PERFORMANCE COMPARISON OF QUERY EVALUATION TECHNIQUES IN PARALLEL TEXT RETRIEVAL SYSTEMS

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

A. Aylin Tokuç

September, 2008

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.

Prof. Dr. Ayhan Altıntaş

Approved for the Institute of Engineering and Science:

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

ABSTRACT

PERFORMANCE COMPARISON OF QUERY EVALUATION TECHNIQUES IN PARALLEL TEXT RETRIEVAL SYSTEMS

A. Aylin Tokuç
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2008

Today's state-of-the-art search engines utilize the inverted index data structure for fast text retrieval on large document collections. To parallelize the retrieval process, the inverted index should be distributed among multiple index servers. Generally the distribution of the inverted index is done in either a term-based or a document-based fashion. The performances of both schemes depend on the total number of disk accesses and the total volume of communication in the system.

The classical approach for both distributions is to use the Central Broker Query Evaluation Scheme (CB) for parallel text retrieval. It is known that in this approach the central broker is heavily loaded and becomes a bottleneck. Recently, an alternative query evaluation technique, named Pipelined Query Evaluation Scheme (PPL), has been proposed to alleviate this problem by performing the merge operation on the index servers. In this study, we analyze the scalability and relative performances of the CB and PPL under various query loads to report the benefits and drawbacks of each method.

Keywords: parallel text retrieval, central broker query evaluation, pipelined query evaluation, term-based distribution, document-based distrribution.

ÖZET

PARALEL METİN ERİŞİM SİSTEMLERİNDE SORGU İŞLEME TEKNİKLERİNİN KARSILAŞTIRILMASI

A. Aylin Tokuç
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2008

Günümüz modern ağ arama motorları, büyük döküman kolleksiyonlarında hızlı metin erişimi yapabilmek için ters dizin yapısını kullanırlar. Erişim işleminin paralalleştirilmesi için ters dizinin, dizin sunucular arasında dağıtılması gerekmektedir. Ters dizinin dağıtımı genellikle terim-bazlı ya da döküman-bazlı olarak yapılır. Her iki dağıtım şeklinin de performansı sistemdeki toplam disk erişimi sayısına ve toplam iletişim hacmine bağlıdır.

Paralel metin erişiminde klasik yöntem her iki dağıtım yöntemi için de Merkezi Simsar Sorgu İşleme Yöntemi'ni kullanmaktır. Bu yöntemde merkezi simsarın birleştirme işlemlerinden dolayı çok yüklenerek işlem hızını belirleyen darboğaz konumuna geldiği bilinmektedir. Yakın geçmişte birleştirme işleminin dizin sunucularda gerçekleştirilmesine dayalı, Boru Hattı Sorgu İşleme Yntemi alternatif bir metod olarak önerilmiştir. Bu çalışmada Merkezi Simsar ve Boru Hattı Sorgu İşleme Yöntemleri'nin ölçeklenebilirlik ve göreceli performanslarını çözümleyip, değişken sorgu ağırlıklarında lehte ve alehte özelliklerini ortaya çıkaracağız.

Anahtar sözcükler: paralel metin işleme, merkezi simsar sorgu işleme, boru hattı sorgu işleme, terim-bazlı dağıtım, döküman-bazlı dağıtım.

To my mother...
who gave and taught me unconditional love and acceptance.

Acknowledgement

First of all, I would like to thank my advisor Prof. Dr. Cevdet Aykanat for his valuable guidance and help throughout this thesis. I would also like to thank Prof. Dr. Özgür Ulusoy and Prof. Dr. Ayhan Altıntaş for taking the time to read and evaluate my thesis.

Very special thanks to Tayfun Küçükyilmaz: my officemate, my neighbor, my friend. He guided me both in academic and personal life, listened to my problems, spent his time to solve them, prepared me tea, let me enter his house and play with his cat whenever I wanted. I can never repay his kindness.

I appreciate Berkant Barla Cambazoglu for implementing the first version of ABC-Server, for taking time to consider our ideas and suggest his during the implementation of the newer version.

I am grateful to my officemates and friends: Özlem Gür, Cihan Özturk, Enver Kayaaslan, Gonenç Ercan, Erkan Okuyan, Izzet Baykan and Ata Türk for their companionship and comments. I also thank my dear friends from the department of computer engineering: Sengör Altıngövde for his great ideas and critiques about the topic, Onur Küçüktunç for taking time to proofread this work, and to Hayrettin Gürkök for just being my friend.

I want to express my gratitude for Onur Yazıcı, who stayed up late online and listened to my complaints and for Özer Temeloğlu, who helped me in every way he could.

I also thank my parents, Zeynep and Özcan, my sister Ayça and the rest of my family for their understanding and support. This work would not have been possible without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The growing use of the internet has a significant influence 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 increasing very rapidly. The staggering increase in the data volume and query processing load create new challenges for text retrieval research. In order to satisfy user needs when large volumes of data is being processed, usage of parallel methods become inevitable. Parallel frameworks provide better average response times and higher throughput rates compared to sequential methods.

Most common method for storing large document collections is using inverted indexes. In the inverted index data structure, there is an associated list of documents for each term. These lists of documents are also called posting/inverted lists. To parallelize the retrieval process, the inverted index should be distributed among multiple index servers. The query responses are generated by combining the partial answer sets produced by the index servers.

In general, distribution of the inverted index can be performed in either document-based or term-based fashion. In both distributions, the responsibility of processing query terms and storing associated inverted lists is distributed among parallel processors. In document-based distribution, a set of documents in the dataset is assigned to a particular index server. In this distribution, during

query processing, each index server contributes to the final answer set by the similarities of the documents assigned to itself. Hence, each query must be sent to all index servers. The answer sets produced by the index servers are merged to form the final answer set. In term-based distribution, each inverted list is assigned to an index server. For each query, a subquery should be sent to the index servers containing at least one term within the query. Only the index servers receiving a subquery is required to respond with a partial answer set in order to compute the final answer set. In this distribution, the partial answer sets are not sufficient to decide whether a document is qualified to be in the final answer set or not. The results of all participant index servers should be accumulated since the terms of a document are scattered throughout separate index servers.

Both distributions have benefits and drawbacks. The performance of both distribution schemes depend on the total number of disk accesses and the total volume of communication in the system.

It is easy to divide the documents evenly across the index servers when document-based distribution is used, hence the storage cost is almost balanced. Furthermore, a query is sent to all index servers, and all index servers contribute to the final answer set causing a balanced workload, enabling maximum parallelism during the processing of a single query by inter-query parallelism. Another advantage of this scheme is that the index servers can compute the final answer sets, which reduces both the load over the central broker and the amount of intermediary communication. The most significant disadvantage of document-based distribution is that multiple disk accesses are required for a single query term.

On the other hand, in term-based distribution, during the processing of a query, only a related subset of index servers is required and utilized for generating the response. In this distribution, only a single disk access is required for a query term. When the system is loaded with sufficiently many queries, since each index server does not necessarily contribute to each query, it is possible to process several queries at once, utilizing the system throughput. The widely accepted disadvantages of term-based distribution are increased communication volume, heavy processing load on the central broker and possible imbalance on index server workloads.

The common approach for both distribution schemes is to have a central broker which divides user queries into subqueries, sends these subqueries to index

servers, and merges the answer sets in order to generate the final answer set. We refer to this scheme as the Central Broker Query Evaluation Scheme (CB). Since the central broker is heavily loaded and becomes a bottleneck, an alternative query evaluation schemetechnique, named Pipelined Query Evaluation Scheme (PPL), has been proposed by Moffat et al. [36] as an alternative to CB. In this scheme, query processing and merging of partial answer sets are performed in a distributed manner across all index servers.

Experimental results reported in a recent study [35] show that, even for small number of processors, a speed-down is observed on query throughput of CB and PPL. Moffat et al. [35] proposed using full system replication for increasing the query throughput rates. We do agree that it is hard to obtain scalable speedups for the CB and PPL schemes mainly because of the high communication-to-computation ratio in distributed query evaluation. However, we expect decent speedups on throughput rates for small-to-medium number of processors by utilizing appropriate algorithmic and implementation enhancements.

The objective of this paper is to investigate efficient parallelization of the CB and PPL. The relative performances of the CB and PPL under various query loads is explored. The pros and cons of each scheme are identified along with detailed implementation, scalability, and performance discussions.

The organization of this paper is as follows: in Chapter 2, we provide related work about parallel text retrieval, inverted index data structure and query processing. In Chapter 3, we present the basics of the query processing techniques we have investigated (CB and PPL). In Chapter 4, we present our experimental framework and analyze our results. Finally, in Chapter 5, we conclude and discuss the future directions of this study.

Chapter 2

Text retrieval problem

The growing use of the internet has a significant influence on text retrieval systems. The size of the text collection available on the internet is growing at an astonishing rate. It is very hard to access the data needed without the help of a text retrieval system. A text retrieval system processes user queries and outputs a set of documents related to the user query.

The data available on the internet is expanding very rapidly. As a result, the amount of data processed for answering a user query is also increasing. It is not a feasible solution to process this enormous data with the techniques used for small data collections as sequential full text search. A different representation of the dataset is needed for effective query processing. Until the early 90's, suffix arrays and bitmaps were sufficient to store the data available, and were used by a majority of text retrieval systems [14]. However, these data structures are not efficient and require large disk space for large scale collections. To alleviate the indexing problem, inverted index data structure [45, 51] is proposed. After its proposal, inverted index data structure replaced the other popular methods and became the de facto method for indexing large document collections.

2.1 Inverted index data structure

An inverted index contains an inverted list (also called posting list) for every term in the data collection. A posting is a pointer to a list of documents containing that term. For large collections, the inverted lists are stored on disk, but the index part generally fits into the main memory. Each posting p for term t_i consists of a document id field $p.d$ and a weight field $p.w$ for each document d_j containing t_i. $p.w$ is the result of the weight function [21] $w(t_i, d_j)$ and shows the relevance between t_i and d_j.

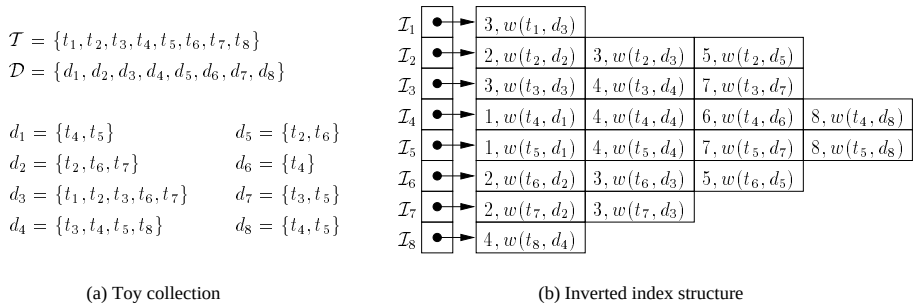

Figure 2.1: The toy document collection used throughout the paper (from [10]).

Fig. 2.1-a shows the document collection that we will use throughout the examples in the paper. There are 8 terms, 8 documents and 21 postings in this toy dataset. The inverted index built for this collection is shown in Figure 2.1-b.

2.2 Parallel Text Retrieval

Parallel text retrieval system architectures fall into two categories: inter-query parallel and intra-query parallel. In inter-query parallel systems processing of each query is handled by a single processor, whereas in intra-query parallel systems, multiple processors in the system actively takes place during the evaluation of a query. Both systems have advantages and disadvantages. Inter-query parallel systems are preferable for their better throughput rates, while intra-query parallel architectures obtain better average response times. Further details of the comparison between these architectures are provided in [42, 4].

In this work, we focus on intra-query parallel text retrieval systems on shared-nothing parallel architectures.

2.3 Inverted Index Distribution

To set up an intra query parallel text retrieval system, the inverted index for the data collection should be distributed among index servers. The storage loads of the index servers should be considered in this process. Each index server should keep an approximately equal amount of posting entries. Let $\text{SLoad}(S_j)$ denote the storage load of index server S_j. If there are K index servers in the system and P postings in the dataset, then

$$\text{SLoad}(S_j) \simeq \frac{|P|}{K}, \qquad \text{for } 1 \leq j \leq K, \tag{2.1}$$

There are two mainly accepted ways of performing the distribution of the inverted index: term-based distribution, also known as global index organization, and document-based distribution, also know as local index organization [34].

In the term-based distribution, the inverted lists for terms are distributed among the index servers. In this technique, all index servers are responsible for processing their own set of terms, that is, inverted lists are assigned to index servers atomically. A query is sent only to index servers containing terms of that query in its local index. Since different terms reside in different index servers, the probability of utilizing different index servers by different queries is very high, allowing high intra-query concurrency in processing. But since only partial scores for the documents are calculated on index servers, this distribution leads to high communication volume in the system. Also, updating a term-based distributed index is a nontrivial problem.

As an alternative to term-based distribution, the inverted index can be partitioned in a document-based fashion. In document-based distribution, each index server contains a portion of the document collection and an index server stores only the postings that contain the document identifiers assigned to it. Each query is sent to all index servers. This strategy reduces the volume of communication by computing the final similarity scores on index servers but requires more disk

seek operations. Also, it is easy to divide the documents evenly across the index servers when document-based distribution is used.

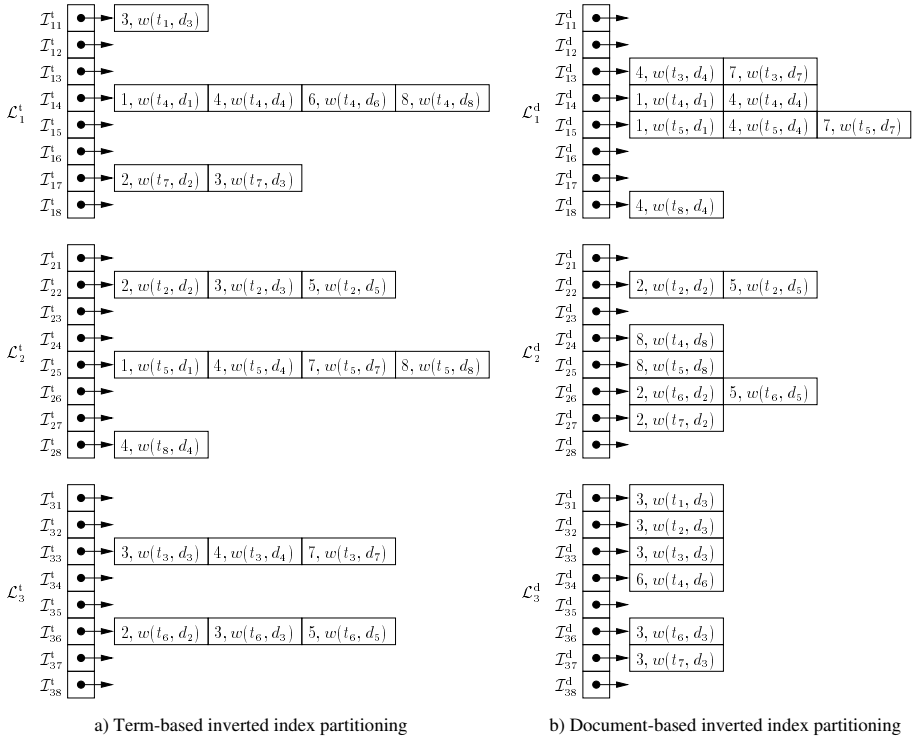

Figure 2.2: 3-way term-based and document-based distribution of our toy inverted index (from [10]).

The term-based and document-based distribution strategies are illustrated on our toy document collection for a 3-processor parallel text retrieval system in Figure 2.2-a and Figure 2.2-b [10]. The postings are assigned to index servers according to term and document ids in a round robin fashion, as in [44].

There is a wide literature on inverted index distribution problem in parallel text retrieval systems starting from early 90's. Tomasic and Garcia-Molina [44] and Jeong and Omiecinski [27] are the early papers evaluating term-based distribution versus document-based distribution of indexes that come into prominence.

Four different methods to distribute an inverted index on a shared-nothing system with different hardware configurations are discussed in [44]. Term- and document-based distribution of the index correspond to the system and disk organizations described in the paper. Performance of the system is measured employing simulation over a synthetic dataset. Similarities of documents and

queries are calculated using the boolean model. They concluded that document-based distribution performs better when there are longer documents in the data collection, whereas term-based distribution is better for document collections containing short terms.

The performance of term- and document-based distributions is measured on a shared-everything multiprocessor system with multi disks in [27]. They worked on a synthetic dataset and used boolean model to evaluate the similarities. They focused on term skewness in their experiments. Two heuristics for load balancing in term-based distribution is proposed. In the first heuristic, inverted index is distributed focusing on the posting sizes instead of number of terms, i.e., the inverted index is distributed with equal posting sizes instead of equal number of terms. In the second heuristic, the term frequencies are considered along with the posting sizes. According to their simulation results, term-based distribution is superior if the distribution of terms is less skewed in the dataset whereas document-based distribution is better otherwise.

MacFarlen et al. [33] explored the effect of distribution method on a text retrieval system. They used a probabilistic model to compute similarity values. They concluded that document-based distribution is performing better in their framework.

Baeza-Yates and Ribeiro-Neto [41] also applied term- and document-based distribution schemes on a shared-nothing parallel system. They used vector space model for document ranking and worked on a real life dataset. Their results show that term-based distribution performs better than document-based distribution in the presence of fast communication channels, opposing the conclusions of [44, 27, 33]. Bardue et al. [3] also confirm their results.

Cambazoglu et al. [10] conducted experiments on a 32-node shared-nothing PC cluster. Their results show that term-based distribution yields better throughput for batch query processing. They also note that document-based distribution should be preferred if the queries are submitted infrequently.

Interested reader should refer to excellent tutorial by Zobel and Moffat [50], which contains a very nice and extensive survey of studies on index distribution problem together with the explanation of many key techniques used in indexing.

2.4 Query Processing

In text retrieval, the main objective of query processing is to find out the relevant documents to a user query and displaying them to the user. Models such as boolean, vector space, fuzzy-set, and probabilistic have been proposed [48] in order to accomplish this goal. The vector space model, due to its simplicity, robustness and speed [42], is the most used and widely accepted model among others. In modern information retrieval systems, ranking based model replaces the boolean model because of its effectiveness and ability to sort out the retrieved documents.

The similarity of a user query is calculated for documents in the collection. A set of documents is returned to the user according to the result of these similarity calculations. This document set is sorted in decreasing order with respect to similarity to the user query.

To calculate the cosine similarity between a query $\mathcal{Q} = \{t_{q1}, t_{q2}, \ldots, t_{qQ}\}$ of size $\mathcal{Q}$ and the document d_j in a text retrieval system adopting vector space model, the formula

$$\mathrm{sim}(\mathcal{Q}, d_j) = \frac{\sum_{i=1}^{Q} w(t_{qi}, d_j)}{\sqrt{\sum_{i=1}^{Q} w(t_{qi}, d_j)^2}} \tag{2.2}$$

is adopted assuming all query terms have equal importance. The *tf-idf* (term frequency-inverse document frequency) score [42] is usually used to compute the weight $w(t_i, d_j)$ of a term t_i in a document d_j as

$$w(t_i, d_j) = \frac{f(t_i, d_j)}{\sqrt{|d_j|}} \times \ln \frac{D}{f(t_i)}, \tag{2.3}$$

where $f(t_i, d_j)$ is the number of times term t_i appears in document d_j, $|d_j|$ is the total number of terms in d_j, $f(t_i)$ is the number of documents containing t_i, and D is the number of documents in the collection.

To calculate the similarity measures, the parallel text retrieval system implemented in this work uses tf-idf together with the vector space model [48]. In a

traditional sequential text retrieval system, there are several stages in processing of a user query. Assume that a user query $\mathcal{Q} = \{t_{q1}, t_{q2}, \ldots, t_{qQ}\}$ is going to be processed. During the process, each query term t_{qi} is considered in turn. For each query term t_{qi}, inverted list I_{qi} is fetched from the disk. Then all postings in I_{qi} are traversed, and the weight $p.w$ of each posting p in I_{qi} is added to the score accumulator for document $p.d$. When all inverted lists for the query terms are processed, documents are sorted in decreasing order of similarity scores, and they are returned to the user in order of relevance.

To reduce the overheads in term-based query processing, a number of strategies are proposed. Exploring hypergraph partitioning to reduce the total volume of communication in the central broker while balancing the index storage on each processor [8], and user-centric approaches utilizing the query term frequency information to balance the query loads of index servers [35, 32] are among these strategies. The aim of PPL proposed by Moffat et al. [36] is also reducing the overheads in term-based distribution. In PPL, partial answers are transferred between index servers and only the final answer set is sent to the central broker. Their results show that the revised method is competitive with document-based distribution in terms of query throughput, but has problems with load balancing which has been shown to be a general issue [3].

2.5 Accumulator Limiting

In literature, ranking-based text retrieval [4, 17, 42, 48] is well-studied both in terms of efficiency [11, 30] and effectiveness [11, 13, 47]. Many optimizations are proposed [6, 22, 31, 38, 39, 43, 46, 49] to decrease the query processing times and to use the memory more effectively. These optimizations focus on either limiting the number of processed query terms and postings (short-circuit evaluation) [40, 23, 2] or limiting the memory allocation for accumulators (prunnig) [39, 37, 12]. The main differences between these optimizations are the processing order of postings and stopping conditions for processing.

Buckley and Lewit [6] proposed an algorithm which traverses query terms in decreasing order of frequencies and limits the number of processed query terms by not evaluating the inverted lists for high-frequency terms whose postings are

not expected to affect the final ranking. Harman and Candela [20] used an insertion threshold on query terms, and the terms whose score contribution are below this threshold are not allowed to allocate new accumulators. Moffat et al. [38] proposed two heuristics which place a hard limit on the memory allocated to accumulators. Turtle and Flood [46] presented simulation results for the performance analysis of two optimizations techniques, which employ term ordered and document ordered inverted list traversal. Wong and Lee [49] proposed two optimization heuristics which traverse postings in decreasing magnitude of weights.

For a similar strategy, Persin [39] proposed a method which prunes entries in inverted indexes and a query term's processing is stopped when a particular condition is met. Moffat and Zobel [37] then proposed that each posting list can have a different stopping condition, according to the size of posting list. They also stated that their accumulator limiting can achieve comparable effectiveness, even when 1 percent of the total accumulators are allowed for update. Altingovde et al. [1] also confirmed this result. During the evaluation of a query in an index server, to reduce the memory constraints and increase scalability, the concept of *accumulator limiting* is has been proposed in [50]. To reduce the number of accumulators, only documents with rare query terms are allowed to have an accumulator.

The optimizations for fast query evaluation can be classified as safe or approximate [46]. Safe optimizations guarantee that best-matching documents are ranked correctly. Approximate optimizations may trade effectiveness for faster production of a partial ranking, which does not necessarily contain the best-matching documents, or may present them in an incorrect order.

There exists a significant amount of related work in the field of database systems. The interested reader may refer to prior works by Lehman and Carey [29], Goldman et al. [18], Bohannon et al. [5], Hristidis et al. [24], Elmasri and Navathe [15], and Ilyas et al [25] for more information about fast query evaluation optimizations in database systems.

Chapter 3

Parallel Query Processing Schemes

The ABC-Server Parallel Text Retrieval System [10] is implemented in C using the LAM/MPI [7] library. In this work, it is running on a 48-node Beowulf PC cluster, located in the Computer Engineering Department of Bilkent University.

3.1 Central Broker Query Evaluation Scheme (CB)

CB is a master-client type of architecture. In this architecture, there is a single central broker, which collects the incoming user queries and redirects them to the index servers in the nodes of the PC cluster. The index servers are responsible from generating partial answer sets to the received queries, using the local inverted indices stored in their disk. The generated partial answer sets are later merged into a global answer set at the central broker, forming an answer to the query. Figure 3.1 displays the CB architecture.

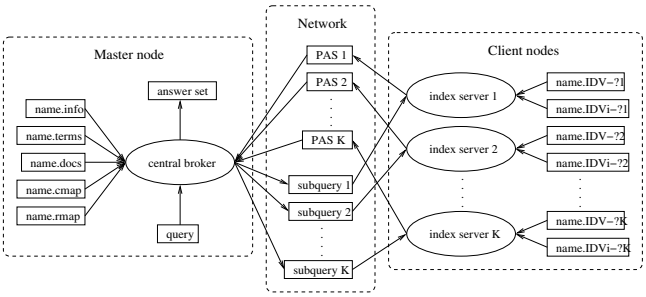

Figure 3.1: The architecture of the ABC-server parallel text retrieval system (from [8]).

3.1.1 Implementation Details

The psuedocodes of codes running on central broker and index servers are shown in Algorithm 1 and Algorithm 2 respectively. Further implementation details will be given by analyzing the steps of processing a query in this section.

A query is first read from a pre-created file by the query submitter interface. This interface spawns a number of child processes each of which concurrently submits queries to web interface of the central broker via network.

Before answering queries, the central broker initializes itself by creating some data structures. First of all it creates a *trie*(also known as radix tree or prefix tree), in which it keeps the terms and their ids. When a query is received from a user, the id of a query term is accessed in O(w) memory accesses where w is the length of that term. In this framework, a trie is preferred instead of a hash table because of its smaller memory requirement. An array to store the mapping information is also created. Finally, the central broker initializes a TCP port over which the queries will be submitted, after creating its receive buffers and allocating memory for statistical purpose data structures. Also the index servers initialize their send buffer and accumulator arrays as well as their data structures for statistical purposes. Both the central broker and the index servers use a queue while processing the user queries.

The central broker enqueues each incoming query to its queue as a query item. When the central broker dequeues a submitted query from the queue, it identifies the responsible index servers and records the number of index servers it is sending a subquery to. Then a packet is sent to responsible index servers. This packet

consists of the query id and the terms of the query.

Each index server periodically checks for incoming subqueries from the central broker. If a subquery is received, it is enqueued as a subquery item to the queue of index server. When it is dequeued from the queue, the index server inspects if any of the terms resides in its local term list. If no term appears in its local index, it replies with an empty packet to the central broker. If there are terms that belong to that index server's local index, then the index server reads its posting list and updates the scores of documents on its accumulator array. Each index server has a static accumulator array with size of the document collection. Another highly deployed technique for storing accumulator arrays is to use a dynamic accumulator array and to insert an accumulator for a document only if the weight field is larger than a predefined threshold (accumulator limiting). In ABC-Server implementation, accumulator limiting is not deployed since the main focus of this work is to compare the relative performances of CB and PPL.

When all terms of the subquery are processed, the index server selects top scored accumulators from the accumulator array by using expected linear time randomized selection algorithm. Selected accumulators are sorted to finalize the partial answer set. Then the prepared partial answer set is copied to the send buffer. However, if there is an ongoing send operation, this process is delayed until that particular finish operation is finalized. The static accumulator array is cleared for future use and the contents of the send buffer is sent to the central broker by a non-blocking send operation (Isend).

The central broker has receive buffers allocated for each index server and periodically checks them. If a partial answer set is detected, the contents of the receive buffer is inserted into the master queue.

When a partial answer set is dequeued from the queue, the central broker merges it with other partial answer sets received from other index servers. Details of the merge operation change based on the scheme used.

After the merge operation, central broker checks whether it is going to receive other partial answer sets for this query. If this is the last partial answer set, the merge operation produces the final answer set. Top s accumulators are extracted from the final answer set and they are displayed to the user.

Algorithm 1 Central Broker algorithm running on Master.

Require: CON: A port through which a client connects, Q: Master queue, q_k: Subquery of query q that will be sent to index server IS_k, $\prod = \{\mathcal{T}_1 \cap \mathcal{T}_2 \cap \ldots \mathcal{T}_k\}$: partitioning of document, collection among index servers, SEND: Non-Blocking send operation, IRECV: Non-Blocking receive operation, AS[q]: Answer set for query q.

```
1: for each index server IS_k do
2:   issue an IRECV
3: while true do
4:   TEST whether a query is received from a client over CON
5:   if TEST(CON) = true then
6:     for each query q received over con do
7:       ENQUEUE(Q,q)
8:   for each index server IS_k do
9:     TEST whether a message containing PAS is received
10:    if TEST = true then
11:      ENQUEUE(Q,(PartialAnswerSet_queryid))
12:      issue a new IRECV
13:  if Q ≠ ∅ then
14:    x ←DEQUEUE(Q)
15:    if type(x) = query then
16:      q ← x
17:      for each index server IS_k do
18:        q_k ← q ∩ T_k
19:        if q_k ≠ ∅ then
20:          subqueryProcessorCount(q) ← subqueryProcessorCount(q) +1
21:          SEND(q_k) to processor index server IS_k
22:    else
23:      ▷ type(x) = PartialAnswerSet
24:      PAS ← x
25:      MERGE partial answer set PAS with AS[PAS.queryId]
26:      subqueryProcessorCount(q) ← subqueryProcessorCount(q) −1
27:      if subqueryProcessorCount(q) = 0 then
28:        DISPLAY(AS[q]) to client via CON
```

Algorithm 2 Central Broker algorithm running on Index Servers.

Require: Q: Index Server queue, IndexServersList: List of nodes that are running the index server code, q_k: Subquery of query q that will be sent to index server IS_k, ISEND: Non-Blocking send operation, SendBuf: An accumulator array of size MAX SEND SIZE, Sending: boolean variable, PAS[q]: Partial answer set generated for query q, which holds (d_j,$score_j$) pairs, D_k: set of documents that reside on the inverted index of index server IS_k, $w_{i,j}$: weight of t_i in d_j calculated using tf-idf scheme, $score_j$: total score of document j, calculated in p, k: Requested number of documents per index server.

```
1: for each processor p ∈ IndexServersList do
2:   while true do
3:     TEST whether a message containing a subquery is received from central broker
4:     if TEST = true then
5:       ENQUEUE(Q,q_k)
6:     if Q ≠ ∅ then
7:       q ←DEQUEUE(Q)
8:       for each t ∈ q_k do
9:         for each d ∈ D_k do
10:          if t_i ∈ d_j then
11:            compute w_{i,j}
12:            score_j ← score_j + w_{i,j}
13:      PAS[q] ← SELECT top k documents from PAS[q] according to their score
         fields
14:      SORT PAS[q] according to their document id fields
15:      if Sending then
16:        Wait for previous send to finish
17:        Sending = FALSE
18:      Copy PAS[q] to SendBuf
19:      Sending = TRUE
20:      ISEND SendBuf to central broker
```

3.1.2 Term-based distribution

In parallel query processing, initial distribution of the inverted index is kept in a term-to-processor mapping array. The central broker creates an array to store this mapping information. If term-based distribution is used, the mapping array is accessed with the id of the term and returns the id of the processor possessing that term.

Upon reception of a query, the central broker parses the query terms to find out their ids with the help of the trie and which processors they are mapped to using the term-to-processor array. Subqueries are sent only to responsible index servers, i.e., to the index servers which have at least one query term mapped to it. The subquery packet consists of the query id and the subset of the query terms residing on that index server.

In term-based distribution, in order to improve performance of ABC-Server system, communication volume is decreased by performing accumulator size restriction on the partial answer sets. A predefined number of accumulators with highes scores are selected employing expected linear time randomized selection algorithm. The selected accumulators are sorted using quicksort algorithm according to their document id fields. Sending a sorted accumulator list from the index servers enables the central broker to merge received partial answer sets in linear time. It is possible to merge received accumulators with the already existing ones upon arrival of a partial answer set(2-way merge), or merge all patrial answer sets at once(k-way merge).

3.1.2.1 2-way merge

If the received accumulators are the first partial answer set for that query, then they become the accumulators for this query and a new receive buffer is allocated. Otherwise, they are merged with the existing ones. Since both existing accumulators and received accumulators are sorted according to their document id fields, the merge operation is performed in linear time in the following way:

Space for merged accumulators is allocated. The length of this accumulator array equals to sum of the received and existing accumulator sizes. If this is the

first partial answer set received for that query, central broker does not perform any operation, it simply becomes the accumulators for the query. Otherwise, it is merged with the existing accumulators.

For the merge operation, there are two pointers, initially pointing to the beginning of the received and existing accumulator arrays, both of which are sorted according to document id fields. The doc id fields of both pointers are compared, and the one with the smaller id is copied into the merged accumulator array. If doc id field of both pointers are equal, then the score fields' sum is taken. Pointer of the processed accumulator list is advanced. When a pointer reaches to the end of a list, remaining accumulators of the other array are copied to the tail of the merged accumulators array. The merged accumulators become the accumulators for the current query at the end of the process. Received and previously existing merged accumulators are then deallocated.

This algorithm runs in linear time with respect to the size of accumulator array. But since it merges existing and received accumulators each time a new partial answer set is received, it should run $k - 1$ times in total to merge all k partial answer sets and form the final answer set. Also, it copies the data from an array to another each time it runs, causing an allocation/deallocation overhead.

3.1.2.2 K-way merge

K-way merge is an alternative to 2-way merge. Since it is known that central broker in CB becomes a bottleneck, the disadvantages of 2-way merge can be alleviated by using this approach.

In this technique, the central broker waits for the merge operation until all partial answer sets for the current query is received. Let k represent the number of received accumulator arrays. Space for merged accumulators is allocated. Length of this accumulator array is the sum of all k received accumulator sizes. There are k pointers pointing to the heads of the received arrays. As in 2-way merge, the pointer with the smallest doc id field is found, and it is copied into the merged accumulator array. If there are other accumulators in other lists with same doc id, their score fields are added to the score field of the merged accumulator. The processed accumulator pointers are advanced. This process is repeated until all

but one of the k list pointers reach to end. The last remaining lists unprocessed accumulators are copied to tail of the merged accumulators list. All k lists are deallocated.

This algorithm runs in O($k \times m$) time where m denotes the size of received accumulators. There is less allocation/deallocation overhead since all accumulators are merged at once. But all partial answer sets for a query waits for the last one to be received, causing an increase in memory requirements of the central broker.

The merged accumulators are stored in doc-id order. Top s accumulators are extracted from the final answer set using expected linear time randomized select algorithm with respect to the score fields of the accumulators. Then the extracted accumulators are sorted according to their score fields using quicksort. Extraction takes O($a + slogs$) time, where a is the number of accumulators in the final answer, and s is the number of answers to be retrieved.

In term-based distribution, accessing a term's inverted list requires a single disk access, but reading the list (i.e., posting I/O) may take a long time since the whole list is stored at a single processor. Similarly, the partial answer sets transmitted by the index servers are long. Hence, the overhead of term-based partitioning is mainly at the network, during the communication of partial answer sets. Especially, in cases where the partial answer sets are long or inverted lists keep additional information such as information on term positions, this communication overhead becomes a bottleneck.

3.1.3 Document-based distribution

If document-based distribution is used, subquery packet including term ids and the query id is sent to all index servers without mapping the terms to processors.

On an index server, when a subquery packet is dequeued from the queue, the index server inspects if any of the terms reside on its local term list since it may receive unrelated terms when document-based partitioning is used. If no terms are related to that index server, a packet only containing the query id is sent to the central broker.

For each index server containing at least one of the query terms, accumulators should be created as it is done in term-based distribution. Selection is performed over these accumulators, but since the accumulators created contains the final scores for those documents, selecting the top s accumulators is sufficient. Then these top s accumulators are sorted according to their score fields and sent to the central broker. Sending a sorted accumulator list from the index servers enables the central broker to merge received partial answer sets in linear time. 2-way merge and k-way merge can be performed to merge these accumulators.

3.1.3.1 2-way merge

If the received accumulators are the first partial answer for that query, then they become the accumulators for this query and a new receive buffer is allocated. Otherwise, they are merged with the existing ones. Since both existing accumulators and received accumulators are sorted, the merge operation is performed in linear time.

A space of length "s" is allocated for merged accumulators. Since received partial answer sets contain the final scores for documents, it is redundant to save the accumulators with scores smaller than the score of the s^{th} accumulator. If this is the first partial answer set received for that query, central broker does not perform any operation, it simply becomes the accumulators for the query. Otherwise, it is merged with the existing accumulators.

For the merge operation, there are two pointers, initially pointing to the beginning of the received and existing accumulator arrays, both of which are sorted according to score fields. The score fields of both pointers are compared, and the one with the larger value is copied into merged accumulators array. Pointer of the processed accumulator list is advanced. This process is repeated until there are s items in the merged accumulators array, or one of the accumulator arrays is fully processed. If the latter case occurs, the tail of the not fully processed list is added to merged accumulators list. The merged accumulators becomes the accumulators for the current query at the end of the process. Received and previously existing merged accumulators are deallocated.

This algorithm runs in linear time with respect to the size of accumulator

array. But since it merges existing and received accumulators each time a partial answer set is received, it should run $k-1$ times to merge all k partial answer sets and form the final answer set. Also, it copies the data from an array to another each time it runs,causing an allocation/deallocation overhead. The merged accumulator always keeps to top s accumulators received so far, so memory overhead of this algorithm is small compared to term-based 2-way merge.

3.1.3.2 K-way merge

K-way merge is an alternative to 2-way merge. Since it is known that central broker in CB becomes a bottleneck, the disadvantages of 2-way merge should be alleviated by using this algorithm.

In this technique, the central broker waits for the merge operation until all partial answer sets for the current query is received. Let k represent the number of received accumulator arrays. Space of size s for merged accumulators is allocated. There are k pointers pointing to the heads of the received arrays. As in 2-way merge, the pointer with the largest score field is found, and it is copied into the merged accumulators array. The processed accumulator pointers are advanced. This process is repeated until all but one of the k list pointers reach to end or the merged accumulators store s top accumulators. If the former case occurs, the last remaining lists unprocessed accumulators are copied to the tail of the merged accumulators list. When the top s accumulators are stored in the merged accumulators array, all k lists are deallocated.

This algorithm runs in O($k \times s$) time. There is less allocation/deallocation overhead since all accumulators are merged at once. But all partial answer sets for a query waits for the last one to be received, causing an increase in memory requirements of the central broker.

If document-based distribution is used, top s accumulators are already in score order in merged accumulators array and no further processing for extraction is necessary.

In document-based distribution, disk accesses are the dominating overhead in total query processing time, especially in the presence of slow disks and a fast network. $O(K)$ disk seeks are required in the worst case to read the inverted list

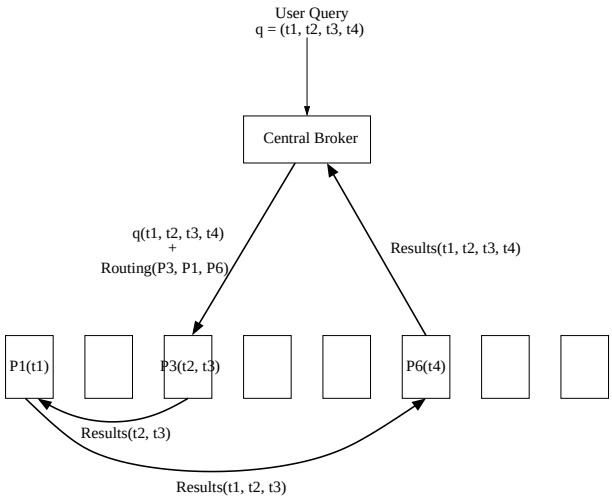

Figure 3.2: Pipelined query evaluation architecture.

of a term since the complete list is distributed at many processors. However, the inverted lists retrieved from the disk are shorter in length, and hence posting I/O is faster. Moreover, in case the user is interested in only the top s documents, no more than s accumulator entries need to be communicated over the network, since no document with a rank of $s+1$ in a partial answer set can take place among the top s documents in the global ranking.

3.2 Pipelined Query Evaluation Scheme (PPL)

The architecture of PPL is very similar to term-based distribution using 2-way merge. In this architecture, there is a single central broker, which collects the incoming user queries and redirects them with a routing information to the first index server in the list. The index servers are responsible from generating partial answer sets to the received queries, using the local inverted indices stored in their disk and forwarding these partial answer sets to the next index server in the route. The generated partial answer sets are later merged into a global answer set at the last index server of the route, forming an answer to the query. The final answer set is sent to the central broker to be displayed to user. Figure 3.2 displays the PPL architecture.

Algorithm 3 Pipelined algorithm running on Master.

Require: CON: A port through which a client connects, Q: Master queue, q_k: Subquery of query q that will be sent to index server IS_k, $\text{sq}_q[p]$: Subquery that will be sent to processor p for query q, $\prod = \{\mathcal{T}_1 \cap \mathcal{T}_2 \cap \ldots \mathcal{T}_k\}$: partitioning of document, SEND: Blocking send operation, IRECV: Non-Blocking receive operation, AS[q]: Answer set for query q.

1: **for** each index server IS_k **do**
2: issue an IRECV
3: **while** true **do**
4: TEST whether a query is received from a client over CON
5: **if** TEST(CON) = true **then**
6: **for** each query q received over con **do**
7: ENQUEUE(Q,q)
8: **for** each index server IS_k **do**
9: TEST whether a message containing AS is received
10: **if** TEST = true **then**
11: ENQUEUE(Q,(AnswerSet))
12: issue a new IRECV
13: **if** $Q \neq \emptyset$ **then**
14: $x \leftarrow$DEQUEUE(Q)
15: **if** type(x) = query **then**
16: $q \leftarrow x$
17: **for** each index server IS_k **do**
18: $q_k \leftarrow q \cap \mathcal{T}_k$
19: **if** $q_k \neq \emptyset$ **then**
20: subqueryProcessorCount(q) $\leftarrow$ subqueryProcessorCount(q) +1
21: ProcSendList(q) $\leftarrow$ ProcSendList(q) $\cup IS_k$
22: ORDER $\leftarrow$ SHUFFLE(ProcSendList)
23: **for** each index server $IS_k \in$ ORDER **do**
24: MSG $\leftarrow$ MSG $\cup \{k, q \cap \mathcal{T}_k\}$
25: SEND MSG to the first index server in ORDER
26: **else**
27: $\triangleright$ type(x) = AS
28: AS[g] $\leftarrow x$
29: DISPLAY(AS[q]) to client via CON

Algorithm 4 Pipelined algorithm running on Index Servers.

Require: Q: Index Server queue, IndexServersList: List of nodes that are running the index server code, sq_q: Subquery received about query q, ISEND: Non-blocking send operation, SendBuf: An accumulator array of size MAX SEND SIZE, Sending: boolean variable, PAS[q]: Partial answer set generated for query q, which holds (d_j,$score_j$) pairs, D: set of documents that reside on the inverted index of p, $w_{i,j}$: weight of t_i in d_j calculated using tf-idf scheme, $score_j$: total score of document j, calculated in p, k: Requested number of documents per index server.

```
1: for each processor running except myself do
2:   issue an IRECV
3: while true do
4:   for each processor running except myself do
5:     TEST whether a MSG containing a subquery and an ORDER is received from
       master ∨ a MSG containing a PAS and an ORDER is received from another
       IS
6:     if TEST = true then
7:       ENQUEUE(Q, MSG)
8:   if Q ≠ ∅ then
9:     MSG ←DEQUEUE(Q)
10:    for each t ∈ q_k in MSG do
11:      for each d ∈ D_k do
12:        if t_i ∈ d_j then
13:          compute w_{i,j}
14:          score_j ← score_j + w_{i,j}
15:    PAS[q] ← SELECT top k documents from PAS[q] according to their score
       fields
16:    SORT PAS[q] according to accumulators' document id fields
17:    MSG ← MSG \ {k, q_k}
18:    if Sending then
19:      Wait for previous send to finish
20:      Sending = FALSE
21:    if ORDER ≠ ∅ then
22:      Copy MSG and PAS[q] to SendBuf
23:      ISEND SendBuf to the next IS ∈ ORDER
24:      Sending = TRUE
25:    else
26:      Copy PAS[q] to SendBuf
27:      ISEND SendBuf to master
28:      Sending = TRUE
```

3.2.1 Implementation Details

The psuedocodes of codes running on central broker and index servers are shown in Algorithm 3 and Algorithm 4 respectively. Further implementation details will be given by analyzing the steps of processing a query in this section.

Before answering queries, the central broker first creates a *trie*, in which it keeps the terms and their ids. As in CB, the central broker creates an array to store the mapping information. The mapping array is accessed with the id of a term and it returns the processor having that term in its inverted list. Finally the central broker initializes a TCP port which the queries will be submitted over, after creating its receiving buffers and allocating memory for statistical purposed data structures. Also the index servers initialize their send buffer, receive buffers, and accumulator arrays as well as their data structures for statistical purposes. Both the central broker and the index servers use a queue while processing the user queries in all implementations.

A query is first read from a pre-created file by the query submitter interface. This interface spawns a number of child processes which concurrently submits queries to web interface of the central broker via network.

The central broker enqueues the incoming queries to its queue as a query item. When the central broker processes a submitted query from the queue, it identifies the responsible index servers. The central broker parses the query terms to find out their ids with the help of trie and which processors they are mapped to using the term-to-processor mapping array. An ordering of these index servers is created. This ordering becomes the routing order for that query. Different techniques can be adopted for the creation of the ordering, three of which are described below in detail.

3.2.1.1 Processor ordered routing

The list of responsible index servers' ids are sorted in increasing order to form the routing order. In this ordering, index servers with small ids rarely perform a merge operation while on the other hand the index servers with large ids suffer from the load of preparing the final answer sets. This approach causes an imbalance in

work loads. There is no possibility of deadlock in this routing order since a topological ordering of the index servers is same as the id ordering of them, and the graph is unidirectional and acyclic(DAG).

3.2.1.2 Random ordered routing

The list of responsible index servers' ids are sorted in increasing order to form the routing order. To shuffle this list, Fisher-Yates shuffle is adopted. Let there be r responsible index servers. While $r > 1$, a random number j between 1 and r is identified, the j^{th} element of the responsible index server list is swapped with the r^{th} element, and r is decremented by 1. This ordering balances the workloads of the index servers, but there is a possibility of deadlock if send/receive operations block the processors.

3.2.1.3 Random cyclic ordered routing

The list of responsible index servers' ids are sorted in increasing order. Let there be r responsible index servers. A random number between 1 and r is identified. Let j denote this random number. This query routes among the responsible index servers in the following order: $j, j+1, \ldots, r-1, r, 1, 2, \ldots, j-1$. This approach balances the workloads of the index servers, but there is a possibility of deadlock if send/receive operations block the processors.

A packet containing query id, query terms and routing order is prepared and sent to the first index server in the routing.

Each index server periodically checks for incoming subqueries from the central broker and partial answer sets from other index servers. If a packet is received, it is enqueued to the queue of index server.

When a packet is dequeued, the index server extracts the terms that are related to it, reads its posting list and updates the scores of documents on its accumulator array. Each index server has a static accumulator array with size of the document collection.

If the dequeued packet is received from another index server, the index server

merges its local partial answer set with the received partial answer set by adding the scores of the received accumulators to its static accumulator array.

If there are more index servers in the routing list, the index server selects top scored accumulators from the accumulator array by using expected linear time randomized selection algorithm. The selected accumulators are not sorted as done in CB, since the merge operation does not require a sorted list in PPL. If this index server is the last one in the routing, then prepares the final answer set by extracting top s accumulators from its static accumulator array and sorting them according to their score fields.

The extracted accumulators are copied to the sending buffer if there exists no ongoing send operation. The static accumulator array is cleared for future use and contents of the sending buffer is sent to its destination by non-blocking send operation(Isend).

The central broker has receive buffers allocated for each index server and periodically checks them. If an answer set coming from an index server is detected, the contents of the receive buffer is inserted into the master queue.

When an answer set is dequeued from the queue, the central broker displays it to the user.

3.3 CB vs PPL

Consider an example scenario where there are $K = 6$ processors, the inverted lists of the terms are held on processors $P_1(t_1)$, $P_3(t_2, t_3)$, and $P_6(t_4)$. A query with four terms, $q = (t_1, t_2, t_3, t_4)$, arrives to the system. In Fig, 3.3(a), we depict the behaviour of CB scheme for this scenario. Upon reception of a query, central broker determines the related index servers, which are P_1, P_3 and P_6 in this case, partitions the query into subqueries according to the term distribution and sends these subqueries to the related index servers. Index servers P_1, P_3 and P_6 evaluate these subqueries over their local index, and return the respective partial answer sets, which are then merged at the central broker to form the final answer set.

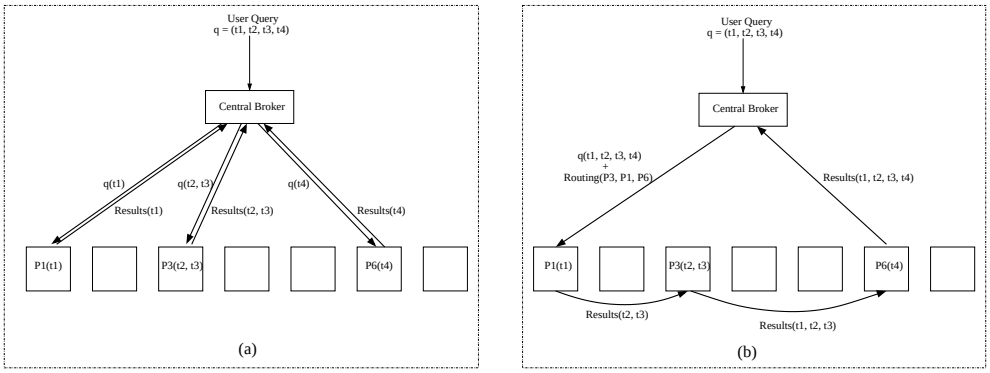

Figure 3.3: (a) Example for CB query evaluation. (b) Example for PPL query evaluation.

In the pipelined approach, if processor ordered routing order is used, evaluation of the query begins on P_1, which processes the list corresponding to term t_1 to produce an initial set of accumulators. This set is passed to P_3, which processes the lists for t_2, t_3 merges its results with the forwarded partial answer set to produce a modified partial answer set. The modified set is passed to P_6, which applies the updates generated by the index list for t_4 to produce a final set of accumulators. These final set of accumulators are returned to the central broker. The only work the central broker need to do is receive each query, plan its route through the processors, and return the answer lists to the user, as is shown in Fig. 3.3(b).

Chapter 4

Experimental Results

4.1 Experimental Setting

4.1.1 Environment

Our experiments are conducted on a Beowulf cluster of 48 processors. Each processor in the cluster runs Mandrake Linux 10.1, is a 3.0 GHz Intel Pentium IV with 1 GB memory and 80 GB hard disk. The cluster is connected by a 1 GB network switch. The parallel query processing algorithms discussed in Chapter 3 is implemented in C using the LAM/MPI [7] library.

4.1.2 Dataset

The document collection is the result of a large crawl performed over the '.edu' domain (i.e., the educational US Web sites). The properties of the dataset used in our experiments are presented in Table 4.1. The entire collection is 30 GB and contains 1,883,037 Web pages. After cleansing and stop-word elimination, 3,325,075 distinct index terms remain. The size of the inverted index constructed using this collection is around 2.7 GB. In term-based (document-based) partitioning, terms (documents) are alphabetically sorted and assigned to K index servers in a round-robin fashion using the distribution scheme of [44]. No compression is

applied and posting list indexes are kept in memory.

Size(MB)	30.000
Documents(10^3)	1.883
Total Terms(10^3)	787.221
Distinct Terms(10^3)	3.325
Index(MB)	3.200

Table 4.1: Properties of the *crawl+* document collection.

4.1.3 Queries

In the process of constructing queries, it is assumed that the real life 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. It is also assumed that the terms of a query are dependent to each other in some way. To construct each query, a term is selected randomly. Then a random document containing that particular term is picked. The other query terms are extracted from that document randomly.

In our experiments, short queries contain 1 to 3 terms and medium sized queries contain 4 to 6 terms. Long query experiments are not conducted since 97 percent of realistic web queries consist of less than 7 words [26]. For *crawl+* dataset average number of terms in the short queries is 2.04 and for medium queries it is 4.99.

In all of the experiments, 20,000 queries are submitted to the system. Processing times of the first 10,000 queries are excluded from the throughput and average response time statistics. The first 10,000 queries are used to warm up the system. The queries are submitted via a query submitter interface. It forks child processes which act as users submitting queries to the system. The number of child processes determines the number of concurrent queries in the system. Experiments for 50, 100, 150 and 200 concurrent queries are conducted to measure the system performance under various query loads.

4.2 Experimental Results

The performances of the query processing schemes are compared in terms of two quality measures: average response time and throughput. Average response times represents the response time for a query(seconds per query), and it is calculated by averaging the running time of each query except the initial 10,000 warm up queries. Throughput represents the number of queries answered by the system per second, and is calculated by dividing system's query evaluation time to the total number of queries submitted. Since we used 10,000 warm up queries, the query evaluation time of the system refers to the processing time of the second 10,000 queries . All results reported are the averages of 5 runs.

4.2.1 Central Broker Scheme

In CB experiments, K denotes the number of collaborating processors. For example, when $K = 16$, the central broker and 15 index servers are working simultaneously to process a query. Our system is homogenous, i.e., the central broker and index servers have exactly the same hardware configuration. This setup is prepared for the sake of fair comparison of the CB and PPL.

4.2.1.1 Term-based Distribution

The implementation details of term-based distribution are described in Section 3.1.2. As mentioned before, the merge operation on the central broker can be done in two different ways: 2-way merge and k-way merge. The results of term-based distribution with these two merge operations for $K = 4, 8, 12, 16, 24, 28, 32$ and 40 processors are given in Fig. 4.1. There are concurrently 100 queries in the system.

As seen in Fig. 4.1, the throughput of the CB with term-based distribution (CB-TB) increases up to $K = 24$ processors for both query types. For number of processors larger than 24, overall system efficiency goes down due to the parallelization overhead: as the number of processors increase, throughput does not

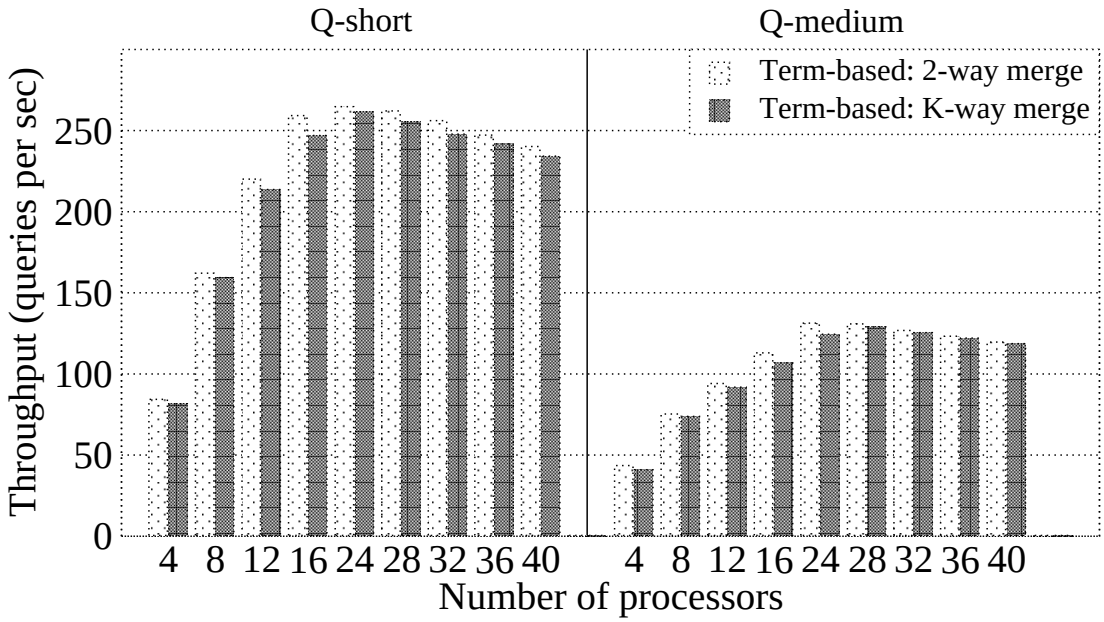

Figure 4.1: Comparison of CB with term-based distribution for 2-way vs k-way merge.

increase. This is a common phenomenon in all parallel systems [19]. A significant decrease in the throughput is not observed since in term-based distribution the total volume of communication does not increase with increasing number of processors, but instead is related with the number of responsible processors for a query. Throughput does not drop down since the processing load of the central broker does not increase with increasing number of processors.

In Fig. 4.1, a slight difference in the throughput rates of 2-way merge and k-way merge is observed. For both types of queries and for all processor counts 2-way merge gives better results than k-way merge. Hence, for comparisons of CB-TB with other schemes, 2-way merge results will be presented for the rest of the paper.

4.2.1.2 Document-Based Distribution

As in term-based distribution, when CB with document-based distribution (CB-DB) is adopted, there are two different ways to perform the merge operation on the central broker: 2-way merge and k-way merge. The relative performances of these two approaches are investigated for $K = 4, 8, 12, 16, 24, 28, 32$ and 40

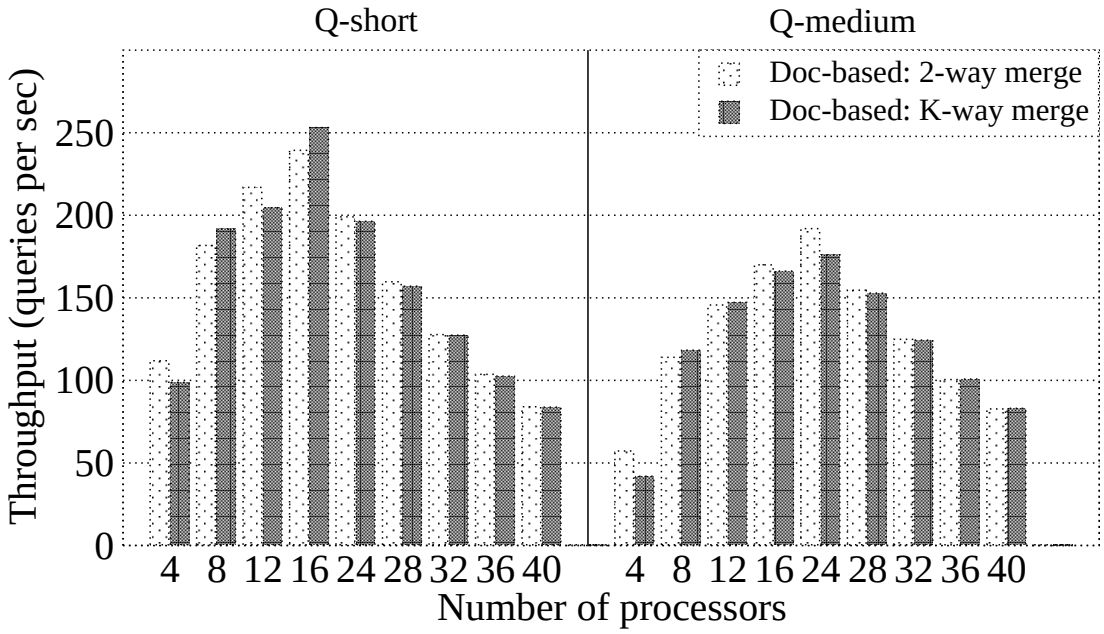

Figure 4.2: Comparison of CB with document-based distribution for 2-way vs k-way merge.

processors. There are concurrently 100 queries in the system.

As seen in Fig. 4.2, for short queries, throughput increases up to 16 processors whereas for medium queries throughput increases up to 24 processors. In document-based distribution, a decrease in the throughput is observed because of the increasing volume of communication during answering a query. For all processor counts, it is observed that almost all index servers contribute during the query processing, probably since the documents are distributed in a round robin fashion, instead of a clustering based distribution. Thus, as the number of processors increase, the total volume of communication during the processing of a query increases in CB-DB as well. As a consequence, the merge load of the central broker increases, which also generates a processing bottleneck on the central broker, causing throughput rates to decrease with increasing number of index servers.

In Fig. 4.2, a slight difference is observed in the throughput rates of 2-way merge and k-way merge. Apart from 8-way and 16-way short query experiments, 2-way merge performs equally or better than k-way merge. Hence 2-way merge is selected to represent document-based distribution for the rest of the paper since it performs better and to be able to make fair comparisons between CB-TB and

CB-DB.

4.2.2 Pipelined Scheme

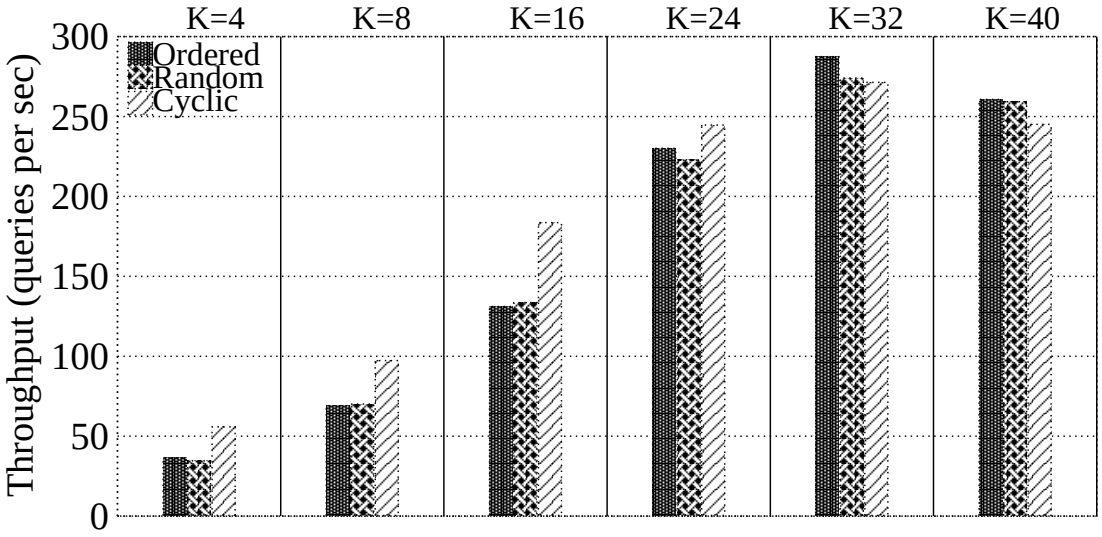

Figure 4.3: Comparison of processing orderings in PPL for short queries.

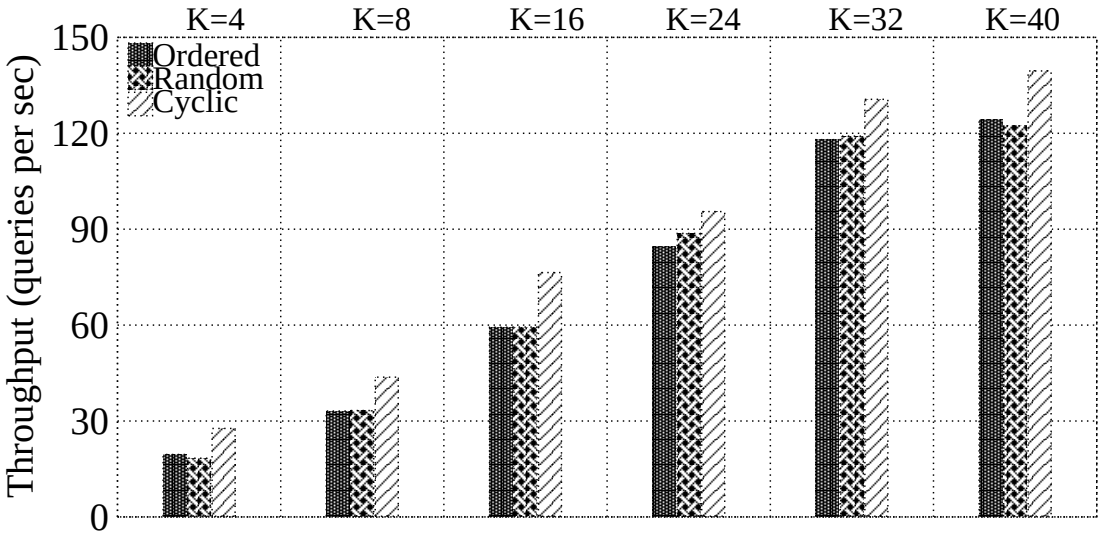

Figure 4.4: Comparison of processing orderings in PPL for medium queries.

In PPL experiments, K denotes the number of collaborating processors. For example, when $K = 16$, there are 16 index servers. The central broker and 16 index servers are working simultaneously on processing a query. The central

broker is not counted as a collaborating processor since the only work of the central broker is to dispatch queries and to display the incoming final answer sets to the users.

The system is homogenous, i.e., the central broker and index servers have exactly the same hardware configuration. This setup is prepared in order to be able to compare CB and PPL fairly.

In PPL, the routing order of a query can effect the performance of the system. The three different ordering algorithms presented in Section 3 are analyzed. The comparison of throughput for these methods using $4, 8, 16, 24, 32$ and 40 processors can be seen in Fig. 4.3 and Fig. 4.4 for short and medium queries, respectively.

The results show that, cyclic ordering performs better than the other ordering schemes for all processor counts with medium queries and for processor counts up to $K = 24$ processors for short queries. Thus, for comparisons of PPL with other schemes randomized cyclic routing results will be given for the rest of the paper.

4.2.3 CB-TB vs CB-DB vs PPL

In Fig. 4.5 CB-TB, CB-DB and PPL are compared in terms of throughput over short queries. PPL performs poorly for small number of processors but achieves high throughput rates after $K = 16$ processors. The overhead of parallelization is observed as a decrease in throughput of CB-DB after $K = 16$ processors whereas this decrease is observed in CB-TB and PPL after $K = 32$ processors. In general CB-TB performs well for all processor counts and PPL seems to be more scalable than the other schemes.

In Fig. 4.6 CB-TB, CB-DB and PPL are compared in terms of throughput over medium queries. All three schemes observe an increase in throughput rates up to $K = 24$ processors and then start to degrade due to parallelization overhead for larger processor counts. For small number of processors, CB-DB performs much better than the term-based schemes. PPL performs poorly for small number of processors but achieves high throughput rates after $K = 32$ processors. PPL

throughput rates continue to increase with increasing number of processors and the efficiency of PPL does not start to decrease until 40 processors, unlike CB schemes. Even though PPL is more scalable, the throughput values can not compete with CB.

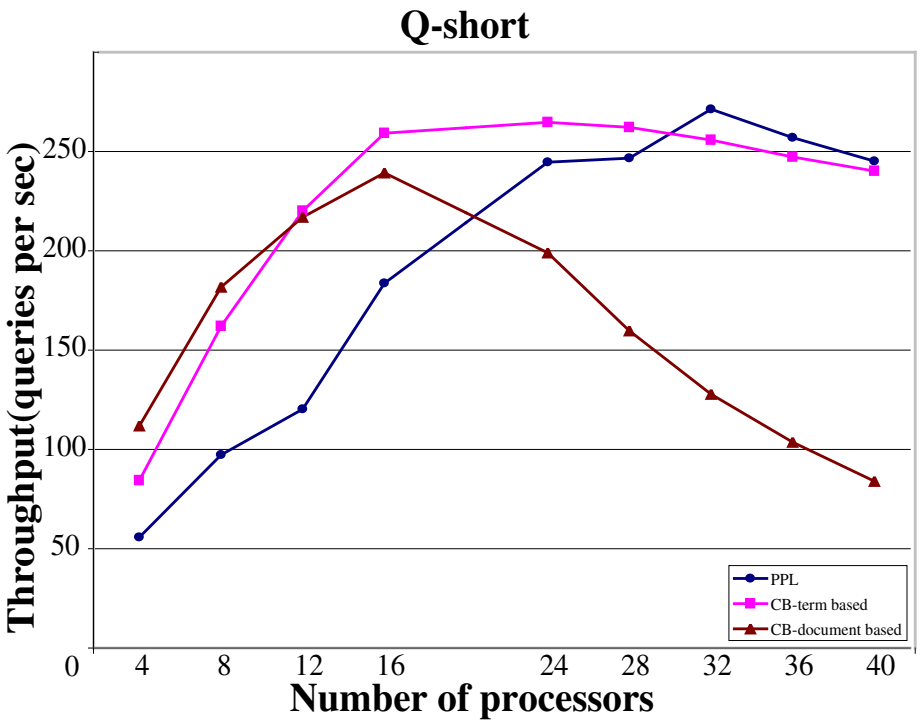

Figure 4.5: Comparison of CB and PPL for short queries in terms of throughput.

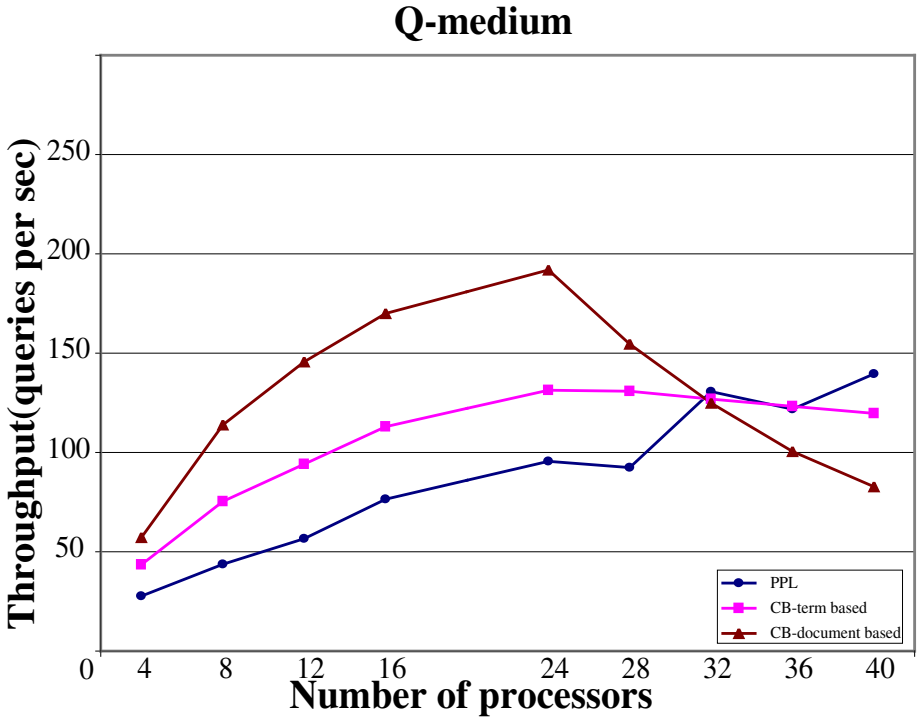

Figure 4.6: Comparison of CB and PPL for medium queries in terms of throughput.

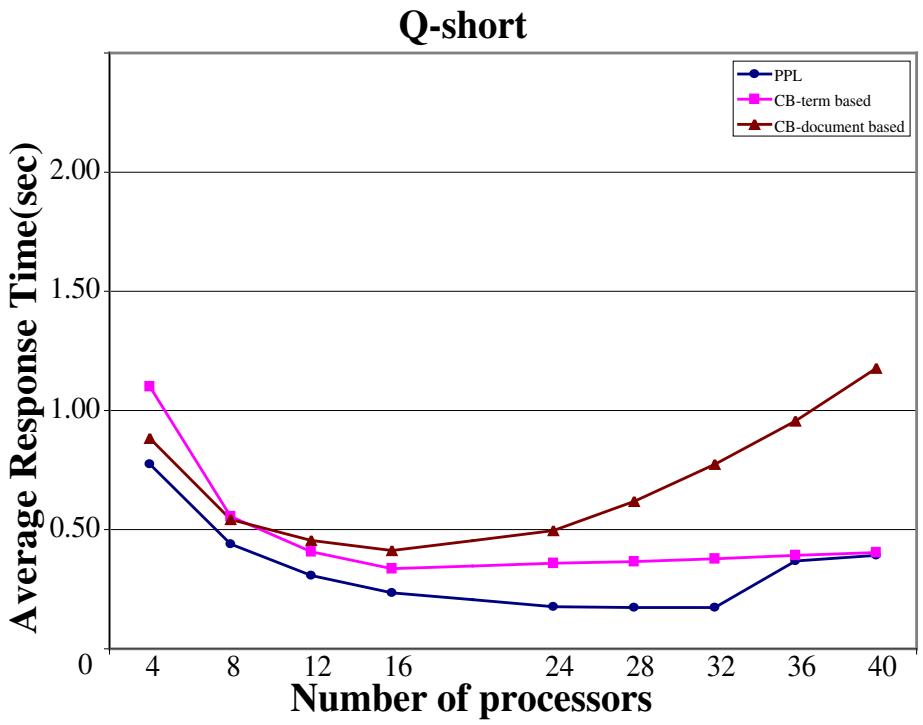

Figure 4.7: Comparison of CB and PPL for short queries in terms of average response time.

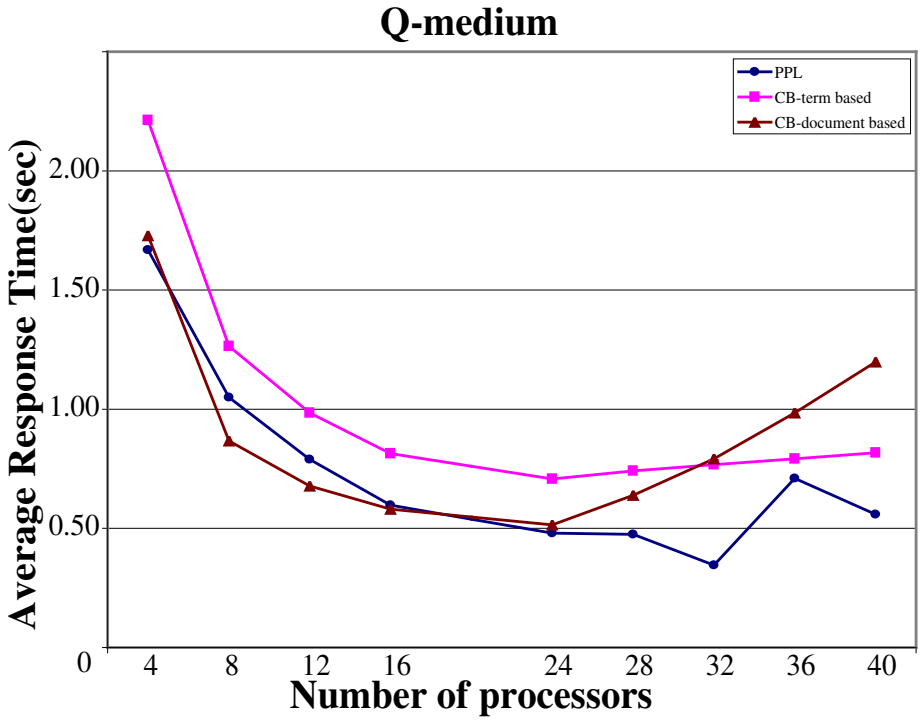

Figure 4.8: Comparison of CB and PPL for medium queries in terms of average response time.

In Fig. 4.7 and Fig. 4.8 CB-TB, CB-DB and PPL are compared in terms of average response times over short and medium queries, respectively. For both query types, CB schemes observe an decrease in average response times up to $K = 24$ processors whereas PPL average response times decrease up to $K = 32$ processors and then start to degrade due to parallelization overhead for larger processor counts. This also indicates to the scalability of PPL scheme.

4.2.4 Variation of system performance under different query loads

To investigate the alteration of the system performance under various query loads, experiments with different number of concurrent queries in the system are carried out. We made experiments for $C = 50, 100, 150$ and 200, where C denotes the number of children spawned in the query submitter interface. The results are presented in Fig. 4.9 and Fig. 4.10 for throughput and Fig. 4.11 and Fig. 4.12 for average response time, $K = 16$ and 32 respectively.

As seen in Fig. 4.9, for $K = 16$ processors, CB-TB and CB-DB schemes' throughput do not change significantly when the concurrent user load on the system is increased whereas PPL throughput slightly increases. Fig. 4.10 shows that, when $K = 32$ processors are used, CB-TB and CB-DB schemes' throughput slightly decreases when the concurrent user load on the system is increased whereas PPL throughput still slightly increases. This indicates that PPL is more resilient to increase in user loads. These results proves that the central broker becomes a bottleneck in query processing. For heavily loaded systems, CB performs poorer. System resources are wasted for enqueue/dequeue operations.

As expected, average response times increase when number of concurrent queries increase in the system. This result can be seen in Fig. 4.11 and Fig. 4.12. Best average response time results are obtained for PPL. CB-TB performs better than CB-DB. Also, the increase in average response times of PPL is smaller compared to CB, because the central broker becomes a bigger bottleneck as the number of concurrent queries in the system increases.

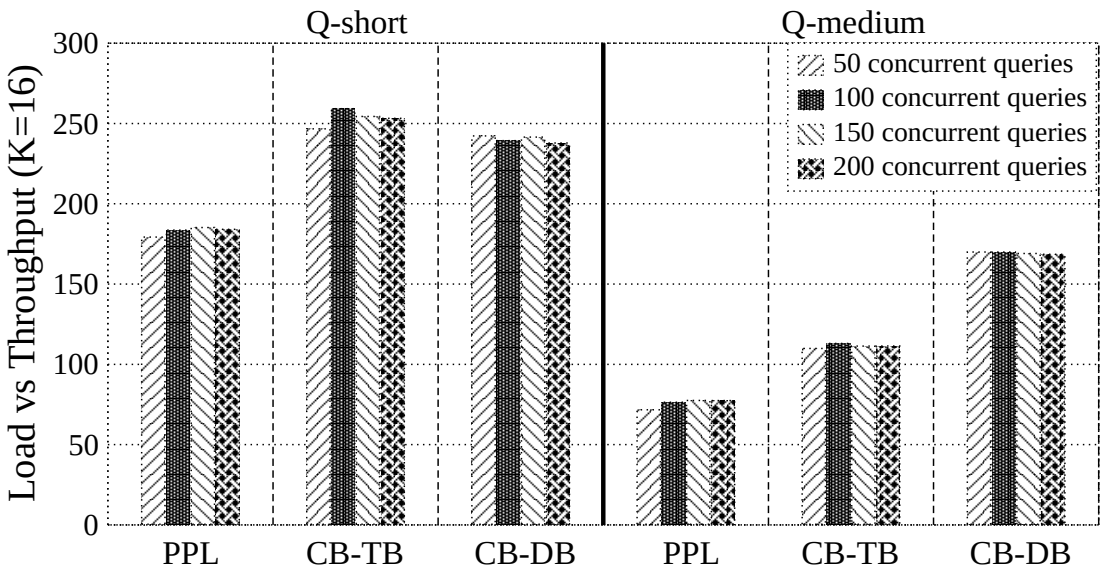

Figure 4.9: Throughputs of CB and PPL compared with changing query load for K = 16.

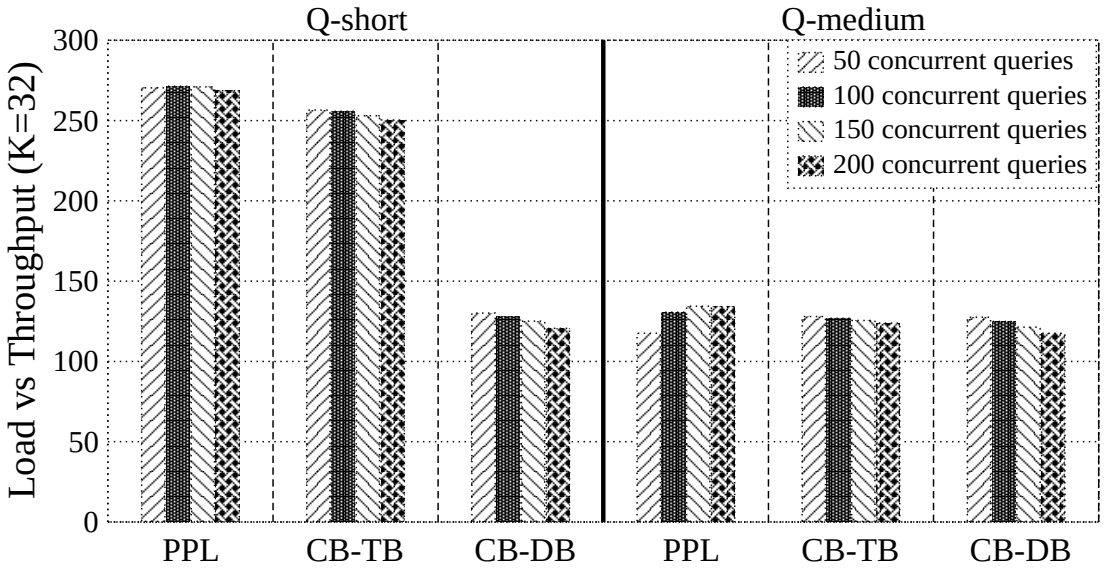

Figure 4.10: Throughputs of CB and PPL compared with changing query load for K = 32.

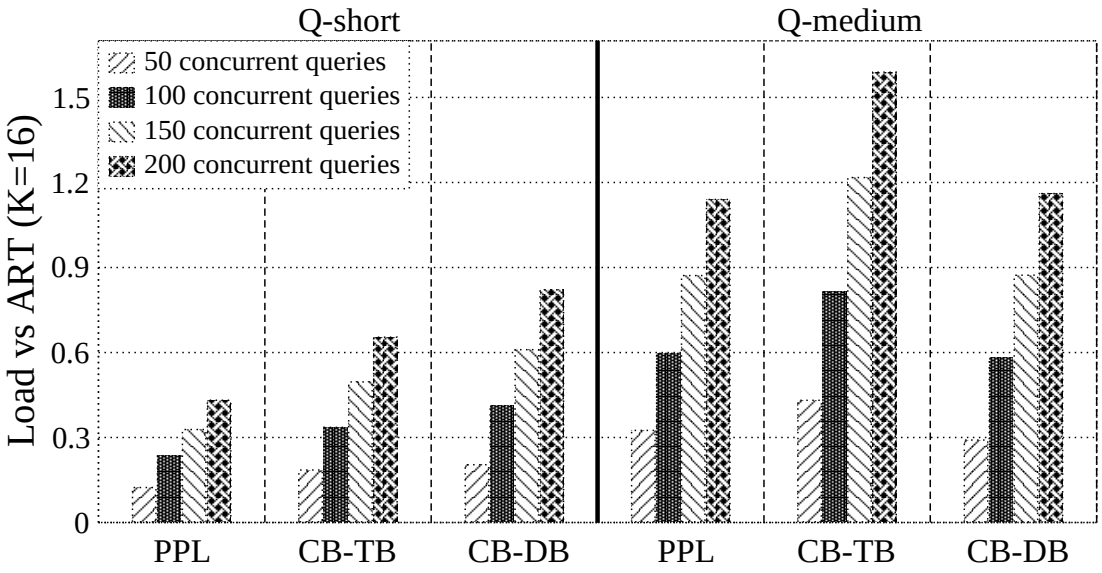

Figure 4.11: Average response times of CB and PPL compared with changing query load for K = 16.

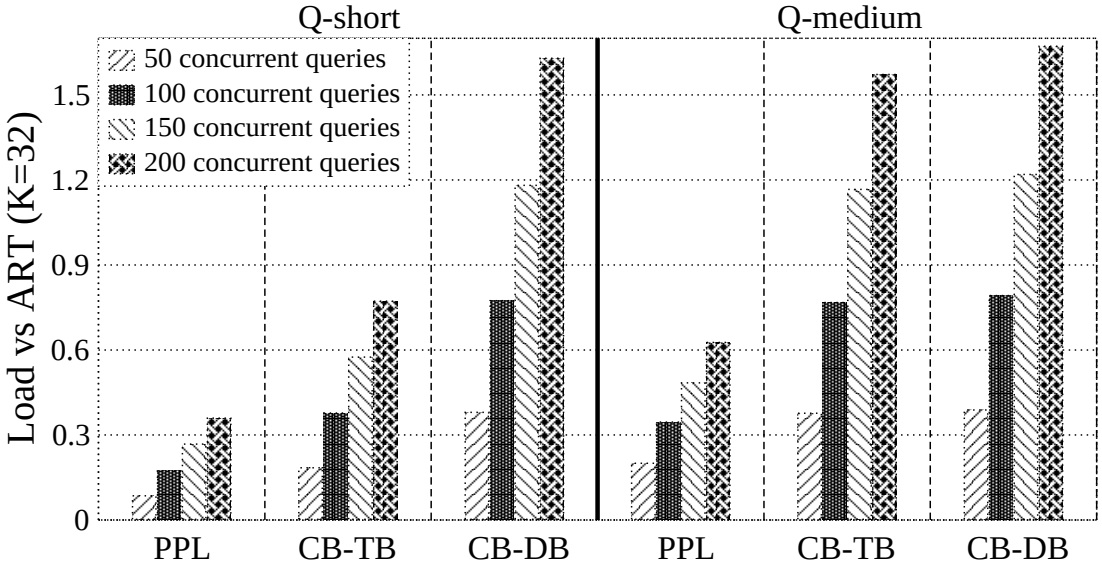

Figure 4.12: Average response times of CB and PPL compared with changing query load for K = 32.

4.2.5 Accumulator Size Restriction and Quality

In term-based distribution of the inverted index, communication overhead significantly reduces the system performance. In light of the previous research [37], our text retrieval system communicates accumulators with size up to 1 percent of the size of the document collection when term-based distribution is used. In the selection phase, index servers select the top 1 percent of the accumulators from their static accumulator arrays and send them to the central broker. The aim of limiting the number of accumulators sent is to keep communication costs at minimum without causing a visible change in the final answer set.

Since the proposed optimization is approximate, a series of experiments are conducted to measure if there is a decrease in the quality of the top s scores presented to the user. To verify that limiting the number of accumulators being sent does not cause a decrease in quality, Fagin et al. [16] developed various correlation measures to compare two lists, based on techniques for comparing two permutations [28]. Carmel et al. [12] also adopted one of their measures, namely a variation of Kendall's tau method.

In this modification of Kendall's tau, for each document pair i, j, penalties are assigned according to appearance of i and j in resulting top s lists. The penalty values are defined as:

- *Case 1:* if both i and j appear in both lists, and if their appearance order is changed, we assign a penalty of 1, otherwise no penalty is assigned.
- *Case 2:* if one of the lists contain both i and j but the other list contains only one of them, then a penalty of 1 is assigned if the lower rank document appears. No penalty is assigned if the higher order term appears, since it is ahead in both lists.
- *Case 3:* if only i appears in one list and only j appears in the other, a penalty of 1 is assigned, since their ordering is changed.
- *Case 4:* if both i and j appear in one list but neither appears in the other, a penalty of $1/2$ is assigned since it is not known which one would appear in higher rank.

For each query, the penalty scores are summed. Note that the result is between 1 and $k\ (3k\ -\ 1)\ /\ 2$. To get a normalized similarity value, the sum is divided by this number, and then the result is subtracted from 1 to define 1 as the highest similarity value, whereas 0 is defined to be the minimum.

Using the accumulator limiting scheme described above, remarkable improvements in the average response times and throughput is obtained. For example, CB without employing accumulator limiting for short queries gives average response time of 18.4 seconds per query and processes 7.1 queries per second on 8 processors, where on the other hand same scheme with accumulator limiting gives average response time of 1.5 queries per second and outputs 73.0 queries per second. For PPL, the difference is even more dramatic: on 8 processors it processes 87.4 queries per second with an average response time of 1.1 seconds per query.

The result set generated employing document-based distribution in CB gives precision value of 1.0 is assumed, since it is possible to compute final scores of the documents in index servers. Comparing the other result sets to it yields to Table 4.2. These experiments are conducted for 1500 queries, each returning top 100 answers.

	Qshort		*Qmed*	
	Avg.Sim.	*Avg.Penalty*	*Avg.Sim.*	*Avg.Penalty*
CB(doc-based)	1.0000	0.00	1.0000	0.00
CB(restricted)	0.9914	128.39	0.9749	374.79
PPL(restricted)	0.9964	53.10	0.9972	41.72

Table 4.2: Average similarity and penalty sum results.

According to Table 4.2, short query evaluation accuracy is not affected from accumulator limiting. But for medium queries, CB-TB with accumulator restriction gives a relatively low similarity measure than PPL. Since the accumulators with low scores are not sent to the central broker, in CB-TB, the documents in the final answer set may switch places, causing a decrease in the similarity scores. However, since the scores on the index servers are merged using the static accumulator arrays, some of these low scored accumulators may be included in the final answer set, causing PPL to perform better than CB-TB.

As seen from Table 4.2, restricting the number of accumulators communicated does not cause a notable decrease in the quality of our answer set but it brings a remarkable increase in the quality of service.

Chapter 5

Conclusion

The vast volume of data available online raises more challenges in information retrieval research. The need to access desired data by numerous users concurrently forces the text retrieval system designers to be innovative since the classical approaches does not work efficiently and effectively on very large document collections. In order to satisfy user needs when a large volume of data is being processed, usage of parallel methods become inevitable since parallel systems provide better average response times and higher throughput rates compared to sequential methods.

In this thesis, a parallel text retrieval system on a shared-nothing architecture is implemented. The system adopts inverted index for distributing the dataset and uses vector space model for calculating the similarities between documents and query terms. The dataset is distributed in a round-robin fashion.

ABC-Server Parallel Text Retrieval System uses two query evaluation schemes: Central Broker (CB) and Pipelined (PPL). CB can be used with both term-based or document-based distributed inverted indexes whereas PPL is valid only for term-based distribution. The query evaluation performance of CB-TB, CB-DB and PPL are measured and compared according to their throughput, average response time and quality performances on a 48 node PC cluster on varying query loads.

The results indicate that, CB-DB gives the best throughput rates whereas

PPL gives the best average response times in general. This result reveals that in CB, some query response times are significantly larger than the average, which increases the average response time without affecting the throughput. This proves the existence of a bottleneck on the central broker.

Another interesting result is that, when the dataset is distributed in a document based fashion, the throughput begins to decrease as the number of processors increase after some point, whereas the throughput decreases slightly after a point when term-based distribution is used. This is also a consequence of the bottleneck on the central broker, due to merge operations.

The *crawl+* document collection used for the experiments is not sufficiently large for exploring the scalability of ABC-Server for practical uses. The future aspects of this study include running the experiments on a larger document collection and compare its results.

Although the system is implemented very carefully for the best performance, there are still some possible enhancements. The system adopts expected linear time selection algorithm for the selection phase both in the central broker and index servers. It is possible to use a minimum heap to select top accumulators as proposed in [48]. Cambazoglu [9] compared the performances of both and concluded that the min heap implementation gives better results. Another possible improvement is to use a min heap instead of keeping the running minimum in k-way merge. Including these in the next version of ABC-server is planned.

Bibliography

[1] I. S. Altingovde, E. Demir, F. Can, and Özgür Ulusoy. Incremental cluster-based retrieval using compressed cluster-skipping inverted files. *ACM Trans. Inf. Syst.*, 26(3):1–36, 2008.

[2] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 372–379, New York, NY, USA, 2006. ACM.

[3] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. *spire*, 00:0010, 2001.

[4] R. Baeza-Yates and Ribeiro-Neto. *Modern information retrieval*. Addison-Wesley., New York, NY, USA, 1999.

[5] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 163–174, New York, NY, USA, 2001. ACM.

[6] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR '85: Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 97–110, New York, NY, USA, 1985. ACM.

[7] G. Burns, R. Daoud, and J. Vaigl. Lam: an open cluster environment for mpi. In *Proceedings of the Supercomputing Symposium*, pages 379–386, 1994.

[8] B. B. Cambazoglu. *Models and algorithms for parallel text retrieval*. PhD thesis, Bilkent University, 2006.

[9] B. B. Cambazoglu and C. Aykanat. Performance of query processing implementations in ranking-based text retrieval systems using inverted indices. *Inf. Process. Manage.*, 42(4):875–898, 2006.

[10] B. B. Cambazoglu, A. Catal, and C. Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems ? In *ISCIS 2006*, pages 717–725, 2006.

[11] F. Can, I. S. Altingövde, and E. Demir. Efficiency and effectiveness of query processing in cluster-based retrieval. *Inf. Syst.*, 29(8):697–717, 2004.

[12] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 43–50, New York, NY, USA, 2001. ACM.

[13] C. L. A. Clarke, G. V. Cormack, and E. A. Tudhope. Relevance ranking for one to three term queries. *Inf. Process. Manage.*, 36(2):291–311, 2000.

[14] W. B. Croft and P. Savino. Implementing ranking strategies using text signatures. *ACM Trans. Inf. Syst.*, 6(1):42–62, 1988.

[15] R. Elmasri and S. Navathe. *Fundamentals of database systems.* Addison-Wesley., Reading, MA, 2003.

[16] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM Journal on Discrete Mathematics*, 17(1):134–160, 2003.

[17] W. B. Frakes and R. Baeza-Yates. *Information retrieval: Data structures and algorithms.* Prentice Hall., Englewood Cliffs, NJ, 1992.

[18] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 26–37, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[19] A. Grama, A. Gupta, E. hong Han, and V. Kumar. *Introduction to Parallel Computing.* Addison-Wesley., New York, NY, USA, 2003.

[20] D. Harman and G. C. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41:581–589, 1990.

[21] D. W. Harman. An experimental study of factors important in document ranking. In *SIGIR '86: Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 186–193, New York, NY, USA, 1986. ACM.

[22] D. J. Harper. *Relevance feedback in document retrieval systems: An evaluation of probabilistic strategies*. PhD thesis, The University of Cambridge, 1980.

[23] D. Hawking. Efficiency/effectiveness trade-offs in query processing (from theory into practice workshop, 1998 sigir conf.). *SIGIR Forum*, 32(2):16–22, 1998.

[24] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 850–861. VLDB Endowment, 2003.

[25] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 754–765. VLDB Endowment, 2003.

[26] B. J. Jansen, A. Spink, J. Bateman, and T. Saracevic. Real life information retrieval: a study of user queries on the web. *SIGIR Forum*, 32(1):5–17, 1998.

[27] B.-S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):142–153, 1995.

[28] M. Kendall and J. D. Gibbons. *Rank correlation methods*. Edward Arnold, London, 5 edition, 1990.

[29] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[30] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 129–140. VLDB Endowment, 2003.

[31] D. Lucarella. A document retrieval system based upon nearest neighbor searching. *Journal of Information Science*, 14(1):25–33, 1988.

[32] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *InfoScale '07: Proceedings of the 2nd international conference on Scalable information systems*, pages 1–9, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[33] A. MacFarlane, J. McCann, and S. Robertson. Parallel search using partitioned inverted files. *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 209–220, 2000.

[34] C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[35] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355, New York, NY, USA, 2006. ACM.

[36] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3):205–231, 2007.

[37] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[38] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, 30(6):733–744, 1994.

[39] M. Persin. Document filtering for fast ranking. In *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 339–348, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[40] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[41] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *DL '98: Proceedings of the third ACM conference on Digital libraries*, pages 182–190, New York, NY, USA, 1998. ACM.

[42] G. Salton and M. J. McGill. *Introduction to modern information retrieval*. McGraw-Hill., New York, 1983.

[43] A. F. Smeaton and C. J. van Rijsbergen. The nearest neighbour problem in information retrieval: an algorithm using upperbounds. In *SIGIR '81: Proceedings of the 4th annual international ACM SIGIR conference on Information storage and retrieval*, pages 83–87, New York, NY, USA, 1981. ACM.

[44] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document informatioon retrieval systems. In *PDIS '93: Proceedings of the second international conference on Parallel and distributed information systems*, pages 8–17, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[45] A. Tomasic, H. García-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 289–300, New York, NY, USA, 1994. ACM.

[46] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, 1995.

[47] R. Wilkinson, J. Zobel, and R. Sacks-davis. Similarity measures for short queries. In *In Fourth text retrieval conference (TREC-4*, pages 277–285, 1995.

[48] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[49] W. Y. P. Wong and D. L. Lee. Implementations of partial document ranking using inverted files. *Inf. Process. Manage.*, 29(5):647–669, 1993.

[50] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[51] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full text databases. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 352–362, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.