otherwise, if

$f_{d,t}$ is greater than or equal to the addition threshold, and if d is already present in position pos, then $sim_{q,d,t}$ is summed to the current value of the position pos. Otherwise, if $f_{d,t}$ is smaller than the addition threshold, then $sim_{q,d,t}$ is discarded and the processing goes to the next query term.

If $f_{d,t}$ of the last term entry in the current inverted list block is greater than or equal to the addition threshold, then the processing continues on the next inverted list block of the current query term. In this way, we reduce disk traffic by reading only one block of the inverted list at a time. The time complexity to insert an element in the hash table is $O(1)$. So, the time complexity for accumulating the partial similarities depends on the number of term entries processed.

After all terms in the query (or subquery) are processed, each non-zero position in the hash table is divided by the norm of the related document. Next, the quicksort algorithm is used to sort the documents in decreasing order of accumulated similarity. The time complexity for the sorting phase is $O(n \cdot log\ n)$, where n is the number of documents in the set to be sorted. The time complexity for identifying the non-zero positions in the hash table and for dividing accumulated similarities by the norm of the related document is $O(h)$, where h is the size of the hash table. Given this time complexity and aiming at improving speedup, we minimized the hash table size with the increase in the number of processors in the network. For the local index partitioning, as the number of processors doubles the hash table size is divided in half. For the global index partitioning, as the number of processors increases the hash table size decreases in a small proportion. The reason is that the pruning of candidate documents must be less rigorous in the global index partitioning for guaranteeing retrieval effectiveness.

Finally, the *S-Process* sends the top r documents of the local ranking to the *B-Process*. The write() system call applied to the socket is used to send the local ranked answer set to the *B-Process*.

Chapter 5

Experimental Results

In this chapter, we present the experimental results on the real case implementation of query processing in the distributed text database. We compare the performance impact on query processing of both the local and the global index partitioning strategies, in regard to retrieval effectiveness and retrieval efficiency.

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 used in the experiments comprise the disks 1 and 2 of the TREC-3 collection [Har94]. 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, as presented in Table 5.1.

	TREC Query Set	Artificial Query Set
Number of queries	50	2000
Origin of terms	topics 151-200	vocabulary
Number of terms on average	21	2

Table 5.1: TREC query set and artificial query set.

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 [ANZ97]. The number of terms per query is 2 or 3.

5.2 Metrics

Next, we define the various performance metrics used in our experiments.

1. **Recall** of a ranking method for some value r is the fraction of the total number of relevant documents that were retrieved in the top r:

$$R_r = \frac{number\ relevant\ that\ are\ retrieved}{total\ number\ relevant}$$

   For example, there are 70 relevant documents; if 50 documents are retrieved in answer to some query and 35 of them are relevant, then the recall at 50 is $R_{50} = 50\%$, since 35/70 of the relevant documents were selected within the top 50. Recall is usually reported at 11 standard points - $0\%, 10\%, \ldots, 100\%$ [WMB99].

2. **Precision** of a ranking method for some value r is the fraction of the top r ranked documents that are relevant to the query:

$$P_r = \frac{number\ retrieved\ that\ are\ relevant}{total\ number\ retrieved}$$

   For example, if 50 documents are retrieved in answer to some query and 35 of them are relevant, then the precision at 50 is $P_{50} = 70\%$ [WMB99].

3. **Interpolated Precision**. Let r_j, $j \in 0, 10, 20, \ldots, 100$, be a reference to the j-th standard recall level. Then,

$$P(r_j) = max_{\ r_j \leq r \leq r_{j+1}}\ P(r)$$

   which states that the interpolated precision at the j-th standard recall level is the maximum known precision at any recall level between the j-th recall level and the $(j+1)$-th recall level. The utilization of precision interpolation procedure is often necessary, since the recall levels for a query might be distinct from the 11 standard recall levels [BYRN99].

4. **11-pt Effectiveness** is the average of interpolated precision at 11 standard recall levels $(0\%, 10\%, \ldots, 100\%)$ [BYRN99].

5. **Retrieval Effectiveness** is a measure of how useful (or relevant) are the answers produced by an algorithm. We measured retrieval effectiveness from the 11-pt effectiveness metric.

6. **Processing Time** is the elapsed time in seconds to process a batch of queries in the distributed system using p processors. It comprehends the elapsed time since the broker takes the first query out of the batch of queries until the broker merges all the local answer sets for the last query in the batch.

7. **Speedup** is defined as:

$$S = \frac{processing\ time\ of\ sequential\ algorithm}{processing\ time\ of\ parallel\ algorithm\ with\ p\ processors}$$

   Ideally, when running a parallel algorithm on p processors, we would obtain perfect speedup, or $S = p$. In practice, perfect speedup is unattainable either because the problem cannot be decomposed into p equal subtasks, the parallel architecture imposes control overhead (e.g., scheduling or synchronization), or the problem contains an inherently sequential component [BYRN99].

8. **Load Imbalance** is the non-uniform spread of a given computation across a number of nodes in a parallel computer. It is given by:

$$S = \frac{maximum\ processing\ time\ of\ processors}{average\ processing\ time\ of\ processors}$$

   Ideally all nodes should have the same computational load [MMR00].

9. **Retrieval Efficiency** is a measure of the algorithm ability to provide high query processing rates. We measured retrieval efficiency from the processing time, speedup and load imbalance metrics.

10. **Seek Time** is the time to move the arm, located over the disk surface and containing a read/write head, to the desired track [HP90].

5.3 Retrieval Effectiveness

In this section, we compare the retrieval effectiveness between the original filtering technique (Section 3.2) and its adaptation to the distributed processing (Section 3.3). We show that for the same retrieval effectiveness the filtering technique adaptation requires less resources than the original one. The experimental results are based on the sequential query processing algorithm.

We measured the retrieval effectiveness from interpolated precision at 11 standard recall levels (from 0 percent through 100 percent) and from 11-pt effectiveness. We retrieved only the top 200 documents for each query and the results are average values over all 50 TREC queries.

Afterwards, we present the retrieval effectiveness obtained for both index partitioning strategies, adopting the filtering technique adaptation in the distributed query processing algorithm. We show that we obtain approximately the same retrieval effectiveness as the sequential algorithm, no matter the index partitioning strategy considered.

5.3.1 Filtering Technique

The objective of this experiment is to adjust the values of the insertion constant c_{ins} and addition constant c_{add}. For more details about c_{ins} and c_{add}, please refer to Section 3.2. In order of reducing the number of accumulators required and the percentage of term entries processed by the filtering algorithm, without causing a deterioration in the retrieval effectiveness, the values for c_{ins} and c_{add} were obtained as follows:

1. First, we measured the retrieval effectiveness with $c_{ins} = c_{add} = 0$, that is, for the basic algorithm described in Figure 3.1. The value of retrieval effectiveness we found with the basic algorithm was 10.68%.

2. Second, we fixed $c_{add} = 0$ and increased c_{ins} until achieving the largest value that still gives an equal or even better retrieval effectiveness as the basic algorithm. This experiment is shown in Figure 5.1. The value we chose was $c_{ins} = 1 \times 10^{-2}$.

3. Finally, we fixed $c_{ins} = 1 \times 10^{-2}$ and measured retrieval effectiveness for increasing values of c_{add}, observing also the percentage of term entries processed by each value of c_{add}. The results are shown in Figure 5.2 and Table 5.2.

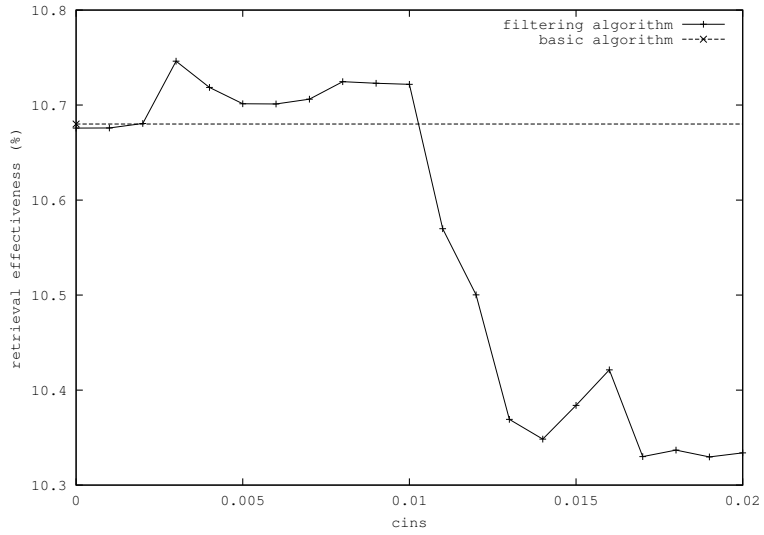

Figure 5.1: Retrieval effectiveness for increasing values of c_{ins} ($c_{add} = 0$).

c_{add}	Processed term entries (%)	Retrieval effectiveness (%)
0	100	10.72
3.3×10^{-4}	70	10.68
5.7×10^{-4}	50	10.52
1.64×10^{-3}	20	10.34
3.1×10^{-3}	10	10.01

Table 5.2: Percentage of term entries processed and retrieval effectiveness for increasing values of c_{add} ($c_{ins} = 1 \times 10^{-2}$) using the filtering algorithm.

The main saving yielded by this technique is a sharp reduction in the number of accumulators. The basic algorithm requires 340, 394 accumulators, while the filtering technique using $c_{ins} = 1 \times 10^{-2}$ results in 13, 189 accumulators. Furthermore, for this number of accumulators, we obtain even better retrieval effectiveness (10.72%) in comparison to the basic algorithm (10.68%), using only 4% of the accumulators.

The technique also yields substantial savings in either disk traffic and CPU processing time. For $c_{add} = 3.1 \times 10^{-3}$, processing only 10% of all term entries, the deterioration in the overall retrieval effectiveness is only 0.67%.

Unfortunately, this filtering technique does not work very well for the distributed processing. Next, we describe the results using the adaptation presented in Section 3.3 of this filtering technique to the distributed processing.

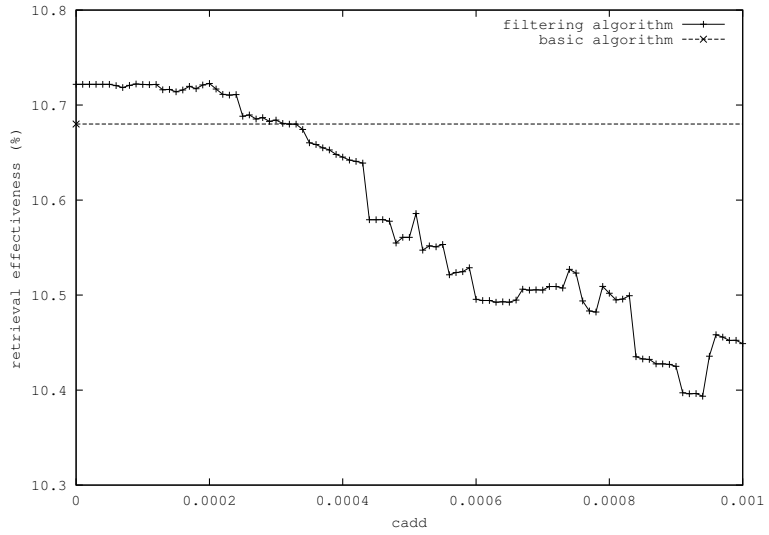

Figure 5.2: Retrieval effectiveness for increasing values of c_{add} ($c_{ins} = 1 \times 10^{-2}$).

5.3.2 Distributed Filtering Technique

Using the distributed filtering algorithm and following the same procedure used in the previous section, we adjusted the values of the insertion constant c_{ins} and addition constant c_{add}. The value we chose for c_{ins} was 6×10^{-3} and the results for c_{add} are shown in Table 5.3.

c_{add}	Processed term entries (%)	Retrieval effectiveness (%)
0	100	10.75
1×10^{-4}	70	10.73
1.8×10^{-4}	50	10.61
5.2×10^{-4}	20	10.43
1.03×10^{-3}	10	10.11

Table 5.3: Percentage of term entries processed and retrieval effectiveness for increasing values of c_{add} ($c_{ins} = 6 \times 10^{-3}$) using the distributed filtering algorithm.

In the distributed filtering technique, we observe a sharper reduction in the number of accumulators compared to the original filtering technique. The latter requires 13,189 accumulators (4% of the amount used by the basic algorithm), and the former results in 7,096 accumulators (2% of the amount used by the basic algorithm). Moreover, for this number of accumulators, we observe a small improvement on retrieval effectiveness (10.75%) against the original filtering algorithm (10.72%).

The distributed technique also yields larger savings in the number of term entries processed. Comparing Table 5.2 and Table 5.3, we observe that the distributed filtering technique gives a little greater retrieval effectiveness than the original one, while processing the same percentage of term entries.

Next, we show in details the retrieval effectiveness obtained using the distributed filtering technique in the distributed query processing algorithm, for both the local and global index partitioning strategies.

Local Index Partitioning

For the local index partitioning, we adopt $c_{ins} = 6 \times 10^{-3}$ and $c_{add} = 1.03 \times 10^{-3}$, that allows us to process only 10% of all term entries with a deterioration in the overall retrieval effectiveness of only 0.57%.

We are evaluating answer precision through the first 200 documents in the top of the ranking. Then, each processor needs to send to the broker at most the first 200 documents in the top of its ranking, in order of guaranteeing that final answer precision is not diminished. The values of recall versus precision for the local index partitioning are shown in Table 5.4. We note that the precision at levels of recall higher than 80% drops to 0 because not all relevant documents have been retrieved.

recall (%)	precision (%)
0	46.82
10	24.42
20	16.43
30	10.95
40	6.10
50	2.34
60	1.59
70	1.48
80	1.11
90	0.00
100	0.00
11-pt effectiveness (%)	10.11

Table 5.4: Recall versus interpolated precision for the local index partitioning.

Global Index Partitioning

For the global index partitioning we maintain the same value for the addition constant ($c_{add} = 1.03 \times 10^{-3}$), but reduce the value of the insertion constant to 5×10^{-3}. The reason is that each processor has no information on documents inserted in the local answer set of the others. This implies that in the global index partitioning the number of candidate documents must be larger than in the local index partitioning.

The fact that the local rankings consider only partial information related to its subquery also implies that more documents have to be sent to the broker. We adopted the cutoff factor given by $(c \times p \times r)$ (see Equation 2.3), fixing $c = 6$ and $r = 200$. The values of recall versus precision for the global index partitioning are shown in Table 5.5. As we can observe, there is no significant variation in the answer precision for different number of processors.

recall (%)	precision (%)			
	p=1	p=2	p=3	p=4
0	46.82	46.34	45.85	44.88
10	24.42	23.56	23.25	23.23
20	16.43	15.06	14.91	14.52
30	10.95	9.82	9.93	9.65
40	6.10	4.25	5.40	4.29
50	2.34	2.57	2.61	2.72
60	1.59	1.59	1.59	1.51
70	1.48	1.48	1.48	1.45
80	1.11	1.11	1.11	1.11
90	0.00	0.00	0.00	0.00
100	0.00	0.00	0.00	0.00
11-pt effectiveness (%)	10.11	9.61	9.65	9.40

Table 5.5: Recall versus interpolated precision for the global index partitioning.

5.4 Retrieval Efficiency

In this section, we first analyze the costs involved in the main phases of the algorithm. We show the analysis for the 50 TREC queries only, because it is analogous for the 2000

artificial queries.

Second, we compare overall query processing performance between the local index partitioning and the global index partitioning. We discuss the results for the 50 TREC queries, which are larger and force parallelism, and the results for the 2000 artificial queries, which are smaller and allow concurrency in our system. The metrics used are:

- Processing time;
- Speedup;
- Load imbalance.

5.4.1 Cost Analysis

We distinguish six main phases during distributed query processing:

1. Establishment of network connection and transfer of a query;
2. Disk seeking;
3. Reading of inverted lists from disk and accumulation of document weights;
4. Ranking of the local answer set;
5. Transference of the local answer sets;
6. Merging of the local answer sets.

The phases 2 to 4 are executed by the server processes while phase 6 involves only the broker process. The phases 1 and 5 represent the network communication cost among broker and servers for transferring the queries and the local answer sets.

The percentage of contribution of each phase, averaged by processor, to the time consumed in the processing of the 50 TREC queries is illustrated in Figure 5.3 and Figure 5.4, for the local and global index partitioning, respectively. As it can be seen, in the local index partitioning the disk seeking stands as the dominant cost. The reason is that each processor has to execute all terms in the query, which implies in performing as many seek operations as the number of terms in the query.

On the other hand, in the global index partitioning disk seeking time becomes proportionally smaller as the number of processors increases in the network. The reason is that

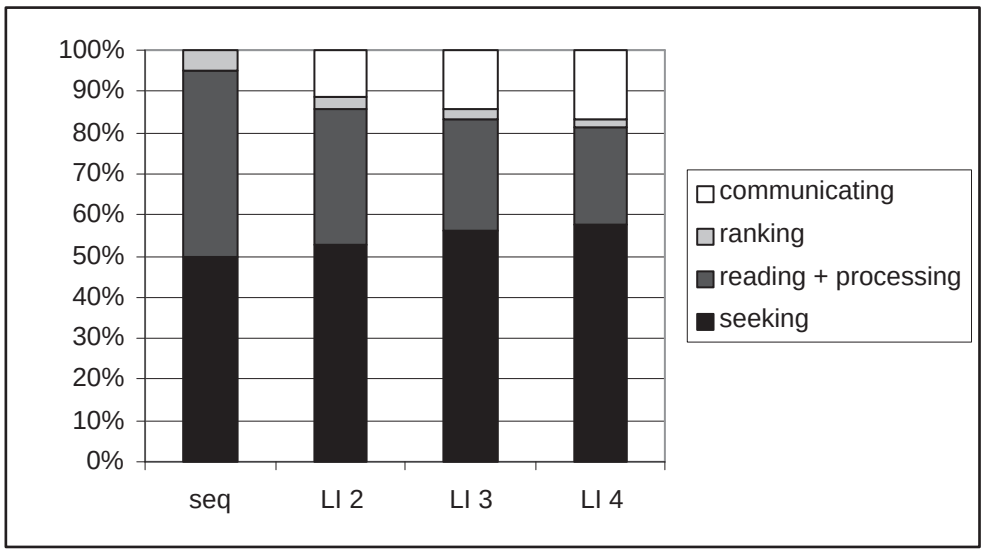

Figure 5.3: Percentage of contribution of the costs averaged by processor to execute the 50 TREC queries in LI.

each processor has to execute only the subquery relative to its local subcollection. So, as the number of processors exceeds the average number of terms in the query, each processor tends to process only a single term and consequently, perform only a single disk seeking operation.

However, in the global index partitioning network communication time becomes proportionally larger with the increase in the number of processors in the network, until turning out to be the dominant cost. Since each processor has no information on the documents in the local answer set of the others, it has to send to the broker a larger local answer set that grows with the number of processors. Thus, on the positive side disk seeking cost decreases to a minimum, because the number of disk seeking operations, while on the negative side network communication cost might increase considerably, because larger local answer sets have to be sent to the broker.

Therefore, the global index partitioning allows trading disk seeking to network communication. Depending on the size of the text collection, the size of the queries, the speed of the disk and network, such trading might become quite advantageous, as presented in Section 5.4.2.

For better understanding the reasons of the differences between the two index partitioning strategies, we detailed the costs involved in the main phases of the algorithm to

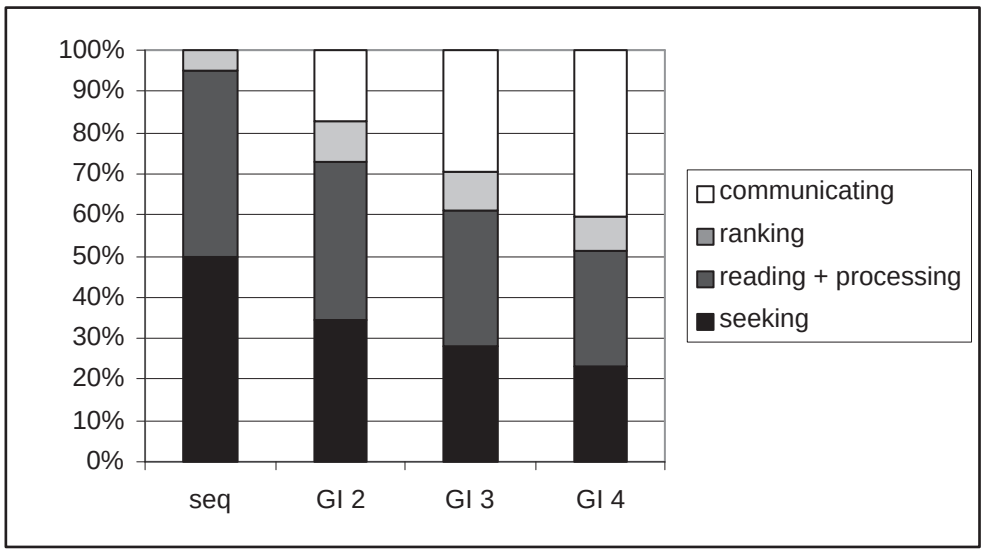

Figure 5.4: Percentage of contribution of the costs averaged by processor to execute the 50 TREC queries in GI.

execute the 50 TREC queries, as shown in Table 5.6 and Table 5.7 for the local and global index partitioning, respectively. The time consumed by each phase, averaged by processor, to execute the 50 TREC queries is illustrated in Figure 5.5. The cost analysis is as follows.

Disk Seeking

The sequential query processing system is composed by the following phases: i) disk seeking; ii) reading of inverted lists from disk and accumulation of document weights; and iii) ranking of the answer set.

For efficient parallelization of the system, it is important to find out the most time consuming phase of the algorithm. In the sequential system, it is disk seeking. For this reason, we believe that the parallelism of disk seeking operations is a crucial point for obtaining an efficient distributed system.

The global index partitioning requires only one I/O request per term, while the local index partitioning requires p requests, where p is the number of processors in the network. This implies that, in the global index partitioning, as the number of processors doubles, the average number of seeks per processor is divided in half (see Table 5.7). On the other hand, in the local index partitioning, the number of seeks per processor keeps the same as the number of processors increases (see Table 5.6).

Number of processors	Cost	Counts				
		P_1	P_2	P_3	P_4	Average
2	Seeks	2.126	2.126			2.126
	$< d, f >$ read	1.686.706	1.556.698			1.621.702
	Documents ranked	185.637	169.196			177.417
	$< d, w >$ transferred	10.000	10.000			10.000
3	Seeks	2.126	2.126	2.126		2.126
	$< d, f >$ read	1.203.331	911.459	1.117.988		1.077.593
	Documents ranked	138.273	95.845	120.715		118.278
	$< d, w >$ transferred	10.000	9.908	10.000		9.969
4	Seeks	2.126	2.126	2.126	2.126	2.126
	$< d, f >$ read	720.274	961.006	783.981	809.217	818.620
	Documents ranked	83.483	102.154	86.499	82.697	88.708
	$< d, w >$ transferred	9.740	10.000	10.000	10.000	9.935

Table 5.6: Cost by processor to execute the 50 TREC queries in LI.

As shown in Figure 5.5, for the local index partitioning, the time consumed by disk seeking decreases as the number of processors increases, although it should be kept the same. This apparently contradictory effect is due to the small size of the collection we used in comparison to the main memory available. Thus, if a term repeatedly occurs in the query set, then its inverted list is not read from disk again and again; on the contrary, the operating system keeps the inverted list in the memory cache, whose reading time is quite smaller.

Another justification is related to the inter-seek distance through the disk [HP90]. The seek time depends on the distance traversed by the disk head from the current to the next position, that is, the smaller the inter-seek distances, the smaller the time consumed by the seek operations. This implies that, as the size of the local inverted lists decreases with the increase in the number of processors in the local index partitioning, the inter-seek distance becomes smaller, which reduces the time consumed by the seek operations.

Both the memory cache and the inter-seek distance also affect the seek time in the global index partitioning. Nevertheless, they cause an stronger impact on the local index partitioning, where the inverted lists become smaller with the increase in the number of processors in the network.

Although the local index partitioning could be favored by the false effect on disk seek time, the global index partitioning still performs better in what concerns disk access.

Number of processors	Cost	Counts				
		P_1	P_2	P_3	P_4	Average
2	Seeks	1.424	702			1.603
	$< d, f >$ read	2.238.990	920.288			1.579.639
	Documents ranked	535.277	264.505			399.891
	$< d, w >$ transferred	118.208	105.586			111.897
3	Seeks	896	766	464		709
	$< d, f >$ read	1.467.274	1.074.933	617.071		1.053.093
	Documents ranked	323.256	322.539	180.572		275.456
	$< d, w >$ transferred	155.925	157.176	121.584		144.895
4	Seeks	740	684	254	448	532
	$< d, f >$ read	1.221.137	1.017.853	321.296	598.992	789.820
	Documents ranked	272.173	292.251	102.173	168.838	208.859
	$< d, w >$ transferred	171.325	188.090	84.756	137.889	145.515

Table 5.7: Cost by processor to execute the 50 TREC queries in GI.

Reading and Processing of Inverted Lists

The number of document-frequency pairs $< d, f >$ read from disk and averaged by processor is approximately equal for both the index partitioning strategies, as it can be seen at Table 5.6 and Table 5.7. The reason is that the distributed filtering technique equalizes the pruning of term entries processed from the distributed index, no matter how it is partitioned. Also, as the number of processors doubles, the average amount of document-frequency pairs read per processor is divided in half.

Therefore, regarding the reading and processing of inverted lists, both the local and global index partitioning present comparable performance.

Ranking of the Local Answer Set

The selection of candidate documents must be less severe in the global index partitioning, because each processor has no information on the documents inserted in the local answer set of the others. This implies that, in the global index partitioning, the local answer set is larger than in the local one. Averaged by processor, the number of documents ranked in the former is about 2 times larger than in the latter (see Table 5.6 and Table 5.7). Consequently, the cost related to the ranking of documents is higher in the global index partitioning than in the local one, as shown in Figure 5.5.

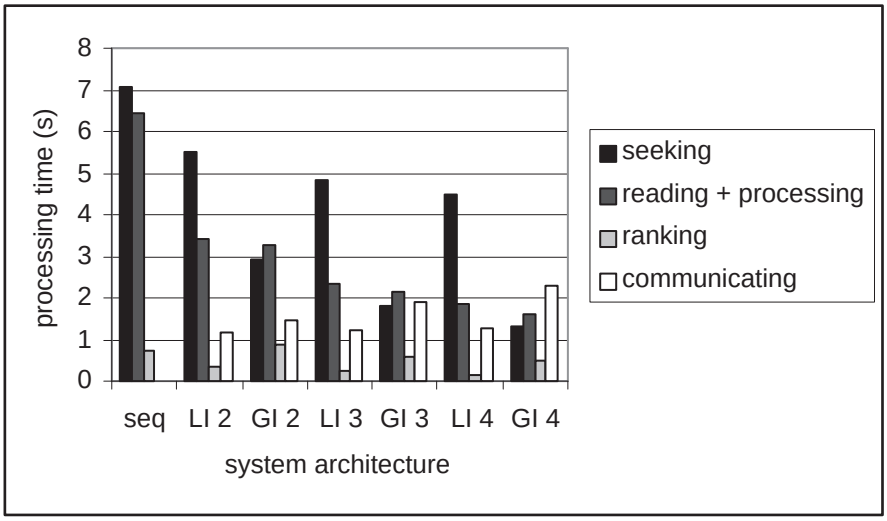

Figure 5.5: Cost averaged by processor to execute the 50 TREC queries.

Therefore, regarding the ranking of the local answer set, the local index partitioning performs better. As the cost of operation at CPU is not expensive in comparison to the disk access cost, the global index partitioning still remains the most advantageous, as we shall see at Section 5.4.2.

Network Communication

In our system, the communication between the broker and servers is based on a message passing mechanism. It comprehends the algorithm phases 1 and 5; the phase 1 is responsible for establishing network connection and transferring a query, and the phase 5 for transferring the local answer sets. We put forward the hypothesis that phase 5 is predominant in the communication time, based on the two following facts. First, phase 5 transfers a far larger amount of data than phase 1. Second, the time consumed in connection establishment by phase 1 is insignificant, which we confirmed by the implementation of a connectionless data transmission and subsequent observation that communication time was almost the same as the connection-based data transmission.

The hypothesis was confirmed through the empirical results, because the local index partitioning that transfers smaller local answer sets consumes less time in communication than the global index partitioning that transfers larger local answer sets (see Figure 5.5). The reason why the local index partitioning transfers smaller local answer sets that the global index partitioning is as follows.

In the local index partitioning, the final result is completed in a single processor for its local document set. In this way, if the broker intends to generate a final answer set composed by r documents, then it is necessary for each processor returning at most the r documents in the top of its ranking.

On the other hand, with the global index partitioning, the final results for any given document cannot be guaranteed to be completed in a single processor. This implies that each processor needs to return more than the r documents in the top of its ranking, if it is supposed for the broker to generate a final answer set composed by r documents. According to the cutting strategy we adopted, the size of the local answer set to be sent to the broker enlarges as the number of processors increases in the network. With 2 processors in the network, the number of documents averaged by processor to be returned to the broker is 11 times larger in the global index partitioning than in the local one; with 4 processors, it is 14 times larger (see Table 5.6 and Table 5.7).

Therefore, regarding the communication between broker and servers, the local index partitioning performs better. However, the positive counter effects still favored the global index partitioning, as we shall see at Section 5.4.2.

Merging of the Local Answer Sets

Figure 5.6 compares the 50 TREC queries total processing time with the merging time at the broker, for the local and global index partitioning. In the local index partitioning, the broker uses a simple multiway merge to fuse the ranked local answer sets and produce the final ranked answer set. On the other hand, in the global index partitioning, the merging is more complicated. The reason is that, in the global partitioning strategy, the broker cannot use the rankings generated by the processors, because such rankings contain only the partial similarities between each document and each term present in the subqueries. Then, it is necessary to sum the partial similarities into the total similarity for documents present in different local answer sets and finally, do a sort to produce the final ranked answer set.

Therefore, regarding the merging of the local answer sets at the broker, the local index partitioning performs better. However, this negative effect did not disfavor the global index partitioning, because the total merging time was much smaller than the processing time consumed by the slowest processor.

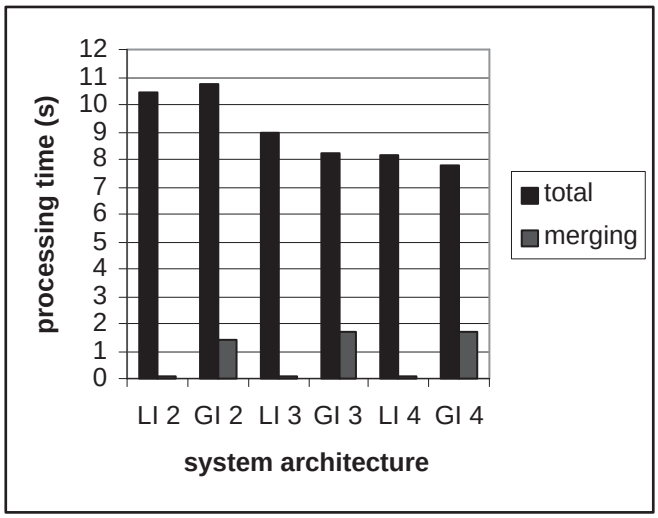

Figure 5.6: Cost of merging at the broker for the 50 TREC queries.

Summary of the Cost Analysis

In summary, for the 50 TREC queries, both the local index partitioning and the global index partitioning are compared in the following aspects of cost:

1. Disk Seeking: In the global index partitioning, as the number of processors doubles, the average number of seeks per processor is divided in half. On the other hand, in the local index partitioning, it is kept the same. As disk seeking is the most time consuming phase of the algorithm, it is the dominant effect to make the global index partitioning the best.

2. Reading and Processing of Inverted Lists: The average amount of document-frequency pairs read per processor is approximated in both index partitioning strategies. Therefore, regarding the reading and processing of inverted lists, both index partitioning strategies present comparable performance.

3. Ranking of the Local Answer Set: Averaged by processor, the number of documents ranked in the global index partitioning is about 2 times larger than in the local index partitioning. Therefore, regarding the ranking of the local answer set, the local index partitioning performs better. As the cost of operation at CPU is not expensive in comparison to the disk access cost, the global index partitioning still remains the most advantageous.

4. Network Communication: With 4 processors in the network, the number of documents averaged by processor to be returned to the broker is 14 times larger in the global index partitioning than in the local one. Therefore, regarding network communication, the local index partitioning performs better. However, the positive counter effects still favored the global index partitioning.

5. Merging of the Local Answer Sets: In the local index partitioning, the broker uses a multiway merge to fuse the ranked local answer sets, which cannot be done in the global index partitioning, because the local rankings generated with this latter index organization is based only in partial information present in the subqueries. Therefore, regarding the merging of the local answer sets at the broker, the local index partitioning performs better. However, this negative effect did not disfavored the global index partitioning, because the total merging time was much smaller than the processing time consumed by the slowest processor.

Therefore, in the local index partitioning the disk seeking stands as the dominant cost. In the global index partitioning, there is a trading of disk seeking to network communication that might be very advantageous, depending on the size of the text collection, the size of the queries, the speed of the disk and network.

5.4.2 Overall Query Processing Performance

In this section, we compare query processing performance between the local and global index partitioning strategies using both the TREC query set and the artificial query set. The TREC query set is larger - 21 terms per query on average -, which forces a parallel query service. In a different manner, the artificial query set is much smaller - 2 terms per query on average -, which allows a concurrent query service. It follows the results and the corresponding interpretations.

TREC Queries

Figure 5.7 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.

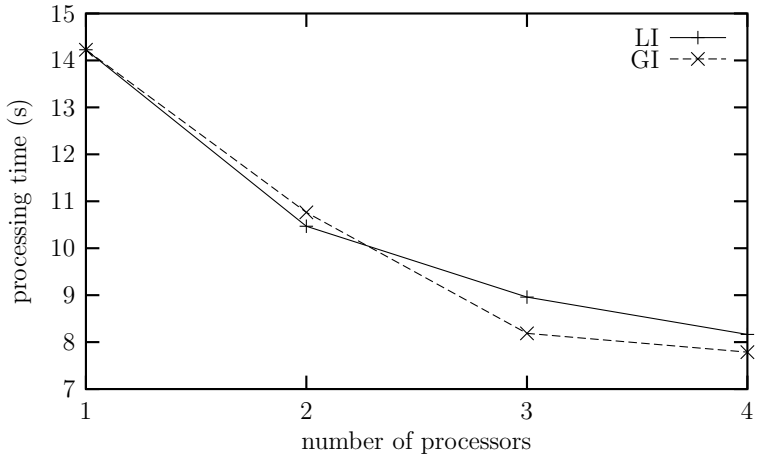

Figure 5.7: Processing time for the 50 TREC queries.

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

Figure 5.8 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 large size of TREC queries.

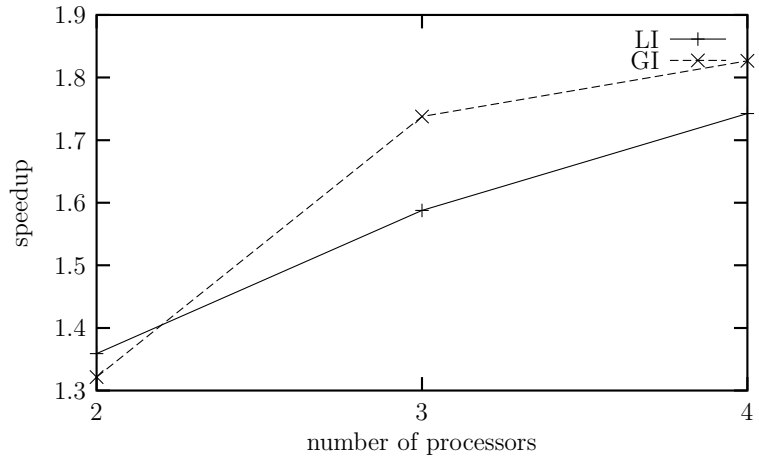

Figure 5.8: Speedup for the 50 TREC queries.

Figure 5.9 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.

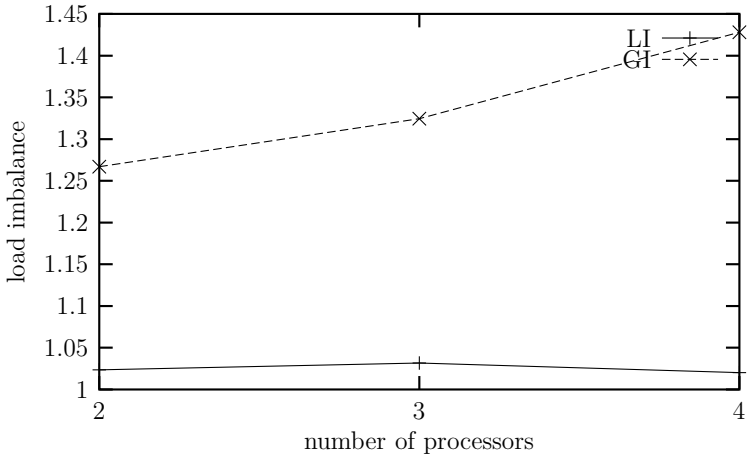

Figure 5.9: Load imbalance for the 50 TREC queries.

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, then 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, then 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 5.10 and Figure 5.11 that show the processing time and speedup, respectively. Also, the relative performance improvement would increase with the number of processors, as shown in Table 5.8. 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.

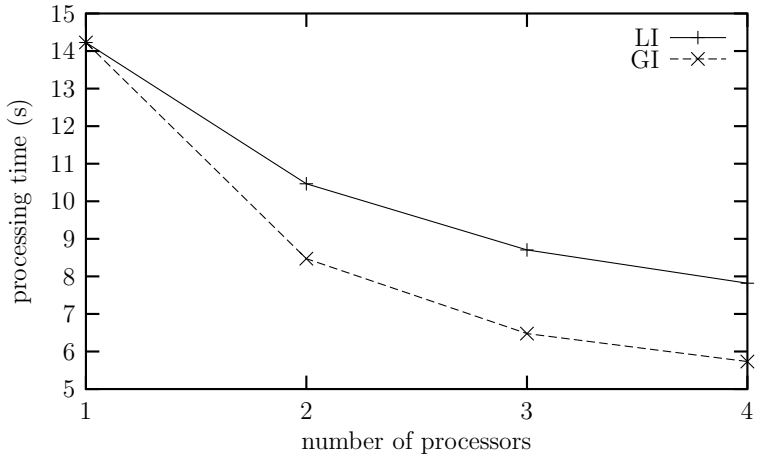

Figure 5.10: Processing time in the load balanced scenario for the 50 TREC queries.

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

Table 5.8: Processing time in the load balanced scenario for the 50 TREC queries: GI as percentage of LI.

Artificial Queries

Figure 5.12 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 5.9. As it can be seen, the global index partitioning might be twice as faster than the local index partitioning. The reason is as follows.

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

Table 5.9: Processing time for the 2000 artificial queries: GI as percentage of LI.

In the local index partitioning, all the processors are forced to process the 2 terms (on

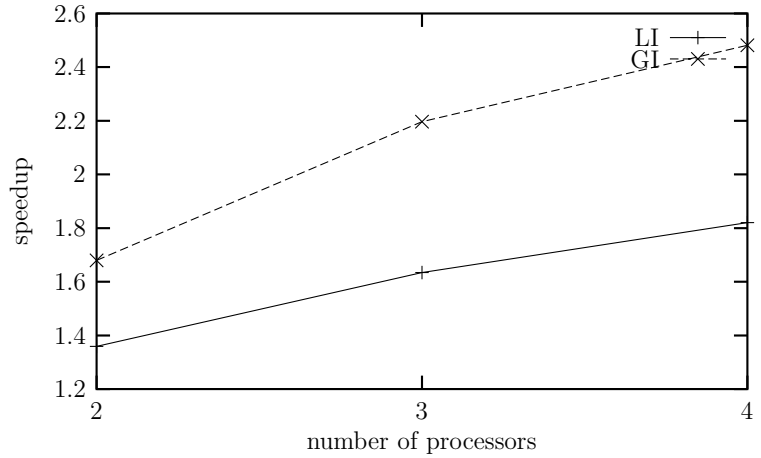

Figure 5.11: Speedup in the load balanced scenario for the 50 TREC queries.

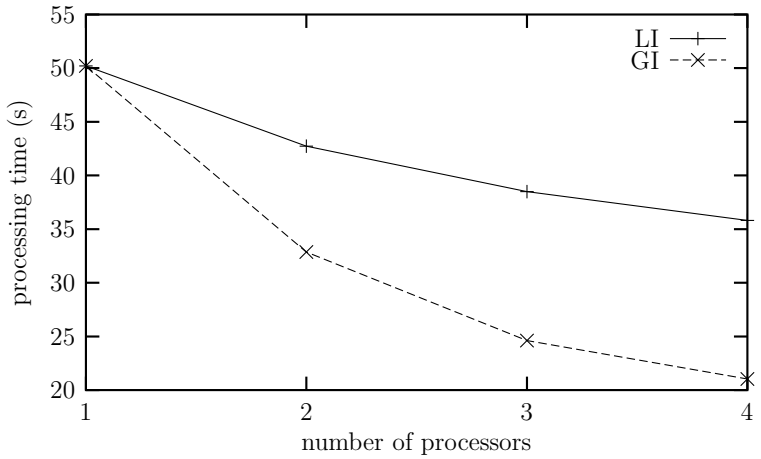

Figure 5.12: Processing time for the 2000 artificial queries.

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 5.13 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 5.14 shows the load imbalance while processing the 2000 artificial queries. For

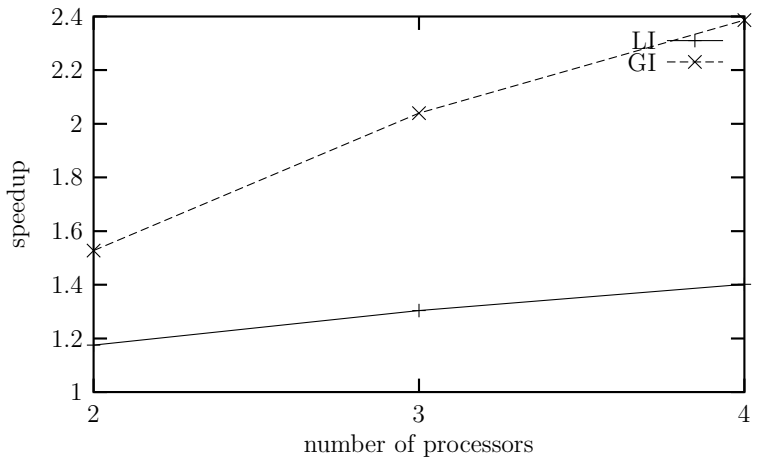

Figure 5.13: Speedup for the 2000 artificial queries.

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.

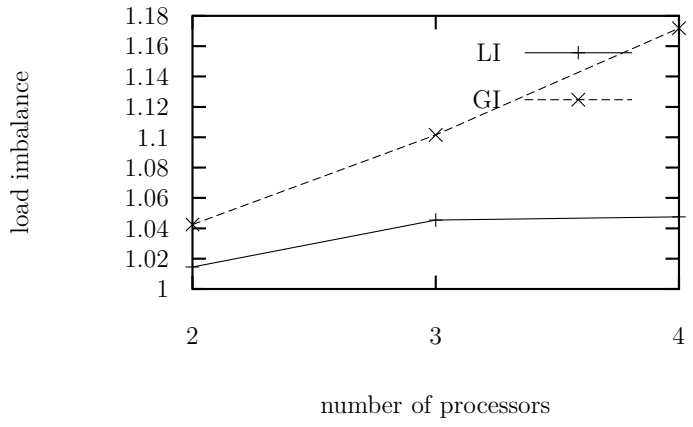

Figure 5.14: Load imbalance for the 2000 artificial queries.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work, we study query processing in a distributed text database. We have implemented a real distributed architecture supporting concurrent query service and compared the impact of two different types of inverted file partitions on system performance. Documents are retrieved and ranked using the vector space model along with a document filtering technique, that allows significant reduction in ranking costs without degradation in retrieval effectiveness.

Experimental results on retrieval effectiveness show that both the local index partitioning and the global index partitioning provide approximately the same effectiveness as the sequential algorithm. Further, results show that the queries are processed in only 2% of the memory of the basic algorithm for ranking and that only 10% of all term entries in the inverted lists are required, which reduces disk traffic and CPU processing time.

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 a query, as follows. First, the processing time with the global index partitioning might be twice smaller as that with the local index partitioning. Second, the speedup in the global index partitioning might be 1.7 times as that in 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 a query.

6.2 Future Work

6.2.1 Two Types of Brokers

In this work, there is only one broker responsible for scheduling queries to the different servers and merging intermediate results into final results. In future work, we are interested in implementing two types of brokers, one for query scheduling and another for merging of intermediate results, in order of relieving the bottleneck in the merging task. Both types of broker has different constraints. The scheduling broker can reside in a simpler processor, because the computational load for scheduling queries to servers is not very high. On the other hand, the merging broker requires a more complex processor, because of the much more higher workload for merging intermediate results of the various queries.

We also want to investigate the possibility of sending part of the work of the merging broker to the other processors in the network. The distribution of the merging task with other processors should be a decision made dynamically, only when the workload is so high that the scheduling broker is not able by itself to accomplish it on time.

6.2.2 Multiprogramming in the Server

In this work, the queries are processed in the servers once at a time, that is, we consider only one query per server at the same time. In future work, in order of increasing system throughput and decreasing response time, we intend to make use of multiprogramming in the server and evaluate the system performance while varying the multiprogramming level (number of simultaneous queries per server). Also, we want to compare the effect of multiprogramming level in various operating systems.

6.2.3 Minimization of Network Traffic

As the number of processors increases in the network, the number of intermediate results to be sent to the broker enlarges and consequently, the network traffic becomes higher. In future work, in order of minimizing network traffic, we are interested in investigating alternatives for diminishing the amount of information sent to the broker. One alternative would be the transfer of compressed data. Another alternative would the fusion of intermediate results between processors, which might generate small answer sets to be sent to the broker.

6.2.4 Performance Evaluation with Web Data

Other future direction of research is to evaluate the behavior of our system while processing Web data, that comprises very large collections and very short queries. We suspect that the advantages of the global index partitioning over the local index partitioning might decrease as the size of the collection increases. In other words, for larger collections, the time consumed by disk seeking becomes proportionally small in comparison to the time consumed by reading inverted lists, communicating through the network and merging at the broker. However, for very small queries, the global index partitioning might allow high concurrency, which is not present with the local index partitioning. In light of these facts, it is important to study the relation between the growth of the size of the collection and the amount of bytes read and transferred to the broker. Moreover, we intend to model queries using a Zipf-like term distribution [BYC00] and to specify a query arrival distribution, in order of simulating Web data and workload respectively.

6.2.5 New Strategies for the Global Index Partitioning

In this work, for the global index partitioning, the inverted lists are evenly distributed by size between processors. In future work, we intend to study new strategies to generate the global index by exploiting usage statistics and other measures, in order of achieving better speedup and load balance. First, we want to evaluate separately the different strategies, in order of quantifying the gain obtained with each of them and subsequently, determining which are really good. Second, we want to investigate the possibility of combining the good strategies into an optimal global index partitioning strategy. As a strategy is designed to benefit a specific metric, such as speedup and load balance, the combination might disfavor some of them. So, it is necessary to identify the tradeoffs over the range of partitioning strategies.

Besides of distributing the inverted lists in subsets Q_{p_i}, each one held by the processor p_i, we also aim to minimize the difference between the minimum and average size S_{p_i} of Q_{p_i} and consequently, minimizing the lost of disk space. The disk lost can be measured by the ratio between the minimum space and the average space occupied in the different disks. Figure 6.1 illustrates the global index partitioning, where V_I is the size of the index vocabulary and V_Q is the size of the query vocabulary. Next, we describe suggestions of global index partitioning strategies and the corresponding metrics each one favors.

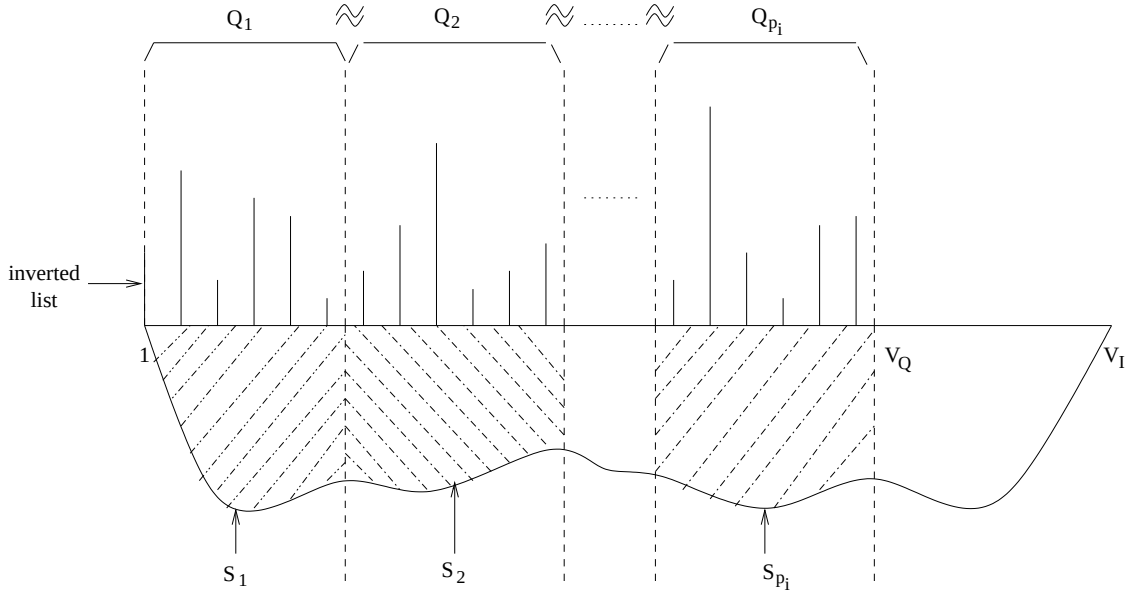

Figure 6.1: Global index partitioning.

Global Index Partitioning by Frequency or Time

In the global index partitioning by size, if a processor receives more query terms than others, then load balance will not be uniform. We suppose that if the distribution of the global inverted lists among processors follows the distribution of terms in a query, then we will obtain a better load balance. Besides the frequency of terms in a query, we believe that load balance is also related to the term processing time, which is a function of its inverted list size. Therefore, in order of optimizing load balance, we suggest two new strategies for distributing inverted lists among processors, namely global index partitioning by frequency and global index partitioning by time.

In the global index partitioning by frequency, each processor p_i holds the inverted list set Q_{p_i} composed of t_{p_i} terms. The sum of the frequency in a query of the terms in Q_{p_1}, held by processor p_1, is approximately equal to the sum of the frequency in a query of the terms in Q_{p_2}, held by processor p_2, which is approximately equal to the sum of the frequency in a query of the terms in Q_{p_i}, held by processor p_i.

In the global index partitioning by time, each processor p_i holds the inverted list set Q_{p_i} composed of t_{p_i} terms. The sum of processing time of the terms in Q_{p_1}, held by processor p_1, is approximately equal to the sum of processing time of terms in Q_{p_2}, held by processor p_2, which is approximately equal to the sum of processing time of the terms in Q_{p_i}, held

by processor p_i.

Global Index Partitioning by Co-occurrence

In the global index partitioning, the best scenario is when a single processor holds the inverted lists for all the terms in a query, which enables that processor to execute the query by itself without need to cooperate with any other processor. In this way, the neighbor processors are released to execute other queries, which allows a higher concurrent query service and consequently a better speedup. Moreover, in this best scenario, only one local answer set is generated for a query, which reduces network traffic and computing time in the merging broker. Therefore, in order of improving speedup in our system, besides network traffic and computing in the merging broker, we suggest the global index partitioning by co-occurrence, by which the terms that co-occur in the same query are stored in the same processor.

In the global index partitioning by co-occurrence, each processor p_i holds the inverted list set Q_{p_i} composed by different pairs of terms $\{t_x, t_y\}$ that co-occur in the same query. We are considering the Web query pattern in which queries are composed by two terms on average, which explains the clustering of terms in pairs.

6.2.6 Global Index in Two Levels

The global index can be structured in two levels[1]:

- Index for the most frequent queries (I_S);
- Index for the remaining of the queries (I_U).

Let V_I be the size of the index vocabulary and V_Q the size of the query vocabulary. The size of I_S is given by $V_{Q'}$, $V_{Q'} < V_Q$, and the size of I_U is given by $V_I - V_{Q'}$. The value of $V_{Q'}$ must be optimum in function of query processing time. Figure 6.2 illustrates the global index in two levels.

I_S is usually smaller and can be stored in main memory; I_U is usually bigger and must be stored in secondary memory. We believe that such scheme of storage favors query processing performance, because I_S, whose terms are accessed more frequently, is stored in main memory, whose reading time is smaller.

[1]To the best of our knowledge, this seems to be the index organization adopted by the Infoseek search engine.

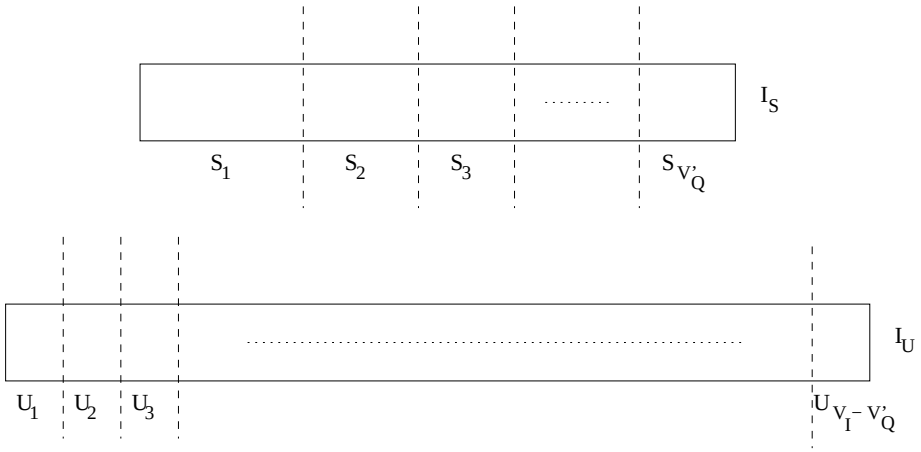

Figure 6.2: Global index in two levels.

In future work, we are interested in verifying if such assumptions are correct.

6.2.7 Caching of Query Results and Inverted Lists

The work in [SMZ+ar] describes two caching schemes that reduces computing and I/O requirements to support the functionality of a Web search engine. Their strategy for caching query results is to keep in memory the list of documents associated with a given query, and for caching inverted lists is to keep in memory the list of documents associated with a given query term. In future work, we intend to study how the caching of query results and inverted lists can improve the performance of our system or favor one of the index partitioning strategies.

Bibliography

[ACPtNT95] Thomas E. Anderson, David E. Culler, David A. Patterson, and the Now Team. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[ANZ97] Marcio Drummond Araujo, Gonzalo Navarro, and Nivio Ziviani. Large text searching allowing erros. In Ricardo Baeza-Yates, editor, *Proceedings of the Fourth South American Workshop on String Processing*, pages 2–20, Valparaiso, Chile, November 1997. Carleton University Press.

[Bar98] Ramurti 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.

[BYC00] Ricardo Baeza-Yates and Carlos Castillo. Relating Web characteristics. Technical report, Computer Science Department, University of Chile, October 2000. http://www.todocl.cl/stats/rbaeza.pdf.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto, editors. *Modern Information Retrieval*. ACM Press New York, Addison Wesley, 1999.

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval - Data Structures and Algorithms*. Prentice Hall, 1992.

[Har94] Donna Harman. Overview of the third text retrieval conference. In Donna 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.

[HCT98] D. Hawking, N. Craswell, and P. Thistlewaite. Overview of TREC-7 very large collection track. In E.M. Voorhess and D.K.Harman, editors, *Proceed-*

ings of the Seventh Text Retrieval Conference, pages 257–268, Gaithersburg, Maryland, U.S.A., November 1998. NIST Special Publication 500-242.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 2ª edition, 1990.

[MMR00] A. MacFarlane, J.A. McCann, and S.E. 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.

[Per94] Michael Persin. Document filtering for fast ranking. In *Proceedings of the 17th ACM SIGIR Conference*, pages 339–348, Dublin, Ireland, July 1994. ACM/Springer.

[PZSD96] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[RNB98] Berthier A. Ribeiro-Neto and Ramurti A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM Conference on Digital Libraries*, pages 182–190, 1998.

[SHMM99] C. Silverstein, M. Henzinger, H. Marais, and M. Moriez. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, Fall 1999.

[SMZ+ar] Patricia Correa Saraiva, Edleno Silva Moura, Nivio Ziviani, Rodrigo Fonseca, Wagner Meira, Cristina Murta, and Berthier 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).

[TGM93] Anthony Tomasic and Hector 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.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes - Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Inc., 2ª edition, 1999.

[Ziv93] Nivio Ziviani. *Projeto de Algoritmos Com Implementações em Pascal e C.* Livraria Pioneira Editora, 2ª edition, 1993.