PARALLEL TEXT RETRIEVAL ON PC CLUSTERS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Aytül Çatal

September, 2003

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Cevdet Aykanat(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Uğur Doğrusöz

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

PARALLEL TEXT RETRIEVAL ON PC CLUSTERS

Aytül Çatal
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2003

The inverted index partitioning problem is investigated for parallel text retrieval systems. The objective is to perform efficient query processing on an inverted index distributed across a PC cluster. Alternative strategies are considered and evaluated for inverted index partitioning, where index entries are distributed according to their document-ids or term-ids. The performance of both partitioning schemes depend on the total number of disk accesses and the total volume of communication in the system. In document-id partitioning, the total volume of communication is naturally minimum, whereas the total number of disk accesses may be larger compared to term-id partitioning. On the other hand, in term-id partitioning the total number of disk accesses is already equivalent to the lower bound achieved by the sequential algorithm, albeit the total communication volume may be quite large. The studies done so far perform these partitioning schemes in a round-robin fashion and compare the performance of them by simulation. In this work, a parallel text retrieval system is designed and implemented on a PC cluster. We adopted hypergraph-theoretical partitioning models and carried out performance comparison of round-robin and hypergraph-theoretical partitioning schemes on our parallel text retrieval system. We also designed and implemented a query interface and a user interface of our system.

Keywords: Parallel text retrieval, inverted index, parallel query processing, inverted index partitioning, system performance.

ÖZET

PC KÜMELERİ ÜZERİNDE PARALEL METİN ERİŞİMİ

Aytül Çatal
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2003

Ters dizin bölümleme problemi paralel metin erişim sistemleri için araştırıldı. Hedef, bir PC kümesi üzerine dağıtılmış ters dizin üzerinde hızlı ve verimli sorgulamayı başarmaktır. Dizin kayıtlarının belge numaraları veya kelime numaralarına göre dağıtıldığı ters dizin bölümlemesi için alternatif stratejiler düşünülmüş ve değerlendirilmiştir. Her iki bölümleme planının performansı toplam disk erişim sayısına ve sistemdeki toplam iletişim hacmine bağlıdır. Belge numarası bazlı bölümlemede, toplam disk erişim sayısı kelime numarası bazlı bölümleme ile kıyaslandığında daha büyük olabilirken, toplam iletişim hacmi doğal olarak en az miktardadır. Diger bir taraftan, kelime numarası bazlı bölümlemede, toplam iletişim hacmi oldukça büyük olabilse de, toplam disk erişim sayısı seri algoritma tarafından ulaşılan alt sınıra zaten eşittir. Şu ana kadar yapılmış çalışmalar, bu bölümleme planlarını sıralı bir biçimde icra etmektedirler ve performanslarını simulasyonla karşılaştırmaktadırlar. Bu çalışmada, paralel metin erişim sistemi bir PC kümesi üzerinde tasarlandı ve programlanması gerçekleştirildi. Hiperçizge kuramsal bölümleme modellerini seçtik ve sıralı ve hiperçizge kuramsal bölümleme planlarının performans karşılaştırmasını paralel metin erişim sistemimiz üzerinde gerçekleştirdik. Bundan başka, sistemimizin sorgulama arayüzünü ve kullanıcı arayüzünü tasarladık ve programlanmasını gerçekleştirdik.

Anahtar sözcükler: Paralel metin erişimi, ters dizin, paralel sorgulama, ters dizin bölümlemesi, sistem performansı.

Acknowledgement

I would like express my gratitude to my supervisor Prof. Dr. Cevdet Aykanat for his trust, invaluable guidance and help for my thesis.

Special thanks go to Berkant Barla Cambazoğlu. Throughout my thesis, he has always been very helpful to me. His invaluable ideas, suggestions and help have been essential to my thesis. I appreciate all the time that he has spent for the development of my thesis.

I also would like to thank Prof. Dr. Cevdet Aykanat , Prof. Dr. Özgür Ulusoy, Assist. Prof. Dr. Uğur Doğrusöz and Berkant Barla Cambazoğlu for taking their time for reading my thesis and commenting on it.

I thank my housemate Sultan Erdoğan for her invaluable friendship and support. I would like to express my thanks to all of my friends for making life enjoyable.

My parents, my sister and my brother have always been on my side. I love them very much and I would like to express my gratitude to them for their endless love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In traditional text retrieval systems, terms are used to index and retrieve documents. An index is a structure that is common to all text retrieval systems, and in general form, it identifies for each term a list of documents that the term appears in. The users formulate their information needs through the queries, which are basically composed of terms and submit their queries to the system. For each submitted user query, the text retrieval system retrieves the documents that are relevant to the query, rank them according to the degree of similarity to the query, and returns them to the user for presentation.

In recent years, the internet has become very popular being an indispensable resource for information. The number of the internet users increases, as the access to the internet is getting easier and cheaper. The growing use of the internet has a significant influence and importance on text retrieval systems. The size of the text collection available online is growing at an astonishing rate. At the same time, the number of users and the queries submitted to the text retrieval systems are also increasing very rapidly [17, 1]. The staggering increase in the data volume and query processing load create new challenges for text retrieval research.

In order to evaluate text retrieval systems, two basic criteria are used: Effectiveness and efficiency. Effectiveness is commonly measured in terms of precision and recall [8]. Precision is the quality of the documents presented to the user,

that is, how many of the retrieved documents are relevant. Recall is the measure of how many relevant documents are retrieved over the whole collection. On the other hand, efficiency measures how fast the results are obtained. This may be computed using the standard empirical statistics measures such as the response time and the throughput. The throughput refers to the number of queries answered in a specific unit of time. So far, most research in text retrieval area has centered around the effectiveness. However, most users have been satisfactory with the accuracy of text retrieval systems, whereas they have become in favor of the systems that respond in a short time [9]. In recent years, in order to increase the efficiency of text retrieval systems, various attempts have been made to introduce parallelism to the text retrieval systems [20]. In this thesis, our main focus is on the inverted index organizations for efficient query processing in parallel text retrieval systems.

For efficient query processing, an indexing mechanism has to be used in text retrieval systems. There exists different indexing techniques in the literature. Some important ones are suffix arrays, signature files and inverted indices [28]. Each of them have their own strong and weak points. Until the early 90's signature files and suffix arrays were very popular, however along the years inverted indices have been traditionally the most popular indexing technique due to its simplicity, robustness and good performance. Therefore, in this work, we consider inverted indices as our indexing mechanism.

In parallel systems, in order to index the collection using inverted indices, a strategy on the distribution of the inverted indices has to be followed. The works in [27, 11, 18] describe two basic partition strategies to organize the index. In the first partitioning strategy, an inverted index is generated for the whole collection and distributed among the processors according to the term-ids. In the second one, distribution of the inverted index among the processors is performed based on the document-ids (Ids are associated with the terms and the documents for identification).

In query processing, many models have been proposed to determine the relevance of the documents to the terms of the query. Among these, the vector-space

model is the most widely accepted model [28, 5], as its performance is superior or almost as good as the known alternatives. In this work, we employed the vector-space ranking model with cosine similarity measure by using *tf-idf* (term frequency-inverse document frequency) weighting metric, which is one of the well-known metrics that gives good retrieval effectiveness [28, 29].

In this thesis, we have designed and implemented a parallel text retrieval system. For efficient query processing, we have worked on different inverted index organizations. We have investigated how these index organizations affect the system by determining the critical parameters that these organizations depend on. Furthermore, in our implementation, we have adapted the data structures that are efficient for the storage and time requirements of our text retrieval system. We have also considered the effectiveness of our system by choosing the vector-space model as our retrieving and ranking method for the documents in the collection of the system.

The rest of the thesis is as follows. Chapter 2 briefly presents sequential text retrieval systems. Chapter 3 overviews parallel text retrieval systems by giving the related work on the inverted index partitioning and our objective in this study. Chapter 4 describes the implementation in detail. Chapter 5 gives the experimental results. Finally, we conclude and point at some future work.

Chapter 2

Sequential Text Retrieval

2.1 Indexing

2.1.1 Index Structure

A naive way to search a query on a set of documents is to scan the whole text sequentially. This option is applicable for small document collections. However, when the document collection is large, it is advisable to build an index to speed up the search. Indexing is one of the most important parts for the process of making the collection efficiently searchable. There are three main indexing techniques: suffix arrays, signature files and inverted indices. We emphasize on inverted indices. Suffix arrays and signature files were popular until the early 90's to index the collections. However, nowadays inverted indices outperform them, and have become the best choice among indexing techniques [28]. Many commercial and academic text retrieval systems use inverted indices [2]. For instance, many web search engines and journal archives use them.

Suffix trees are an indexing mechanism, which treats the text as a one long string. Each position in the whole collection is considered as a suffix of the collection. That is, the string starting from that position to the end of the

collection is identified as a suffix. So, two suffixes starting at different positions are lexicographically distinct. It is important to note that not all the positions in the collection need to be indexed. Therefore, in the collection index points are determined such that only retrievable suffixes are indexed [2].

Signature files are a word-oriented method for indexing documents, which means that the whole collection is taken as a sequence of words. A hash function is used to map every term of the document, accordingly each document is associated a *signature*, where the bits of the signature corresponding to those hash values are set to one [2, 6].

An inverted index is typically composed of two elements: an index for each term in the lexicon (vocabulary), where the set of distinct words in the whole collection is referred as collection vocabulary and an inverted list for each index. An inverted list entry is known as a posting and keeps a *document-id, weight* pair. The index entry of a term is composed of the id of the term and a pointer to the start of the inverted list of the term [2].

In general, an inverted index structure is based on a word-oriented mechanism to index a collection. This assumption limits the types of queries to be answered to some extent, for instance phrase search becomes costly to perform. Suffix trees are efficient for phrase search. However, suffix trees have a high space requirement. Suffix arrays are implemented to reduce space requirements of suffix trees. The common shortcoming of suffix trees and suffix arrays is their costly construction process. The construction of both signature files and inverted indices is rather easy. On the other hand, signature files have a high search complexity compared to other techniques. Therefore, this technique is not preferred for very large texts.

Each of these indexing methods have their own strong and weak points. Generally, in applications where the queries are based on words and when the size of the collection is large, inverted index outperforms other techniques considerably. Also, due to its simplicity and good performance, inverted index mechanism has been the best choice of indexing techniques along the years [2].

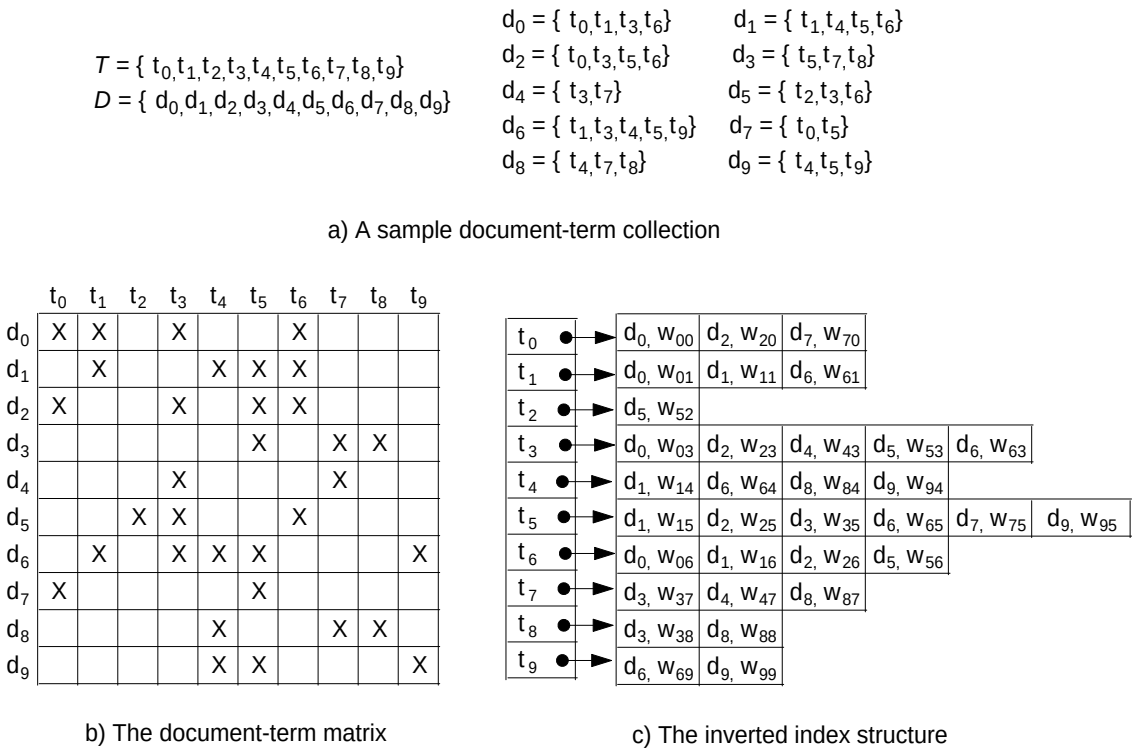


Figure 2.1: A sample collection.

Figure 2.1-a shows our sample document-term collection, which we will use to describe our models and other inverted index models. The document set and term set of the sample collection are called D and T, respectively. There are 10 documents, 10 terms and 33 posting entries in the collection. We use P to denote the posting set. Figure 2.1-b shows the document-term matrix representation of our collection. This is a sparse matrix, as documents do not include most of the terms. Along with these, Figure 2.1-c demonstrates the inverted index structure of our collection.

In general, as the collection grows larger, inverted lists reach to a size that cannot be stored in main memory. The index part is usually small to fit into main memory, and inverted lists are stored on the disk [28].

2.1.2 Stop word elimination, case folding and stemming

In order to improve the effectiveness of the indexing techniques, there are three important mechanisms that are widely used: Stop word elimination, case folding and stemming. A stop word list is a list of most frequently used words of the language such as “the”, “a”, “an”, “and” and etc... These words are eliminated from the index. It is very advantageous to use a stop word list. Since this kind of words appear in almost every document, their inverted lists are very long. Therefore indexing of these common words increases storage cost, besides that retrieving the postings of their inverted lists raises the search time considerably. Furthermore, as they are common in many documents, indexing them does not improve effectiveness. Consequently, most text retrieval systems eliminate stop words before indexing.

The other process is case folding, which is simply replacing all uppercase letters of a word with lowercase equivalents. For example, all combinations of a word such as “mpi”, “MPI”, “Mpi” will be indexed and searched as “mpi”. This process also makes the search easier and faster, and most of the users do not differentiate between case sensitive and case insensitive queries. Also, it reduces the indexing structure size by decreasing the number of distinct terms.

Stemming is reducing the word to its grammatical root by stripping one or more suffixes off the word. For example, the word “stem” is the stem for the variants stemmed, stemming and stems. Stemming is accepted as a factor that enhances the retrieval performance, because it lessens the variants of a root word to a common concept. Furthermore, it decreases the size of the indexing structure as the number of distinct terms is reduced.

From these three mechanisms, we employed only stop word elimination and case folding. Implementation of stemming process requires a detailed knowledge of the language in question and a great deal of effort. There are many exceptions of the rules of a language, and also one finds exceptions to exceptions and so on. A stemmer used as an example in [28] is given with more than five hundreds rules and exceptions. Therefore, we preferred not to incorporate stemming into

our indexing mechanism.

2.2 Query Processing

In this section, we will examine two types of queries: Boolean and ranked queries. Also, we will discuss shortly how to process them.

The oldest way to build a query is combining the terms with Boolean operators like AND, OR and NOT. For example, consider the following query: (*text OR data OR image*) *AND retrieval*, where the parenthesis indicate operation order. This query returns the documents including the phrases *text retrieval*, *data retrieval* and *image retrieval*. Note that, the words in the phrases need not be adjacent, nor appear in any particular order.

With the classical Boolean text retrieval systems, ranking of the retrieved documents is normally not provided. A document either matches the Boolean query or not. Additionally, obtaining relevant results is not a matter of how the query is constructed with Boolean operators. Because, connecting the query terms with the AND operator would cause many documents, which are likely to be relevant, not to appear in the result set. Using OR connectives would be ineffective, since too many documents will match and very few of them are likely to be relevant to the query.

The problems based on Boolean queries are solved with ranked queries. In order to rank the queries, different methods are used in text retrieval systems. Some of them are the vector-space model, probabilistic models, fuzzy-set models and neural network models. Among them, the most popular one is the vector-space model due to its performance and simplicity. For further information about other models, one can check [2].

In the vector-space model, the *degree of the similarity* between the query and each document in the collection is calculated. The relevance of the documents matching the query is determined by sorting the retrieved documents of their

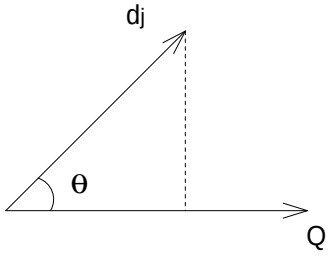


Figure 2.2: The cosine of θ is adopted as *sim(d_j, q)*.

degree of similarity in decreasing order. In the vector-space model, both the documents and the queries in the collection are represented as T dimensional vectors as shown in Figure 2.2. T is the total number of index terms in the collection. The vector-space model measures the degree of the similarity of the document d_j with respect to the query q by calculating the correlation between the vectors $\overrightarrow{d_j}$ and $\overrightarrow{q}$. This correlation is given by the *cosine similarity measure* [21], which is shown in Equation 2.1. In the equation, $\|\overrightarrow{q}\|$ and $\|\overrightarrow{d_j}\|$ are the norms of the query and the document vectors respectively, and $\|.\|$ denotes the inner product operation. Since $\|\overrightarrow{q}\|$ is the same for all the documents, it does not affect the ranking, while $\|\overrightarrow{d_j}\|$ provides a normalization in the space of documents [2].

$$sim(\overrightarrow{q}, \overrightarrow{d_j}) = \frac{\overrightarrow{q} \cdot \overrightarrow{d_j}}{\|\overrightarrow{q}\| \times \|\overrightarrow{d_j}\|} = \frac{\sum_{i=1}^{T} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{T} w_{i,j}^2 \times \sum_{i=1}^{T} w_{i,q}^2}} \tag{2.1}$$

Index term weight $w_{i,j}$, which is the weight of term t_i in a particular document d_j, can be calculated in several ways [19]. Here, we only mention the most effective one that tries to balance the *intra-cluster* similarity and the *inter-cluster* dissimilarity, as most successful clustering algorithms try to do.

Intra-cluster similarity is measured by the frequency of term t_i inside document d_j. This is called as the t_f *factor*, which is a measure of how well that term expresses the document content. Inter-cluster similarity considers the frequency of term t_i in the whole collection. It is meant that as the frequency of a term increases in the whole collection, it becomes less important for the particular document d_j, since that term could not distinguish that document from other documents in the collection. Therefore, this measure is referred to as inverse document frequency, *idf factor*. This factor is calculated as shown in Equation 2.2,

where N is the total number of documents in the collection and n_i is the number of documents in which term t_i appears.

$$idf_i = log\frac{N}{n_i} \tag{2.2}$$

The best known term-weighting metric, which is called $t_f - idf$ *metric*, uses these factors. It is given in Equation 2.3, which is the multiplication of the term frequency by the inverse document frequency. In our work, we also preferred to use $t_f - idf$ metric with the vector-space model.

$$w_{i,j} = f_{i,j} \times log\frac{N}{n_i} \tag{2.3}$$

Chapter 3

Parallel Text Retrieval

As the electronic text available online and query processing loads increase, text retrieval systems are turning to distributed and parallel storage and searching. In this chapter, we will briefly review parallel architectures and give some approaches to parallel text retrieval.

Parallel computing is the simultaneous use of more than one computational resource to solve a problem. The parallel formulation of a problem can be performed with respect to the instructions and/or the data that is manipulated by the instructions of the problem. Not all the problems have efficient parallel formulations. It means that it may be more costly dividing the problem and assigning it to multiple processors. However, as long as instruction and/or data requirements of the problem is large, and the problem is suitable for decomposition into subproblems, it is more beneficial to solve the problem in parallel [14].

In parallel architectures, processors can be combined in various ways. Flynn [7] describes a taxonomy for classifying parallel architectures. This taxonomy is based on concept of streams, which are a sequence items operated on by a CPU. These streams can either be instructions to the CPU or data manipulated by the instructions. Four broad classes are described for parallel architecture:

- SISD - Single Instruction Single Data Stream

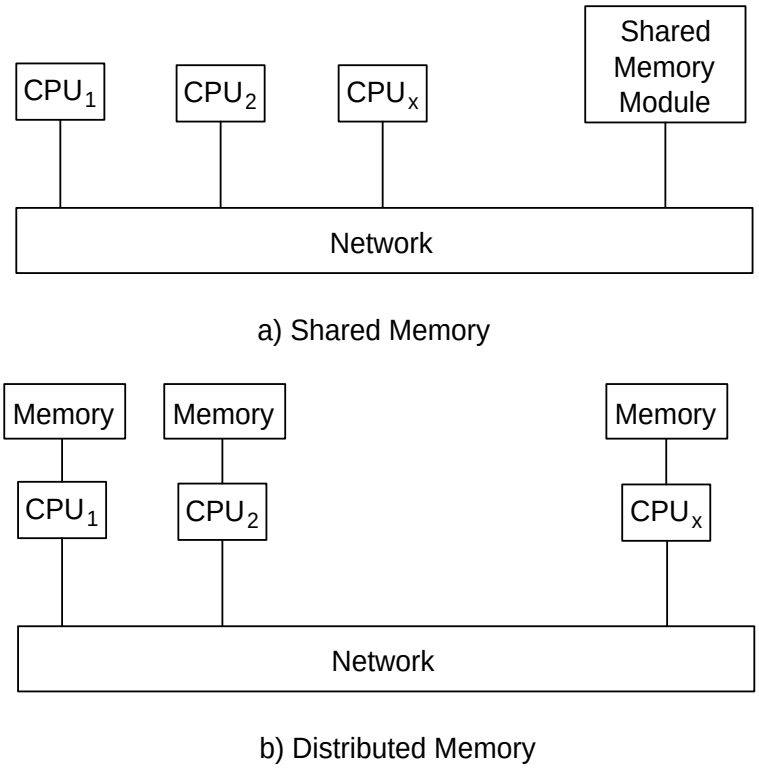


Figure 3.1: Types of memory organizations.

- SIMD - Single Instruction Multiple Data Stream
- MISD - Multiple Instruction Single Data Stream
- MIMD - Multiple Instruction Multiple Data Stream

The SISD class includes the traditional uniprocessor personal computers, running sequential programs. The SIMD class describes the architecture, where N processors operate on N data streams by executing the same instruction at the same time. MISD architecture is relatively rare. In this class, N processors operate on the same data stream, where each processor executes its own instruction stream simultaneously on the same data item [13]. The MIMD class is the most compelling and the most popular parallel architecture. In this architecture N processors operate independently N different instruction streams on N different data stream. The processors in this architecture may have their own memories or share the same memory. These are called as shared memory or distributed

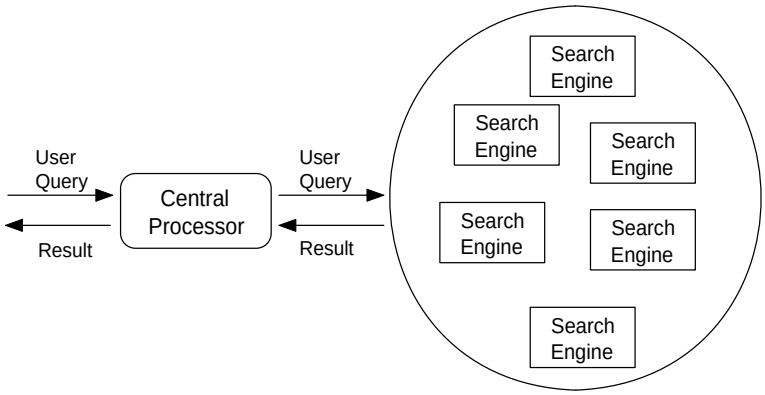


Figure 3.2: Inter-query Parallelism.

memory architectures that are illustrated in Figure 3.1.

When parallel text retrieval architectures are examined, it is seen that there are basically two general categories: Inter-query parallelism and intra-query parallelism. Inter-query parallelism means parallelism among queries. In this type, user queries are collected by a central processor. The central processor sends each query to an available client query processor, and queries are served concurrently by the client processors. This means that each client processor behaves like an independent search engine. This is demonstrated in Figure 3.2, which can be also found in [2]. Since each query is served by a single processor, this architecture is called inter-query parallelism.

In intra-query parallel architectures, a single query is distributed among the processors. In this case, a central processor collects and redirects an incoming user query to all client query processors. Each processor processes the incoming query, constitutes its own partial answer set and returns them to the central processor, where all the partial answer sets are merged to a single final result and returned for presentation to the user. This architecture is named as intra-query parallelism as all the client query processors cooperate to evaluate the same query. This is depicted in Figure 3.3, which is shown also in [2].

In this work, we focus on intra-query parallelism on a shared- nothing MIMD parallel architecture. This means that communication between the processors is through messages, and each processor has its own local disk and memory.

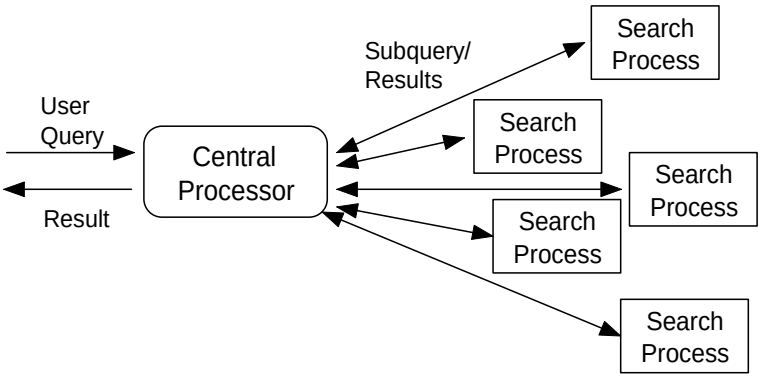


Figure 3.3: Intra-query Parallelism.

3.1 Inverted Index Partitioning for Parallel Query Processing

As mentioned earlier, in a traditional text retrieval system, the efficiency is measured by the response time and the throughput of the system. The response time to a query is affected by many factors. Mainly, these are the query-dependent, collection-dependent and system-dependent factors. The number of the terms in a query and the query term frequencies are in the query-dependent factors. The size of the collection and the frequencies of the terms in the collection are included in the collection-dependent factors. Lastly, the query processing time is affected by the system-dependent factors such as disk and CPU performance parameters.

In parallel query processing, some additional factors are included that affect the query processing time. Some important ones are the parallel architecture used, the number of processors, the network performance parameters and the index organization.

The main interest in this thesis is inverted index partitioning on a shared-nothing architecture, as mentioned previously. Inverted index partitioning is a preprocessing step for parallel query processing and its organization has a crucial effect on the efficiency of the system [9]. As the organization of the inverted index heavily determines the time elapsed on the network and disk access [27, 18, 11, 24].

Besides the efficient usage of the network and the disks, the balance of the storage costs of the disks should be taken into consideration while partitioning the inverted index [27, 18, 11]. Assume that the system has K processors and there are $|P|$ posting entries in the collection, so each storage site in $S = \{S_0, \cdots, S_{K-1}\}$ should be assigned to approximately an equal number of posting entries to balance the storage, as shown in Equation 3.1. $SLoad(S_i)$ shows the posting storage of site S_i.

$$SLoad(S_i) \simeq \frac{|P|}{K}, \qquad \text{for } 0 \leq i \leq K-1 \tag{3.1}$$

3.1.1 Inverted Index Partitioning

Several ways can be followed while partitioning the inverted index of a collection. The posting entries of the inverted index can be distributed among the processors in a random manner, or by following a specific methodology. There are basically two main methods for partitioning of the inverted index in parallel systems.

In the first method, the document-ids in the collection are evenly distributed across the processors. Each processor is responsible from a different set of documents. Considering that the documents are evenly distributed, each processor has a posting list of size that is given in Equation 3.1. Since this partitioning is based on the document-ids, this organization is called document-id partitioning. The second method is term-id partitioning. The inverted index of the whole collection is distributed across the processors according to the term-ids. In this case, each processor is responsible from its own set of terms.

The reason that document-id or term-id partitioning methods are mainly used is that they have some advantages in terms of system parameters. Document-id partitioning balances the storage costs of the disks and also uses the network efficiently by minimizing the total volume of communication in the parallel system. On the other hand, term-id partitioning uses disks efficiently by reducing the total volume of disk accesses in the system. We will discuss this in more detail in Section 3.2.

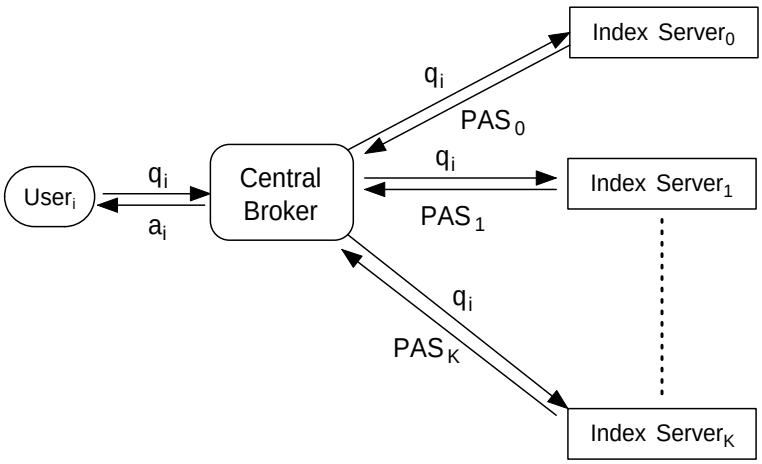


Figure 3.4: Query processing for document-id partitioning scheme.

3.1.2 Parallel Query Processing

In this section, we will describe the processing of the queries on a shared-nothing, intra-query parallel architecture. Typically, in a shared-nothing parallel system, there is a central processor, which we name as central broker and a set of client processors, which we call index servers. The central broker collects the incoming user queries, inserts them in a queue and redirects the queries to the related index servers. The index servers retrieve the documents based on the degree of the similarity of the documents to the query, which is calculated based on the vector-space model. The index servers form their partial answer sets, which are composed of the retrieved document-ids and their weights and send them to the central broker. The partial answer sets obtained from the index servers are collected and merged by the central broker. Finally, using a ranking-metric the central broker orders the documents according to their relevance and returns to the user. The query distribution among the index servers and processing steps differ somewhat depending on the index partitioning schemes.

Figure 3.4 illustrates query processing for document-id partitioning. In this scheme, the central broker takes a query (q_i) out of the queue and sends it to all index servers. Each index server reads its own posting lists corresponding to the terms of the query and forms its partial answer set (PAS). Partial answer sets returned from each index server is merged and sorted by the central broker and

sent to the user as the final answer set (a_i) of the query.

In term-id partitioning, when the central broker takes the query out of the queue, it checks which index servers hold inverted lists of the query terms. Accordingly, the central broker breaks the query into subqueries and send them to the related index servers. These index servers form their partial answer sets and send them to the central broker. The central broker collects and merges all the partial answer sets returned and sends the final answer set to the user.

3.2 Inverted Index Partitioning Strategies for Parallel Query Processing

As mentioned in Section 3.1.1, there are two main inverted index partitioning methods: Document-id and term-id partitioning. Several strategies can be followed on the partitioning of the inverted index according to these two methods. In this section, we will discuss these strategies by considering the system parameters. Especially, we focus on the efficiency of the network and disk usage in terms of the total volume of communication and the total number of disk accesses.

3.2.1 Document-Id Partitioning

In this partitioning scheme, the inverted index is distributed across the index servers according to the document-ids, so each index server has a distinct set of documents. This simplifies the communication of the index servers with the central broker in a remarkable way. Recall that, for a user query, the index servers send their partial answer sets, which contain the document-ids and their weights, to the central broker through the network. Since each index server has a distinct set of documents, there is no overlapping in the partial answer sets. So, this scheme naturally achieves the minimum total volume of communication through the network. However, in this partitioning scheme, the total number of disk accesses may be large, since each index server has its own local inverted index

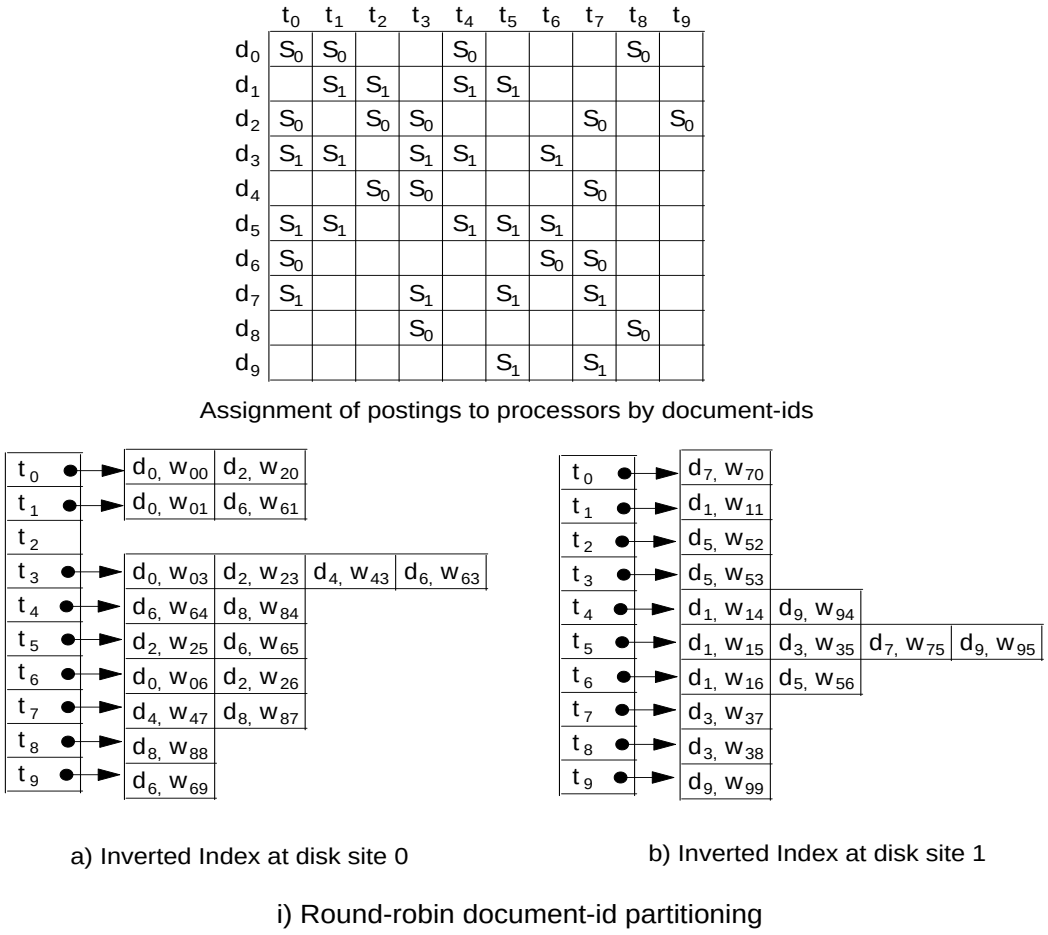


Figure 3.5: 2-way round-robin document-id partitioning of our sample collection.

and a term may have postings on several disks. For a user query, all the index servers that have the terms of the query access their disks to read the postings lists of the corresponding terms. So, the total number of disk accesses in the system may be quite large.

In the literature, document-id partitioning is performed in a round-robin fashion [27, 18, 11]. Namely, the document-ids are distributed across the processors one-by-one. Figure 3.5 illustrates partitioning of the inverted index of our sample collection among two processors according to the document-ids in a round-robin fashion.

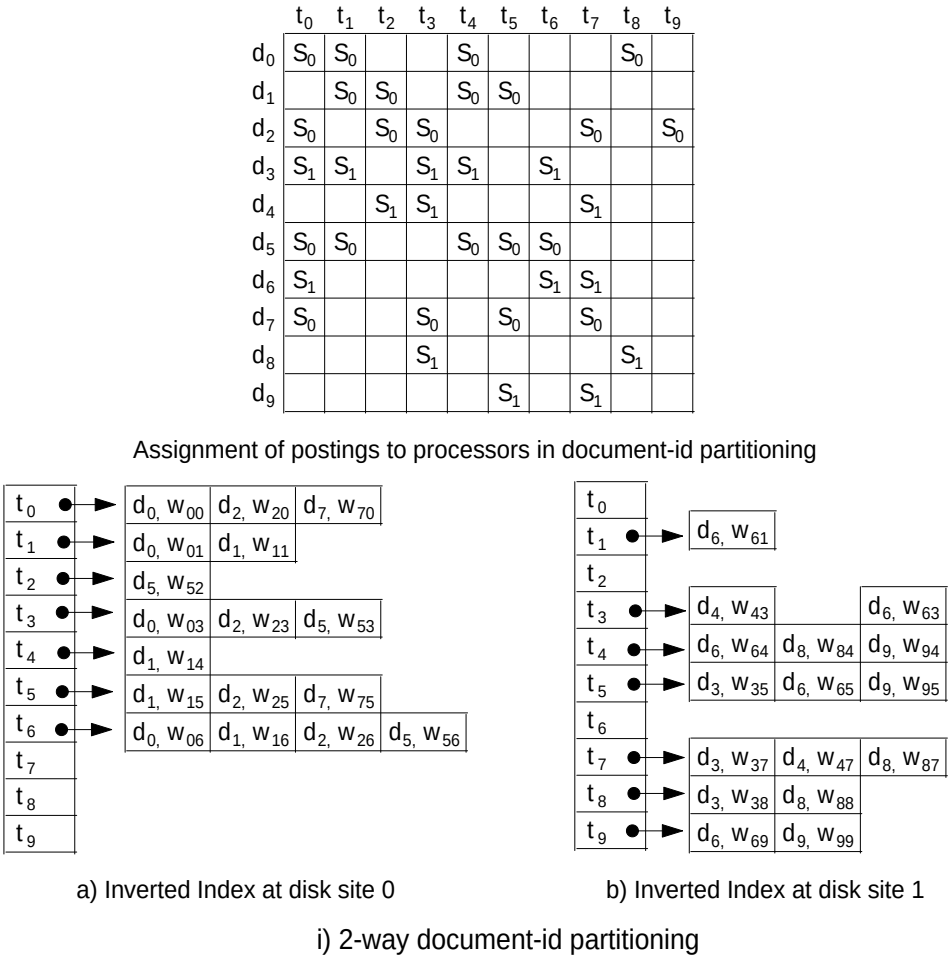


Figure 3.6: 2-way document-id partitioning of our sample collection.

As in document-id partitioning the total volume of communication is minimum, it is precious to decrease the number of disk accesses. When a strategy is followed for the partitioning of the inverted index by the document-ids, the objective should be to reduce the number of the terms that is indexed at several disks. This can be accomplished by clustering more related documents on the same disks. Namely, by allocating the documents that have more terms in common, the number of the terms that has postings on several disks can be minimized. In this respect, no strategy is followed to improve the efficiency of the system in round-robin partitioning scheme.

The above idea is explained in Figure 3.6. Our objective is to minimize the

number of the terms that are indexed for both disks. Assume that all the terms of the collection are queried once. In this example with such a query set, the total number of disk accesses is 14. This is the number of the distinct terms that appear only on one site plus two times the number of the terms that appear on both sites, as these terms are accessed by both index servers. On the other hand, in round-robin partitioning shown in Figure 3.5, there are 19 disk accesses with the same formulation. Only for t_2 there is one disk access while the other terms are accessed twice in round-robin partitioning. Hence, by gathering the related documents together, the total number of disk accesses is reduced by 26.3% in this example partitioning.

3.2.2 Term-Id Partitioning

In term-id partitioning scheme, the inverted index of the collection is distributed across the index servers based on the term-ids, so each index server is responsible for a distinct set of terms. This minimizes the total number of disk accesses in the system as a whole. The total number of disk accesses is already equivalent to the lower bound achieved by the sequential algorithm. Since for a query term, only one disk access is done by the index server, which has the postings of this term. However, in this partitioning scheme, while the number of disk accesses is minimum, the total volume of communication may be large. Two terms indexed at different index servers may have postings that share the same documents. So, partial answers sets transmitted from different index servers may include the same documents. Repetition of the documents at the network causes increase in the total volume of communication.

Studies so far focus on term-id partitioning in a round-robin fashion [27, 18, 11]. In this partitioning scheme, the term-ids are distributed among the processors one-by-one. Figure 3.7 shows partitioning of the inverted index of our sample collection among two processors according to the term-ids in a round-robin fashion.

When the partitioning of the inverted index is by the term-ids, the total

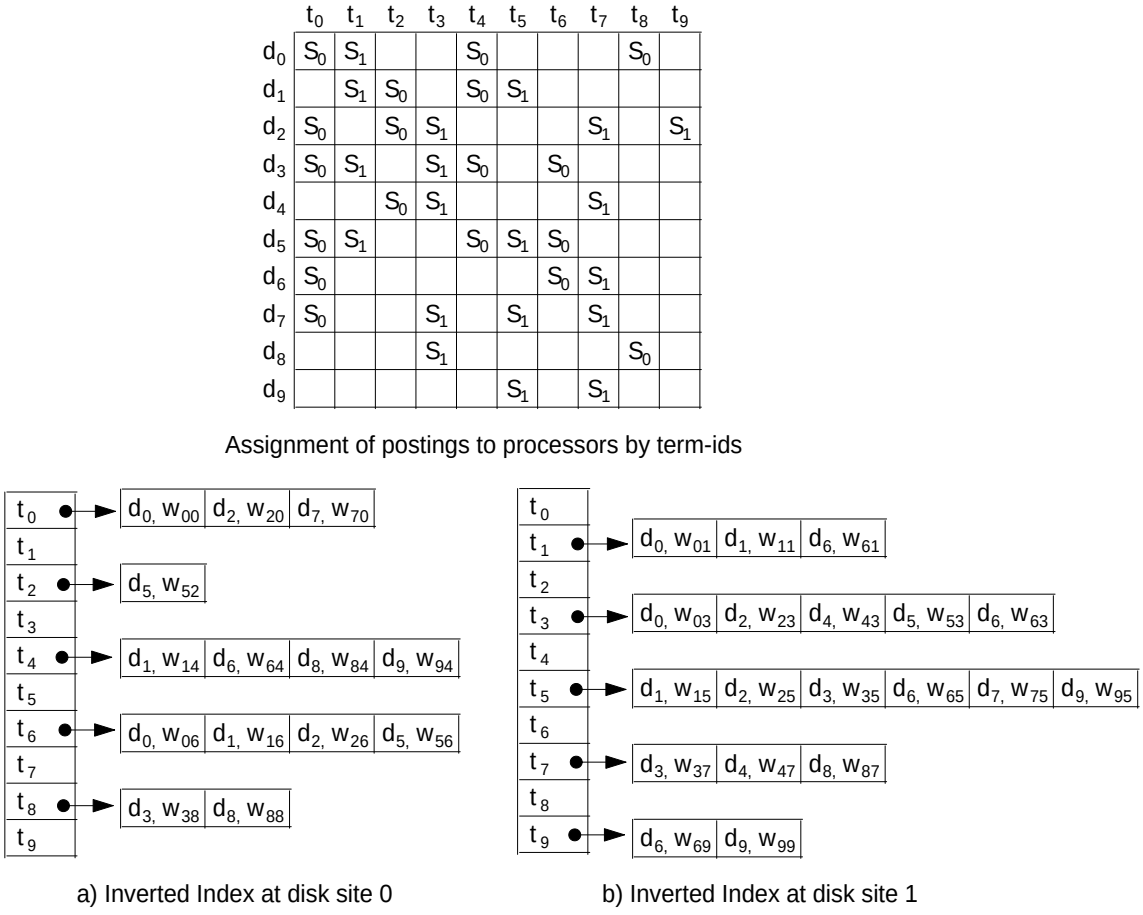


Figure 3.7: 2-way round-robin term-id partitioning of our sample collection.

volume of communication should be taken into consideration, as it may be quite large. Repetition of the documents at the network can be reduced by assigning a document to a minimum number of index servers. By clustering more related terms on the same index servers, the number of the documents that belong to several disks can be decreased. The terms are said to be more related in the sense that they appear in more common documents. In round-robin partitioning scheme, the distribution of the documents among the processors is not considered, in view of that, the size of the total volume of communication is not considered.

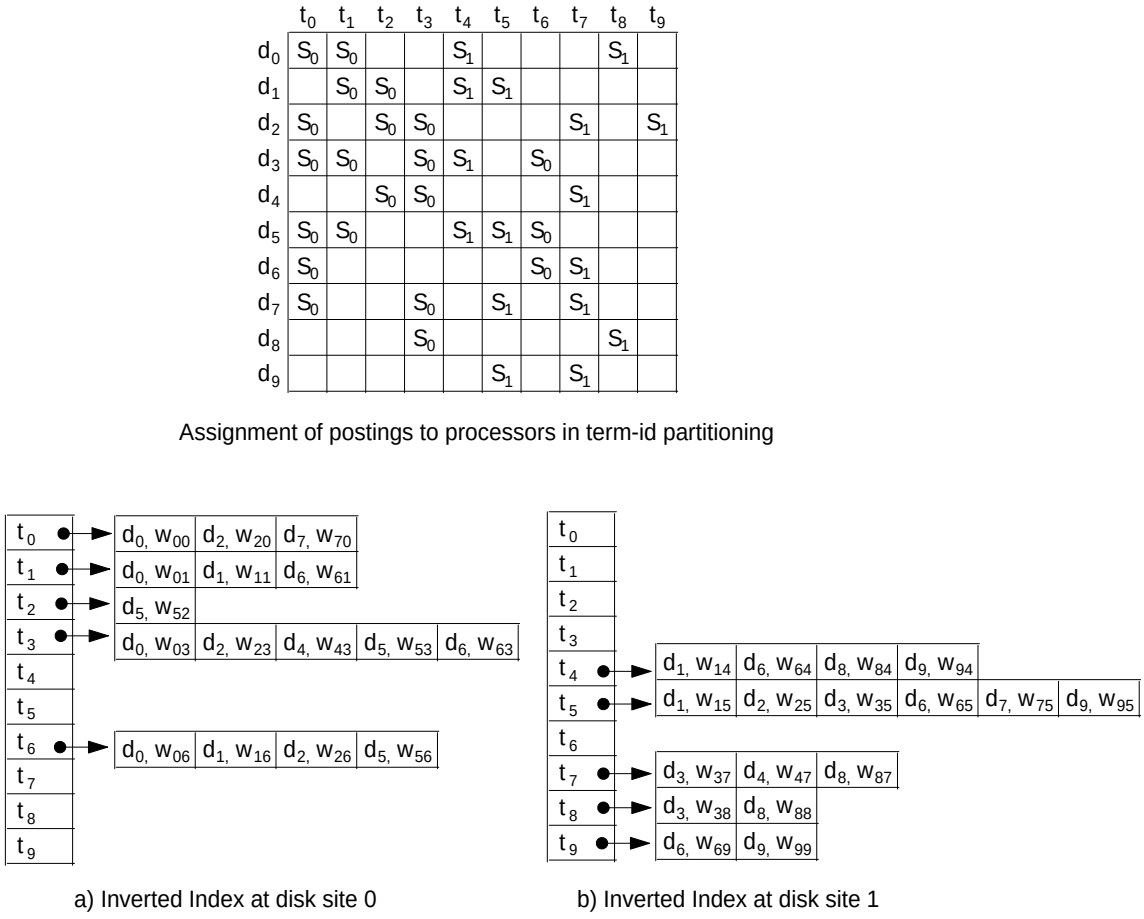
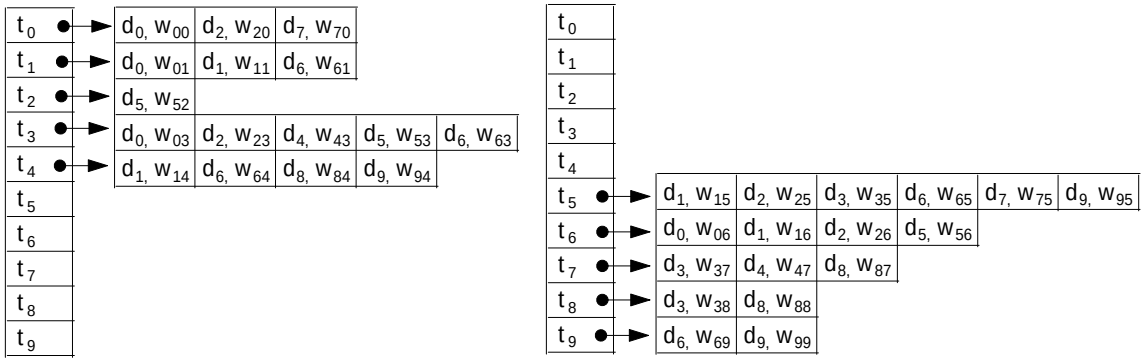


Figure 3.8: 2-way term-id partitioning of our sample collection.

Figure 3.8 exemplifies what we discuss above. Our objective in this partitioning example of our sample collection is to reduce the total volume of communication by decreasing the number of the documents that appear on both sites. When all the documents of the collection are requested once, the total number of posting entries to be transmitted by both index servers will be 15. This is the number of the distinct documents that appear only on one site plus two times the number of the documents that appear on both sites, as these documents are sent by both index servers. In round-robin partitioning shown in Figure 3.7, the number of posting entries to be transferred is 19 with the same formulation. So, relative to the round-robin partitioning by employing the proposed objective, the total volume of communication is decreased by 21% in this example.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
d_0	S_0	S_0			S_0				S_1	
d_1		S_0	S_0		S_0	S_1				
d_2	S_0		S_0	S_0				S_1		S_1
d_3	S_0	S_0		S_0	S_0		S_1			
d_4			S_0	S_0				S_1		
d_5	S_0	S_0			S_0	S_1	S_1			
d_6	S_0						S_1	S_1		
d_7	S_0			S_0		S_1		S_1		
d_8				S_0					S_1	
d_9						S_1		S_1		

Load balanced term-id assignment of postings to processors



a) Inverted Index at disk site 0 b) Inverted Index at disk site 1

i) Load balanced term-id partitioning

Figure 3.9: 2-way, load balanced term-id partitioning of our sample collection.

3.3 Related Work

There are a number of papers about parallel text retrieval systems [27, 11, 18, 3, 10, 23, 15] in the literature. In this section, we will focus on there papers [27, 11, 18] that put emphasis on the organization of the inverted index, as it strongly affects the performance of parallel text retrieval systems.

Jeong and Omiecinski [11] investigate round-robin term-id and document-id partitioning schemes on a shared everything multiprocessor system with multiple disks. They examine the performance of these partitioning schemes by simulation, using a synthetic data set that is created by following Zipf's law [21]. They use

a Zipf-like probability distribution that is based on the correlation between the rank and the frequency of the terms in the collection. They use the Boolean model to process the queries.

In term-id partitioning, they propose two heuristics to balance the storage costs: Partition by Term I and Partition by Term II. In Partition by Term I, instead of terms, posting lists are distributed evenly across the disks. Under the assumption that the query term frequencies are uniform, their heuristic provides approximately equal number of disk accesses. Their approach is described on our sample inverted index in Figure 3.9. As can be seen in the figure, the postings of sites 0 and 1 are 16 and 17, respectively. Namely, load balance is achieved by this heuristic.

In the case that the query term frequencies are not uniform, they assumed that term frequencies of the terms in the query could be estimated. Their other heuristic, Partition by Term II, takes the access frequencies of the terms also into consideration. They formulated this heuristic as balancing the sum of the posting size multiplied by the term access frequencies.

Although Jeong and Omiecinski [11] point out some problems associated with term-id and document-id partitioning, they do not mention how to solve them. They emphasize the importance of balancing the storage costs of the disks and propose the heuristics for term-id partitioning that are stated above. In their heuristics, they do not consider the minimization of the total volume of communication, which may become a problem in term-id partitioning. For example, when all documents are requested once, their load balanced heuristic given in Figure 3.9 transfers 19 postings, while the example that clusters the related terms together, depicted in Figure 3.8, transfers 15 postings.

Tomasic and Garcia-Molina [27] examine four methods to partition the inverted index on a distributed shared nothing system. These methods comprise Disk, I/O Bus, Host and System organizations. The Disk and System organizations are the same as the round-robin document-id and term-id partitioning schemes respectively. In I/O Bus organization, documents are partitioned to the I/O buses, then for each bus an inverted list is built and distributed across the

disks. In the Host organization the documents are partitioned to the hosts, then for each host an inverted list is built and distributed across the disks. Namely, the I/O Bus and Host organization differ in that, whether document-id partitioning is done across the I/O buses or hosts. In their example, these two organizations are same, as each host has exactly one I/O bus. They experiment the performance of their methods by simulation with synthetic data sets. Queries are answered by an answer set model, without examining the text of the documents. Also, the Boolean model is used to process the queries.

In their work, the main focus is on the comparison of the performance of these four partitioning schemes. The simulation results show that the Host organization perform well for full-text systems while the System organization (term-id partitioning) is better on the abstracts of the texts. Their results are reasonable, because Host organization is a hybrid scheme of term-id and document-id partitioning, so with this organization they balance the drawbacks of these two schemes. Hence, the Host organization performs better than the other schemes. Also, it is meaningful that the System organization (term-id partitioning) is better on the abstracts. When the collection is composed of the abstracts, the posting lists will be very short compared to a full-text system, and this prevents the problem that the total volume of communication becomes very costly in term-id partitioning.

Yates and Ribeiro-Neto [18] also investigate two partitioning schemes on a shared nothing system: Global index organization and local index organization, which are equivalent to round-robin term-id (System organization) and document-id (Disk organization) partitioning respectively. Instead of using the Boolean model they use the vector-space model as their ranking strategy. They experiment the performance by a simulator that is coupled with a simple analytical model. They partition the global index among the machines in lexicographical order by assigning each of them roughly an equal portion of the whole index. In that sense, they do balancing in term-id partitioning. Their results show that the global index organization performs better than the local index organization in the presence of fast communication channels. Their results in general support the trade-offs in system parameters. For example, it is expected that term-id

Table 3.1: A comparison of the previous works on inverted index partitioning

	Tomasic and Garcia-Molina	Jeong and Omiecinski	Yates and Riberio-Neto
Year	1993	1995	1999
Target architecture	DMA	multi-disk PC	NOW
Index residence	disk	disk	disk
Ranking model	boolean	boolean	vector-space
Partitioning model	round-robin	load-balanced	load-balanced
Datasets	synthetic	synthetic	real
Experimental setup	simulation	simulation	simulation

partitioning gains superiority over document-id partitioning in the presence of fast communication channels, as this lessens the problem with the total volume of communication in term-id partitioning.

The main focus of the works done so far is based on the organization of the inverted index. In these works, the performance of different inverted index partitioning schemes is compared by simulation under different parameters. Table 3.1 gives the major points of the previous works done on inverted index partitioning.

Chapter 4

Implementation

In this work, we have designed and implemented a parallel text retrieval system. In our system, we have investigated different partitioning schemes based on the document-ids and the term-ids. Besides round-robin partitioning, we have worked on the partitioning schemes that try to solve the problems with document-id and term-id partitioning in terms of the system parameters. Concisely, in document-id partitioning scheme, our objective is to minimize the total number of disk seeks by clustering more related documents on the same disks while in term-id partitioning scheme, our goal is to reduce the total volume of communication by allocating more related terms on the same disks. To achieve this, we use the hypergraph-theoretical index partitioning model, which handles nicely both load balancing and problems associated with index partitioning schemes. For theoretical background of these models one can refer to [4]. In addition to these, we also designed and implemented a query interface and a user interface of our parallel text retrieval system.

4.1 Preprocessing Modules

In this section, we will describe the programming modules to generate the data sets, their inverted indices and the query sets to submit to the system. By passing

through these modules, we prepare the inputs to our parallel text retrieval system. Namely, the queries that the system evaluates and the data sets that the queries are searched on are created by these modules.

4.1.1 Data Set Generation

4.1.1.1 Real-Life Data Set Generator

In this phase, real data sets are converted to inverted indices. We have worked on the *Radikal* data set, which is a Turkish newspaper. This repository includes a variety of news about politics, economics, sport, art, culture and daily life. It is 11.2 MB in size.

Modules followed in this phase are corpus creation, document vector creation and lastly inverted index creation. Firstly, from the *Radikal* data set a corpus is created. Stop word elimination [28] is employed on the corpus. That is, the most frequently words, like "the", "a", "an", "and" and etc. are eliminated from the corpus. Thereafter, from the created corpus all the needed data is extracted. These are the number of distinct documents and terms in the corpus. In our corpus there are 6,888 distinct documents (D) and 125,399 distinct terms (T). Also, the number of distinct documents that each term appears in and the total number of terms that a document includes are determined. With this gathered data, the document vector is created. In the document vector, for each document-id there is a row, which includes the total number of terms and the term-ids with their frequencies. A document may not comprise most of the terms. Therefore, the terms that do not appear in this document are not indexed to prevent redundancy. Finally, inverted index is obtained from the document vector. The index is *inverted* in the sense that the key values *terms* are used to find the records *documents*, rather that the other way around.

4.1.1.2 Synthetic Data Set Generator

This phase includes the creation of the synthetic inverted indices with various probability distributions. In the indexed file of the natural-language text or terms (keywords), the distribution of postings per access term was shown in [11] to follow Zipf's law. Zipf's law [21] states that there is an inverse relationship between the frequency of the terms and their ranks in a corpus of natural language text. Namely, the rank of a term decreases, as its frequency increases in the corpus. The constant rank-frequency law of Zipf is stated as Equation 4.1 below:

$$\text{Frequency x rank} \simeq \text{constant} \tag{4.1}$$

We have used Zipf-like probability distribution [12] to model the data skewness of posting entries. Documents are created by W distinct terms, which follow the probability distribution function $Z(t_i)$, given in Equation 4.2, with independent and identicall trials.

$$Z(t_i) = c/i^{(1-\theta)} \text{ where } c = 1/\sum_{i=1}^{T}(1/i^{(1-\theta)}),\ 1 \leq i \leq T \tag{4.2}$$

By changing the value of θ, different data skewness can be obtained. θ is calculated as shown in Equation 4.3.

$$\theta = \log(Fraction\ of\ Posting\ Entries)/\log(Fraction\ of\ Terms) \tag{4.3}$$

If we want to have a data skewness, where 20% of the terms comprise 80% of the posting entries, we use the *80-20* rule to calculate θ. θ value for *80-20* data skew is $\theta = \log 0.8 \ / \log 0.2 = 0.1386$. The value of θ varies between 1 to 0. If θ equals to 1, we have the uniform distribution, as θ decreases to 0, we get closer to the pure Zipf distribution.

This programming module needs four inputs in order to generate the synthetic data set. These are θ, D, T and W. By changing these parameters, we can obtain a variety of data sets. The skewness of the data set, the total number of distinct documents and terms in the data set and also the total number of terms that a document contains can be changed by these parameters. As in real data set generator, the following modules carry out document vector generation and the creation of the inverted index.

4.1.2 Query Set Generation

4.1.2.1 Real-Life Query Set Generator

We assumed that query patterns are similar to patterns in the documents. That is, the probability of a term occurring in a query is proportional to that term's frequency in the document collection as a whole. In the literature, [24] also models their query sets under this assumption. Accordingly, we created our query sets randomly by extracting the terms from the corpus, which we used as our data set. Also, the number of the terms in a query can be changed with this programming module.

4.1.2.2 Synthetic Query Set Generator

In our synthetic query set generator module, we used the term generating probability distribution $Q(t)$, shown in Equation 4.4. In the work by [11], the query sets are also generated by using this distribution.

$$Q(t) = \begin{cases} C\text{x}Z(t_i) & if\ 1 \leq t \leq uT \\ 0 & Otherwise \end{cases} \text{ where } 1 = \sum_{i=1}^{uT}(C\text{x}Z(i)) \tag{4.4}$$

The parameter value, u affects the probability that a term appears in a query. As u decreases, the probability of choosing more frequent terms appearing in the collection increases. Also, we can change the skewness of the query set by altering the value of θ, which is a parameter of the Zipf-like probability distribution $Z(t_i)$, given in Equation 4.2. For instance, to generate a uniform query set, we set θ to 1, by determining the skewness of the query set as *50-50*. In a uniform query model, each term has the same access probability.

4.2 Parallel Implementation

We implemented the shared-nothing parallel text retrieval system in C language using the Message Passing Interface (MPI) parallel language package. In this

section, we will see the details of the implementation of our parallel text retrieval system.

There are three key components in the system. These are the user, the central broker and index servers. Both the central broker and the index servers use a queue while processing the user queries. As mentioned earlier, the system works roughly as follows: the user inserts the queries into the system, the central broker takes these queries into its queue and sends them to the related index servers. The index servers process sent query terms, form their partial answer sets and send them to the central broker. The central broker merges the partial answer sets and returns the final answer set to the user.

Here, we will discuss our text retrieval system in view of the central broker in more detail. Three basic steps are followed by the central broker repeatedly. The first step is to check the incoming queries. If there is a query submitted by a user, the central broker inserts the query into the queue. Secondly, it checks whether there is a partial answer set sent by an index server at the network. If so, it is inserted into the queue. Finally, the central broker checks the queue.

The queue of the central broker can contain both the queries coming from the user and the partial answer sets coming from the index servers. If there is a user query in the queue, the central broker prepares the subquery packet for that query. In the subquery packet preparation, for term-id partitioning, the index servers that have the postings of the terms in the query are determined. The central broker records the number of partial answer sets needed for that query, and sends the prepared subquery packet to all related index servers. If a partial answer set is in the queue, the central broker examines whether a partial answer set corresponding to the same query is sent before. If not, it allocates an empty accumulator array for that query. An accumulator array has an entry for each document-id in the collection. So, its size is D. The central broker uses this array to store the weights of the document-ids returned in the partial answer sets. At the beginning, the accumulator array is empty, i.e. the weights of all the documents are set to zero. The next step is to merge the partial answer set with the partial answer sets in the accumulator array of that query. Namely, the

central broker updates the weights of the document-ids sent in the partial answer set. Also, the central broker checks whether all partial answer sets are sent for that query. If all the partial answer sets are collected for that query, the central broker sends the final answer set to the user of the query.

An index server has two main steps to follow continuously in the system. The first one is to check whether there is a subquery packet sent by the central broker at the network. If so, it is inserted into the queue. The second step is to examine the queue. The queue of an index server contains subquery packets sent by the central broker. If there is a packet in the queue of the index server, an empty accumulator array of size D is allocated for the partial answer set of the subquery. Then, the index server reads the posting lists of the terms of the subquery and updates the weights of the document-ids retrieved for that subquery in the accumulator array. When all terms of the subquery are processed, the partial answer set is ready to be sent. So, the index server sends the partial answer set of that subquery to the central broker.

4.2.1 Communication

The communication between the central broker and the index servers is performed by the Message Passing Interface, MPI. Message passing is a programming paradigm used widely on parallel computers with distributed memory. We used basically *MPI_SEND* and *MPI_RECEIVE* to send and receive packets between the central broker and the index servers. As explained in Section 4.2, the index servers send their partial answer sets, which are collected in the accumulator arrays of size D, to the central broker. However, most accumulator arrays are not full, since not all the documents are retrieved for each query. So to avoid redundant data transmission through the network, we pack the data before sending, and unpack after it is retrieved, by means of *MPI_PACK* and *MPI_UNPACK* respectively.

4.3 Data Structures

While implementing the system, appropriate data structures tried to be selected to reduce time and storage costs. In this section, we will remark some important details about the system and its implementation.

4.3.1 The Trie Data Structure

A trie is basically a type of general tree, containing words and/or numbers. The trie is an immensely useful data structure when storing strings in memory. In text retrieval, the trie has been built to contain whole words of the collection and maintain a count of how many times a word occurs. This data structure enables the retrieval of a word in $O(k)$ time, where k is the length of the indexed word. By this way, this structure makes the search of the collection and the weighting of the documents by the recurrence of a particular word quite simple and fast [28, 16].

We used the trie data structure to index the terms in the collection and incoming query terms. Figure 4.1 illustrates a simple example about how indexing is done on our trie data structure. In this example, the first word to index is *car*, so the characters of the *car* are inserted into the trie. For each new coming word, the first level of the trie is searched, if there is a match to the first character of the word, then one level down is checked whether there is a match to the second character of the word. If there is a match, then one level down is checked, this search is continued until no match is found. In the example, our second word is *cat*. On the first level, there is a match for character *c*, then we go one level down. The second level also matches to character *a*. On the third level, there is no match for character *t*, so we insert *t*. Our third word is *tea*, since there is no match on the first level, we go right and insert all characters of this word into the trie. All incoming words are indexed by following this way.

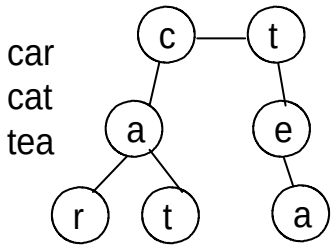


Figure 4.1: An example on the trie data structure.

4.3.2 Accumulators

A simple dynamic array of size D is used for accumulators, as mentioned in Section 4.2. A dynamic search structure such as a tree or a hash table is appropriate if the accumulators are expected to have much fewer entries than the number of documents. It is because in the tree or hash table implementation, space is required to index documents' ids, whereas in array implementation, by allocating the array as the size of document number, we get rid of indexing the ids of the documents. So, when many documents are retrieved for a query, a tree or a hash table requires a great deal of space compared to the array implementation. In large collections, where the number of the documents and the terms is very huge, the number of nonzero elements is much fewer than the number of the documents. So, it is advisable to use these dynamic structures for accumulators. However, in our implementation, we deal with small collections, so using arrays is reasonable in our case.

4.4 Simulation of the Disk

As stated in Section 4.1.1, our data set is small in size (11.2 MB). Consequently, after processing a few queries, the operating system takes a large portion of posting lists into the memory by paging, so there will be no disk accesses and I/O after a point of the experimentation. As large data sets are used in real systems, we simulated the disks to make the implementation more realistic.

Table 4.1: Values used for the cost components in the simulation

Cost type	Symbol	Cost
Seek time	T_{ds}	8.5 ms
Rotational latency	T_{rl}	4.2 ms
Reading a disk block	$T_{I/O}$	13 μs

There are three main parameters used to measure the time to read data from the disk [22]. The first and the most costly one is the *disk seek time*. The seek time is the time taken to move the arm to the correct cylinder.

The disk drive is constantly rotating. The head must be positioned on the correct track of the cylinder. That is, once the arm is placed on the correct cylinder, the head waits to position until the correct place on the track is just reaching the head. This time is called the *rotational latency*.

The third parameter is the *block reading time*. It is the time to read a block. In our simulation, our disk blocks size are 512-byte. The formula for calculating the time to read posting list T_R, for a term t_i is given in Equation 4.5, where BF stands for the blocking factor of the disk.

$$T_R = T_{ds} + T_{rl} + \lceil \frac{PSize(t_i)}{BF} \rceil \times T_{I/O} \tag{4.5}$$

In Table 4.1 the cost parameters are given, which are the typical values for a today's PC cluster.

4.5 Query Interface

In order to model the query interface of our system, we have used CGI, which is abbreviation for *Common Gateway Interface*. This is an interface standard that provides a method of executing a server-side program (script) from a web site to generate a web page with dynamic content. Any programming language that produces an executable file can be used to write CGI scripts conforming to this standard. Most often these scripts are written in Perl, Python, C, C++, or TCL.

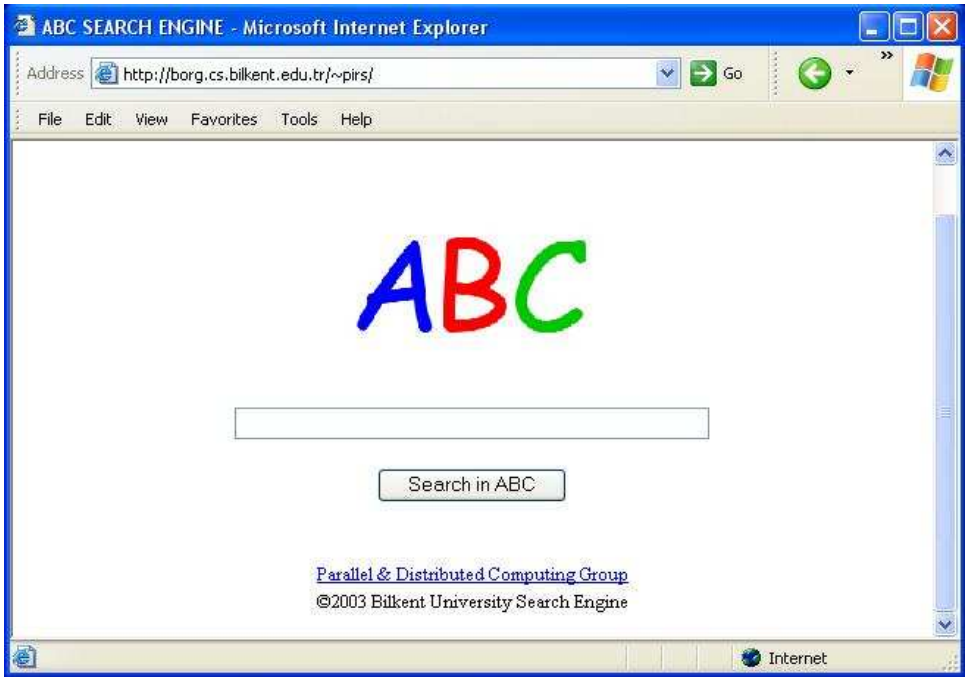


Figure 4.2: ABC website.

We have used Perl due to its flexibility.

Figure 4.2 shows our web site [25], that provides an interface for the users who wants to use our search engine. It runs on the *Radikal* data set. If a user submits a query and clicks the *Search* button, our CGI script executes. This script will execute the program, which is on the client side. It takes the query and directs it to the central broker. Then, it begins waiting for the packet from the central broker, which will process the query, produce the answer set and send it back to the client.

In order to achieve interprocess communication between the client side and the server side, we have used sockets. A socket is a generalized interprocess communication channel. Like a pipe, a socket is represented as a file descriptor. But, unlike pipes, sockets support communication between unrelated processes, and even between processes running on different machines that communicate over a network. In our case, processes run on the same machine, we maintain communication channel between the processes via sockets.

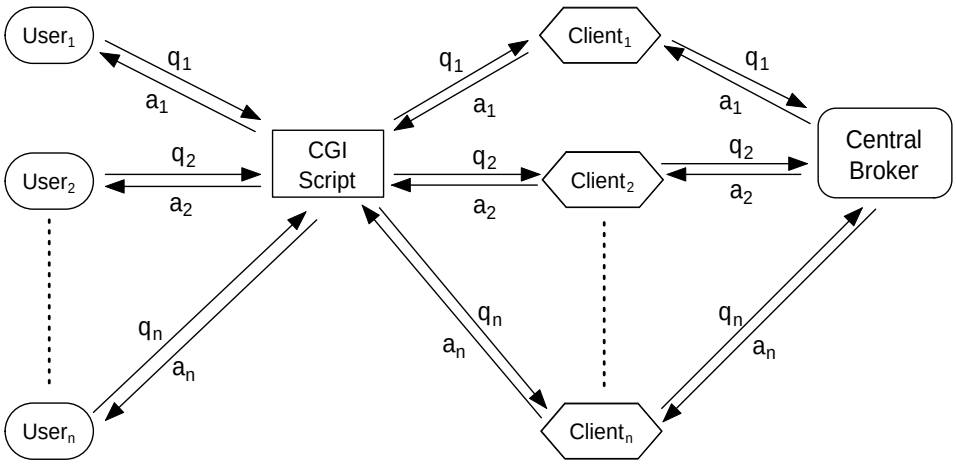


Figure 4.3: The query interface.

Figure 4.3 illustrates how the system works, where q and a are abbreviations for *query* and *answer set* respectively. Users send their queries through our web site. For each inserted user query CGI script makes a system call to execute a client program, through which user queries are sent to the central broker. As explained in Section 4.2, one of the steps that central broker continuously pass is to check the network for the user queries inserted to the system. Clients can connect to the central broker while central broker checks the network for user queries. The central broker takes the queries from the clients that request for connection. After sending the queries, clients begin to wait for the final answer set of the query from the central broker. The central broker closes the connection with the client when it sends the answer set. Finally, the answer set of the query is transmitted to the user again via CGI script for presentation.

4.6 User Interface

We designed a web site called *ABC* [25], which provides a user interface for our parallel text retrieval system. Our design is based on the design of the web page for *Google* [26].

In Figure 4.4, the query *ankara sanat* is inserted to *ABC*. When the user clicks

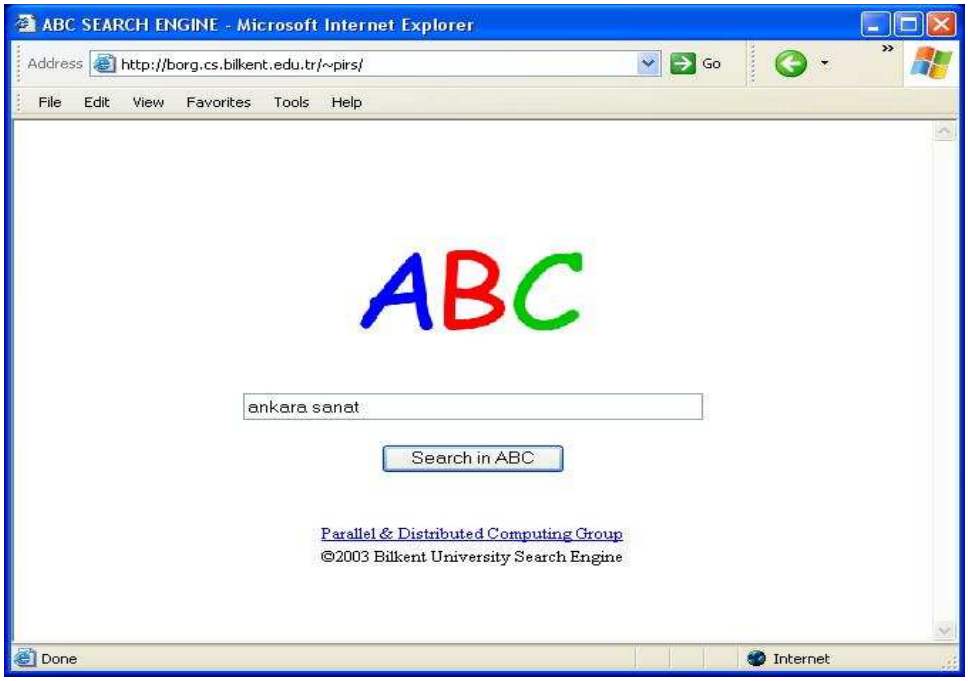


Figure 4.4: A query is inserted.

on the *Search in ABC* button, the answer set will return as the format seen in web page shown in Figure 4.5. In this search the documents of the *Radikal* data set is searched on the central broker[1]. We put the links to the documents returned, the category names that the documents belongs to and the full address of the document links on our web page. When the user clicks the link for a document, the page of the document pops up. For example, when the user clicks on the *03anado.html* link in Figure 4.5, then the page given in Figure 4.6 will return to the user. In our design, we change the colors of the links related to whether the page is visited or not. Also, we include the number of documents returned and the time that search lasted on our web page.

[1]Borg: the parallel machine of Bilkent University Computer Science Department.

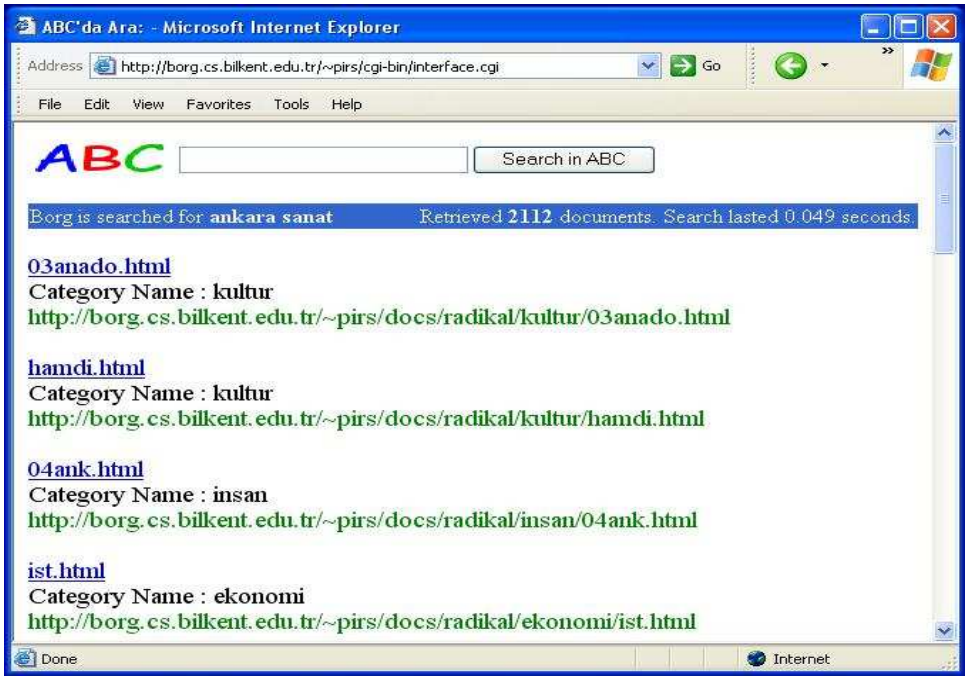


Figure 4.5: The answer set returned for the query.

Figure 4.6: A document returned for the query.

Chapter 5

Experimental Results

As the parallel text retrieval platform, a 24-node PC cluster is used. The central broker contains an Intel Pentium III 500 Mhz processor and 2 GB of RAM. The processing nodes of the cluster are the index servers of the system. The index servers are equipped with 128 MB of RAM, and are interconnected by a 3COM Superstack II 3900 Fast Ethernet switch. Each index server has an Intel Pentium II 400 Mhz processor and runs the Debian GNU Linux operating system. Our parallel text retrieval system is developed in the C language, using the MPI library as the communication interface.

5.1 Scalability

In this section, we will discuss experimental results that we obtained by changing the number of processors, the query count and the number of terms in the queries. An excessive number of experiments are conducted on the *Radikal* data set. Here, we only present some significant ones.

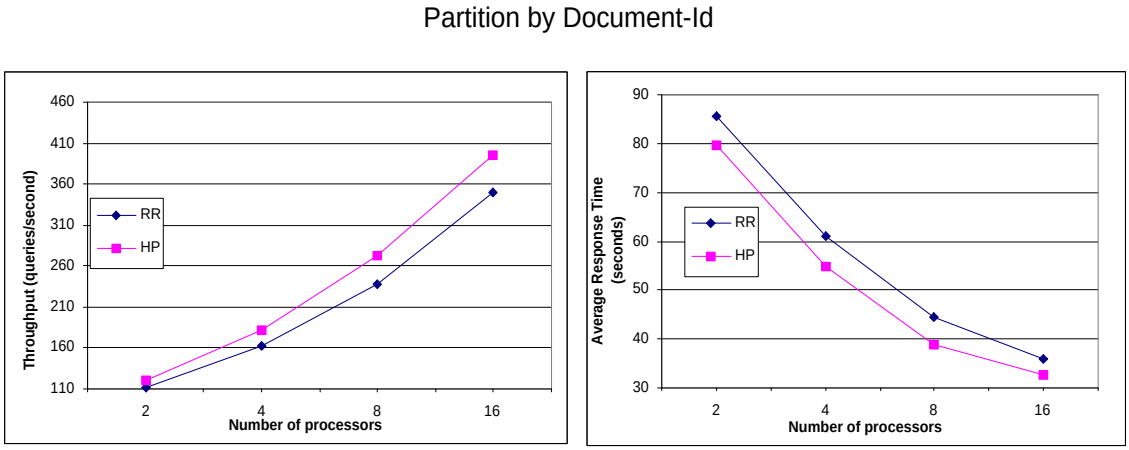


Figure 5.1: 18,000 distinct terms of the collection is sent in a query set.

5.1.1 Document-Id Partitioning

As mentioned earlier, in document-id partitioning the essential problem is that the number of disk accesses may be quite larger compared to term-id partitioning. The performance difference between round-robin partitioning and hypergraph-theoretical partitioning [4] can be observed best, when all the terms of the collection are sent to the system once in a query set. However, as our system capacity does not allow to send all the terms of the collection, we generate a query set formed by 18,000 distinct terms of the collection, where all queries have a single term. Figure 5.1 shows that hypergraph-theoretical partitioning (HP) has a better performance than round-robin partitioning (RR), with this query set. Average response time is the time elapsed while answering a single query on the average. In both partitioning models, the system performance improves with the increase in the number of processors, when both the system throughput and average response time are considered. Although the *Radikal* data set is not so large, query processing in parallel improves the system performance. This can be observed in the forthcoming figures as well.

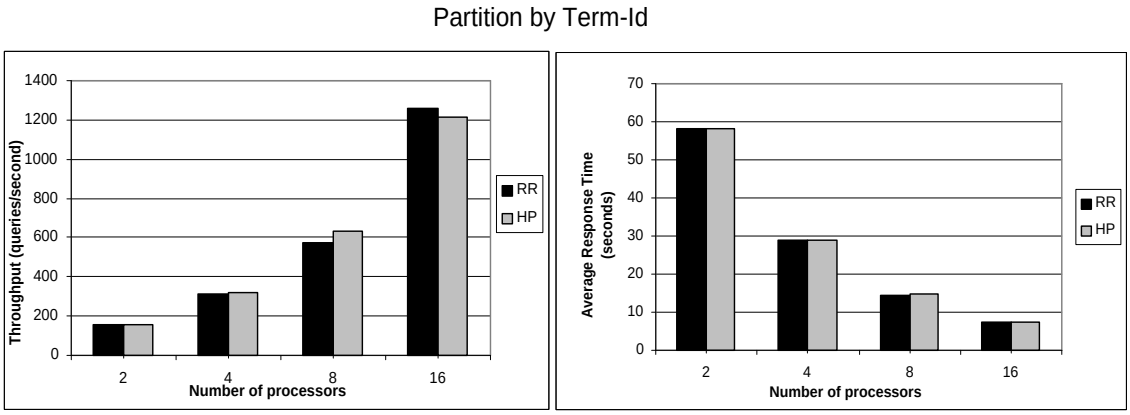


Figure 5.2: 18,000 distinct terms of the collection is sent in a query set.

5.1.2 Term-Id Partitioning

In term-id partitioning, there is no significant performance difference between round-robin and hypergraph-theoretical partitioning schemes with the query set that includes 18,000 distinct terms. The system throughput and average response time is depicted in Figure 5.2 for both round-robin and hypergraph-theoretical partitioning, as the number of processors increases. As previously stated, the total number of disk accesses is minimum in term-id partitioning, whereas the total communication volume may be larger compared to the document-id partitioning. In order to minimize the total communication cost, the hypergraph-theoretical model tries to cluster more related terms on the same disks. In this experiment, since all the queries are composed of a single term, the system could not completely make use of the hypergraph-theoretical distribution, where related terms are allocated together on the same disks. That is, index servers sent their partial answer sets to the central broker through the network and the size of the partial answer sets for a single term could not be reduced in partitioning by term-id. However, the balanced distribution of the postings may increase the system performance.

In general, we could not observe the performance of hypergraph-theoretical term-id partitioning very well in the experiments. The main reason for this is that the data set is small in size. In term-id partitioning, the performance of the system

is affected considerably by the size of the data set. As previously explained, Figure 3.4 displays query processing in document-id partitioning. It is basically the same for term-id partitioning when the general structure is considered. Index servers process the subquery terms and send their partial answer sets through the network to the central broker. When the data is small, reading the posting lists of the terms does not take much time, i.e. the I/O time is not long. Also, as in term-id partitioning the total number of disk seeks is minimum, index servers complete the processing of the subqueries in a rather short time. Since the processing time is short, as more and more queries are sent to the system, index servers will send continuously their partial answer sets to the central broker. Although our central broker has a higher capacity compared to index servers, it cannot handle all the partial sets at the same time, so partial answer sets begin to accumulate in the queues and network becomes a bottleneck. In term-id partitioning, the goal of the hypergraph-theoretical model is to reduce the total volume of communication, but as the network becomes a bottleneck, the performance of the model cannot be observed well.

Another important point is that, as explained earlier, the queries are constructed in accordance with the patterns of the terms in the whole collection. Also, we try to build a query with related terms, by choosing the terms from the same document while constructing a query with more than one term. In real systems, user queries consist of related terms. For example, the queries in the form {*Football, Network, Freud*} are not too frequent. However, taking the terms from the same document may not always work to construct a query with related terms. In term-id partitioning, since hypergraph-theoretical partitioning allocates more related terms on the same storage sites, the system performance could be improved even more when real system queries are sent to the system. Concisely, we expect better performance from hypergraph-theoretical partitioning in real systems, as the collection size is much larger and queries are meaningful, i.e. the terms in a query are closely related to each other.

In order to observe the performance difference between round-robin and hypergraph-theoretical term-id partitioning, we submit a single document, which has 227 terms, to the system as a query. This kind of queries are used in text

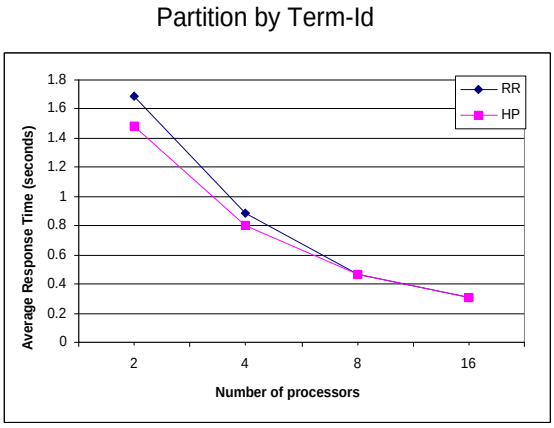


Figure 5.3: A single document is sent as a query.

categorization systems. Figure 5.3 shows the performance comparison of round-robin and hypergraph-theoretical partitioning. Hypergraph-theoretical partitioning performs a little better than round-robin partitioning. The performance difference could be observed better when a document with more terms is questioned in a larger collection.

5.2 Skewness

In this section, the experiments conducted with synthetic data sets and query sets will be presented. We will investigate the effect of the skewness of the data sets and the query sets on the performance of hypergraph-theoretical document-id and term-id partitioning.

5.2.1 Document-Id Partitioning

Our data sets contain 200,000 distinct terms (T) and 100,000 distinct documents (D). Each document consists of 100 terms (W). With these parameters we generated four data sets, which are uniform (*50-50*), low skewed (*60-40*), medium skewed (*70-30*) and high skewed (*80-20*). Recall that, *80-20* skewness means that 20% of the terms comprise 80% of the posting entries, which is explained in

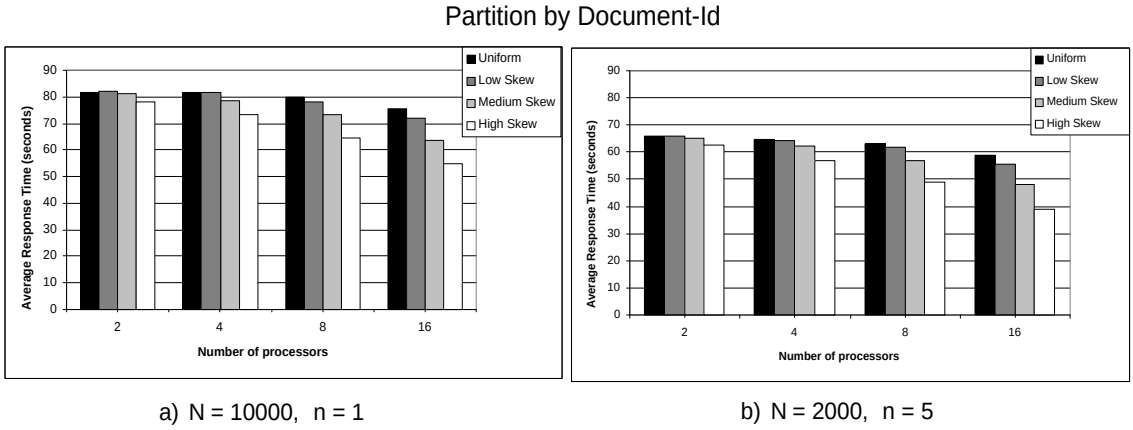


Figure 5.4: The effect of uniform term distribution in a query set.

Section 4.1.1.2.

Figure 5.4 shows the performance of document-id partitioning for two uniform query sets. The first query set includes 10,000 queries (N=10000), each with a single term (n=1), which is depicted in Figure 5.4-a. The second query set consists of 2,000 queries, and each query has five terms in it, as shown in Figure 5.4-b. For both uniform query sets, the performance of hypergraph-theoretical document-id partitioning improves, as the skewness of the data sets increases. The reason why partition by document-id has better performance on more skewed data sets is based on the way that the hypergraph-theoretical model follows to partition a data set. Recall that, the goal in this partitioning scheme is to minimize the number of terms that appear on several disks by clustering the related documents on the same disks. By this way, the total number of disk accesses can be reduced. In a skewed environment, while the documents include a small portion of the terms of the collection very frequently, a large portion of the terms appear infrequently in the documents. So, nearly all documents are related in the sense that most of them include the same terms. Therefore, while the hypergraph-theoretical model clusters the related documents together, the documents are differentiated by the infrequent terms. With this partitioning scheme, the documents that include the same infrequent terms are allocated on the same disks. In a uniform query set, each term has the same access probability. When an infrequent term is queried

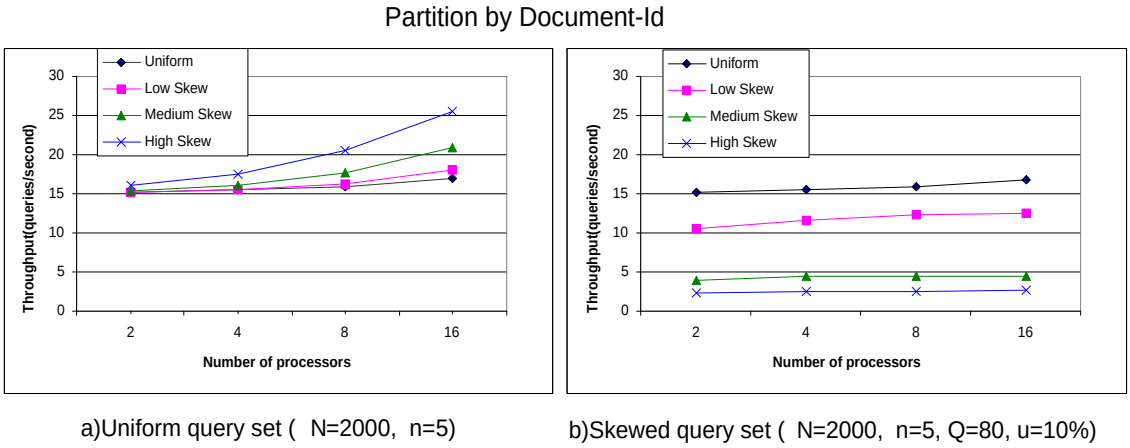


Figure 5.5: Comparison between uniform and skewed query sets.

in this partitioning scheme, the number of disk accesses can be reduced considerably, since the postings of this term are clustered on the same disk. However, in a uniform environment, all terms have the same frequency in the collection. So, while clustering the related documents, a term may appear on several disks.

Also, when we compare Figure 5.4-a with Figure 5.4-b, we realize no significant performance difference between them. In fact, instead of sending the terms one-by-one in the queries, when we submit them together in a query, the total volume of communication decreases, since the size of the partial answer sets sent through the network decreases. However, in document-id partitioning the total volume of communication is minimum. So, the network cost is less important with respect to the disk cost in document-id partitioning. Therefore, a reduction in the network cost does not affect the system performance considerably.

Lastly, we analyze the effect of skewed term distribution in a query set, and compare it with a uniform query set. We generated a high skewed (*80-20*) query set, where the parameter u is set to 10%. The generation of the skewed query set is explained in Section 4.1.2.2. Our skewed query set consists of 2,000 queries, where each query has five terms in it. As can be seen from Figure 5.5, the throughput of the system is much less with the skewed query set than the one with the uniform query set. Also, the performance of the system does not improve as the number of processors increases with the skewed query set. Such a query

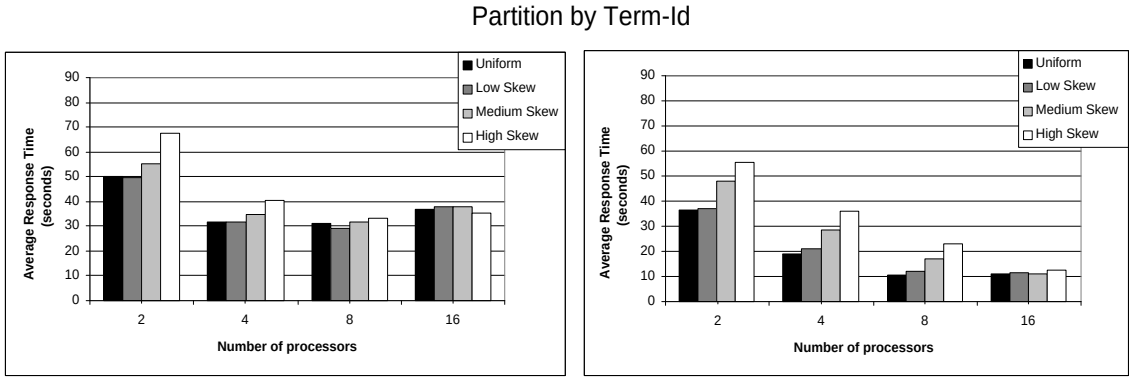


Figure 5.6: The effect of uniform term distribution in a query set.

set includes only a small portion of the terms. So, only a few index servers, that have the postings of these terms are active in query processing. Therefore, the increase in the number of processors does not improve the system performance. In addition, the performance degradation becomes much severe as the skewness of the data set increases. Recall that, in a high skewed environment, a small portion of the terms comprise a large portion of the posting lists. So, in more skewed environments, the posting lists of the frequent terms grow larger in size. Since, the skewed query model includes frequent terms heavily, the size of the posting lists that are retrieved are much larger compared to the less skewed environments. That is why we observe such a severe performance degradation.

5.2.2 Term-Id Partitioning

In the experiments that we performed with hypergraph-theoretical term-id partitioning, we used the same data sets and query sets that we generated for the experiments of document-id partitioning.

Figure 5.6 shows the performance of term-id partitioning for two uniform query sets. For both uniform query sets, the performance of term-id partitioning decreases, as the skewness of the data sets increases. In more skewed environments, the size of the posting list varies a great deal, as the frequency of the

terms in the collection differs to a large extent. So, when the partitioning is based on the term-ids, to have balanced storage, the number of terms that the disks include can change in amount. Since, in a uniform query set, each term has the same access probability, the disks that contain more terms are utilized more. The unbalanced disk utilization causes a decrease in the system performance.

Also, we observe a considerable performance difference between the experiments depicted in Figure 5.6-a and 5.6-b. As discussed in the previous section, the total volume of communication is less when we submit the terms together. In term-id partitioning, the total volume of communication may be quite large in the system. So, a reduction in the network cost improves the system performance significantly.

Another important point to mention in Figure 5.6-a is that the performance of the system does not improve, as the parallel system works with more than four processors. As discussed previously, in our system, the network becomes a bottleneck, when too many queries are sent to the system. Compared to document-id partitioning, term-id partitioning is affected more by the network congestion, since the total volume of communication may be quite large in term-id partitioning scheme. As more processors are involved in the system, the number of partial answer sets sent through the network increases. So, the rise in the number of processors increases the network congestion more, after the point that the network becomes a bottleneck. This can be observed in Figure 5.6-b, when the number of processors increases from eight to sixteen. Since the size of the partial answer sets sent in this experiment is less, the congestion at the network starts, when more processors are involved in the system.

Finally, we analyze the effect of uniform and skewed query sets, shown in Figure 5.7-a and 5.7-b. The performance of the system degrades with the skewed query set, and the increase in the number of processors does not improve the system performance. As we discussed for the experiments of document-id partitioning, with the skewed query set, the terms that appear most frequently in the documents are queried, so only the index servers that have these terms are involved in the processing of the queries. Also, in a skewed environment, since the

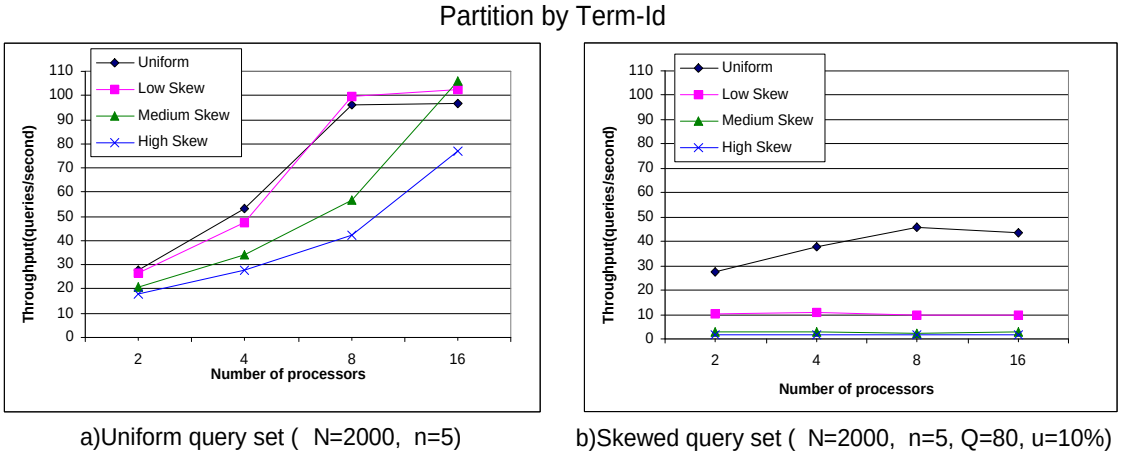


Figure 5.7: Comparison between uniform and skewed query sets.

posting lists of these terms are larger, the performance of the system decreases more.

5.3 Document-Id versus Term-Id Partitioning

All experiments that we conducted so far show that the performance of term-id partitioning is considerably better than document-id partitioning. The main reason is that the data size is small and consequently, the posting lists are short. So, the partial answer sets transmitted through the network are small in size. The total time elapsed at the network to transfer the partial answer sets becomes much less important compared to the time that elapsed by the disks to access and read the posting lists. Since network cost is less than disk cost in our system architecture. Recall that, the *disk seek time* constitutes the highest cost while reading a data from the disk, which is explained in Section 4.5. Also, the total number of disk accesses is minimum in term-id partitioning, whereas in document-id partitioning, the total volume of communication is minimum. Since the disk cost dominates the network cost, the increase in the performance of the system is higher by the minimization of the disk cost compared to the performance improvement by the minimization of the network cost. However, note that the main reason in this performance difference is based on the small size of the data

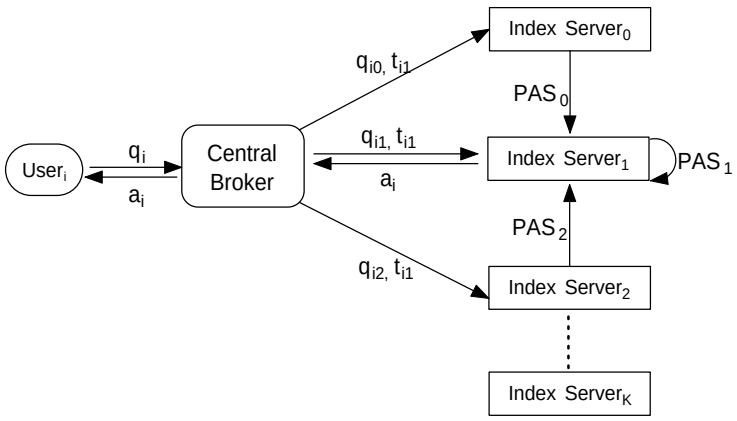


Figure 5.8: An alternative system structure.

that we use in our system.

Document-id partitioning could perform better in a system, where the network cost dominates the disk cost. Since, in this case, the minimization of the total volume of communication appears much more important in the improvement of the system efficiency. Besides this, in a system, where the posting lists are kept in the memory, the document-id partitioning could perform much better than term-id partitioning.

5.4 An Alternative System Structure

In order to reduce the congestion in the network, a different system structure can be designed for our text retrieval system. Figure 5.8 shows a possible system structure demonstrated by a simple example to make the representation more clear. In this system, index servers communicate with each other also. When a user query is inserted into the system, the central broker determines which index servers hold the corresponding posting lists and additionally, it also determines which index server will be responsible to form the final answer set. Accordingly, the central broker sends the subqueries to the related index servers (q_{i0} is the subquery of q_i corresponding to index server 0, and the tag t_{i1} means that index server 1 will be responsible for the final answer set of q_i). In this example, only

index servers 0, 1 and 2 have the posting lists corresponding to the q_i query and the central broker assigns index server 1 to form the final answer set of the query. So, when index servers 0 and 2 finish to process the subqueries, they send their partial answer sets to index server 1. Index server 1 constructs the final answer set by merging these partial answer sets with its own partial answer set and transfers the final answer set to the central broker. By using this structure, the congestion in the network can be reduced. In such a system, considerable performance improvement can be observed in parallel query processing.

Chapter 6

Conclusion

In this thesis, we have designed and implemented a parallel text retrieval system on a shared-nothing architecture. We have investigated the performance of query processing in our parallel text retrieval system. In our system, we adapted the inverted indices as our indexing mechanism and the vector-space model to rank the relevance of the documents to a query. Our main focus was on the inverted index organizations for efficient parallel query processing.

Two inverted index partitioning schemes are presented, which are mainly used for parallel text retrieval systems: Document-id and term-id partitioning schemes. The choice of these index partitioning schemes heavily depends on the advantages that they bring. Document-id partitioning achieves naturally the minimum total volume of communication in the system. On the other hand, in term-id partitioning the total volume of disk accesses is already equivalent to the lower bound achieved by the sequential algorithm. However, there are some weaknesses of both partitioning schemes. In document-id partitioning scheme, the total number of disk accesses may be quite large, whereas in partitioning by term-id, the total volume of communication of the system may increase a great deal. In the literature, the partitioning based on these schemes is done in a round-robin fashion. Round-robin partitioning schemes do not consider the problems associated with document-id and term-id partitioning schemes. We have investigated the

partitioning schemes that try to solve these problems. In document-id partitioning, our objective was to minimize the number of documents that belong to several disks in order to reduce the total number of disk accesses in the system. In term-id partitioning scheme, our goal was to minimize the number of terms that appear on several disks to reduce the total volume of communication in the system. The hypergraph-theoretical partitioning model meets these objectives by clustering more related documents on the same disks in document-id partitioning, and by allocating more related terms on the same disks in term-id partitioning.

In addition, we have analyzed the performance of round-robin and hypergraph-theoretical models through an implementation executed on a real-life data set. We have also studied the effect of the skewness of the synthetic data sets and query sets on the performance of hypergraph-theoretical document-id and term-id partitioning schemes. The experimental results and the impact of the parallel text retrieval system structure are discussed in Chapter 5. In that chapter, we have also proposed an alternative system structure in order to decrease the congestion at the network that we have observed after a point of time in our experimentations. As a further study of this thesis, this alternative system can be implemented as the underlying structure of our parallel text retrieval system.

Bibliography

[1] A. Arasu, J. Cho, H. Garcia-Molia, A. Paepcke, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technology*, 2001.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Publishing, Wokingham, UK, 1999.

[3] B. Cahoon and K. McKinley. Performance evaluation of a distributed architecture for information retrieval. In *Proceedings of the 19th Annual International SIGIR on Research and Development in Information Retrieval*, pages 110–118, August 1996.

[4] B. B. Cambazoglu. *Ph.D. Thesis, in Preparation*, 2003.

[5] P. Efraimidis, B. Mamalis, P. Spirakis, and B. Tampakas. Parallel text retrieval on a high performance supercomputer using the vector space model. In *ACM SIGIR conference on research and development in information retrieval*, pages 58–66, 1995.

[6] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. on Office Information Systems*, 2(4):267–288, October 1984.

[7] M. J. Flynn. Very high-speed computing systems. In *Proceedings IEEE*, volume 54, pages 1901–1909, 1966.

[8] W. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* PrenticeHall, 1992.

[9] O. Frieder, D. A. Crossman, A. Chowdhury, and G. Frieder. Efficiency considerations for scalable information systems. *Journal of Digital Information*, April 2000.

[10] D. Hawking. Padre-a parallel document retrieval engine. In *Proceedings of the Third Fujitsu Parallel Computing Workshop*, 1994.

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

[12] D. E. Knuth. *The Art of Computer Programming:Sorting and Searching.* Addison-Wesley, Massachussetts, 1973.

[13] V. Kumar, A. Grama, A. Gupta, and G. Karypris. *Introdution to Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[14] A. Macfarlane, S. Robertson, and J. Maccan. Parallel computing in information retrieval an updated review. *Journal of Documentation*, 53(3):274–315, 1997.

[15] B. Mamalis, O. Spirakis, and Tampakas. Parallel techniques for efficient searching over very large text collections. In *Proceedings of the 5th Text Retrieval Conference (TREC-5)*, 1996.

[16] S. Nilsson and M. Tikkanen. Implementing a dynamic comprossed trie. In *Proceedings WAE'98*, pages 2–22, August 1998.

[17] L. Page and S. Brin. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh World-Wide Web Conference*, 1998.

[18] B. A. Riberio-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the 3rd ACM International Conference on Digital Libraries*, pages 182–190, June 1998.

[19] G. Salton. *Information Retrieval: Data Structures and Algorithms.* Addison-Wesley, Massachussetts, 1989.

[20] G. Salton and C. Buckley. Parallel text search methods. *Communications of the ACM*, 31(2):202–215, February 1998.

[21] G. Salton and M. McGill. *Introduction to Modern Information Retrieval: Data Structures and Algorithms.* McGraw Hill, New York, 1983.

[22] B. Salzberg. *File Structures: An Analytic Approach.* Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1988.

[23] C. Stanfil. Partitioned posting files: A parallel inverted file structure for information retrieval. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 413–428, 1990.

[24] C. Stanfil, R. Tau, and D. Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of ACM-SIGIR Conference*, pages 88–97, 1989.

[25] ABC Search Engine. http://borg.cs.bilkent.edu.tr/ pirs/.

[26] Google homepage. http://www.google.com/.

[27] A. Tomasic and H. G. Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1993.

[28] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufman Publ., San Francisco, 1999.

[29] W. Y. P. Wong and D. L. Lee. Implementations of partial document ranking using inverted files. *Information Processing and Management*, 29(5):647–669, October 1993.