MODELS AND ALGORITHMS FOR PARALLEL TEXT RETRIEVAL

A DISSERTATION 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
DOCTOR OF PHILOSOPHY

By
Berkant Barla Cambazoğlu
January, 2006

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

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 dissertation for the degree of doctor of philosophy.

Prof. Dr. Volkan Atalay

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

Prof. Dr. Fazlı Can

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

Prof. Dr. Enis Çetin

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

Prof. Dr. Özgür Ulusoy

Approved for the Institute of Engineering and Science:

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

ABSTRACT

MODELS AND ALGORITHMS FOR PARALLEL TEXT RETRIEVAL

Berkant Barla Cambazoğlu
Ph.D. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
January, 2006

In the last decade, search engines became an integral part of our lives. The current state-of-the-art in search engine technology relies on parallel text retrieval. Basically, a parallel text retrieval system is composed of three components: a crawler, an indexer, and a query processor. The crawler component aims to locate, fetch, and store the Web pages in a local document repository. The indexer component converts the stored, unstructured text into a queryable form, most often an inverted index. Finally, the query processing component performs the search over the indexed content. In this thesis, we present models and algorithms for efficient Web crawling and query processing. First, for parallel Web crawling, we propose a hybrid model that aims to minimize the communication overhead among the processors while balancing the number of page download requests and storage loads of processors. Second, we propose models for document- and term-based inverted index partitioning. In the document-based partitioning model, the number of disk accesses incurred during query processing is minimized while the posting storage is balanced. In the term-based partitioning model, the total amount of communication is minimized while, again, the posting storage is balanced. Finally, we develop and evaluate a large number of algorithms for query processing in ranking-based text retrieval systems. We test the proposed algorithms over our experimental parallel text retrieval system, Skynet, currently running on a 48-node PC cluster. In the thesis, we also discuss the design and implementation details of another, somewhat untraditional, grid-enabled search engine, SE4SEE. Among our practical work, we present the Harbinger text classification system, used in SE4SEE for Web page classification, and the K-PaToH hypergraph partitioning toolkit, to be used in the proposed models.

Keywords: Search engine, parallel text retrieval, Web crawling, inverted index partitioning, query processing, text classification, hypergraph partitioning.

ÖZET

PARALEL METİN GETİRME İÇİN MODELLER VE ALGORİTMALAR

Berkant Barla Cambazoğlu
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Ocak, 2006

Son on yılda arama motorları hayatımızla bütünleşik bir hale gelmişlerdir. Arama motorları teknolojisi şu anda paralel metin getirmeye dayanmaktadır. Bir paralel metin getirme sistemi temel olarak üç bileşenden oluşmaktadır: tarayıcı, indeksleyici ve sorgu işleyici. Tarayıcı bileşeni Ağ'da bulunan sayfaları bulmayı, getirmeyi ve yerel bir metin ambarında saklamayı amaçlar. İndeksleme bileşeni saklanmış olan düzensiz metinleri sorgulanabilir bir yapıya dönüştürür ki bu yapı çoğu zaman bir ters dizindir. Sorgu işleme bileşeni ise indekslenmiş içerik üzerinde aramayı gerçekleştirir. Bu tezde, etkin Ağ tarama ve sorgu işleme için modeller ve algoritmalar önerilmiştir. Paralel Ağ tarama için, işlemciler arası iletişim miktarını en aza indiren ve işlemcilerin sayfa indirme isteklerinin sayısını ve saklama yüklerini dengeleyen karma bir model önerilmiştir. Ek olarak, metin ve kelime bazlı ters dizin bölümleme için modeller önerilmiştir. Metin bölümlemeye dayalı modelimizde saklama yükü dengelenirken sorgu işleme sırasında karşılaşılacak disk erişim miktarı en aza indirilmektedir. Kelime bölümlemeye dayalı modelimizde ise yine saklama yükü dengelenirken toplam iletişim hacmi en aza indirilmektedir. Bunlara ek olarak, sıralamaya dayalı metin getirme sistemleri için çok sayıda sorgu işleme algoritması uygulanmış ve değerlendirilmiştir. Önerilen algoritmalar 48 düğümlü bir PC kümesi üzerinde çalışmakta olan deneysel paralel metin getirme sistemimiz Skynet üzerinde denenmiştir. Tezde ayrıca gride uyarlanmış SE4SEE arama motorumuzun tasarım ve uygulama detayları tartışılmıştır. Pratikteki katkılarımız arasından, SE4SEE içinde kullanılan Harbinger metin sınıflandırma sistemi ve önerilen modellerde kullanılmak üzere geliştirilen K-PaToH hiperçizge bölümleme aracı sunulmuştur.

Anahtar sözcükler: Arama motoru, paralel metin getirme, Ağ tarama, ters dizin bölümleme, sorgu işleme, metin sınıflandırma, hiperçizge bölümleme.

Acknowledgement

I would like to acknowledge the valuable help, guidance, and availability of Prof. Dr. Cevdet Aykanat throughout the study. I thank the following people for their contributions in the thesis: Ata Türk (Chapter 2), Aytül Çatal (Chapters 3 and 5), Evren Karaca (Chapter 6), Tayfun Küçükyılmaz (Chapters 6 and 7), and Bora Uçar (Chapter 8). I also thank Prof. Dr. Fazlı Can for his inspiration for the contribution in Chapter 4. I am grateful to Prof. Dr. Volkan Atalay, Prof. Dr. Fazlı Can, Prof. Dr. Enis Çetin, and Prof. Dr. Özgür Ulusoy for reading my thesis. Finally, I thank to my friends Ali, Ata, Bayram, Engin, Evren, Funda, Kamer, Özgün, Sengör, and Tayfun, who turned our department into a lovely place.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The exponential rate at which the Web grows led to an explosion in the amount of publicly accessible digital text media. In the last decade, various text retrieval systems addressed the issues in discovery, fetching, storage, compression, indexing, querying, filtering, and presentation of this vast content. In this age of information, search engines act as important services, providing the community with the information hidden in the Web and, due to their frequent use, stand as an integral part of our lives. The last decade has witnessed the design and implementation of several state-of-the-art search engines [100]. The wide-spread use of these systems resulted in an increase in the number of submitted user queries. At the time of this writing, the Google search engine, a popular search engine on the Web, has indexed more than four billion Web pages. Today, the popular search engines process millions of user queries per day over their index. This explains the heavy research interest on text retrieval well.

Currently, text retrieval research is focused on the two major criteria by which the systems are evaluated: effectiveness and efficiency. Effectiveness is a measure of the quality of the returned results. The two frequently used metrics for effectiveness are precision and recall. Precision is the ratio of the number of retrieved

documents that are relevant to the total number of retrieved documents. Recall is the ratio of the number of retrieved documents that are relevant to the number of relevant documents.

So far, most research is concentrated on the effectiveness part, and it is highly speculated that the research on effectiveness in text retrieval is about to reach its limits. Efficiency criteria, which is used to evaluate the computational performance of retrieval systems, took relatively little interest. We believe that efficiency and effectiveness are two closely related issues. Improving efficiency can indirectly improve effectiveness via relaxation on some query processing thresholds and cutoff values (e.g., term count limits on the size of user queries, thresholds in similarity calculations between documents and queries, and cutoff values in document ranking and presentation). Consequently, we believe that the efficiency component deserves more attention than it currently had.

During the last two decades, text retrieval research addressed the issues mostly in sequential computer systems. The massive size of today's document collections when coupled with the ever-growing number of users querying the documents in these collections necessitates parallel computing systems. Although both parallel computing and text retrieval research lend their roots to several decades ago, research on parallel text retrieval is relatively young and evolving. Unfortunately, so far, most efforts towards efficient retrieval remained as a trade secret due to the commercial nature of the text retrieval systems. This thesis focuses on efficient query processing in parallel text retrieval systems, in particular on efficient parallel Web crawling, inverted index organizations, and query processing.

1.2 Background

A traditional search engine is typically composed of three pipelined components [5]: a crawler, an indexer, and a query processor. The crawler component is responsible for locating, fetching, and storing the content on the Web. The downloaded content is concurrently parsed by an indexer and transformed into

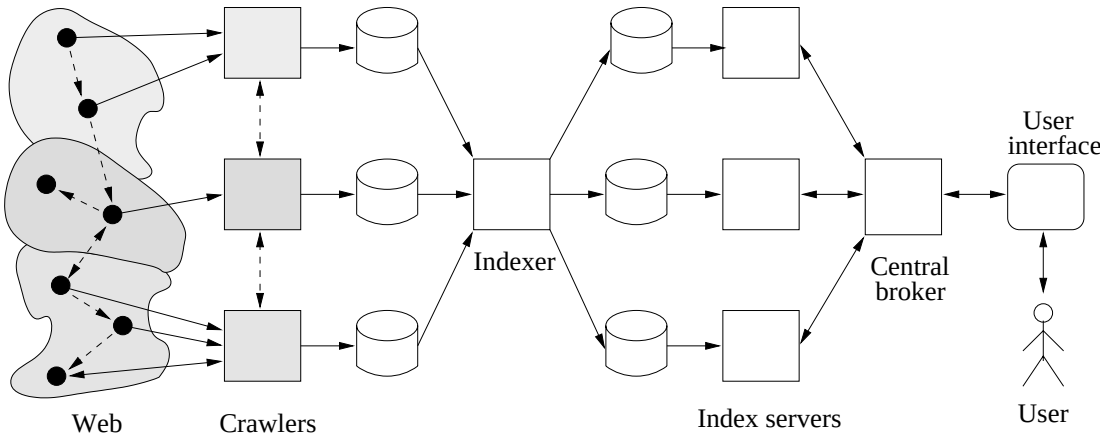


Figure 1.1: Architecture of a traditional search engine.

an inverted index [113, 133], which represents the content in a compact and efficiently queryable form. The query processor is responsible for evaluating user queries over the index and returning the users pages relevant to their queries.

Figure 1.1 depicts the picture of a general architecture for a traditional shared-nothing parallel text retrieval system. This is the architecture for which we are developing models and algorithms. In this architecture, the Web is partitioned among a number of software programs, called Web crawlers. Each crawler is responsible for downloading a subset of pages on the Web. The crawlers locate the pages by following the hyperlinks among the pages. After they are downloaded, the pages are stored in the local hard disks of the processors. A concurrently running indexer is responsible for converting the documents into a queryable form, which is often an inverted index. The constructed inverted index is partitioned and stored in a distributed manner among the local disks of the processors in the parallel system. While all these happen in the background, the users submit queries to the retrieval system through a user interface. A submitted query is sent to the central broker, where it is split into subqueries. These subqueries are then submitted to index servers. Index servers access their local disks, determine the set of documents matching the subquery, and send these answer sets back to the central broker. The central broker merges these partial answer sets and puts the documents into a sorted order according to the similarity of the documents to the query. Finally, the user is returned a set of best-matching documents.

1.3 Contributions

The contributions of this thesis can be categorized into two as theoretical and practical. The theoretical contributions, which are presented in Chapters 2, 3, and 4, include the proposed models and algorithms that aim to improve the efficiency of Web crawling and query processing in both sequential and/or parallel text retrieval systems. The practical contributions, which are presented in Chapters 5, 6, 7, and 8, involve the software systems developed throughout the study. These systems are implemented mostly to evaluate the practical performance of the proposed, theoretical models. In what follows, we list a brief overview of our particular contributions together with the organization of the thesis.

In Chapter 2, we give a taxonomy of implementations for Web crawling and present a page-to-processor assignment model for efficient parallel Web crawling. The proposed model is a hybrid model that combines our previously proposed Web crawling models [21, 117], which are based on graph and hypergraph partitioning, into a single more powerful model. This hybrid model minimizes the total inter-processor communication overhead while balancing the page storage loads of processors as well as the page download requests issued by the processors.

In Chapter 3, we propose two inverted index partitioning models for term-based and document-based indexing in parallel and distributed text retrieval systems [25]. The proposed hypergraph-partitioning-based models aim to improve the query processing efficiency of the text retrieval system, by producing an intelligent assignment of posting entries to the processors. Specifically, in the term-based inverted index partitioning model, the total volume of communication among the index servers and the central broker is minimized while the posting storage load of index servers is balanced. In the document-based partitioning model, the number of disk accesses performed by the index servers to retrieve the inverted lists is minimized while, again, the posting storage is balanced.

In Chapter 4, we introduce a taxonomy for the query processing algorithms in ranking-based text retrieval systems using inverted indices. We investigate the complexity of a large number of query processing implementations, several of

which are proposed by us [18]. We conduct a comparative study on the performance of these implementations in terms of their time and space efficiency. We report performance results over a large collection of Web pages.

In Chapter 5, we introduce our prototype parallel text retrieval system, Skynet. Although Skynet has all the ingredient a traditional search engine would require, it is by no means developed as a fully-functional, complete search engine. In particular, this system is designed and implemented in order to act as a test-bed on which we would evaluate the models and algorithms proposed in Sections 3 and 4.

In Chapter 6, we describe the design details and an architectural overview of our SE4SEE (Search Engine for South-East Europe) application [19, 24]. SE4SEE is a grid-enabled Web search engine, which we developed as a regional application throughout the EU-funded SEE-GRID FP-6 project, utilizing our expertise in Web crawling and text classification. The SE4SEE application can be defined as a personalized, on-demand, category-based, country-specific search engine. In this chapter, we provide performance results for this search engine over a geographically distributed grid infrastructure.

In Chapter 7, we present our prototype text classification system, Harbinger, as well as the machine learning toolkit that the classification system utilizes [20]. Although we have other ongoing works that this system uses, the Harbinger text classification system is mainly employed in SE4SEE for the purpose of classifying Web pages into categories. We provide a manual for this system in Appendix B.

Finally, in Chapter 8, we provide algorithmic details of a multi-level direct K-way hypergraph partitioning implementation, namely the K-PaToH toolkit [6]. This implementation is important in that the solution qualities of the proposed models presented in Chapters 2 and 3 heavily rely on the solution quality of the hypergraph partitioning. Experiments presented in this chapter indicate that K-PaToH proves to be more efficient in terms of both execution time and solution quality compared to our previously used hypergraph partitioning tool PaToH.

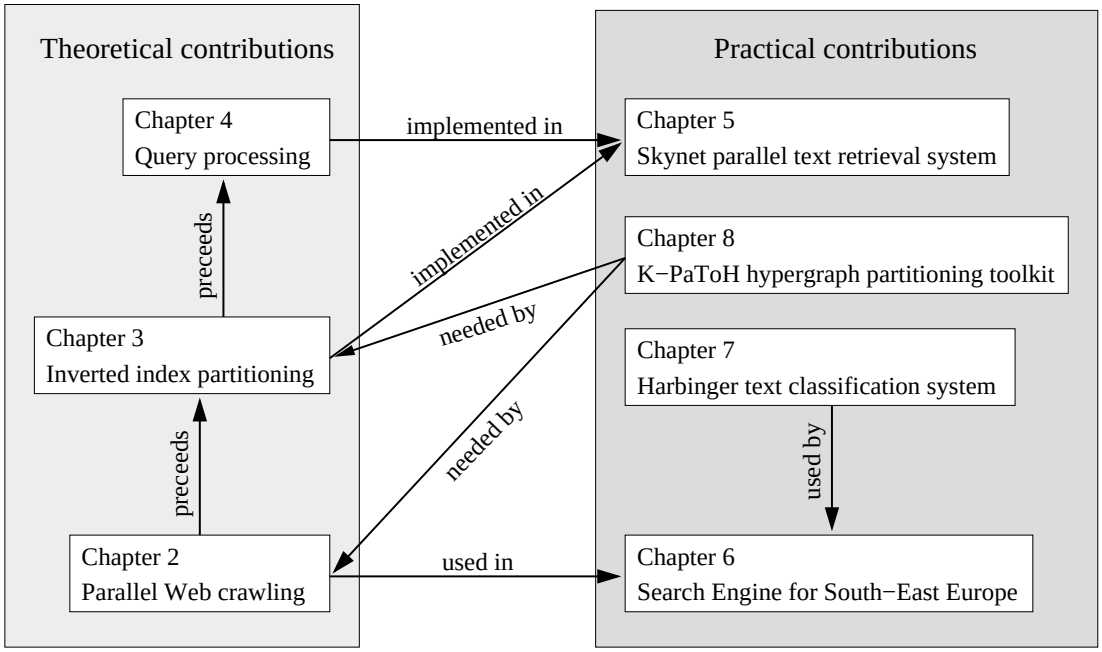


Figure 1.2: The graph representing the dependency between the contributions of the thesis.

Since this is a rather lengthy thesis, we provide the dependency graph in Figure 1.2 to the reader in order to visualize the inter-relation between the contributions of the thesis. The text on the arcs represents the type of the dependency between the chapters of the thesis. Chapters 2, 3, and 4 should be read in that order since the content in these chapters respectively mention Web crawling, inverted index partitioning, and query processing, which are the three components successively pipelined in a text retrieval system. If the reader has no background knowledge on hypergraph partitioning, we highly recommend reading Chapter 8 (at least Section 8.1) because the models described in Chapters 2 and 3 require a good understanding of hypergraphs and hypergraph partitioning. For the sake of the presentation, in these chapters, we partially duplicate some background information about hypergraph partitioning. The reader interested in practical work may safely move to Chapter 5, where we present the implementation of the Skynet parallel text retrieval system, and Chapter 6, where we present the details of our grid-enabled search engine, SE4SEE.

Chapter 2

Parallel Web Crawling Model

The need to quickly locate, gather, and store the vast amount of material on the Web necessitates crawling the Web via parallel computing systems. In this chapter, we propose a model, based on multi-constraint hypergraph partitioning, for efficient data-parallel Web crawling. The model aims to balance the amount of data downloaded and stored by the processors as well as balancing the number of page download requests issued by the processors. The model also minimizes the total communication overhead incurred during the link exchange between the processors.

Section 2.1 makes an introduction to Web crawling and introduces a taxonomy of parallel Web crawling architectures. Section 2.2 presents an overview of the issues in parallel Web crawling. Section 2.3 surveys the previous work, mostly on data-parallel Web crawling. Section 2.4 defines the hypergraph partitioning problem. In Section 2.5, we present the proposed Web crawling model, which is based on hypergraph partitioning. In Section 2.6, performance results are provided on a sample Web repository for the proposed model. The chapter is concluded in Section 2.7 together with some future work.

2.1 Introduction

Web crawling is the process of locating, fetching, and storing the pages on the Web. The computer programs that perform this task are referred to as Web crawlers. The Web crawlers have vital importance for the search engines, which keep a cache of the Web pages for providing quick access to the information in them. In order to enlarge their cache and keep the information within up-to-date, search engines run crawlers to download the content on the Web. Unfortunately, only a few search engine designs [100] are published in the literature due to the commercial value they have. Similarly, the crawling process and the details of Web crawlers mostly remain as a black art.

In general terms, the working of a Web crawler is as follows. A typical Web crawler, starting from a set of seed pages, locates new pages by parsing the downloaded pages and extracting the hyperlinks (in short links) within. Extracted links are stored in a FIFO fetch queue for further retrieval. Crawling continues until the fetch queue gets empty or a satisfactory number of pages are downloaded. In short, the link structure of the Web is followed to explore and retrieve the content on the Web. Usually, many crawler threads execute concurrently in order to overlap network operations with the processing in the CPU, thus increasing the throughput of page download.

The dynamically changing topology of the Web (new page additions and deletions, changes in the inter-page links), and the changes in pages' content requires the crawling process to be a continuous process. Furthermore, due to the enormous size of the Web and the limitations on data transfer rates at accessing the pages, crawling is a slow process. It is reported by the Google search engine that crawling the whole Web requires a full month of downloading even with the huge computing infrastructure Google has. Currently, crawling the Web by means of sequential computing systems is infeasible due to the need for vast amounts of storage, computational power, and high download rates.

The recent trend in construction of cost-effective PC clusters makes the Web crawling problem an appropriate target for parallel computing. In parallel Web

crawling, each processor is responsible from downloading a subset of the pages. The processors can be coordinated in three different ways: independent, master-slave, and data-parallel. In the first approach, each processor independently traverses a portion of the Web and downloads a set of pages pointed by the links it discovered. Since some pages are fetched multiple times, in this approach, there is an overlap problem, and hence, both storage space and network bandwidth are wasted. In the second approach, each processor sends its links, extracted from the pages it downloaded, to a central coordinator. This coordinator, then assigns the collected URLs to the crawling processors. The weakness of this approach is that the coordinating processor becomes a bottleneck. In the third approach, pages are partitioned among the processors such that each processor is responsible from fetching a non-overlapping subset of the pages. Since some pages downloaded by a processor may have links to the pages in other processors, these inter-processor links need to be communicated in order to obtain the maximum page coverage and to prevent the overlap of downloaded pages. In this approach, each processor freely exchanges its inter-processor links with the others.

In this work, our focus is on data-parallel Web crawling architectures. In these architectures, the partitioning of the Web among the processors (i.e., page-to-processor assignment) is usually hierarchical or hash-based. The hierarchical approach assigns pages to processors according to URL domains. This approach suffers from the imbalance in processor workloads since some domains contain more pages than the others. In the hash-based approach, either single pages or sites as a whole are assigned to the processors. This approach solves the load balancing problem implicitly. However, in this approach, there is a significant communication overhead since inter-processor links, which must be communicated, are not considered while creating the page-to-processor assignment.

2.2 Issues in Parallel Crawling

The working of parallel crawling system is somewhat similar to that of a sequential crawling system. However, there exist several issues [39] in assignment of Web

pages to crawlers, coordination of crawler activities, and minimization of parallelization overheads. In this section, we present a discussion of the important issues in parallel Web crawling, some of which also apply to sequential crawling systems.

- *Overlap:* In a shared-nothing parallel crawling system, if crawlers are working independent of each other, there is a possibility that the same pages will be downloaded multiple times by different crawlers. This may result in an overhead in storage, use of network bandwidth, and use of processing resources. Therefore, a clever implementation should always avoid the download of the same page by more than one crawlers.

- *Page assignment:* To prevent overlaps, several techniques can be employed to assign pages to crawlers. In one approach, each page may be uniquely assigned to a crawler in the parallel system. A hash function may be used to map the URL of a page to a crawler. A more coarse-grain assignment approach is to assign sites to crawlers as a whole. For example, a crawler could be responsible from downloading Microsoft pages while another crawler downloads pages in the Yahoo site. An even coarser approach is to assign pages to crawlers depending on the URL domains. In this approach, for example, the pages in the .com domain may be downloaded by the same crawler, whereas the pages in the .edu domain are downloaded by another.

- *Coverage:* Another important issue is the ability to locate the pages. A successful crawling system should be able to locate the whole set of pages which are linked by other pages. If there is no communication among the crawlers (i.e., the Firewall scheme in [39]), it is possible that some pages on the Web will never be located.

- *Quality:* Depending on the path the pages are traversed, the quality of indexing may be greatly affected. In general, it is beneficial to crawl high quality pages earlier. In parallel Web crawling, if each crawler independently crawls its portion of the Web, the quality of the retrieved content may be worse than that of a sequential crawler.

- *Inter-processor communication:* In order to address the issues of coverage and quality, inter-processor communication is required. The crawlers pass the inter-processor links, of which source and destination pages are assigned to different processors, among themselves via point-to-point communication. This way, it becomes possible to locate the pages which are accessible by inter-processor links. The frequency that the inter-processor links are passed also determines the quality of the crawling. In general, if the links are more frequently communicated, the quality of the page scores increases.

- *Subnetwork/Web server overload:* During the crawling process, the Web servers should not be overwhelmed with download requests from the crawlers. A crawler that tries to download a whole site in a short amount of time may turn into a denial of service attack. A clever crawling system should be able to distribute the page download requests submitted to the Web servers in a balanced manner. A similar issue arises for the subnetworks. The bandwidth consumption must be balanced, and no subnetworks must be overwhelmed with requests.

- *Revisit frequency:* It should take a similar amount of time for the crawlers to crawl their portions of the Web. This way, freshness of the indexed pages may be close to optimum. An unbalanced load distribution may cause some pages to be crawled several times, whereas some pages are not crawled at all. An adaptive page revisit strategy may be superior in that frequently updated pages are also frequently crawled.

2.3 Previous Work

In the literature, there are many studies concentrating on different issues in Web crawling, such as URL ordering for retrieving high-quality pages earlier [8, 41], partitioning the Web for efficient multi-processor crawling [21, 112], distributed crawling [15, 131], and focused crawling [35, 50]. Despite this vast amount of effort, due to the commercial value of the developed applications, it is still difficult

to obtain robust, and customizable crawling software [68, 109].

The page-to-processor assignment problem in data-parallel Web crawling was addressed by a number of authors. Cho and Garcia-Molina [39] used the site-hash-based assignment technique with the belief that it will reduce the number of inter-processor links when compared to the page-hash-based assignment technique. Boldi et al. [15] applied the consistent hashing technique, a method assigning more than one hash values for a site, in order to handle the failures among the crawling processors. Teng et al. [112] used a hierarchical, bin-packing-based page-to-processor assignment approach. Cambazoglu et al. [22] proposed a graph-partitioning-based model for page-to-processor assignment. This model correctly encapsulates the total volume of communication during the link exchange. The same authors recently proposed another model [117], which encapsulates the number of messages transmitted during the link exchange. In both models, the page storage amounts and number of page download requests of the processors are balanced. The model proposed in this work combines these graph- and hypergraph-partitioning-based models into a single model.

2.4 Hypergraph Partitioning Problem

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$ [12]. Each net $n_j \in \mathcal{N}$ connects a subset of vertices in $\mathcal{V}$. The set of vertices connected by a net n_j are called as the pins of net n_j. Multiple weights $w_i^1, w_i^2, \ldots, w_i^M$ may be associated with a vertex $v_i \in \mathcal{V}$. A cost c_j is assigned as the cost of each net $n_j \in \mathcal{N}$.

$\Pi=\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is said to be a K-way partition of $\mathcal{H}$ if each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, parts are pairwise disjoint, and the union of the K parts is equal to $\mathcal{V}$. A partition Π is said to be balanced if each part $\mathcal{V}_k$ satisfies the balance criteria

$$W_k^m \leq (1+\epsilon)W_{\text{avg}}^m, \qquad \text{for } k=1,2,\ldots,K \text{ and } m=1,2,\ldots,M. \tag{2.1}$$

In Eq. 2.1, each weight W_k^m of a part $\mathcal{V}_k$ is defined as the sum of the weights w_i^m of the vertices in that part. W_{avg}^m is the weight that each part should have in the case of perfect load balancing. ϵ is the maximum imbalance ratio allowed.

In a partition Π of $\mathcal{H}$, an edge is said to be cut if its pair of vertices fall into two different parts and uncut otherwise. In Π, a net is said to connect a part if it has at least one pin in that part. The connectivity set Λ_j of a net n_j is the set of parts connected by n_j. The connectivity $\lambda_j = |\Lambda_j|$ of a net n_j is equal to the number of parts connected by n_j. If $\lambda_j = 1$, then n_j is an internal net. If $\lambda_j > 1$, then n_j is an external net and is said to be at cut.

After these definitions, the K-way, multi-constraint hypergraph partitioning problem can be stated as the problem of dividing a hypergraph into two or more parts such that a partitioning objective defined over the nets is minimized while the multiple balance criteria (Eq. 2.1) on the part weights are maintained. In this work, as the partitioning objective, we use the connectivity-1 metric

$$\chi(\Pi) = \sum_{n_i \in \mathcal{N}} c_i(\lambda_i - 1), \tag{2.2}$$

in which each net contributes $c_i(\lambda_i - 1)$ to the cost $\chi(\Pi)$ of a partition Π.

2.5 Parallel Web Crawling Model

In this section, we propose a model based on multi-constraint hypergraph partitioning for load-balanced and communication-efficient data-parallel crawling. A major assumption in our model is that the crawling system runs in sessions. Within a session, if a page is downloaded, it is not downloaded again, that is, each page can be downloaded just once in a session. The crawling system, after downloading enough number of pages, decides to start another crawl session and recrawls the Web. For efficient crawling, our model utilizes the information (i.e., the Web graph) obtained in the previous crawling session and provides a better page-to-processor mapping for the following crawling session. We assume that

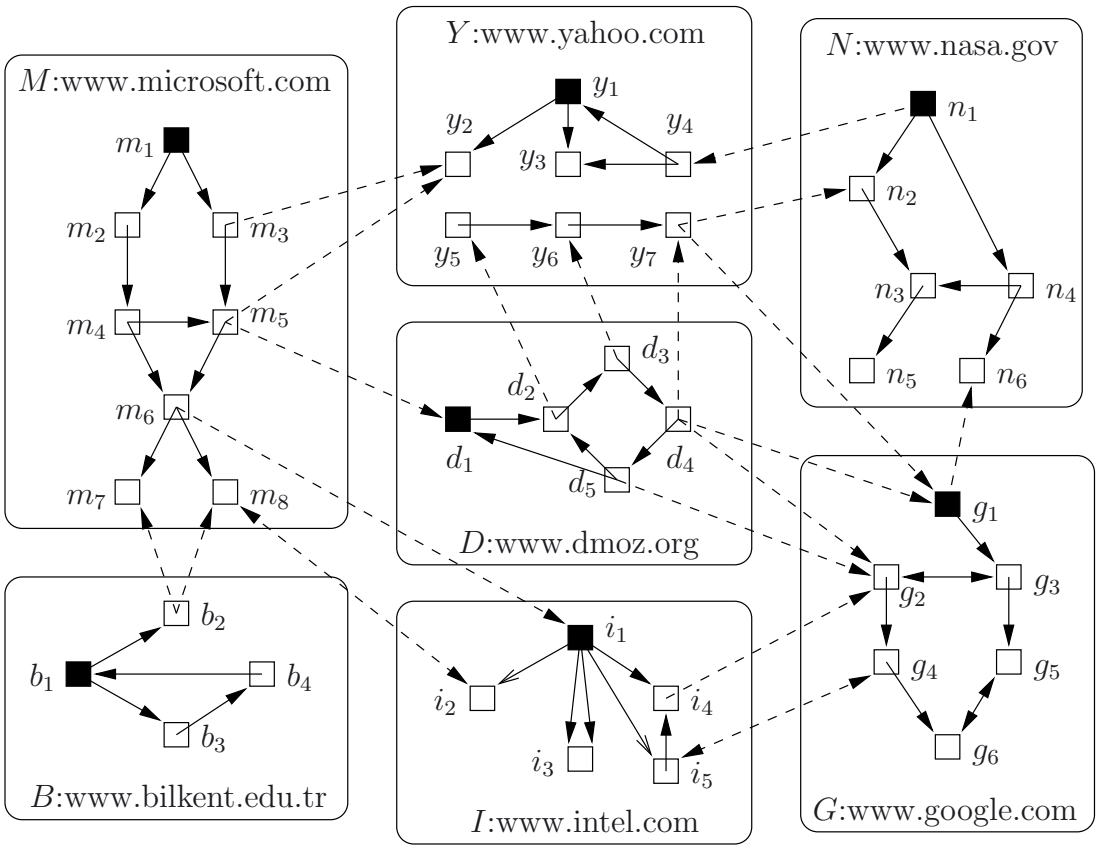


Figure 2.1: An example to the graph structure of the Web.

between two consecutive sessions, there are no drastic changes in the Web graph in terms of page sizes and the topology of the links.

We describe the proposed model using the sample Web graph shown in Figure 2.1. In this graph, which is assumed to be created in the previous crawling session, there are 7 sites. Each site contains several pages, which are represented by small squares. The directed lines between the squares represent the links between the pages. There may be multi-links (e.g., (i_1, i_3)) and bidirectional links between the pages (e.g., (g_5, g_6)). In the figure, inter-site links are displayed as dashed lines. In presentation of the model, we will assume that, in Figure 2.1, each page contains a unit amount of text, and each link has a unit size.

In our model, we represent the link structure between the pages by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$. In this representation, each page p_i corresponds to a vertex

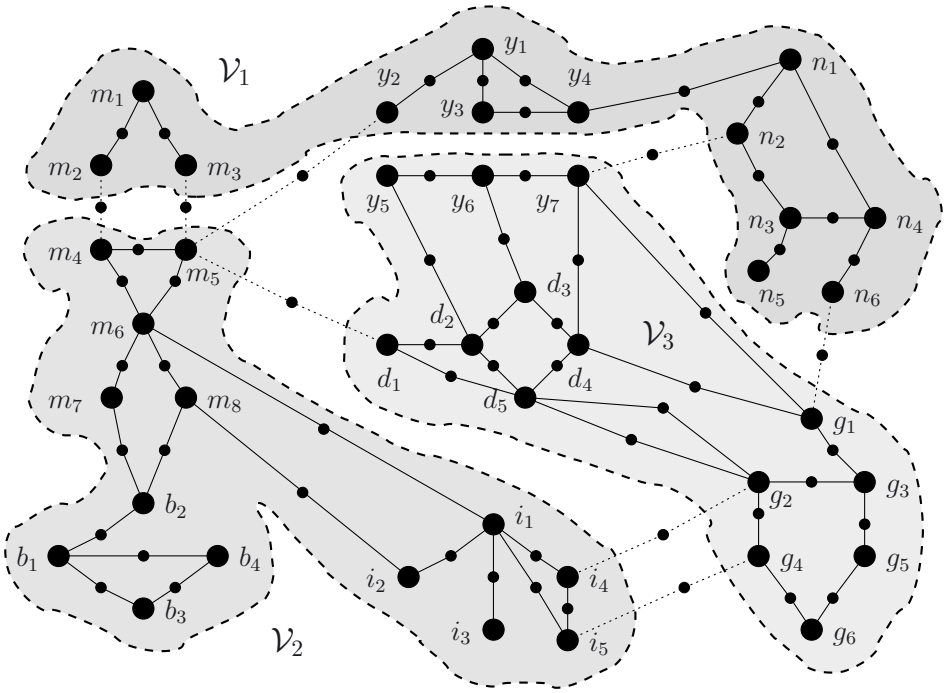
v_i. Two weights, w_i^1 and w_i^2, are associated with each vertex v_i. The weight w_i^1 of vertex v_i is equal to the size (in bytes) of page p_i and represents the download and storage overhead for page p_i. The weight w_i^2 of vertex v_i is equal to 1 and represents the overhead of requesting p_i. This overhead mainly involves the cost of domain name resolution for the page URL.

There are two types of nets in $\mathcal{N}$: two-pin nets and multi-pin nets. There exists a two-pin net n_j between vertices v_h and v_i if and only if page p_h has a link to page p_i or vice versa. Multiple links between the same pair of pages are collapsed into a single two-pin net. The cost c_j of a two-pin net n_j is equal to the total string length (in bytes) of the links (p_i, p_j) and (p_j, p_i) (if any) between pages p_i and p_j divided by the transfer rate of the network (in MB/s). This cost corresponds to the communication overhead of transmitting the links between two processors via point-to-point communication over the network in case p_i and p_j are mapped to different processors.

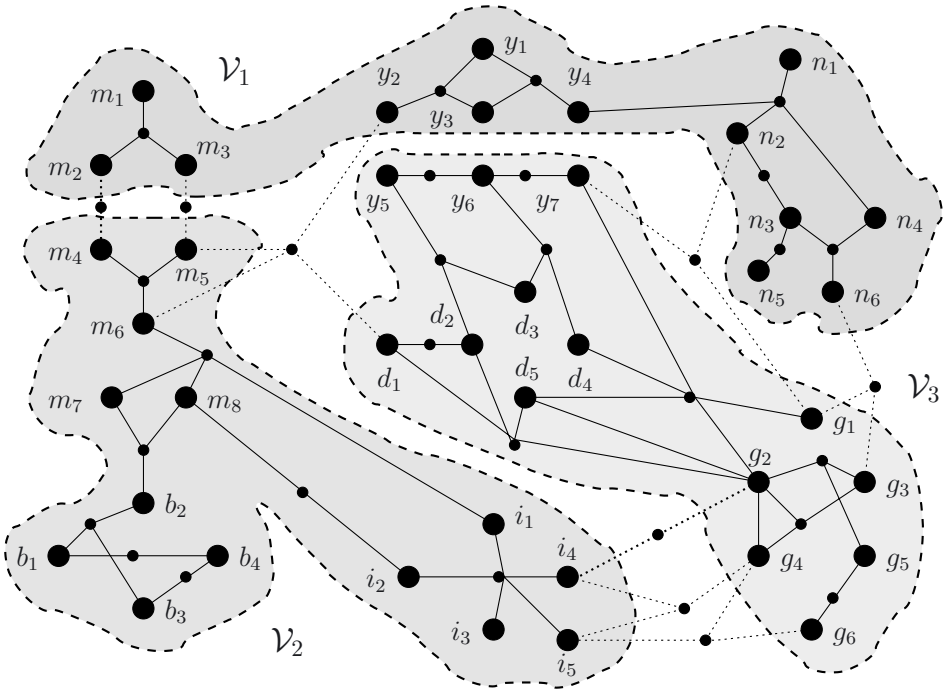
For each page p_i that has one or more outgoing links to other pages, a multi-pin net n_i is placed in the hypergraph. Vertex v_i and the vertices corresponding to the pages linked by p_i form the pins of the multi-pin net n_i. As the cost c_i of multi-pin net n_i, a fixed message startup cost (in seconds) is assigned. This cost represents the cost of preparing a single network packet containing the links of page p_i.

In a K-way partition $\Pi = (\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K)$ of hypergraph $\mathcal{H}$, each vertex part $\mathcal{V}_k$ corresponds to a subset $\mathcal{P}_k$ of pages to be downloaded by processor P_k. That is, every page $p_i \in \mathcal{P}_k$, represented by a vertex $v_i \in \mathcal{V}_k$, is fetched and stored by processor P_k. In this model, maintaining the balance on part weights W_k^1 and W_k^2 (Eq. 2.1) in partitioning hypergraph $\mathcal{H}$, respectively balances the download and storage overhead of processors as well as the number of page download requests issued by the processors. Minimizing the partitioning objective $\chi(\Pi)$ (Eq. 2.2) corresponds to minimizing the total overhead of inter-processor communication that will be incurred during the link exchange between the processors.

Figure 2.2 shows a 3-way partition for the hypergraph corresponding to the sample Web graph in Figure 2.1. For simplicity, the two-pin nets and multi-pin



(a) Only two-pin nets displayed.



(b) Only multi-pin nets displayed.

Figure 2.2: A 3-way partition of the hypergraph representing the sample Web graph in Figure 2.1.

nets are separately displayed in Figures 2.2(a) and 2.2(b), respectively. In this example, almost perfect load balance is obtained since weights (for both weight constraints) of the three vertex parts $\mathcal{V}_1$, $\mathcal{V}_2$, and $\mathcal{V}_3$ are respectively 13, 14, and 14. Hence, according to this partition, each processor P_k, which is responsible from downloading all pages $p_i \in \mathcal{P}_k$, is expected to fetch and store almost equal amounts of data in the next crawling session. In the figure, the pins of the cut nets are displayed with dotted lines. In Figure 2.2(a), two-pin cut nets represent the inter-processor links, which must be communicated between the processors. For example, due to the two-pin net connecting vertices m_5 and d_1 a link is transferred from processor P_2 to P_1. In Figure 2.2(b), multi-pin nets represent the message startup cost of processors. The connectivity-1 cost incurred to the cut by a multi-pin nets gives the number of processors to which a message must be send. For example, due to the cut net which connects vertices m_5, m_6, y_2, and d_1, processor P_2 must send a message to $3-1=2$ processors (i.e., P_1 andP_3). The total number of messages is $(3-1)\times 1+(2-1)\times 7=9$.

2.6 Experiments

2.6.1 Dataset

Experiments are conducted on a large (8 GB) Web repository, made publicly available by Google Inc.[1]. There are 913,569 Web pages in this repository. The number of links between the pages is 4,480,218. There are 680,199 multi-pin nets and 4,480,218 two-pin nets in the hypergraph representing the repository. The number of multi-pin nets is less than the number of Web pages in the repository since some pages do not contain links to other pages. Average net size is 7.59 for multi-pin nets. The total number of pins is 14,120,853. The number of pins due to the multi-pin and two-pin nets are respectively 5,160,417 and 8,960,436.

[1]Google Web repository. Available at: `http://www.google.com/programming-contest/`

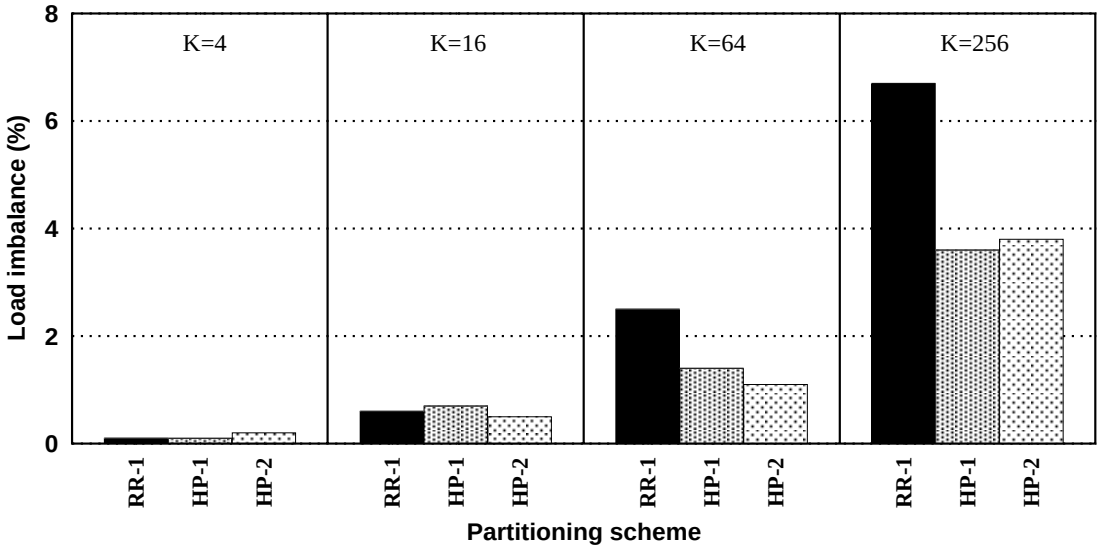


Figure 2.3: The load imbalance in the number of page download requests and storage loads with increasing number K of processors.

2.6.2 Results

We conducted experiments comparing two Web partitioning schemes, RR and HP. The RR scheme is the round-robin assignment scheme, in which pages are assigned to processors in a round-robin fashion. This scheme corresponds to the hash-based page assignment scheme previously used in the literature. The HP scheme is the hypergraph-partitioning-based page assignment scheme introduced in this work. For multi-constraint partitioning of the constructed hypergraph, the state-of-the-art hypergraph partitioning tool PaToH [33] is used with default parameters. The maximum allowed imbalance ratio is set to 0.01 for both constraints. In the experiments, a Gigabit network with a 7.6 ns/byte transfer rate and a fixed message startup cost of 100 ns is assumed.

Figure 2.3 displays the imbalance values obtained by the RR and HP schemes. In the figure, RR-1 and HP-1 represent the page storage imbalance for the RR and HP schemes, respectively. HP-2 represents the imbalance in the number of page download requests issued by the processors. Since RR almost perfectly balances the number of page download requests for all numbers of processors, these results are not displayed. According to Figure 2.3, HP performs better in load balancing

Table 2.1: Communication costs (in seconds) of the partitioning schemes with increasing number K of processors

	Message startup		Link transfer		Total cost	
K	RR	HP	RR	HP	RR	HP
2	58.4	3.7	680.8	18.0	739.2	21.7
4	139.0	6.6	1021.6	30.5	1160.7	37.1
8	229.9	9.8	1191.8	43.3	1421.7	53.1
16	310.9	11.2	1276.9	48.4	1587.8	59.6
32	368.7	12.5	1319.4	52.2	1688.0	64.8
64	404.3	13.3	1340.6	55.1	1744.9	68.4
128	424.9	15.2	1351.4	63.4	1776.2	78.6
256	436.0	18.4	1356.7	76.6	1792.6	95.0

especially as the number of processors increases. At small numbers of processors, the RR scheme already achieves good imbalance values. The HP scheme display almost similar behavior in balancing the storage load (HP-1) and the number of page download requests (HP-2).

Since there is a large performance gap between the RR and HP schemes in minimizing the communication overhead, we display the experimental results as a table for better visibility. Table 2.1 contains the total message startup and data transfer costs observed (in seconds) during the link exchange with increasing number K of processors. On the average, the HP scheme performs around 95% better in reducing the costs of both message startup and link transfer. In general, the overhead due to the total message startup cost increases relatively faster than the overhead of link transfer with increasing number of processors. Although, in our scenario, the total message startup cost seems to be relatively less important, in a faster network (e.g., a 10Gb/s network), this overhead can be dominant. Overall, there is a considerable performance gain in reducing the total communication overhead in favor of the proposed hypergraph-partitioning-based model.

2.7 Conclusions and Future Work

In this chapter, we proposed a hybrid model, which combines two previously proposed Web crawling models. According to the theoretical experiments conducted, the model appears to be quite successful in minimizing the inter-processor communication overheads during the link exchange in data-parallel Web crawling systems. However, we believe that the experiments need to be repeated on a real-life system to observe the improvement in practice. As an on-going work, we are working on a site-based model, where, instead of pages, the sites are assigned to processors for download. This work will enable us to work on larger datasets, which, otherwise, we could not partition due to the memory limitations of the current sequential hypergraph partitioning tools.

Chapter 3

Inverted Index Partitioning Models

Shared-nothing, parallel text retrieval systems require an inverted index, representing a document collection, to be partitioned among a number of processors. In general, the index can be partitioned based on either the terms or documents in the collection, and the way the partitioning is done greatly affects the query processing performance of the system. In this chapter, we propose two novel inverted index partitioning models for efficient query processing on parallel text retrieval systems that employ the term- or document-based inverted index organizations [25]. The proposed models formulate the index partitioning problem as a hypergraph partitioning problem. Both models aim to balance the posting storage loads of processors. As the partitioning objective, the term-based partitioning model tries to minimize the total volume of communication, whereas the document-based model tries to minimize the total number of accesses to the disks during query processing.

The chapter is organized as follows. Section 3.1 introduces inverted indices and sequential query processing. Section 3.2 briefly presents parallel text retrieval architectures together with the inverted index organizations and query processing on intra-query-parallel architectures. Section 3.3 overviews the previous works on

inverted index partitioning. In Section 3.4, we provide some background and notation about hypergraph partitioning. Section 3.5 provides the details of the proposed inverted index partitioning models. Section 3.6 gives experimental results verifying the validity of the proposed work. Section 3.7 concludes.

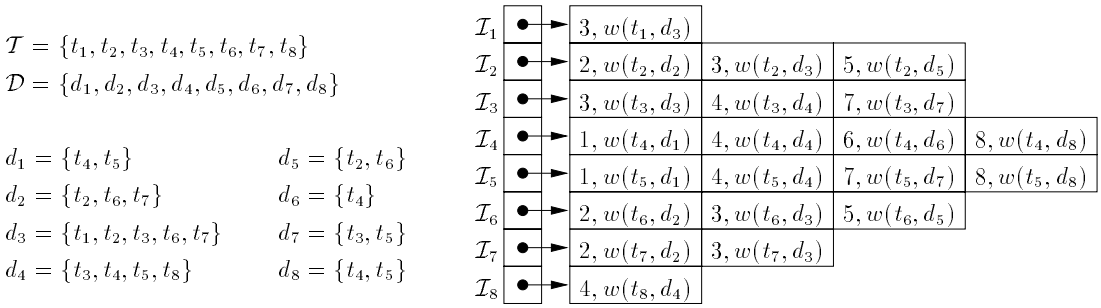
3.1 Introduction

3.1.1 Inverted Index Structure

The basic duty of a text retrieval system is to process user queries and present the users a set of documents relevant to their queries. For small document collections, processing of a query can be performed over the original collection via full text search. However, for efficient query processing over large collections, an intermediate representation of the collection (i.e., and indexing mechanism) is required. Until the early 90's signature files and suffix arrays were available as a choice for the text retrieval system designers. In the last decade, inverted index data structure replaced these structures and currently appears to be the only choice for indexing large document collections.

An inverted index is composed of a set of inverted lists $\mathcal{L} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_T\}$, where $T = |\mathcal{T}|$ is the size of the vocabulary $\mathcal{T}$ of the indexed document collection $\mathcal{D}$, and an index pointing to the heads of the inverted lists. The index part is usually small to fit into the main memory, but inverted lists are stored on the disk. Each list $\mathcal{I}_i \in \mathcal{L}$ is associated with a term $t_i \in \mathcal{T}$. An inverted list contains entries (called postings) for the documents containing the term it is associated with. A posting $p \in \mathcal{I}_i$ consists of a document id field $p.d = j$ and a weight field $p.w = w(t_i, d_j)$ for a document d_j in which term t_i appears. $w(t_i, d_j)$ is a weight which shows the degree of relevance between t_i and d_j using some metric.

Figure 3.1(a) shows the toy document collection that we will use throughout the examples in this chapter. This document collection $\mathcal{D}$ contains $D = 8$ documents, and its vocabulary $\mathcal{T}$ has $T = 8$ terms. There are $P = 21$ posting entries,



(a) Toy collection (b) Inverted index structure

Figure 3.1: The toy document collection used throughout the chapter.

in the set $\mathcal{P}$ of postings. Figure 3.1(b) shows the inverted index built for this document collection.

3.1.2 Query Processing

In query processing, it is important to pick the related documents and present them to the user in the order of documents' similarity to the query. For this purpose, many models have been proposed in the literature [125]. Some examples are the boolean, vector-space, fuzzy-set, and probabilistic models. Among these, the vector-space model, due to its simplicity, robustness, speed, and ability to catch partial matches, is the most widely accepted model [104].

In the vector-space model, the similarity $\text{sim}(\mathcal{Q}, d_j)$ between a query $\mathcal{Q} = \{t_{q_1}, t_{q_2}, \ldots, t_{q_Q}\}$ of size Q and a document d_j is computed using the cosine similarity measure, which can be simplified as

$$\text{sim}(\mathcal{Q}, d_j) = \frac{\sum_{i=1}^{Q} w(t_{q_i}, d_j)}{\sqrt{\sum_{i=1}^{Q} w(t_{q_i}, d_j)^2}}, \tag{3.1}$$

assuming all query terms have equal importance. The *tf-idf* (term frequency-inverse document frequency) weighting scheme [104] is usually used to compute the weight $w(t_i, d_j)$ of a term t_i in a document d_j as

$$w(t_i, d_j) = \frac{f(t_i, d_j)}{\sqrt{|d_j|}} \times \ln \frac{D}{f(t_i)}, \tag{3.2}$$

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. Throughout the thesis, the *tf-idf* weighting scheme is used together with the vector-space model [125].

Processing of a user query follows several stages in a traditional sequential text retrieval system. While processing a user query $\mathcal{Q} = \{t_{q_1}, t_{q_2}, \ldots, t_{q_Q}\}$, each query term t_{q_i} is considered in turn and is processed as follows. First, inverted list $\mathcal{I}_{q_i}$ is fetched from the disk. All postings in $\mathcal{I}_{q_i}$ are traversed, and the weight $p.w$ in each posting $p \in \mathcal{I}_{q_i}$ is added to the score accumulator for document $p.d$. After all inverted lists are processed, documents are sorted in decreasing order of their scores, and highly-ranked documents are returned to the user. The interested reader may refer to Chapter 4 for more details on sequential query processing.

3.2 Parallel Text Retrieval

3.2.1 Parallel Text Retrieval Architectures

In practice, parallel text retrieval architectures can be classified as: inter-query-parallel and intra-query-parallel architectures. In the first type, each processor in the parallel system works as a separate and independent query processor. Incoming user queries are directed to client query processors on a demand-driven basis. Processing of each query is handled solely by a single processor. Intra-query-parallel architectures are typically composed of a single central broker and a number of client processor, each running an index server responsible from a portion of the inverted index. In this architecture, the central broker redirects an incoming query to all client query processors in the system. All processors collaborate and contribute processing of the query and compute partial answer sets

of documents. The partial answer sets produced by the client query processors are merged at the central broker into a final answer set, as a final step.

In general, inter-query-parallel architectures obtain better query processing throughput, whereas intra-query-parallel architectures are better at reducing query response times. Further advantages, disadvantages, and a brief comparison are provided in [9]. In this work, our focus is on intra-query-parallel text retrieval systems on shared-nothing parallel architectures.

3.2.2 Inverted Index Organizations

In a K-processor, shared-nothing, intra-query-parallel text retrieval system, the inverted index is partitioned among K index servers. The partitioning must be performed taking the storage load of index servers into consideration. If there are $|\mathcal{P}|$ posting entries in the inverted index, each index server S_j in the set $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$ of index servers should keep approximately equal amount of posting entries as shown by

$$\text{SLoad}(S_j) \simeq \frac{|\mathcal{P}|}{K}, \qquad \text{for } 1 \leq j \leq K, \tag{3.3}$$

where $\text{SLoad}(S_j)$ is the storage load of index server S_j. The storage imbalance should be kept under a satisfactory value.

In general, partitioning of the inverted index can be performed in two different ways: term-based or document-based partitioning. In the term-based partitioning approach, each index server S_j locally keeps a subset $\mathcal{L}_j^{\text{t}}$ of the set $\mathcal{L}$ of all inverted lists, where

$$\mathcal{L}_1^{\text{t}} \cup \mathcal{L}_2^{\text{t}} \cup \ldots \cup \mathcal{L}_K^{\text{t}} = \mathcal{L} \tag{3.4}$$

with the condition that

$$\mathcal{L}_i^{\mathrm{t}} \cap \mathcal{L}_j^{\mathrm{t}} = \emptyset, \qquad \text{for } 1 \leq i, j \leq K, i \neq j. \tag{3.5}$$

In this technique, all processors are responsible from processing their own set of terms, that is, inverted lists are assigned to index servers as a whole. If an inverted list $\mathcal{I}_i$ is assigned to index server S_j (i.e., $\mathcal{I}_{ji}^{\mathrm{t}} = \mathcal{I}_i$), any index server $\mathcal{S}_k$ other than $\mathcal{S}_j$ has $\mathcal{I}_{ki}^{\mathrm{t}} = \emptyset$.

Alternatively, the partitioning can be based on documents. In the document-based partitioning approach, each processor is responsible from a different set of documents, and an index server stores only the postings that contain the document ids assigned to it. Each index server S_j keeps a set $\mathcal{L}_j^{\mathrm{d}} = \{\mathcal{I}_{j1}, \mathcal{I}_{j2}, \ldots, \mathcal{I}_{jT}\}$ of inverted lists containing subsets $\mathcal{I}_{ji}^{\mathrm{d}}$ of every inverted list $\mathcal{I}_i \in \mathcal{L}$, where

$$\mathcal{I}_{1i}^{\mathrm{d}} \cup \mathcal{I}_{2i}^{\mathrm{d}} \cup \ldots \cup \mathcal{I}_{Ki}^{\mathrm{d}} = \mathcal{I}_i, \qquad \text{for } 1 \leq i \leq T \tag{3.6}$$

with the condition that

$$\mathcal{I}_{ji}^{\mathrm{d}} \cap \mathcal{I}_{ki}^{\mathrm{d}} = \emptyset, \qquad \text{for } 1 \leq j, k \leq K, j \neq k, 1 \leq i \leq T, \tag{3.7}$$

and it is possible to have $\mathcal{I}_{ji}^{\mathrm{d}} = \emptyset$.

In Figure 3.2(a) and Figure 3.2(b), the term- and document-based partitioning strategies are illustrated on our toy document collection for a 3-processor parallel system. The approach followed in this example is to assign the postings to processors in a round-robin fashion according to term and document ids. This technique is used in [114].

3.2.3 Parallel Query Processing

Processing of a query in a parallel text retrieval system follows several steps. These steps slightly differ depending on whether term-based or document-based

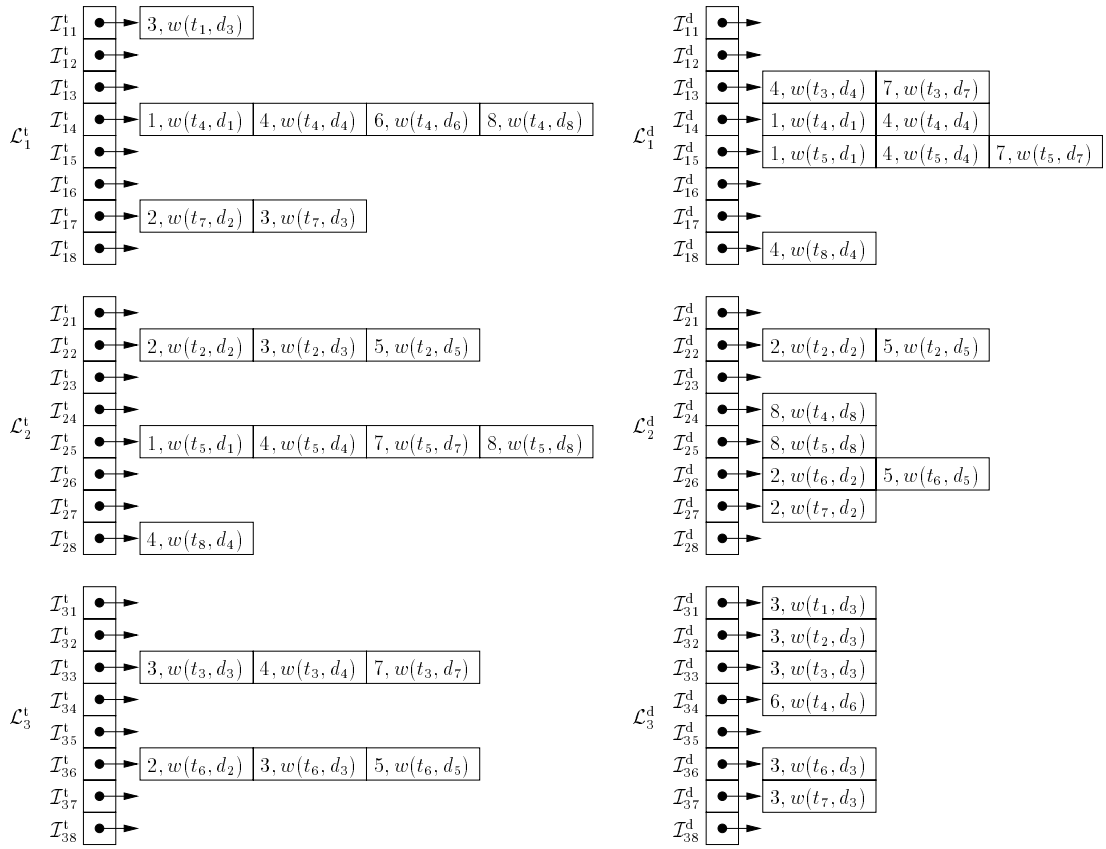


Figure 3.2: 3-way term- and document-based partitions for the inverted index of our toy collection.

inverted index partitioning schemes are employed. In term-based partitioning, since the whole responsibility of a query term is assigned to a single processor, the central broker splits a user query $\mathcal{Q} = \{t_{q_1}, t_{q_2}, \ldots, t_{q_Q}\}$ into K subqueries. Each subquery $\mathcal{Q}_i$ contains the query terms whose responsibility is assigned to index server S_i, that is, $\mathcal{Q}_i = \{q_j : t_{q_j} \in \mathcal{Q} \wedge \mathcal{I}_{q_j} \in \mathcal{L}_i^{\mathrm{t}}\}$. Then, the central broker sends the subqueries over the network to index servers. Depending on the query content, it is possible to have $\mathcal{Q}_i = \emptyset$, in which case no subquery is sent to index server S_i. In document-based partitioning, postings of a term are distributed on many processors. Hence, unless a $K \times T$-bit term-to-processor mapping is stored in the central broker, each index server is sent a copy of the original query, that is, $\mathcal{Q}_i = \mathcal{Q}$.

Once an index server receives a subquery, it immediately accesses its disk and reads the inverted lists associated with the terms in the subquery. For each query term $t_{q_j} \in \mathcal{Q}_i$, inverted list $\mathcal{I}_j$ is fetched from the disk. The weight $p.w$ of each posting $p \in \mathcal{I}_j$ is used to update the corresponding score accumulator for document $p.d$. When all inverted lists are read and accumulator updates are completed, index server S_i transfers the accumulator entries (document ids and scores) in the memory to the central broker over the network, forming a partial answer set $\mathcal{A}_i$ for query $\mathcal{Q}$.

During this period, the central broker may be busy with directing other queries to index servers. For the final answer set to the query to be generated, all partial answer sets related with the query must be gathered at the central broker. The central broker merges the received K partial answer sets $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_K$ and returns the most relevant (highly-ranked) document ids as the complete answer set to the user submitted query $\mathcal{Q}$.

3.2.4 Evaluation of Inverted Index Organizations

The term-based and document-based partitioning schemes have their own advantages and disadvantages. In the term-based partitioning scheme, accessing a term's inverted list requires a single disk access, but reading the list may take long time since the whole list is stored at a single index server. Similarly, the partial answer sets transmitted by the index servers are long. Hence, the overhead of term-based partitioning is mainly at the communication. The communication overhead becomes a bottleneck in parallel architectures where the communication-to-computation ratio is low, or in the case that the entire set of inverted lists are stored in the primary memory, or in cases where the partial answer sets contain additional information such as the positions of the terms in the documents. Previously proposed term-based partitioning schemes do not take this communication overhead into consideration during the partitioning of the inverted index.

In document-based partitioning, the inverted lists retrieved from the disk are shorter in length, and hence posting I/O is faster. Moreover, in case the user

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

Authors	Tomasic and Garcia-Molina	Jeong and Omiecinski	Riberio-Neto and Baeza-Yates
Year	1993	1995	1999
Target architecture	shared-nothing parallel	multi-disk PC	shared-nothing parallel
Ranking model	boolean	boolean	vector-space
Partitioning model	round-robin	load-balanced	load-balanced
Dataset	synthetic	synthetic	real-life

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. However, in document-based partitioning, $O(K)$ disk accesses are required to read the inverted list of a term since the complete list is distributed at many processors. The disk accesses are the dominating overhead in total query processing time, especially in the presence of slow disks and a fast network. If the documents are assigned to sites in a random manner, as done in the previous works, too many disk accesses may be observed.

3.3 Previous Works

There are a number of works on inverted index partitioning problem in parallel text retrieval systems. We briefly overview three publications here. Table 3.1 summarizes and compares these previous works on inverted index partitioning.

Tomasic and Garcia-Molina [114] examine four different techniques to partition the inverted index on a shared-nothing distributed system for different hardware configurations. The system and disk organizations described in their work respectively correspond to the term- and document-based partitioning schemes we previously described. The authors verify the performance of the techniques by

simulation over a synthetic dataset and use the boolean model for similarity calculations between documents and queries. Their results indicate that document-based partitioning performs well for long documents, whereas term-based partitioning is better on short-document collections.

Jeong and Omiecinski [73] investigate the performance of the two partitioning schemes for a shared-everything multiprocessor system with multiple disks. As in [114], they use the boolean ranking model and work on synthetic datasets. They conduct experiments especially on term skewness. For term-based partitioning, they propose two heuristics for load balancing. In their first heuristic, they partition the posting file with equal posting size instead of equal number of terms. In their second heuristic, they also consider the term frequencies besides posting sizes. The results of their simulation show that term-based partitioning is better when term distribution is less skewed in the document collection, and document-based partitioning should be preferred otherwise.

Baeza-Yates and Ribeiro-Neto [103] apply the two partitioning schemes on a shared-nothing parallel system. In their work, they refer to the term- and document-based partitioning schemes as global and local index organizations, respectively. For document ranking, they use the vector-space model and conduct their experiments on a real-life document collection. Their results show that term-based partitioning performs better than document-based partitioning in the presence of fast communication channels.

3.4 Hypergraph Partitioning Overview

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$ [12]. Each net $n_j \in \mathcal{N}$ connects a subset of vertices in $\mathcal{V}$. The set of vertices connected by a net n_j are called as the pins of net n_j and are denoted as $Pins(n_j)$. The size of a net n_j is equal to the number of its pins, that is, $size(n_j)=|Pins(n_j)|$. Similarly, the nets connecting a vertex v_i are called as the nets of a vertex and are denoted as $Nets(v_i)$. The degree of a vertex v_i is equal to the number of its

nets, that is, $deg(v_i) = |Nets(v_i)|$. Each vertex $v_i \in \mathcal{V}$ is associated with a weight w_i. Each net $n_j \in \mathcal{N}$ is associated with a cost c_j.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is a K-way vertex partition if each part $\mathcal{V}_k$ is nonempty, parts are pairwise disjoint, and the union of parts gives $\mathcal{V}$. In Π, a net is said to connect a part if it has at least one pin in that part. The connectivity set Λ_j of a net n_j is the set of parts connected by n_j. The connectivity $\lambda_j = |\Lambda_j|$ of a net n_j is equal to the number of parts connected by n_j. If $\lambda_j = 1$, then n_j is an internal net. If $\lambda_j > 1$, then n_j is an external net and is said to be at cut.

In Π, the weight of a part is equal to the sum of the weights of vertices in that part. A partition Π is said to be balanced if each part $\mathcal{V}_k$ satisfies the balance criterion

$$W_k \leq W_{\text{avg}}(1 + \epsilon), \qquad \text{for } k = 1, 2, \ldots, K, \tag{3.8}$$

where each weight W_k of a part $\mathcal{V}_k$ is defined as the sum of the weights w_i of the vertices in that part, W_{avg} is the weight that each part should have in the case of perfect load balancing, and ϵ is the maximum imbalance ratio allowed.

Given all these definitions, the K-way hypergraph partitioning problem [2] can be defined as finding a partition Π for a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ such that the balance criterion on part weights (Eq. 3.8) is maintained while an objective function defined over the nets is optimized. There are several objective functions developed and used in the literature. The metric used in this work is the connectivity metric

$$\chi(\Pi) = \sum_{n_i \in \mathcal{N}} c_i \lambda_i, \tag{3.9}$$

in which each net contributes $c_i \lambda_i$ to the cost $\chi(\Pi)$ of a partition Π.

3.5 Inverted Index Partitioning Models based on Hypergraph Partitioning

3.5.1 Proposed Work

In the previous works on term- and document-based inverted index partitioning, assignment of postings to index servers is performed without considering the association between the documents and terms. Here, we propose two novel inverted index partitioning models that lessens the overall query processing overhead in intra-query-parallel text retrieval systems. In the proposed models, the inverted index is viewed as a hypergraph and hypergraph partitioning heuristics are employed to obtain a partition of the inverted index. For simplicity, in our term-based model, we assume that the terms appear in the queries with equal probabilities. Similarly, in our document-based model, we assume that the documents are requested with equal probability. For both models, extensions are possible to the unequal probability case.

3.5.2 Term-Based Partitioning Model

In our hypergraph-partitioning model for term-based inverted index partitioning, the inverted index is represented as a hypergraph $\mathcal{H}^{\mathrm{t}} = (\mathcal{V}^{\mathrm{t}}, \mathcal{N}^{\mathrm{t}})$. The terms in the vocabulary $\mathcal{T}$ correspond to the vertices of vertex set $\mathcal{V}^{\mathrm{t}}$. That is, a term t_i, which is an atomic task to be completely processed by an individual index server, is represented as a vertex $v_i \in \mathcal{V}^{\mathrm{t}}$. As the weight of vertices, the number of posting entries in the inverted list of the corresponding term is assigned, that is, $w_i = PSize(t_i)$. This weight represents the storage overhead of inverted list $\mathcal{I}^{\mathrm{t}}_i$.

The documents in collection $\mathcal{D}$ correspond to the nets in the hypergraph. That is, we represent a document d_j by a net $n_j \in \mathcal{N}^{\mathrm{t}}$. The net costs are all set to 1, that is, $c_j = 1$. This cost represents the cost of transmitting an accumulator over the network. A pin is placed between a vertex v_i and a net n_j if and only if document d_j contains term t_i. Hence, in this model, the degree $|Nets(v_i)|$ of a

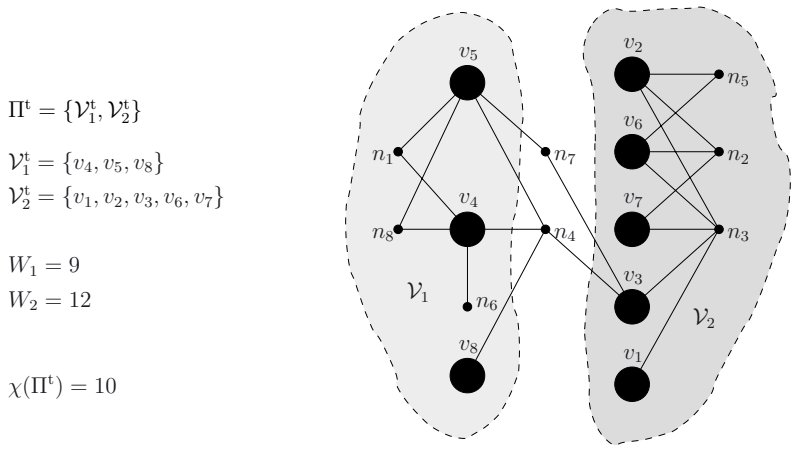
vertex v_i is equal to the number of postings in the inverted list $\mathcal{I}_i$ and also the vertex weight w_i.

In this model, a K-way partition $\Pi^{\mathrm{t}} = \{\mathcal{V}_1^{\mathrm{t}}, \mathcal{V}_2^{\mathrm{t}}, \ldots, \mathcal{V}_K^{\mathrm{t}}\}$ of hypergraph $\mathcal{H}^{\mathrm{t}}$ obtained by hypergraph partitioning corresponds to the set $\{\mathcal{L}_1^{\mathrm{t}}, \mathcal{L}_2^{\mathrm{t}}, \ldots, \mathcal{L}_K^{\mathrm{t}}\}$ of partial inverted indices to be distributed on K index servers $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_K\}$. Due to the load balance constraint in Eq. 3.8, it is guaranteed that each index server will have similar amounts of posting entries after the partitioning. Hence, a balance is obtained at the index servers in terms of posting storage.

In a partition Π^{t}, the connectivity λ_j of a net n_j shows the number of index servers that will transmit an accumulator for document d_j, and hence the volume of communication that will be incurred due to d_j. Consequently, minimizing the partitioning objective (Eq. 3.9) during the partitioning minimizes the total volume of communication that will be incurred during the transmission of accumulators from the index servers to the central broker in case each document in the collection is requested exactly once in query processing.

Figure 3.3 illustrates this with an example. Figure 3.3(a) shows hypergraph $\mathcal{H}^{\mathrm{t}}$ representing our toy inverted index and a 2-way partition Π^{t} obtained on this hypergraph. In Figure 3.3(b) and Figure 3.3(c), the resulting local inverted indices are displayed. According to this partition, net n_4 and n_7 are at the cut. Hence, when either document d_4 or d_7 is requested, both index servers will send accumulators about these documents. For internal nets, only one index server will send an accumulator. For example, although net n_3 is connected to many vertices, after the partitioning, it remains as an internal net, and hence, when document d_3 is requested only index server $\mathcal{S}_2$ will transmit an accumulator.

In the example of Figure 3.3, the part weights $W_1 = 9$ and $W_2 = 12$ correspond to the storage loads (i.e., the number of posting entries) for index servers $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. If each document is requested once, the total number of accumulators to be transmitted by the index servers is $2 \times 2 + 6 \times 1 = 10$, which is exactly equal to the λ-way cut cost of partition Π^{t}.



(a) A 2-way partition Π^t of hypergraph $\mathcal{H}^t$ for the term-based partitioning model.

$\mathcal{I}_{11}^t$				
$\mathcal{I}_{12}^t$				
$\mathcal{I}_{13}^t$				
$\mathcal{I}_{14}^t$	$1, w(t_4, d_1)$	$4, w(t_4, d_4)$	$6, w(t_4, d_6)$	$8, w(t_4, d_8)$
$\mathcal{I}_{15}^t$	$1, w(t_5, d_1)$	$4, w(t_5, d_4)$	$7, w(t_5, d_7)$	$8, w(t_5, d_8)$
$\mathcal{I}_{16}^t$				
$\mathcal{I}_{17}^t$				
$\mathcal{I}_{18}^t$	$4, w(t_8, d_4)$			

(b) Local inverted index $\mathcal{L}_1^t$ at index server $\mathcal{S}_1$.

$\mathcal{I}_{21}^t$	$3, w(t_1, d_3)$		
$\mathcal{I}_{22}^t$	$2, w(t_2, d_2)$	$3, w(t_2, d_3)$	$5, w(t_2, d_5)$
$\mathcal{I}_{23}^t$	$3, w(t_3, d_3)$	$4, w(t_3, d_4)$	$7, w(t_3, d_7)$
$\mathcal{I}_{24}^t$			
$\mathcal{I}_{25}^t$			
$\mathcal{I}_{26}^t$	$2, w(t_6, d_2)$	$3, w(t_6, d_3)$	$5, w(t_6, d_5)$
$\mathcal{I}_{27}^t$	$2, w(t_7, d_2)$	$3, w(t_7, d_3)$	
$\mathcal{I}_{28}^t$			

(c) Local inverted index $\mathcal{L}_2^t$ at index server $\mathcal{S}_2$.

Figure 3.3: A 2-way, term-based partition of the toy collection.

3.5.3 Document-Based Partitioning Model

Our document-based partitioning model uses the dual of the hypergraph in the previous model. In this representation, each document $d_i \in \mathcal{D}$ is represented by a vertex $v_i \in \mathcal{V}^d$. For each vertex v_i, the vertex weight w_i is set equal to the degree $|Nets(v_i)|$ of v_i. This weight shows the number of posting entries that must be stored for document d_i.

There exists a corresponding net $n_j \in \mathcal{N}^d$ for each term $t_j \in \mathcal{V}^d$ in vocabulary $\mathcal{T}$. The cost c_j of a net n_j is assigned as 1, i.e., $c_j = 1$. This cost represents the cost of the disk access for retrieving an inverted list from the disk. A vertex v_i is a pin of a net n_j if and only if term t_j appears in document d_i. In this setting, the degree $deg(v_i)$ of a vertex v_i is equal to the number of distinct terms in the document, that is, $|Nets(v_i)| = |d_i|$. Similarly, the size $|Pins(n_j)|$ of a

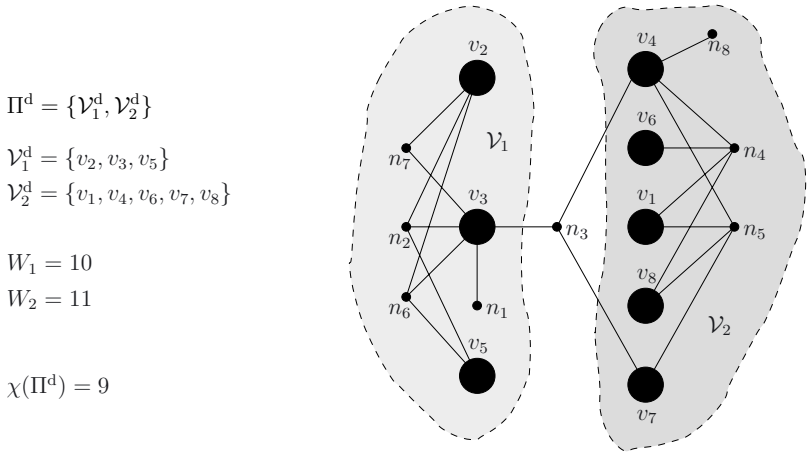
net n_j is equal to the number of documents in which term t_j appears, that is, $|Pins(n_j)|=|\mathcal{I}_j|$.

In a K-way partition $\Pi^{\mathrm{d}} = (\mathcal{V}_1^{\mathrm{d}}, \mathcal{V}_2^{\mathrm{d}}, \ldots, \mathcal{V}_K^{\mathrm{d}})$ of hypergraph $\mathcal{H}^{\mathrm{d}}$, each vertex part $\mathcal{V}_k^{\mathrm{d}}$ corresponds to a subset $\mathcal{D}_k$ of documents whose responsibility is assigned to index server S_k. In other words, every posting in the form of $(i, w(t_j, d_i))$ is stored by index server S_k if and only if $v_i \in \mathcal{V}_k^{\mathrm{d}}$ and $d_i \in \mathcal{D}_k$. Balancing the part weights W_k according to the balance criterion in Eq. 3.8 effectively balances the storage load of processors since each index server is assigned almost equal amount of postings.

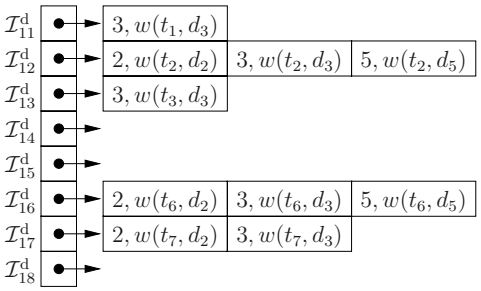
The connectivity set Λ_j of a net n_j corresponds to the set of index servers where inverted list $\mathcal{I}_j$ will be distributed. Each index server $\mathcal{S}_k$ stores a partial list $\mathcal{I}_{kj}^{\mathrm{d}}$ and responds to a subquery containing term t_j. The connectivity λ_j of a net n_j gives the number of disk accesses the overall system must perform to retrieve the inverted lists for term t_j. Consequently, minimizing the partitioning objective $\chi(\Pi)$ (Eq. 3.9) corresponds to minimizing the total number of disk accesses incurred in case every term in the vocabulary is submitted to the system as a single query.

Figure 3.4 illustrates this with an example. Figure 3.4(a) displays hypergraph $\mathcal{H}^{\mathrm{d}}$ representing our toy collection and its 2-way partition Π^{d} in the document-based model. In Figure 3.4(b) and Figure 3.4(c), the resulting local inverted indices are displayed. In partition Π^{d}, the only cut net is n_3, with a connectivity of $\lambda_3=2$. This means that when term t_3 appears in a query, it will be necessary to perform 2 disk accesses in the text retrieval system. All other terms will require a single disk access since their representative nets are all internal.

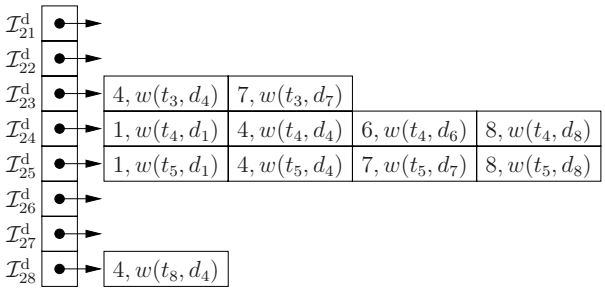
In the example of Figure 3.4, parts weights $W_1 = 10$ and $W_2 = 11$, as in the previous model, show the posting storage amounts of the two index servers. The connectivity cost $2\times1+7\times1=9$ of the partition indicates that if all terms in the vocabulary are submitted in a query, the total number of disk accesses incurred will be 9.



(a) A 2-way partition Π^{d} of hypergraph $\mathcal{H}^{\mathrm{d}}$ for the document-based partitioning model.



(b) Local inverted index $\mathcal{L}_1^{\mathrm{d}}$ at index server $\mathcal{S}_1$.



(c) Local inverted index $\mathcal{L}_2^{\mathrm{d}}$ at index server $\mathcal{S}_2$.

Figure 3.4: A 2-way, document-based partition of the toy collection.

3.6 Experimental Results

3.6.1 Dataset

In the experiments, Financial Times Limited (1991–1994) document collection, known as the FT database, of TREC Disk 4 is used. During the preparation of the global inverted index standard stop-word elimination and cleansing techniques are followed. After preprocessing, the collection obtained contains 210,157 documents, and the total number of distinct terms in the collection is 275,478. The total number of postings in the inverted index is 30,949,837. In the HP scheme, the maximum allowed imbalance ratio is set to 0.10. In the experiments conducted for the term-based inverted index partitioning, accumulators are assumed to be of 8 bytes.

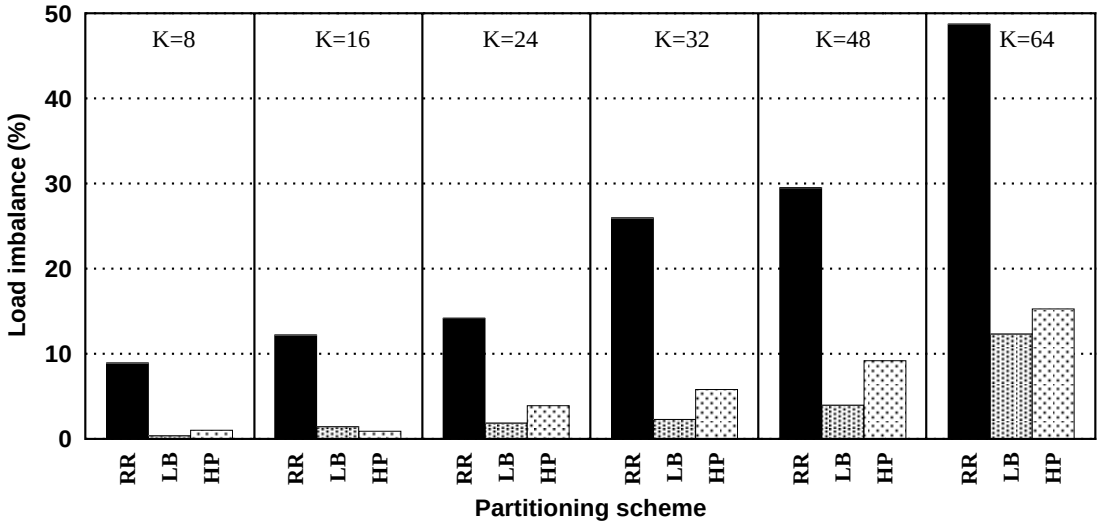


Figure 3.5: The load imbalance in posting storage with increasing number K of index servers in term-based inverted index partitioning.

We provided performance results for three different inverted index partitioning schemes: RR, LB, and HP. RR is the partitioning scheme employed in [114]. In this scheme, alphabetically sorted terms (or documents) are assigned to the index servers in a round-robin fashion. LB is the partitioning scheme introduced in [73]. In this scheme, terms are sorted in decreasing order of document frequency. Then, K parts are obtained on this sorted list such that each part contains almost equal amount of postings. We extend the same method to document-based partitioning. HP is the partitioning scheme of the proposed model. In this scheme, depending on the the type of the organization, a hypergraph representing the inverted index is created. Then, the state-of-the-art hypergraph partitioning tool PaToH [33] is used to partition this hypergraph.

3.6.2 Results on Term-Based Partitioning

Figure 3.5 shows the load imbalance values obtained by the three partitioning schemes in posting storage of index servers for term-based inverted index partitioning. As expected, the LB and HP schemes perform relatively better than the RB scheme in load balancing, due to the explicit effort towards balancing the posting storage. In general, LB performs slightly better than HP. At 64 index

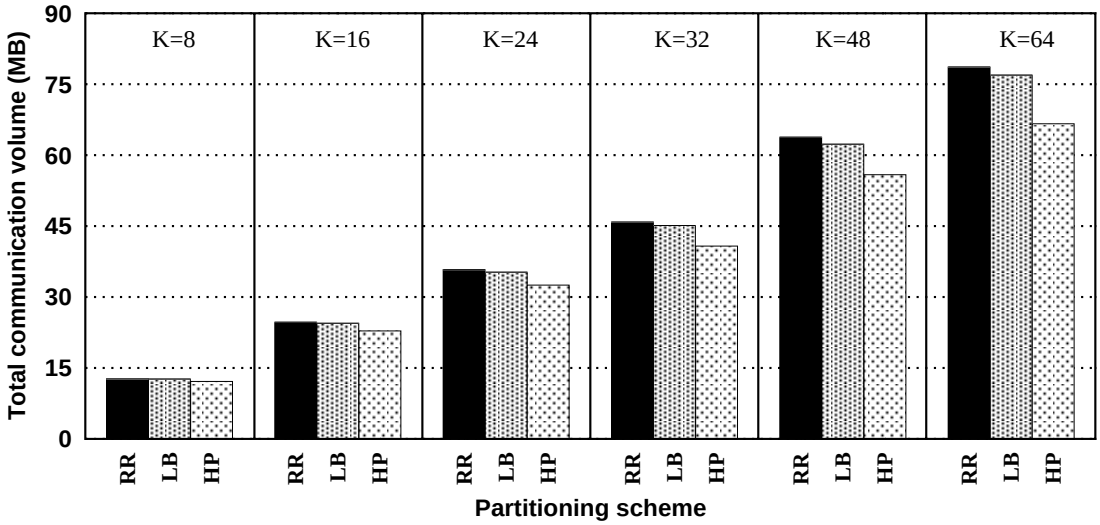


Figure 3.6: The total volume of communication incurred in query processing with increasing number K of index servers in term-based inverted index partitioning.

servers, the imbalance values are 48.71%, 12.31%, and 15.27% for the RR, LB, and HP schemes, respectively.

Figure 3.6 shows the total volume of communication that will be incurred during the transmission of accumulators from index servers to the central broker with the assumption that every document in the collection is requested once during query processing. The performance of HP in minimizing the communication volume is especially notable at high numbers of index servers. At 64 index servers, the HP scheme incurs 15.26% and 13.39% less total volume of communication than the RR and LB schemes, respectively.

3.6.3 Results on Document-Based Partitioning

Figure 3.7 shows the load imbalance values obtained by the three partitioning schemes in posting storage of index servers for document-based inverted index partitioning. According to this figure, in document-based inverted index partitioning, the imbalance values obtained by all schemes are relatively lower compared to term-based inverted index partitioning (Figure 3.5). This is basically due to the fact that documents follow a uniform size distribution, whereas terms

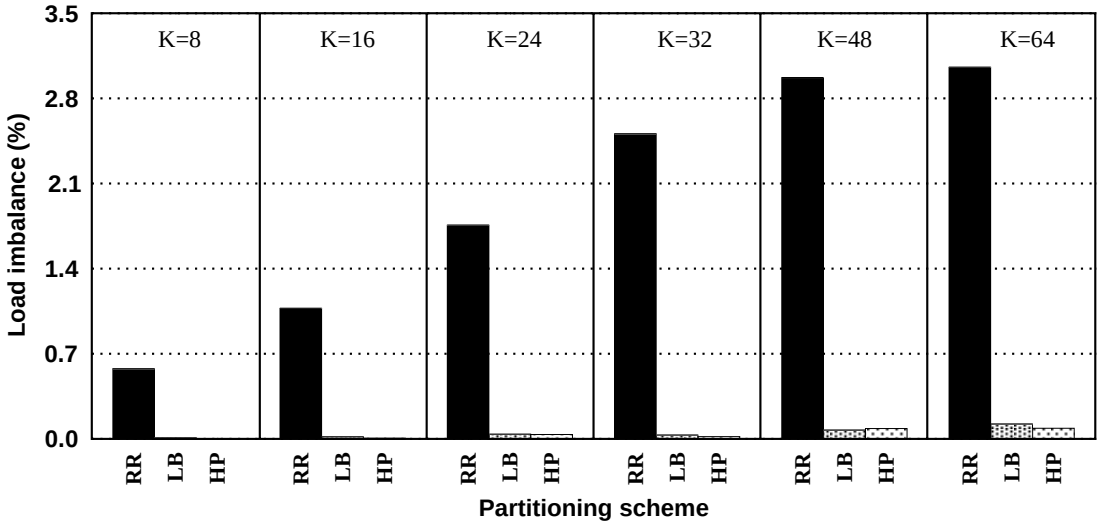


Figure 3.7: The load imbalance in posting storage with increasing number K of index servers in document-based inverted index partitioning.

follow a Zipf-like distribution, which causes a great variation in inverted index sizes. At 64 index servers, the imbalance values observed are 3.01%, 0.12%, and 0.09% for the RR, LB, and HP schemes, respectively.

Figure 3.8 shows the total number of disk accesses incurred in the three different partitioning schemes with the assumption that each term in the vocabulary is submitted as a query. At all numbers of index servers, the HP scheme outperforms the RR and LB schemes in reducing the number of disk accesses. In general, the increasing number of index servers seem to favor the HP scheme. In particular, the improvement of HP over LB rises from 24.8% at 8 index servers to 28.36% at 64 index servers. This behavior is due to the fact that, as the number of index servers increases, the number of pins per part decreases, and hence the hypergraph partitioning heuristics have a better solution space in optimizing the objective function.

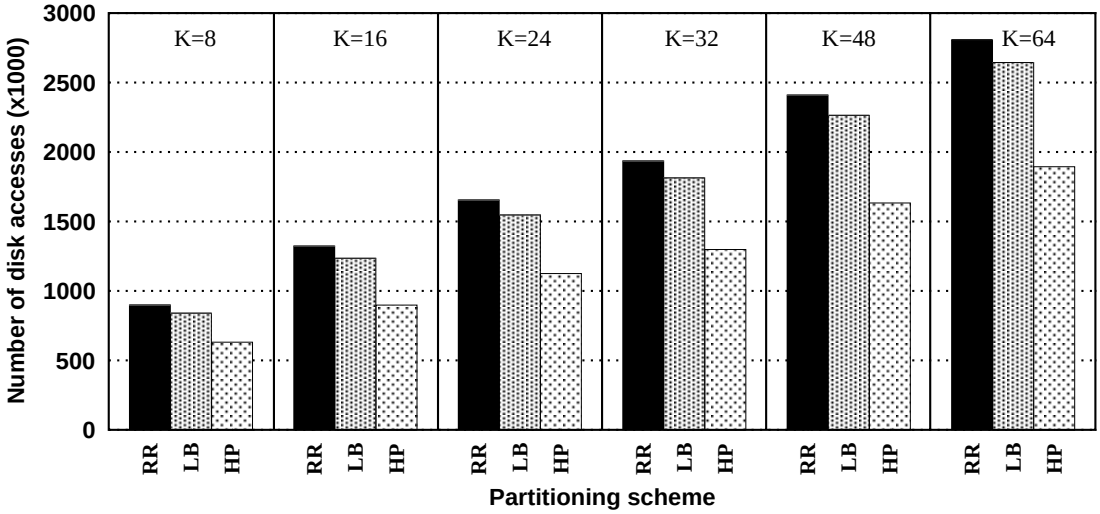


Figure 3.8: The total number of disk accesses incurred in query processing with increasing number K of index servers in document-based inverted index partitioning.

3.7 Conclusions

Although the proposed inverted index partitioning models have no benefit in minimizing the query processing times of individual queries, they are beneficial in reducing the use of system resources (i.e., the network in case of the term-based partitioning model and the disks in case of the document-based partitioning model). These schemes may turn out to be useful in improving overall system efficiency, especially in systems where the resources are shared by other software modules such as a parallel Web crawler, running on the same parallel system with the query processor. In the future, we plan to conduct practical experiments to observe the effect of the proposed partitioning models on a real-life parallel text retrieval system.

Chapter 4

Query Processing Algorithms

Similarity calculations and document ranking form the computationally expensive parts of query processing in ranking-based text retrieval. In this chapter of the thesis, eleven alternative implementation techniques are presented for these calculations [18]. The implementations are classified under four different categories, and their asymptotic time and space complexities are investigated. To our knowledge, six of these techniques are not discussed in the literature before. Furthermore, analytical experiments are carried out on a 30 GB document collection to evaluate the practical performance of different implementations in terms of query processing time and space consumption. Advantages and disadvantages of each technique are illustrated under different querying scenarios, and several experiments that investigate the scalability of the implementations are presented.

The chapter is organized as follows. In Section 4.1, we provide some background information on query processing in ranking-based text retrieval systems. In Section 4.2, we give pointers to the related work on efficient query processing. In Section 4.3, we describe the implementation techniques and present an analysis of their asymptotic time and space complexities. In Section 4.4, we evaluate the practical performance of each technique on a large (30 GB) document collection. In Section 4.5, we present a discussion on advantages and disadvantages of the techniques and conclude.

4.1 Introduction

In the last decade, a shift has been observed from the boolean model of query processing to the more effective ranking-based model. In text retrieval systems employing the ranking-based model, similarity calculations are performed between a user query and the documents in a collection. As a result of these calculations, the user is presented a set of relevant documents, ranked in decreasing order of similarity to the query. The similarity calculations and document ranking, which form the major source of overhead in query processing, can be implemented in many ways, using different data structures and algorithms. The main focus of this work is on advantages and disadvantages of these data structures and algorithms.

Although other strategies may also be employed [45] a document collection is usually represented by an inverted index [113, 133]. An inverted index is composed of two parts: a set of inverted lists and an index into these lists. The set of inverted lists $\mathcal{L} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_T\}$ of size T, where T is the number of distinct terms in the collection, contains a list $\mathcal{I}_i$ for each term t_i in the collection. The index part contains a pointer to each term's inverted list. Each inverted list $\mathcal{I}_i$ keeps entries, called postings, about the documents in which term t_i appears. A posting $p \in \mathcal{I}_i$ includes a document id field $p.d = j$ and a weight field $p.w = w(t_i, d_j)$ for a document d_j containing term t_i, where $w(t_i, d_j)$ is a weight [63] which indicates the degree of relevance between t_i and d_j.

In construction of the inverted index, usually, the *tf-idf* (term frequency-inverse document frequency) weighting scheme [104] is used to compute $w(t_i, d_j)$. In this work, we use the *tf-idf* variant (also shown in Eq. 3.2 of Chapter 3)

$$w(t_i, d_j) = \frac{f(t_i, d_j)}{\sqrt{|d_j|}} \times \ln \frac{D}{f(t_i)}, \tag{4.1}$$

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 processing a query, only the inverted lists associated with the query terms are used. Specifically, if we have a query $\mathcal{Q}=\{t_{q_1}, t_{q_2}, \ldots, t_{q_Q}\}$ of Q distinct query terms, we work on a partial inverted index $\mathcal{L}_\mathcal{Q} \subset \mathcal{L}$ of Q inverted lists, in which each list $\mathcal{I}_{q_i} \in \mathcal{L}_\mathcal{Q}$ is associated with query term $t_{q_i} \in \mathcal{Q}$. The similarity $\text{sim}(\mathcal{Q}, d_j)$ of query $\mathcal{Q}$ to a document d_j can be calculated using the cosine function [104]. Since, in Eq. 4.1, we already approximated cosine normalization by the $\sqrt{|d_j|}$ factor [86], the cosine similarity metric can be simplified as

$$\text{sim}(\mathcal{Q}, d_j) = \sum_{t_{q_i} \in \mathcal{Q}} w(t_{q_i}, d_j), \tag{4.2}$$

assuming that all query terms have equal importance. That is, to calculate the similarity between query $\mathcal{Q}$ and document d_j, we need to accumulate the weights $w(t_{q_i}, d_j)$ for each query term $t_{q_i} \in \mathcal{Q}$ in a memory location dedicated to document d_j. These memory locations are called accumulators. An accumulator a typically keeps an integer document id field $a.d$ and a floating point score field $a.s$, which contains the accumulated similarity value for document $a.d$. After all accumulator updates are completed, sorting them in decreasing order of finalized $a.s$ values gives a ranking of documents.

Both time and space are critical in ranking-based text retrieval. Especially, in cases where the inverted index is completely stored in volatile memory (a common practice for Web search engines) and disk accesses are avoided, similarity calculations and document ranking directly determine the query processing times. Considering the existence of search engines which indexed more than 4 billion pages, it is easily seen that space consumption is also a critical issue. In this work, we present eleven alternative implementations under four different categories for query processing in ranking-based text retrieval, taking time and space needs into consideration. To our knowledge, six of these implementations are not discussed in any publication before.

4.2 Related Work

In the literature, ranking-based text retrieval is well-studied in terms of both effectiveness [30, 42, 123] and efficiency [30, 91]. Some of the basic query processing techniques are described in classical information retrieval books [9, 57, 104, 125]. Many optimizations are proposed for decreasing query processing times and efficiently using the memory [16, 65, 92, 96, 101, 110, 118]. Wong93, These optimizations are based on limiting the number of processed query terms and postings (short-circuit evaluation) or limiting the memory allocated to accumulators. They mainly differ in their choice for the processing order of postings and when to stop processing them.

Buckley and Lewit [16] 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 [64] 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. [96] proposed two heuristics which place a hard limit on the memory allocated to accumulators. Turtle and Flood [118] presented simulation results for the performance analysis of two optimizations techniques, which employ term-ordered and document-ordered inverted list traversal. Wong and Lee [126] proposed two optimization heuristics which traverse postings in decreasing magnitude of weights. For a similar strategy, Persin [101] used thresholds for allocation and update of accumulators.

These optimizations can be classified as safe or approximate [118]. Safe optimizations guarantee that best-matching documents are ranked correctly. Approximate optimizations may trade effectiveness for efficiency producing a partial ranking, which does not necessarily contain the best-matching documents, or may present them in an incorrect order. Our focus in this work is not on partial query evaluation or approximate optimizations. We investigate the complexities of implementations and data structures in total document ranking as well as their performance in practice.

Throughout the chapter, we take an information retrieval point of view in analyzing various implementation techniques. However, there exists a significant amount of related work in the database literature. The interested reader may refer to prior works by Lehman and Carey [87], Goldman et al. [59], Bohannon et al. [14], Hristidis et al. [71], Elmasri and Navathe [52], and Ilyas et al [72].

4.3 Query Processing Implementations

The analysis presented in this work are based on processing of a single query $\mathcal{Q} = \{t_{q_1}, t_{q_2}, \ldots, t_{q_Q}\}$ with Q distinct terms over a document collection with D documents. u denotes the total number of postings in the processed Q inverted lists $\mathcal{I}_{q_i} \in \mathcal{L}_{\mathcal{Q}}$, all of which are stored in the volatile memory. The number of distinct document ids in these postings is denoted by e. The text retrieval system returns the most relevant (highly ranked) s documents to the user as the result of the query. Table 4.1 displays the notation used in the chapter.

Although other orderings are possible, the postings in our inverted lists are ordered by increasing document id since this ordering is strictly required by some of the algorithms we implemented. Moreover, this ordering is necessary in case inverted index is compressed [11, 132]. In postings, we store normalized *tf* scores $(f(t_i, d_j)/\sqrt{|d_j|})$, thus eliminating the need to lookup the document lengths $(|d_j|)$ and allocate a large array to store them. This way, the main space demand is for the accumulators and the postings in the inverted lists. The *idf* component $(\ln(D/f(t_i)))$ is not precomputed in postings but computed during query processing, allowing easy updates over the inverted index.

In a query processing implementation, depending on the operations on accumulators, we distinguish five phases which affect the processing time of a query: *creation*, *update*, *extraction*, *selection*, and *sorting*. Descriptions of these phases are given below.

- *Creation*: Each document d_i is associated with an accumulator a_i, initialized as $a_i.d = i$ and $a_i.s = 0$. Depending on the implementation, either

Table 4.1: The notation used in the work

Symbol	Description
T	The number of distinct terms in the collection
D	The number of documents in the collection
t_i	A term in the collection
d_i	A document in the collection
$\|d_i\|$	The total number of terms in d_i
$\mathcal{L}$	The set of inverted lists
$\mathcal{I}_i$	The inverted list associated with t_i
$p.d, p.w$	Document id and weight fields of a posting p
$f(t_i, d_j)$	The number of times t_i appears in d_j
$f(t_i)$	The number of documents containing t_i
$\mathcal{Q}$	A user query
Q	The number of distinct terms in $\mathcal{Q}$
$\mathcal{L}_\mathcal{Q}$	The partial set of inverted lists processed in answering $\mathcal{Q}$
$a.d, a.s$	Document id and score fields of an accumulator a
u	The total number of postings in all $I_i \in \mathcal{L}_\mathcal{Q}$
e	The number of postings with distinct document ids in all $I_i \in \mathcal{L}_\mathcal{Q}$
s	The number of documents to be returned to the user
B	The number of buckets in the hashing implementation

previously allocated locations are used as accumulators or space is dynamically allocated for accumulators as needed. In this phase, some auxiliary data structures may also be allocated and initialized.

- *Update*: Once an accumulator a_i is created for a document d_i, the weight $p.w$ of each posting p where $p.d = i$ is simply added to the score of accumulator a_i, i.e., $a_i.s = a_i.s + p.w$. It is necessary and sufficient to perform u updates since each posting incurs a single update.

- *Extraction*: The accumulators with nonzero scores (i.e., $a_i.s > 0$) whose updates are completed can be extracted. Such accumulators are located and passed to the selection phase as input. Since an accumulator is extracted exactly once, there are always e extraction operations.

- *Selection*: This phase compares each extracted accumulator score with the previously extracted ones and selects the accumulators having the top s scores. This way, the set $\mathcal{S}_{\text{top}}$ of best-matching documents is constructed.

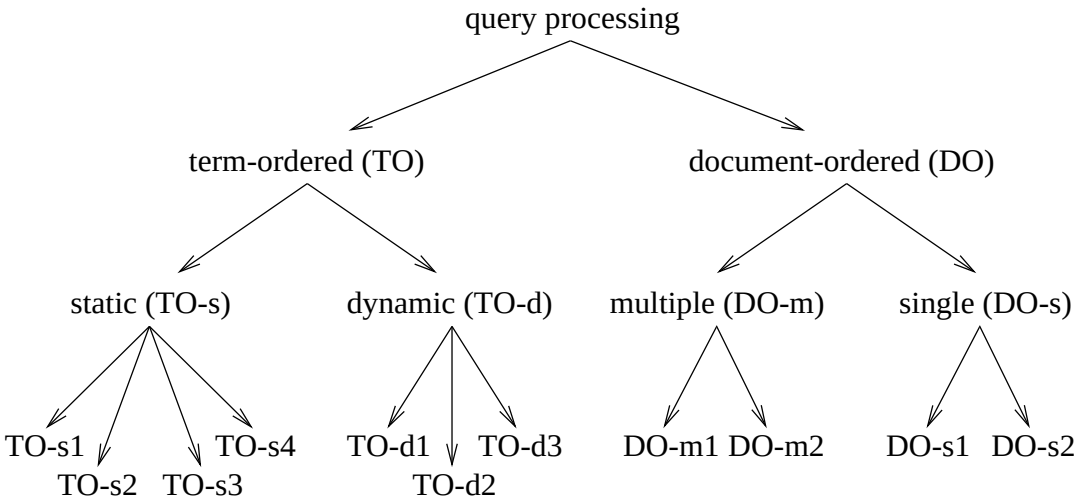


Figure 4.1: A classification for query processing implementations.

- *Sorting*: The accumulators in $\mathcal{S}_{\text{top}}$ are sorted in decreasing order of their scores, and their document ids are returned to the user in this sorted order.

The asymptotic run-time costs for the creation, update, extraction, selection and sorting phases are represented by Time_{C}, Time_{U}, Time_{E}, Time_{S}, and Time_{R}, respectively. We represent the total run-time cost of an implementation by Time_{T} and the storage cost by S. In all analysis, we strictly have $e \leq D$, $e \leq u$, $s \leq e$, and $u \leq QD$. Moreover, we assume $s \ll D$, $Q \ll T$, and $u = O(D)$.

Depending on the processing order of postings, we make a broad classification of query processing implementations as term-ordered (TO) and document-ordered (DO). We further classify TO processing as static (TO-s) and dynamic (TO-d), according to the strategy used in allocation of accumulators. Similarly, we classify DO processing as multiple (DO-m) and single (DO-s), according to the number of accumulators allocated. For TO-s, TO-d, DO-m, and DO-s approaches, we present 4, 3, 2, and 2 implementations, respectively (Figure 4.1). To the best of our knowledge, the implementations TO-s4, TO-d1, TO-d2, TO-d3, DO-m1, and DO-m2 are not discussed in a previous publication.

TO-s($\mathcal{Q}$, $\mathcal{A}$)
for each accumulator $a_i \in \mathcal{A}$ **do**
INITIALIZE a_i as $a_i.d = i$ and $a_i.s = 0$
for each query term $t_{q_j} \in \mathcal{Q}$ **do**
for each posting $p \in \mathcal{I}_{q_j}$ **do**
UPDATE $a_{p.d}.s$ as $a_{p.d}.s + p.w$
$\mathcal{S}_{\text{top}} = \emptyset$
INSERT the accumulators having the top s scores into $\mathcal{S}_{\text{top}}$
SORT the accumulators in $\mathcal{S}_{\text{top}}$ in decreasing order of their scores
RETURN $\mathcal{S}_{\text{top}}$

Figure 4.2: The algorithm for TO-s implementations.

4.3.1 Implementations for Term-Ordered (TO) Processing

In TO processing, inverted lists are sequentially processed. The postings of a term are completely exhausted before the postings of the next term are processed. Extraction and selection phases are performed in an interleaved manner. In TO-s, D accumulators are allocated statically. In TO-d, at most e accumulators are allocated on demand, thus saving space if D is very high.

4.3.1.1 Implementations with Static Accumulator Allocation (TO-s)

In TO-s implementations, an array $\mathcal{A}$ of D accumulators is statically allocated. Each array element $a_i = \mathcal{A}[i]$ is used as an accumulator. Before processing a query, accumulator fields are initialized as $a_i.d = i$ and $a_i.s = 0$. Similarity updates for document d_i are performed over $a_i.s$. Creation and update phases are the same for all TO-s implementations. These implementations mainly differ in extraction, selection, and sorting phases. The algorithm for TO-s implementations is given in Figure 4.2. In this section, we describe four different TO-s implementations.

TO-s1: accumulator array, accumulators with nonzero scores sorted

The most naive implementation is to sort all accumulators in $\mathcal{A}$ in decreasing order of their scores and return the document ids in the first s accumulators. If $e \ll D$, most accumulators are never updated and their score fields remain zero. In this case, it is better to first pick the nonzero accumulators and then sort those [125]. Costs for this approach are as follows:

- *Creation*: Array $\mathcal{A}$ of D accumulators is allocated, and its accumulators are initialized. This type of allocation is a one-time $O(D)$-cost operation independent of the number of incoming queries. However, reinitialization of the accumulators between consecutive queries require $O(e)$ operations. Hence, $\text{Time}_\text{C} = O(e)$.

- *Update*: Each term q_j is considered in turn, and for each posting $p \in \mathcal{I}_{q_j}$ with $p.d = i$, an update is performed over the corresponding accumulator field $a_i.s$, i.e., $a_i.s = a_i.s + p.w$. This phase involves reading and writing a total of u values between two locations. Hence, $\text{Time}_\text{U} = O(u)$.

- *Extraction*: Since it is not known which accumulators have nonzero score fields, the whole $\mathcal{A}$ array must be traversed to locate them. During this traversal, nonzero accumulators are picked and stored at the first e elements of array $\mathcal{A}$. Traversing the whole array and checking the score fields require $O(D)$ comparisons. Hence, $\text{Time}_\text{E} = O(D)$.

- *Selection*: This phase involves no work since the top s scores to be selected already reside within the first e array elements. $\text{Time}_\text{S} = O(1)$.

- *Sorting*: Sorting the first e array elements in decreasing order of the scores gives a ranking. The document ids in the first s array elements are returned as the set $\mathcal{S}_\text{top}$ of best-matching documents. Sorting has a cost of $\text{Time}_\text{R} = O(e \lg e)$.

The running time of this implementation is $\text{Time}_\text{T} = O(e + u + D + 1 + e \lg e) = O(D + e \lg e)$. The storage overhead is $S = O(D)$.

TO-s2: accumulator array, max-priority queue for nonzero accumulators

An improvement over TO-s1 is to use a max-priority queue implemented as a binary heap $\mathcal{H}_{\max}$ to select the top s accumulator scores [96]. The max-heap $\mathcal{H}_{\max}$ contains e accumulators, keyed by their scores. This approach avoids the cost of sorting the whole set of nonzero accumulators if $s < e$.

- *Creation*, *Update*: Similar to TO-s1. $\text{Time}_{\text{C}} = O(e)$, $\text{Time}_{\text{U}} = O(u)$. Note that array $\mathcal{A}$ can be used in order to store the accumulators in $\mathcal{H}_{\max}$. Hence, no extra storage is necessary for implementing the max-priority queue.
- *Extraction*: Similar to TO-s1. $\text{Time}_{\text{E}} = O(D)$.
- *Selection*: Extracted accumulators in the first e elements of array $\mathcal{A}$ are treated as elements of heap $\mathcal{H}_{\max}$, using their score fields as the key and document id fields as the data. Since there are e extracted accumulators, the heap can be built with $O(e)$ operations. After building, the root of $\mathcal{H}_{\max}$ keeps the accumulator with the highest score. The top s accumulators are obtained by repeatedly performing s extract-max operation on $\mathcal{H}_{\max}$. $\text{Time}_{\text{S}} = O(e + s \lg e)$.
- *Sorting*: This phase involves no work since accumulators are extracted from $\mathcal{H}_{\max}$ in sorted order during the selection phase. $\text{Time}_{\text{R}} = O(1)$.

$\text{Time}_{\text{T}} = O(e + u + D + e + s \lg e) + 1 = O(D + s \lg e)$. $S = O(D)$.

TO-s3: accumulator array, min-priority queue for top s accumulators

A variation over TO-s2 is to employ, instead of a max-priority queue, a min-priority queue implemented as a min-heap $\mathcal{H}_{\min}$ [125]. At any time, the min-heap $\mathcal{H}_{\min}$ contains at most s accumulators, keyed by their scores.

- *Creation*, *Update*: Similar to TO-s1. $\text{Time}_{\text{C}} = O(e)$, $\text{Time}_{\text{U}} = O(u)$.

- *Extraction*: The $\mathcal{A}$ array is traversed, and nonzero accumulators are passed to the selection phase. $\text{Time}_{\text{E}} = O(D)$.

- *Selection*: As long as the number of accumulators in $\mathcal{H}_{\min}$ is less than s, extracted accumulators are simply added to $\mathcal{H}_{\min}$. Once it contains s accumulators, $\mathcal{H}_{\min}$ is built. After this point, the root of $\mathcal{H}_{\min}$ keeps a_{smin}, the accumulator with the minimum score observed so far. The score $a.s$ of each extracted accumulator a is compared with $a_{\text{smin}}.s$. If the incoming score $a.s$ is less than the current minimum $a_{\text{smin}}.s$, the accumulator a is simply ignored. Otherwise, accumulator a_{smin} is removed from $\mathcal{H}_{\min}$, and the extracted accumulator a is inserted into $\mathcal{H}_{\min}$. Building the min-heap from the first s extracted accumulators has a cost of $O(s)$. In the worst case, all remaining accumulators must be inserted into $\mathcal{H}_{\min}$. This has a cost of $O((e-s)\lg s)$. Hence, $\text{Time}_{\text{S}} = O(s + (e-s)\lg s)$.

- *Sorting*: Accumulators in $\mathcal{H}_{\min}$ are sorted in decreasing order of scores. $\text{Time}_{\text{R}} = O(s \lg s)$.

$$\text{Time}_{\text{T}} = O(e + u + D + (s + (e-s)\lg s) + s \lg s) = O(D + e \lg s).\ S = O(D).$$

TO-s4: accumulator array, s-th largest score selection

This method relies on the observation that the accumulator with the smallest score to be entered into the set $\mathcal{S}_{\text{top}}$ of top s accumulators can be located in linear time.

- *Creation*, *Update*: Similar to TO-s1. $\text{Time}_{\text{C}} = O(e)$, $\text{Time}_{\text{U}} = O(u)$.

- *Extraction*: This phase involves no work. $\text{Time}_{\text{E}} = O(1)$.

- *Selection*: The accumulator with the s-th largest score can be selected in worst-case linear time by the median-of-medians selection algorithm [44] over the accumulators in $\mathcal{A}$. Instead of this algorithm, the randomized selection algorithm [44], which has expected linear-time complexity, could be used for run-time efficiency in practice. This algorithm returns $a_{s\text{-th}}$, the

accumulator having the s-th largest score and places the remaining $s-1$ accumulators that should appear in $\mathcal{S}_{\text{top}}$ in the array elements following $a_{s\text{-th}}$. Hence, $\mathcal{S}_{\text{top}}$ is formed with $O(D)$ operations. $\text{Time}_{\text{S}} = O(D)$.

- *Sorting*: Accumulators in $\mathcal{S}_{\text{top}}$ are sorted in decreasing order of scores. $\text{Time}_{\text{R}} = O(s \lg s)$.

$\text{Time}_{\text{T}} = O(e + u + 1 + D + s \lg s) = O(D + s \lg s)$. $S = O(D)$.

4.3.1.2 Implementations with Dynamic Accumulator Allocation (TO-d)

If $e \ll D$, array $\mathcal{A}$ contains too many unused accumulators and hence wastes lots of space. In such a case or the case where array $\mathcal{A}$ is too large to fit into the volatile memory, it may be a good idea to use a dynamic data structure $\mathcal{D}$ and allow on-demand space allocation for accumulators. In this approach, accumulators are stored in nodes of $\mathcal{D}$ and are located using their document ids as keys. In this section, AVL tree [83], hashing [70], and skip list [102] alternatives are investigated for this purpose. In what follows, we discuss these three alternatives, starting with the AVL tree. Our time analysis for the hashing and skip list alternatives are expected-time analysis. The algorithm for TO-d implementations is given in Figure 4.3.

TO-d1: AVL tree of accumulators, min-priority queue for top s accumulators

In this implementation, an AVL tree $\mathcal{T}$ containing at most e nodes is used to store the accumulators. Each node of $\mathcal{T}$ keeps an accumulator, pointers to its left and right children, and a balance factor. An AVL tree implementation is preferred over a binary search tree implementation since the postings are stored in each inverted list in increasing order of document ids. In the case of a binary search tree implementation, with such a posting storage scheme, new accumulator insertions may quickly turn the tree into a linked list. Hence, we prefer the AVL

```
TO-D(Q, D)
    for each query term t_{q_i} ∈ Q do
        for each posting p ∈ I_{q_i} do
            if ∃ an accumulator a ∈ D with a.d = p.d then
                UPDATE a.s as a.s + p.w
            else
                ALLOCATE a new accumulator a
                INITIALIZE a as a.d = p.d and a.s = p.w
                D = D ∪ {a}
    S_top = ∅
    for each a ∈ D do
        SELECT(S_top, a)
    SORT the accumulators in S_top in decreasing order of their scores
    RETURN S_top

SELECT(S, a)
    if |S| < s then
        S = S ∪ {a}
    else
        LOCATE a_smin, the accumulator with the minimum score in S
        if a.s > a_smin.s then
            S = (S − {a_smin}) ∪ {a}
```

Figure 4.3: The algorithm for TO-d implementations.

tree data structure, which dynamically balances the height of the tree, making accumulator search less costly.

- *Creation*: If an accumulator needs to be updated in $\mathcal{T}$ and it is not already there, a tree node is dynamically allocated to store the accumulator. The cost of node allocation is constant, i.e., $O(1)$. Hence, $\text{Time}_\text{C} = O(e)$.

- *Update*: For each posting p, nodes of $\mathcal{T}$ are searched to locate the accumulator to be updated, where $a.d = p.d$. If the accumulator is found, its score field $a.s$ is updated as $a.s + p.w$. Otherwise, a new node is allocated and inserted into $\mathcal{T}$, initializing the accumulator in the node as $a.d = p.d$ and $a.s = p.w$. The update cost for an accumulator is proportional with the height of the AVL tree. Hence, $\text{Time}_\text{U} = O(u \lg e)$.

- *Extraction*: When all updates are completed, accumulators can be extracted

from nodes of $\mathcal{T}$ in any order. Each extracted accumulator is passed to the selection phase. Traversing the AVL tree has a cost of $\text{Time}_{\text{E}} = O(e)$.

- *Selection*: The min-priority queue mechanism of TO-s3 is used. $\text{Time}_{\text{S}} = O(s + (e-s)\lg s)$.
- *Sorting*: Similar to TO-s3. $\text{Time}_{\text{R}} = O(s \lg s)$.

$\text{Time}_{\text{T}} = O(e + u \lg e + e + (s + (e-s)\lg s) + s \lg s) = O(u \lg e)$. The storage overheads are $O(e)$ for the AVL tree and $O(s)$ for the min-priority queue. $S = O(e)$.

TO-d2: hashing of accumulators, min-priority queue for top s accumulators

Another implementation alternative which offers dynamic allocation is hashing. Since e is not known until all postings are completely processed, hashing techniques that require static allocation (such as open addressing) cannot be used. Here, we use hashing with chaining [70]. In this implementation, accumulators are placed into B buckets, where each bucket keeps a linked list of accumulators. The bucket b for an accumulator a is determined by applying a hash function on the document id field (e.g., $b = a.d \bmod B$).

- *Creation*: Selecting the appropriate number B of buckets is the most important step in this implementation. Allocating too many buckets may increase space consumption. On the contrary, if too few buckets are allocated, the number of accumulators per bucket increases. Since accumulators are sequentially searched in each bucket, this increases the query processing time. In this implementation, B pointers are needed to keep the list heads. Each list node stores an accumulator and has a pointer to the next node in the linked list. It is necessary to dynamically allocate a total of e list nodes. Hence, $\text{Time}_{\text{C}} = O(B + e)$.
- *Update*: For a posting p, the bucket to be searched is determined by hashing $p.d$ to a bucket. The accumulators in a bucket are searched by following the

links between list nodes. If an accumulator with $a.d = p.d$ is found, its score is updated. If the end of the list is reached or an accumulator with a greater document id is found, the search ends. In this case, a new node which contains an accumulator is allocated, initialized using p, and then inserted into the list. List nodes are maintained in increasing order of document ids. Each bucket stores e/B list nodes on the average. Hence, these many comparisons are necessary to locate an accumulator. $\text{Time}_{\text{U}} = O(ue/B)$.

- *Extraction*: Accumulators are extracted from the buckets and passed to the selection phase. Since exactly e nodes must be extracted, $\text{Time}_{\text{E}} = O(e)$.

- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_{\text{S}} = O(s + (e-s)\lg s)$, $\text{Time}_{\text{R}} = O(s \lg s)$.

$\text{Time}_{\text{T}} = O((B+e)+ue/B+e+(s+(e-s)\lg s)+s\lg s) = O(ue/B+e\lg s)$. The storage overheads are $O(B+e)$ for the hash table and $O(s)$ for the min-priority queue. $S = O(B+e)$.

TO-d3: skip list of accumulators, min-priority queue for top s accumulators

Yet another alternative is to use a skip list $\mathcal{S}$ to store and search the accumulators. Skip lists balance themselves probabilistically rather than explicitly (e.g., rotations in AVL trees). Although they have bad worst-case time complexities, they have good expected-time complexities for insert and find operations and perform well in practice.

- *Creation*: A list node is dynamically allocated in $\mathcal{S}$ to store an accumulator and a set of forward pointers to the following list nodes. The number of forward pointers in each node is determined randomly, but it is limited from above. Since e list nodes must be allocated, $\text{Time}_{\text{C}} = O(e)$.

- *Update*: For each posting p, the nodes in $\mathcal{S}$ are searched to locate the accumulator to be updated, where $a.d = p.d$. For this purpose, forward pointers are used and the skip list is traversed in a manner similar to binary

search. If the accumulator is located in $\mathcal{S}$, its score field is updated as $a.s = a.s + p.w$. Otherwise, a new node is allocated and inserted into $\mathcal{S}$ after initializing its accumulator as $a.d = p.d$ and $a.s = p.w$. The expected update cost for an accumulator is $O(\lg e)$. Hence, $\text{Time}_{\text{U}} = O(u \lg e)$.

- *Extraction*: Nodes of $\mathcal{S}$ are visited sequentially, and accumulators are passed to the selection phase. $\text{Time}_{\text{E}} = O(e)$.
- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_{\text{S}} = O(s + (e-s) \lg s)$, $\text{Time}_{\text{R}} = O(s \lg s)$.

$\text{Time}_{\text{T}} = O(e + u \lg e + e + (s + (e-s) \lg s) + s \lg s) = O(u \lg e)$. The storage overheads are $O(e)$ for the skip list and $O(s)$ for the min-priority queue. $S = O(e)$.

4.3.2 Implementations for Document-Ordered (DO) Processing

Two important features in the inverted index structure let us devise another query processing strategy. First, the postings of a term are stored in increasing order of document ids. That is, while traversing an inverted list, once a document id is seen in a posting, there cannot be a smaller document id in one of the succeeding postings in that list. Second, the number of query terms is limited. We have Q terms to be processed. These observations allow us to process the inverted lists in parallel instead of processing them consecutively. This way, it is possible to compute a complete score for a document before all postings in the lists are completely processed. In DO processing, update, extraction, and selection phases are performed in an interleaved manner. The implementations differ in their choice for the number of accumulators allocated, the data structures employed to store the accumulators, and the processing order of the list heads.

4.3.2.1 Implementations with Multiple Accumulator Allocation (DO-m)

Implementations in the DO-m category use a structure $\mathcal{M}$, which contains at most Q accumulators at any time. Also, an array h of Q elements is used to locate the first unprocessed posting in each inverted list, i.e., each element $h[i]$ points at the posting $\mathcal{I}_{q_i}^{h[i]} \in \mathcal{I}_{q_i}$ that will be processed next in list $\mathcal{I}_{q_i}$. Each accumulator $a \in \mathcal{M}$ is associated with a single inverted list. Accumulators contain a list id field, which is initialized as $a.\ell = i$ if accumulator a is associated with inverted list $\mathcal{I}_{q_i}$. Although any posting with a document id of $a.d$ from any inverted list may update the score field $a.s$, only the postings from list $\mathcal{I}_{q_{a.\ell}}$ may initialize $a.d$. The document id $a.d$ of each accumulator a is equal to a document id in one of the postings in $\mathcal{I}_{q_{a.\ell}}$. No two accumulators in $\mathcal{M}$ can have the same document id and list id. The structure $\mathcal{M}$ can be implemented by a sorted array or a dynamic data structure. These alternatives are described below. The algorithm for DO-m implementations is given in Figure 4.4.

DO-m1: sorted array of accumulators, array of posting pointers, min-priority queue for top s accumulators

In this approach, Q accumulators are kept in an array sorted in decreasing order of document ids.

- *Creation*: An accumulator array $\mathcal{A}$ and an array h for marking current list heads, each of size Q, are allocated. The cost of allocating both arrays is $O(Q)$. After the allocation, each $h[i]$ is initialized to point at the first posting $\mathcal{I}_{q_i}^1 \in \mathcal{I}_{q_i}$, i.e., $h[i] = 1$. In processing a query, there are e initializations over the accumulators in $\mathcal{A}$. Hence, $\text{Time}_C = O(e + Q)$.

- *Update*, *Extraction*: The following procedure is repeated until all postings are processed. If there are less than Q occupied accumulators in $\mathcal{A}$, updates are performed over the accumulators using the postings at the current list heads (pointed by h) which are not currently associated with an accumulator in $\mathcal{A}$. In processing of a posting $p = \mathcal{I}_{q_i}^{h[i]}$, array $\mathcal{A}$ is searched for an

DO-M($\mathcal{Q}$, $\mathcal{M}$)
  $\ell_1 = 0$, $\ell_2 = 1$, $\mathcal{M} = \emptyset$, and $\mathcal{S}_{\text{top}} = \emptyset$
  for each query term $t_{q_i} \in \mathcal{Q}$ **do**
    $h[i] = 1$, i.e., the current head of $\mathcal{I}_{q_i}$ is its first posting $\mathcal{I}_{q_i}^1$
  while $|\mathcal{L}_\mathcal{Q}| > 0$ **do**
    while $|\mathcal{L}_\mathcal{Q}| > 0$ **and** $|\mathcal{M}| < |\mathcal{L}_\mathcal{Q}|$ **do**
      if $\ell_1 = 0$ **then**
        $\ell = \ell_2$
      else
        $\ell = \ell_1$
      $p = \mathcal{I}_{q_\ell}^{h[\ell]}$
      if $\exists$ an accumulator $a \in \mathcal{M}$ with $a.d = p.d$ **then**
        UPDATE $a.s$ as $a.s + p.w$
        $h[\ell] = h[\ell] + 1$
        if $h[\ell] > |\mathcal{I}_{q_\ell}|$ **then**
          $\mathcal{L}_\mathcal{Q} = \mathcal{L}_\mathcal{Q} - \{\mathcal{I}_{q_\ell}\}$
          if $\ell_1 = 0$ **then**
            $\ell_2 = \ell_2 + 1$
      else
        ALLOCATE an accumulator a
        INITIALIZE a as $a.d = p.d$, $a.s = p.w$, and $a.\ell = \ell$
        $\mathcal{M} = \mathcal{M} \cup \{a\}$
        $h[\ell] = h[\ell] + 1$
        if $\ell_1 = 0$ **then**
          $\ell_2 = \ell_2 + 1$
    while $|\mathcal{L}_\mathcal{Q}| > 0$ **and** $|\mathcal{M}| = |\mathcal{L}_\mathcal{Q}|$ **do**
      LOCATE a_{dmin}, the accumulator with the min. document id in $\mathcal{M}$
      $\mathcal{M} = \mathcal{M} - \{a_{\text{dmin}}\}$
      $\ell_1 = a_{\text{dmin}}.\ell$
      SELECT($\mathcal{S}_{\text{top}}$, a_{dmin})
      if $h[\ell] > |\mathcal{I}_{q_\ell}|$ **then**
        $\mathcal{L}_\mathcal{Q} = \mathcal{L}_\mathcal{Q} - \{\mathcal{I}_{q_\ell}\}$
  SORT the accumulators in $\mathcal{S}_{\text{top}}$ in decreasing order of their scores
  RETURN $\mathcal{S}_{\text{top}}$

Figure 4.4: The algorithm for DO-m implementations.

accumulator with $a.d = p.d$. If it is found, a is updated using p. Otherwise, a new accumulator is created in $\mathcal{A}$ and is initialized as $a.d = p.d$, $a.s = p.w$, and $a.\ell = i$. If all Q accumulators in $\mathcal{A}$ are occupied, i.e., associated with a list, the accumulator a_{dmin} with the minimum document id is located, extracted, and passed to the selection phase. Then, $h[a_{\text{dmin}}.\ell]$ is incremented by 1, and hence it points to the posting $p = \mathcal{I}_{q_{a_{\text{dmin}}.\ell}}^{h[a_{\text{dmin}}.\ell]}$ to be processed next. Since the $\mathcal{A}$ array is maintained in decreasing order of document ids, an accumulator can be located in $O(\lg Q)$ time using binary search. Although update of an accumulator is an $O(1)$-time operation once it is located, insertion of a new accumulator after a failed search requires shifting $O(Q)$ accumulators in the array. Considering the fact that there are $u - e$ accumulator updates and e insertions, $\text{Time}_U = O(u \lg Q + eQ)$. Extraction is simple since the accumulator with the smallest document id is always the last element of the array. $\text{Time}_{\text{E}} = O(e)$.

- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_{\text{S}} = O(s + (e - s) \lg s)$, $\text{Time}_{\text{R}} = O(s \lg s)$.

$\text{Time}_{\text{T}} = O((e + Q) + (u \lg Q + eQ) + e + (s + (e - s) \lg s) + s \lg s) = O(u \lg Q + eQ + e \lg s)$. The storage overheads are $O(Q)$ for the sorted array, $O(Q)$ for the array of posting pointers, and $O(s)$ for the min-priority queue. $S = O(Q + s)$.

DO-m2: AVL tree of accumulators, array of posting pointers, min-priority queue for top s accumulators

Instead of a sorted array, an AVL tree $\mathcal{T}$ can be used as a dynamic structure to store the accumulators.

- *Creation*: Array h is allocated and initialized similar to DO-m1. Nodes of AVL tree $\mathcal{T}$ are dynamically allocated. For each accumulator with a distinct document id, a tree node must be allocated although $\mathcal{T}$ contains no more than Q nodes at any time. Hence, $\text{Time}_{\text{C}} = O(e + Q)$.

- *Update*, *Extraction*: Update and extraction phases are similar to DO-m1. However, in processing a posting, both update of an existing accumulator

and insertion of a new one require $O(\lg Q)$ operations in the worst case. Hence, $\text{Time}_{\text{U}} = O(u \lg Q)$. The accumulator with the smallest document id is contained within the left-most leaf node in $\mathcal{T}$. This leaf node can be reached by following the left links iteratively starting from the root of $\mathcal{T}$ until a node with no children is reached. With this approach, extraction is an $O(\lg Q)$-time operation. However, it is possible to improve this by an implementation trick. If each node keeps a link to its parent node, and the node with the smallest document id in $\mathcal{T}$ is remembered by a pointer, it turns out that extraction is an $O(1)$-time operation. Hence, $\text{Time}_{\text{E}} = O(e)$.

- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_{\text{S}} = O(s + (e-s) \lg s)$, $\text{Time}_{\text{R}} = O(s \lg s)$.

$\text{Time}_{\text{T}} = O((e+Q) + u \lg Q + e + (s + (e-s) \lg s) + s \lg s) = O(u \lg Q + e \lg s)$. The storage overheads are $O(Q)$ for the AVL tree, $O(Q)$ for the array of posting pointers, and $O(s)$ for the min-priority queue. $S = O(Q + s)$.

4.3.2.2 Implementations with Single Accumulator Allocation (DO-s)

Implementations in the DO-s category require the use of only a single accumulator a_{dmin} at any time. All updates are performed on this single accumulator. Here, we describe two different implementations that belong to this category. The algorithm for DO-s implementations is given in Figure 4.5.

DO-s1: single accumulator, array of posting pointers, min-priority queue for top s accumulators

In this very simple approach, two passes are made over the list heads. In the first pass, the smallest document id among the currently unprocessed postings is determined. In the second pass, the postings with this smallest document id are picked and used to update a_{dmin}.

- *Creation*: The single accumulator a_{dmin}, which stores the information about the currently minimum document id, is allocated. The h array is allocated

```
DO-S(Q, a_dmin)
    S_top = ∅
    for each query term t_{q_i} ∈ Q do
        h[i] = 1, i.e., the current head of I_{q_i} is its first posting I^1_{q_i}
    while |L_Q| > 0 do
        LOCATE p_dmin, the posting with the minimum document id
                among all I^{h[i]}_{q_i} ∈ I_{q_i}, where I_{q_i} ∈ L_Q
        INITIALIZE a_dmin as a_dmin.d = p_dmin.d and a_dmin.s = 0
        for each posting p = I^{h[i]}_{q_i} where I_{q_i} ∈ L_Q do
            if p.d = p_dmin.d then
                UPDATE a_dmin.s as a_dmin.s + p.w
                h[i] = h[i] + 1
                if h[i] > |I_{q_i}| then
                    L_Q = L_Q − {I_{q_i}}
        SELECT(S_top, a_dmin)
    SORT the accumulators in S_top in decreasing order of their scores
    RETURN S_top
```

Figure 4.5: The algorithm for DO-s implementations.

and initialized as in DO-m1. The cost of reinitializing a_{dmin} is $O(e)$. Hence, $\text{Time}_{\text{C}} = O(e + Q)$.

- *Update*, *Extraction*: A pass is made over the postings pointed by the h array. Within these postings, a posting p_{dmin} with the minimum document id $p_{\text{dmin}}.d$ is found. Accumulator a_{dmin} is initialized as $a_{\text{dmin}}.d = p_{\text{dmin}}.d$ and $a_{\text{dmin}}.s = 0$. With a second pass over these postings, the postings that have this minimum document id are found. The score field $a_{\text{dmin}}.s$ of accumulator a_{dmin} is updated using the weights in each such posting. $h[i]$ for each inverted list $\mathcal{I}_{q_i}$ that contains such a posting is incremented to point at the next posting in the list. Once all updates over a_{dmin} is completed, a_{dmin} is passed to the selection phase. This procedure is repeated until all postings are consumed. Since two passes are made over h for each distinct document id, $\text{Time}_{\text{U}} = O(eQ)$. Extracting a_{dmin} is an $O(1)$-time operation. Hence, $\text{Time}_{\text{E}} = O(e)$.

- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_{\text{S}} = O(s + (e-s)\lg s)$, $\text{Time}_{\text{R}} = O(s \lg s)$.

$\text{Time}_\text{T} = O((e + Q) + eQ + e + (s + (e - s) \lg s) + s \lg s) = O(eQ + e \lg s)$. The storage costs are $O(1)$ for the accumulator, $O(Q)$ for the array of posting pointers, and $O(s)$ for the min-priority queue. $S = O(Q + s)$.

DO-s2: single accumulator, min-priority queue for posting pointers, min-priority queue for top s accumulators

In this implementation, instead of the h array in the DO-s1 implementation, a min-priority queue is used so that there is no need for the first pass, which searches for the minimum document id. Here, we describe an improved version of the implementation described by Kaszkiel et al. [78].

- *Creation*: Similar to DO-s1. However, h is a min-priority queue implemented as a min-heap of postings pointers, keyed by the document ids in the postings they point at. $\text{Time}_\text{C} = O(e + Q)$.
- *Update*, *Extraction*: The min-priority queue h is built using the postings at the list heads. The following procedure is repeated until all postings are processed. The root of h stores posting p_{dmin}, i.e., the posting with the minimum document id among the current list heads. a_{dmin} is initialized as $a_{\text{dmin}}.d = p_{\text{dmin}}.d$ and $a_{\text{dmin}}.s = 0$. h is traversed in reverse order (starting from the Q-th element down to the first element), and the postings with $p.d = p_{\text{dmin}}.d$ are located. Each such posting p is used to update a_{dmin} as $a_{\text{dmin}}.d = p.d$ and $a_{\text{dmin}}.s = a_{\text{dmin}}.s + p.s$. Then, posting p is replaced by the next posting in the inverted list that p belongs to, and h is heapified at the node containing p. This approach avoids building the heap [78] at each pass. After the posting p_{dmin} at the root performs its update, a_{dmin} is extracted and passed to the selection phase. In this approach, the heap is heapified exactly once for each posting, and hence $\text{Time}_\text{U} = O(u \lg Q)$. Extraction has a cost of $\text{Time}_\text{E} = O(e)$.
- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_\text{S} = O(s + (e - s) \lg s)$, $\text{Time}_\text{R} = O(s \lg s)$.

$\text{Time}_\text{T} = O((e+Q) + u \lg Q + e + (s + (e-s) \lg s) + s \lg s) = O(u \lg Q + e \lg s)$. The

storage overheads are $O(1)$ for the accumulator, $O(Q)$ for the min-priority queue of posting pointers, and $O(s)$ for the min-priority queue of top s accumulators. $S = O(Q + s)$.

4.4 Experimental Results

4.4.1 Experimental Platform

In the experiments, a Pentium IV 2.54 GHz PC, which has 2 GB of main memory, 512 KB of L2 cache, and 8 KB of L1 cache, is used. As the operating system, Mandrake Linux, version 13 is installed. All algorithms are implemented in C and are compiled in gcc with -O2 optimization option. Due to the randomized nature of some of the implementations, experiments are repeated 10 times, and the average values are reported. All experiments are conducted after booting the system into the single user mode.

As the document collection, results of a large crawl performed over the '.edu' domain, i.e., the educational US Web sites, is used. The entire collection is around 30 GB and contains 1,883,037 Web pages (documents). After cleansing and stop-word elimination, there remains 3,325,075 distinct index terms. The size of the inverted index constructed using this collection is around 2.7 GB.

In query processing, four different query sets ($\mathcal{Q}_{\text{short}}$, $\mathcal{Q}_{\text{medium}}$, $\mathcal{Q}_{\text{long}}$, and $\mathcal{Q}_{\text{huge}}$) are tried. Each query set contains 100 queries, expect for $\mathcal{Q}_{\text{huge}}$, which contains a single query. The query terms are selected from the sentences within the documents of the collection. Queries in $\mathcal{Q}_{\text{short}}$, which simulate Web queries, are made up of between 1 and 5 query terms. Queries in $\mathcal{Q}_{\text{medium}}$ contain between 6 and 25 query terms. This type of queries is observed in relevance feedback. Queries in $\mathcal{Q}_{\text{large}}$ contain between 26 and 250 query terms and simulate queries observed in text classification. $\mathcal{Q}_{\text{huge}}$ is included for experimental purposes and the results, although mentioned in the text, are partially reported. Properties of the query sets are given in Table 4.2. This table also presents the minimum,

Table 4.2: The minimum, maximum, and average values of the number of query terms (Q), number of extracted accumulators (e), and number of updated accumulators (u) for different query sets

	$\mathcal{Q}_{\text{short}}$	$\mathcal{Q}_{\text{medium}}$	$\mathcal{Q}_{\text{long}}$	$\mathcal{Q}_{\text{huge}}$
$\lvert\mathcal{Q}\rvert$	100	100	100	1
Q_{min}	1	6	26	2,500
Q_{max}	5	25	250	2,500
Q_{avr}	3.0	14.6	142.1	2,500
e_{min}	4	331,524	1,218,640	1,866,703
e_{max}	1,363,584	1,637,894	1,839,661	1,866,703
e_{avr}	375,166	1,109,691	1,723,229	1,866,703
u_{min}	4	367,068	2,625,452	111,028,126
u_{max}	1,964,216	6,861,180	38,760,201	111,028,126
u_{avr}	451,931	2,310,010	16,468,300	111,028,126

Table 4.3: The minimum, maximum, and average values of the number of top documents (s) for answer sets produced after processing query set $\mathcal{Q}_{\text{short}}$

	$\mathcal{S}_{\text{small}}$	$\mathcal{S}_{\text{large}}$	$\mathcal{S}_{\text{full}}$
$\lvert\mathcal{S}\rvert$	10	1000	e
s_{min}	4	4	4
s_{max}	10	1000	1,363,584
s_{avr}	9.94	994	375,166

maximum, and average e and u values observed during the experiments.

For each query set, three answer sets ($\mathcal{S}_{\text{small}}$, $\mathcal{S}_{\text{large}}$, and $\mathcal{S}_{\text{full}}$), each with a different top document count s, are tried. $\mathcal{S}_{\text{small}}$ and $\mathcal{S}_{\text{large}}$ expect the query processing system to return the first 10 and 1000 best-matching documents, respectively. $\mathcal{S}_{\text{full}}$ expects all documents with a nonzero score to be returned to the user. Properties of these answer sets, and the minimum, maximum, and average number of top documents actually returned as answer to queries in $\mathcal{Q}_{\text{short}}$ are displayed in Table 4.3.

4.4.2 Experiments on Execution Time

Figure 4.6 presents the running times of implementations for different types of query and answer sets. Among the static-accumulator implementations in the TO-s category, for $\mathcal{S}_{\text{small}}$ and $\mathcal{S}_{\text{large}}$, the min-priority queue implementation TO-s3 performs the best if queries contain a few terms, i.e., when $\mathcal{Q}_{\text{short}}$ is used. For the same answer sets, the linear-time selection scheme TO-s4 performs slightly better than TO-s3 if $\mathcal{Q}_{\text{medium}}$ or $\mathcal{Q}_{\text{long}}$ is used. For the answer set $\mathcal{S}_{\text{full}}$, the best results are achieved by the max-priority queue implementation TO-s2. The TO-s1 implementation, which requires sorting the nonzero accumulators, is outperformed in all experiments, but the gap between TO-s1 and the others closes as the queries get longer. For $\mathcal{Q}_{\text{huge}}$ and $\mathcal{S}_{\text{full}}$ combination, TO-s1 is almost as good as TO-s2 and TO-s3.

Among the dynamic-accumulator implementations in the TO-d category, for $\mathcal{Q}_{\text{short}}$ and $\mathcal{Q}_{\text{medium}}$, the hashing implementation TO-d2 performs the best. For this implementation, we used an adaptive bucket size $B = u/Q$ due to the time-space trade-off mentioned in Section 4.3. For query sets $\mathcal{Q}_{\text{long}}$, the best results are achieved by TO-d2 and the AVL tree implementation TO-d1, which perform almost equally well. Increasing the number of terms in queries seems to favor TO-d1, which is the fastest implementation for $\mathcal{Q}_{\text{huge}}$.

In the DO-m category, although the run-time complexity for the AVL tree implementation DO-m2 is better than that of the sorted array implementation DO-m1, in practice, DO-m1 is faster than DO-m2 for $\mathcal{Q}_{\text{short}}$ and $\mathcal{Q}_{\text{medium}}$. This shows that the cost of rotations in the AVL tree implementation is higher than the cost of accumulator shifts in the sorted array implementation. However, if queries get longer, DO-m2 starts to perform better than DO-m1. Interestingly, for $\mathcal{Q}_{\text{huge}}$, DO-m2 runs 11 times faster than DO-m1 on the average.

In the DO-s category, for short queries, the two-pass DO-s1 implementation is faster than the one-pass DO-s2 implementation. As the number of query terms increase, DO-s2 starts to perform better. This can be explained by the fact that visiting the list heads in the first pass of DO-s1 brings an additional overhead,

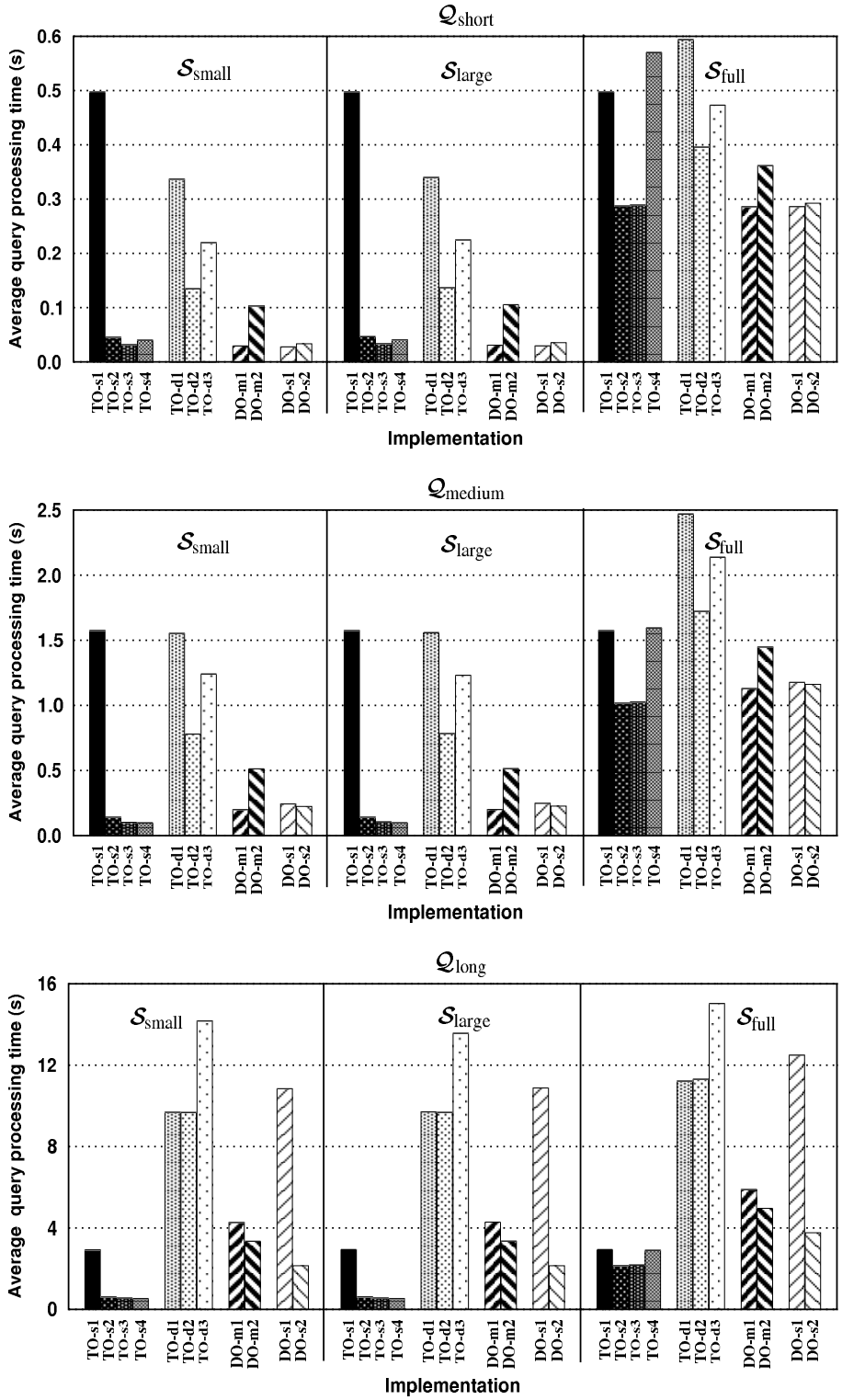


Figure 4.6: Query processing times of the implementations for different query and answer set sizes.

which dominates when queries are long. It is observed that, for $\mathcal{Q}_{\text{huge}}$, DO-s2 runs 35 times faster than DO-s1.

Among all implementations, if all documents with a nonzero score are returned, TO-s2 performs the best with TO-s3 displaying close performance. Otherwise, if answers are partially returned, performance depends on the number of query terms. For example, if queries are short DO-s1 is the best choice, whereas TO-s4 is the fastest implementation for medium and long query sizes.

It should also be noted that, for aggregate querying scenarios, the winners may change. For example, in the case the user is interested in the top 10 documents and 40% or more of the queries come from $\mathcal{Q}_{\text{short}}$ while the remaining 60% or less are of type $\mathcal{Q}_{\text{medium}}$ requiring all top documents, then TO-s3 is preferable to both DO-s1 and TO-s2 in that it provides the best average query processing time. Taking this fact into consideration, we also present normalized running times in Figure 4.7. In order to generate this figure, the execution times are first normalized with the smallest execution time. Then, the normalized time values are averaged and displayed across each query and answer set category.

According to Figure 4.7, DO-s1 and DO-m1 perform better than the rest for query set $\mathcal{Q}_{\text{short}}$. For $\mathcal{Q}_{\text{medium}}$ and $\mathcal{Q}_{\text{long}}$, TO-s3 is better than the others. For $\mathcal{S}_{\text{small}}$ and $\mathcal{S}_{\text{large}}$, TO-s3 is again the best. For $\mathcal{S}_{\text{full}}$, TO-s2 very slightly outperforms TO-s3. On the overall, the local winners of the four categories are TO-s3, TO-d2, DO-m1, and DO-s2, where TO-s3 is also the global winner.

Figure 4.8 displays the percent dissection of execution times for different query processing phases, i.e., creation, update, extraction, selection, and sorting. According to this figure, for TO-s1, the bottleneck is at the sorting phase. However, for most implementations, the sorting overhead is relatively less important, except for the case of short queries with all results retrieved. Overhead of the selection phase is more apparent for short queries. Especially, in the small answer set case, a considerable percentage of execution times for TO-s2, TO-s3, TO-s4, DO-s1, and DO-s2 implementations is occupied by the overhead of this phase. The extraction phase seems to be relatively important for DO-m1 and DO-s1 implementations. The respective reasons of this high overhead for DO-m1 and DO-s1

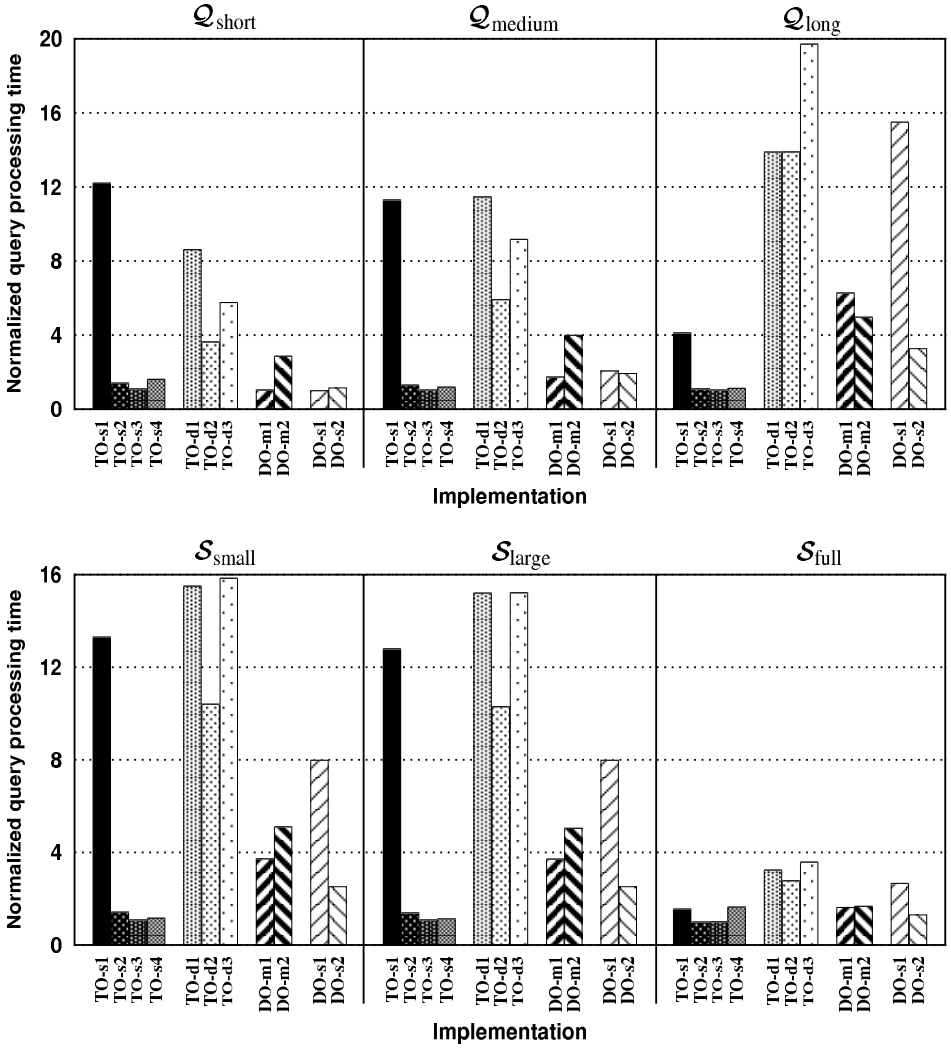


Figure 4.7: Normalized query processing times of the implementations for different query and answer set sizes.

are the high amount of accumulator shift operations and inverted list head traversals. In general, except for the case of short queries with all answers returned, the update phase incurs the highest overhead. This overhead is especially high for TO-d implementations. The creation overhead is usually negligible.

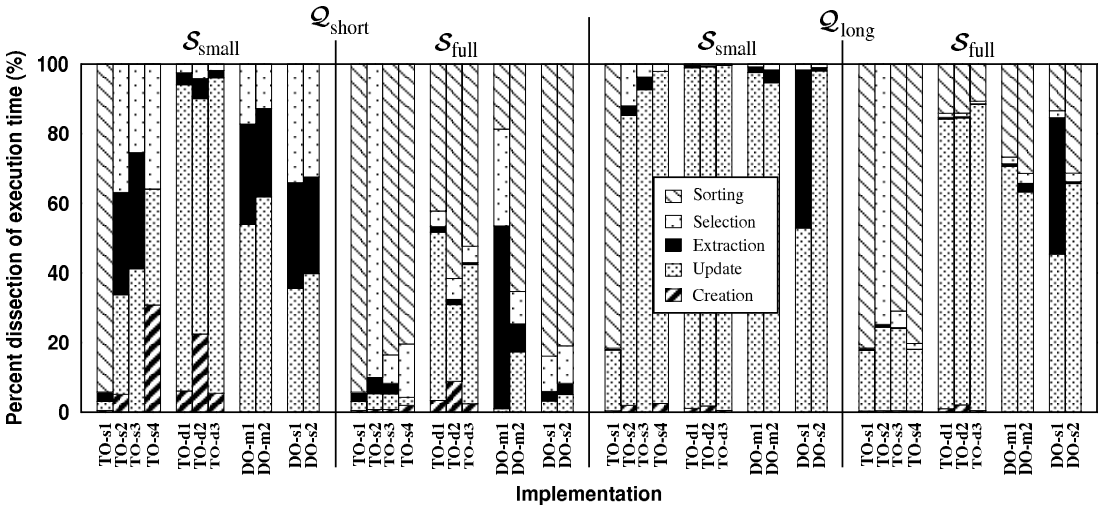


Figure 4.8: Percent dissection of execution times of query processing implementations according to the five different phases.

4.4.3 Experiments on Scalability

In this section, we provide some experimental results that evaluate scalability of the implementations with increasing number of query terms, increasing number of extracted postings, increasing answer set sizes, and increasing number of documents. In the plots, instead of displaying the actual data curves which contain many data points, we give curves fitted by regression and limit the number of data points to 11 in order to simplify drawings and ease understanding. For the same purpose, we provide a single representative curve in cases where more than one curves have a very similar behavior and hence overlap.

4.4.3.1 Effect of Number of Query Terms (Q)

Figure 4.9 shows the query processing performance for varying number of query terms. This plot is obtained by submitting 100 queries, where ith query contains i query terms, and retrieving highly ranked 10 documents at each query. As expected, DO-s1 is the implementation most affected from increasing query sizes. Other DO implementations as well as TO-d implementations are also affected since increasing number of query terms results in more posting updates, i.e.,

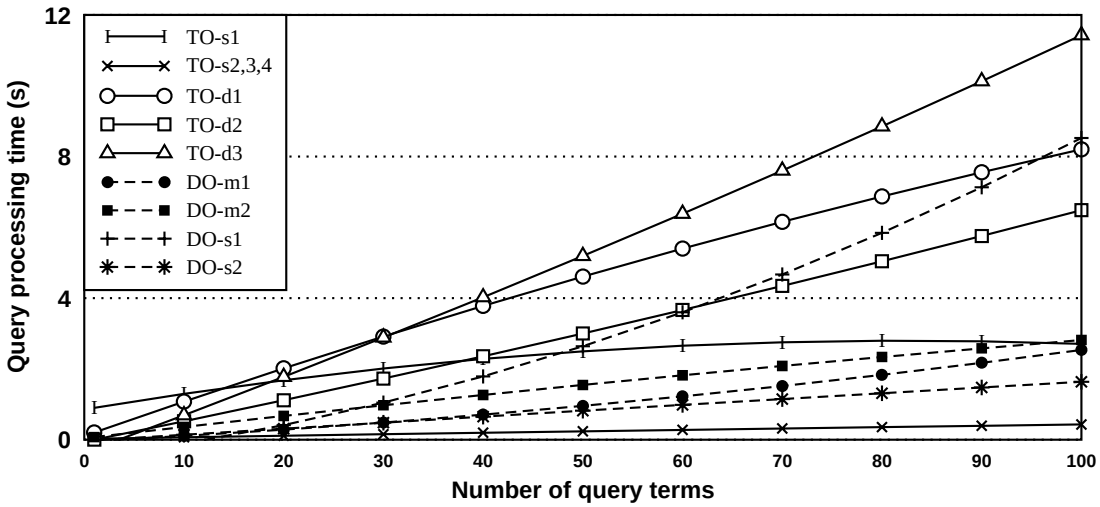


Figure 4.9: Query processing times for varying number of query terms (Q).

increases the overhead of update phase. The impact on TO-s implementations is relatively limited since update operations are not costly and extraction and selection overheads have a considerable overhead for this type of implementations.

4.4.3.2 Effect of Number of Extracted Accumulators (e)

In order to investigate the effect of the number of extracted postings on the query processing performance, we used a query set consisting of 100 queries, where each query has a single term. The queries are such that the ith query incurs $1000 \times i$ extraction operations. As a result, the top 10 documents are retrieved. Figure 4.10 shows the performance variation for increasing number of extracted accumulators. Except for TO-s1, the TO-s implementations are not affected much by the increasing number of extractions since they anyway traverse the whole accumulator array and check every score field. The different behavior of TO-s1 is basically due to the overhead of sorting. Among the TO-d implementations, TO-d2 seems to scale best with increasing e. DO implementations perform quite well since there is only a single term in the queries.

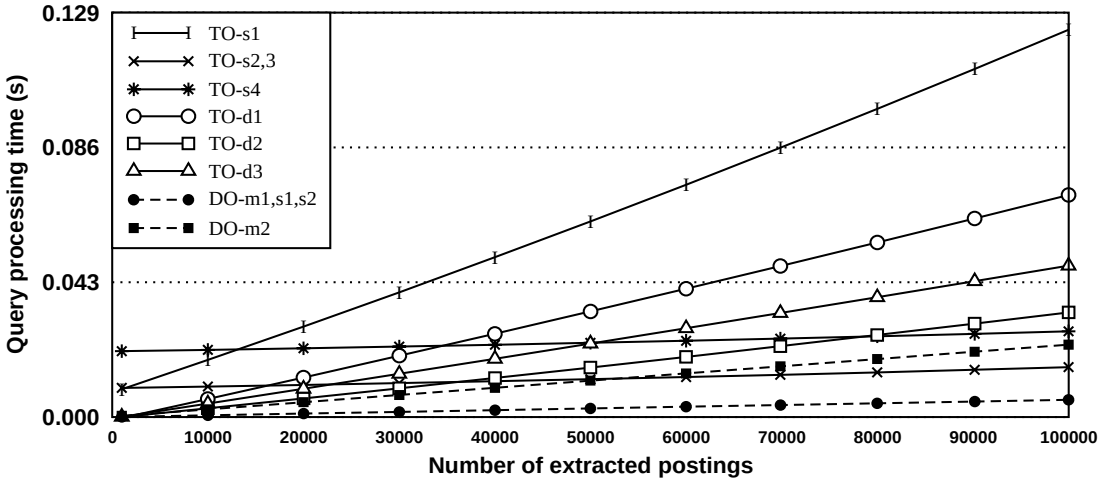


Figure 4.10: Query processing times for varying number of extracted accumulators (e).

4.4.3.3 Effect of Number of Retrieved Documents (s)

Figure 4.11 shows how the performance is affected by increasing size of answer sets. To obtain this plot, we used a single query containing a very frequent term ('university') so that the number of documents returned is high in case all documents with a nonzero score are requested. We had 100 experiments, where, for the ith experiment, the size of the answer set equals $i\%$ of the documents with a nonzero score, i.e., $s_i = i \times e/100$. According to Figure 4.11, as expected, the number of returned documents has no effect on TO-s1 since all nonzero documents are anyway sorted. For TO-s2, the curve is almost linear since the complexity of the selection phase is $s \lg e$ and e is fixed. The linear behavior of TO-s4 is also due to the linear-time selection heuristic employed. All other implementations have a similar behavior which complies with their $O(e \lg s)$ complexity. The performance gap between the curves is due to the overheads of other phases. An interesting observation obtained from Figure 4.11 is that a trade-off can be made between TO-s2, TO-s3, and TO-s4 implementations depending on the percentage of retrieved documents.

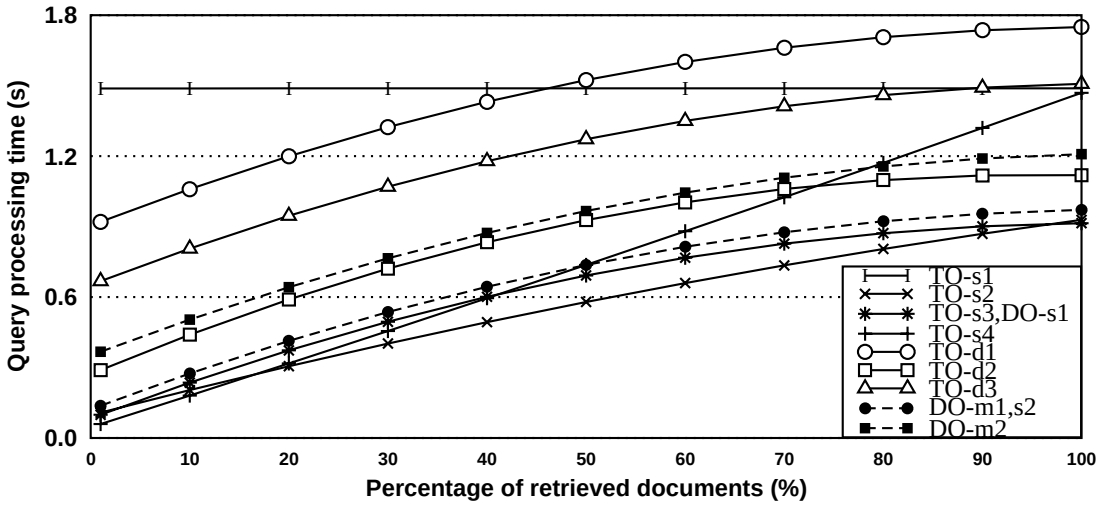


Figure 4.11: Query processing times for varying number of retrieved documents (s).

Table 4.4: The number of documents (D) and distinct terms (T) in collections of varying size

	$\mathcal{D}_{\text{small}}$	$\mathcal{D}_{\text{medium}}$	$\mathcal{D}_{\text{large}}$
D	472,533	943,672	1,883,037
T	1,467,932	2,201,992	3,325,075

4.4.3.4 Effect of Dataset Size (D)

In this section, we investigate the scalability of the implementations with respect to the document collection size. In the experiments, we use document collections of three different sizes ($\mathcal{D}_{\text{small}}$, $\mathcal{D}_{\text{medium}}$, and $\mathcal{D}_{\text{large}}$). $\mathcal{D}_{\text{small}}$ and $\mathcal{D}_{\text{medium}}$ are subsets of the original collection $\mathcal{D}_{\text{large}}$, which was used in the rest of the experiments. Table 4.4 gives the number of documents and number of distinct terms in these collections. In all experiments, we use the medium-length query set $\mathcal{Q}_{\text{medium}}$ with $\mathcal{S}_{\text{small}}$ and $\mathcal{S}_{\text{full}}$ as the answer sets.

Figure 4.12 shows the average query processing times for collections of different sizes. To better illustrate the scalability of the implementations with increasing dataset size, we also provide Table 4.5. This table provides the speedups, which is calculated as $\text{QPT}(\mathcal{D})/\text{QPT}(\mathcal{D}')$, where QPT is the average query processing

Table 4.5: Scalability of implementations with different collection sizes

	$\mathcal{Q}_{\text{medium}}$ and $\mathcal{S}_{\text{small}}$		$\mathcal{Q}_{\text{medium}}$ and $\mathcal{S}_{\text{full}}$	
Imp.	$\frac{\text{QPT}(\mathcal{D}_{\text{medium}})}{\text{QPT}(\mathcal{D}_{\text{small}})}$	$\frac{\text{QPT}(\mathcal{D}_{\text{large}})}{\text{QPT}(\mathcal{D}_{\text{medium}})}$	$\frac{\text{QPT}(\mathcal{D}_{\text{medium}})}{\text{QPT}(\mathcal{D}_{\text{small}})}$	$\frac{\text{QPT}(\mathcal{D}_{\text{large}})}{\text{QPT}(\mathcal{D}_{\text{medium}})}$
TO-s1	2.2	2.2	2.2	2.2
TO-s2	2.0	2.1	2.5	2.4
TO-s3	2.0	2.1	2.5	2.4
TO-s4	2.0	2.1	2.2	2.2
TO-d1	2.2	2.2	2.3	2.3
TO-d2	2.0	2.1	2.2	2.3
TO-d3	2.2	2.6	2.3	2.6
DO-m1	2.0	2.1	2.4	2.4
DO-m2	2.0	2.2	2.3	2.4
DO-s1	2.0	2.0	2.3	2.4
DO-s2	2.0	2.1	2.4	2.4

time, for two document collections $\mathcal{D}$ and $\mathcal{D}'$ such that $|\mathcal{D}| > |\mathcal{D}'|$. According to Table 4.5, for $\mathcal{Q}_{\text{medium}}$ and $\mathcal{S}_{\text{small}}$ combination, there is almost no scalability problem for most of the implementations as we increase the size of the document collection from small to medium, i.e., the query processing times double as the collection size doubles. However, scalability begins to become an issue when we further increase the size of the document collection. The best scalability is observed for DO-s1, whereas the least scalable implementation is TO-d3. In general, the implementations are less scalable in case all answers are returned. This is basically due to the increasing overhead of the sorting phase, which does not scale well.

4.4.4 Experiments on Space Consumption

Figure 4.13 displays the peak space consumption of each implementation. This value is equal to the maximum amount of space allocation for inverted lists, accumulators, and some auxiliary data structures, observed at any time while running the query processor for a query and answer set pair. It excludes the space for the general data structures which are utilized for each query. In all implementations, a data structure is immediately deallocated at the moment it

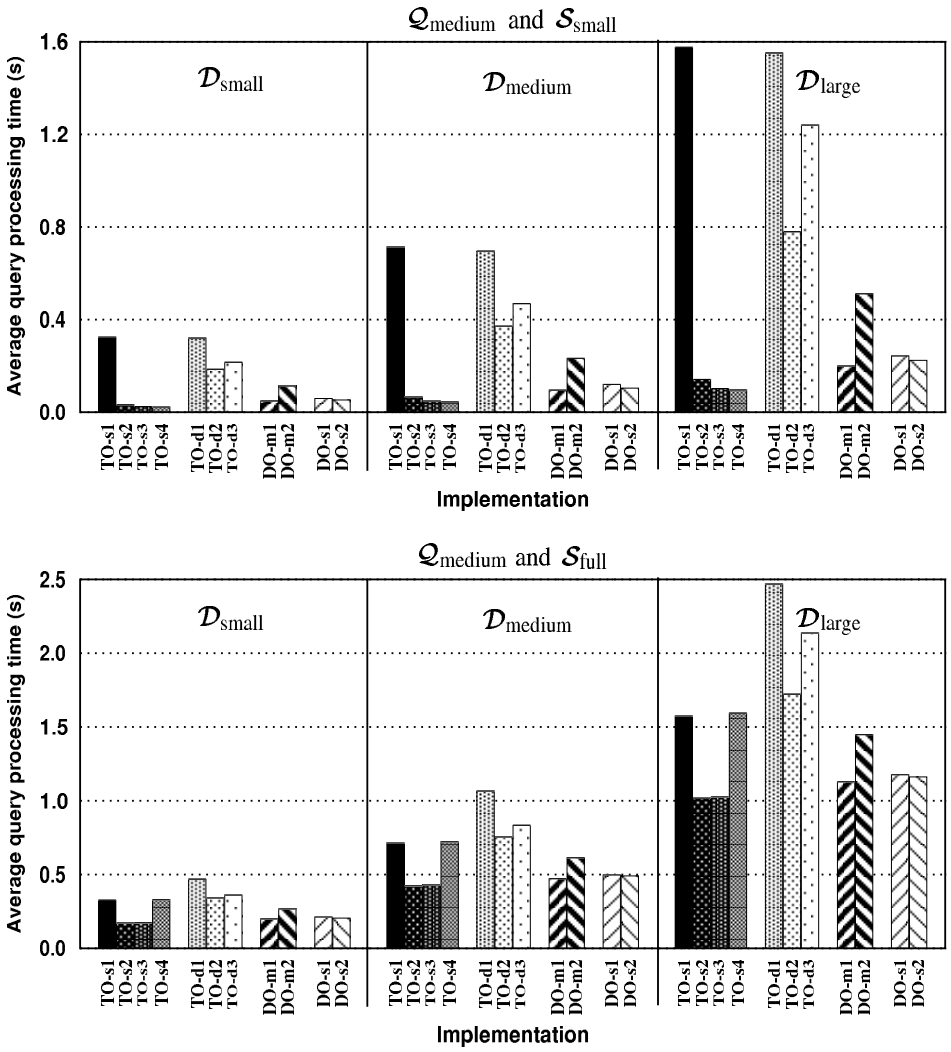


Figure 4.12: Average query processing times for collections with varying number of documents (D).

is no longer needed.

In TO implementations, the peak space consumption is reached when space for accumulators plus an inverted list is allocated. In TO-s implementations, the peak consumption is reached when the space for the inverted list with the highest number of postings is allocated. In DO implementations, it is reached when the space for all inverted lists is allocated and the number of accumulators is at the maximum.

According to Figure 4.13, for short queries, DO implementations are the most

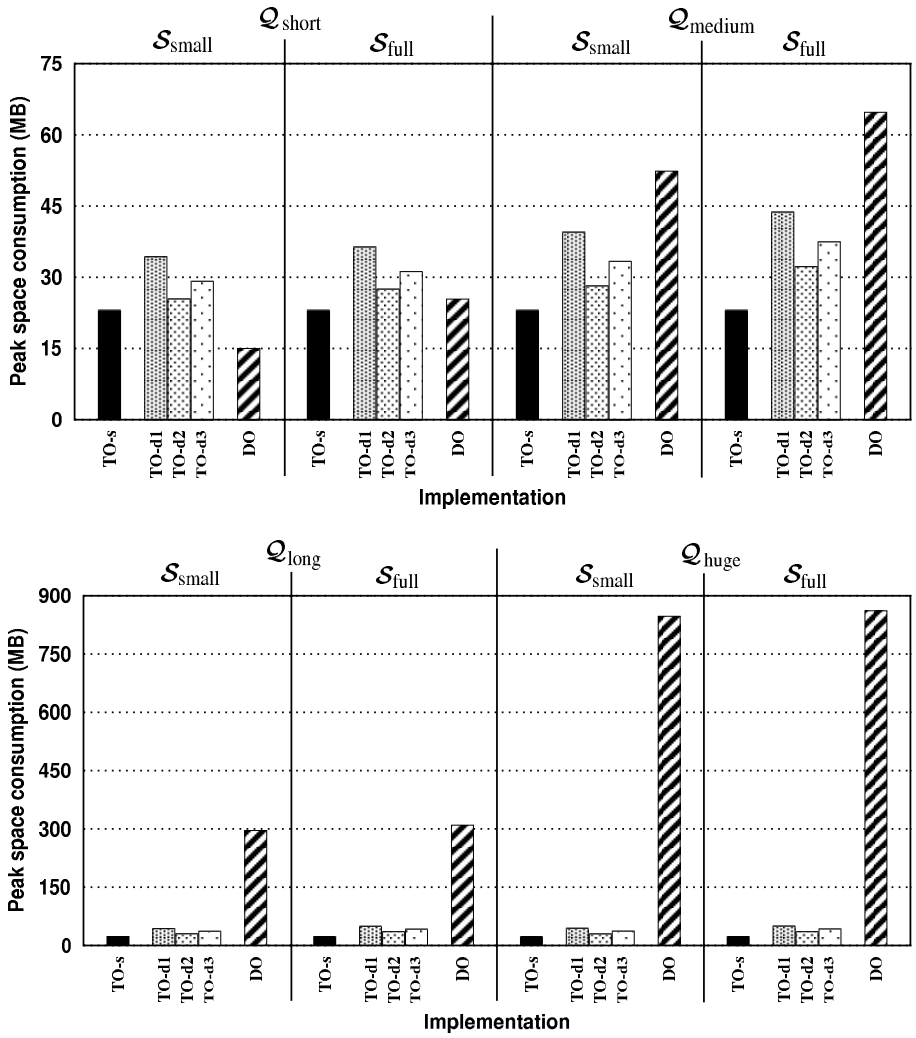


Figure 4.13: Peak space consumption (in MB) observed for different implementations.

space-efficient. However, there is a rapid increase in the space needs of this type of implementations as the queries get longer. This is basically because the storage amount of postings dominates that of accumulators since more inverted lists must be in the memory at the same time. For $\mathcal{Q}_{\text{medium}}$, $\mathcal{Q}_{\text{long}}$, and $\mathcal{Q}_{\text{huge}}$, TO-s implementations require the least amount of space. Among TO-d implementations, TO-d2 is the most space-efficient implementation.

Table 4.6: The run-time analyses of different phases in each implementation technique

Impl.	Time_C	Time_U	Time_E	Time_S	Time_R
TO-s1	$O(e)$	$O(u)$	$O(D)$	$O(1)$	$O(e \lg e)$
TO-s2	$O(e)$	$O(u)$	$O(D)$	$O(e+s \lg e)$	$O(1)$
TO-s3	$O(e)$	$O(u)$	$O(D)$	$O(s+(e-s) \lg s)$	$O(s \lg s)$
TO-s4	$O(e)$	$O(u)$	$O(1)$	$O(D)$	$O(s \lg s)$
TO-d1	$O(e)$	$O(u \lg e)$	$O(e)$	$O(s+(e-s) \lg s)$	$O(s \lg s)$
TO-d2	$O(B+e)$	$O(ue/B)^*$	$O(e)$	$O(s+(e-s) \lg s)$	$O(s \lg s)$
TO-d3	$O(e)$	$O(u \lg e)^*$	$O(e)$	$O(s+(e-s) \lg s)$	$O(s \lg s)$
DO-m1	$O(e+Q)$	$O(u \lg Q+eQ)$	$O(e)$	$O(s+(e-s) \lg s)$	$O(s \lg s)$
DO-m2	$O(e+Q)$	$O(u \lg Q)$	$O(e)$	$O(s+(e-s) \lg s)$	$O(s \lg s)$
DO-s1	$O(e+Q)$	$O(eQ)$	$O(e)$	$O(s+(e-s) \lg s)$	$O(s \lg s)$
DO-s2	$O(e+Q)$	$O(u \lg Q)$	$O(e)$	$O(s+(e-s) \lg s)$	$O(s \lg s)$

* Expected time complexities are given.

4.5 Concluding Discussion

Time complexities for different phases of the algorithms are summarized in Table 4.6. According to this table, in general, TO-s implementations differ in their selection phase whereas the update phase is discriminating for TO-d and DO implementations. Table 4.7 gives the total time and space complexities. The provided space complexities in Table 4.6 do not encapsulate the space cost of inverted lists, which is $O(e)$ for the TO implementations and $O(u)$ for the DO implementations.

It should be noted that different variants, which perform well under certain circumstances, can be created by slight modifications over the algorithms presented in this work. For example, TO-s4 can be modified so that in the extraction phase nonzero accumulators are placed in the first e elements, and the median-of-medians selection algorithm can be run only on these accumulators. In our experiments on this variant (although not reported here), we observed that this implementation is the fastest in processing short queries.

Similarly, DO-s2 can be modified using a pruning strategy such that only the postings having the minimum document id and their left and right children in

Table 4.7: The total time and space complexities for different implementations

Impl.	Time	Space
TO-s1	$O(D+e \lg e)$	$O(D)$
TO-s2	$O(D+s \lg e)$	$O(D)$
TO-s3	$O(D+e \lg s)$	$O(D)$
TO-s4	$O(D+s \lg s)$	$O(D)$
TO-d1	$O(u \lg e)$	$O(e)$
TO-d2	$O(ue/B+e \lg s)^*$	$O(B+e)$
TO-d3	$O(u \lg e)^*$	$O(e)$
DO-m1	$O(u \lg Q+eQ+e \lg s)$	$O(Q+s)$
DO-m2	$O(u \lg Q+e \lg s)$	$O(Q+s)$
DO-s1	$O(eQ+e \lg s)$	$O(Q+s)$
DO-s2	$O(u \lg Q+e \lg s)$	$O(Q+s)$

* Expected time complexities are given.

the heap are checked. This approach performs well on long queries but the bookkeeping overhead dominates at short queries. Similar optimizations are possible for space consumption. For example, TO-s2 and TO-s3 can be modified such that the accumulator array keeps only the scores. This decreases the space consumption to half of its original as long as $s \leq D/2$. Although our results indicate that TO-d implementations perform poorly, for querying scenarios where D and Q are high but e is low, implementations in TO-d category can be both time- and space-efficient.

To summarize, the results show that there is no single, superior implementation. Depending on the properties of the computing system, document collection, user queries, and answer sets, each implementation has its own advantages. Currently, we are working on a hybrid system which will, depending on the parameters, intelligently select and execute the most appropriate implementation taking both time and space efficiency into consideration. Clearly, for a better analysis, the experiments need to be repeated on a larger document collection where D and T are much higher. For this purpose, we have started a large crawl of the Web and plan to repeat the experiments on this larger collection.

Chapter 5

Skynet Parallel Text Retrieval System

As a test-bed infrastructure for evaluating the models and algorithms developed throughout the study, we have built a parallel text retrieval system, named Skynet. This system is currently running on a 48-node PC cluster located at the Computer Engineering Department of Bilkent University. Moreover, as a part of Skynet, we developed a sequential simulation software [27], which allows the performance of theoretical models in parallel text retrieval to be evaluated with different parameters and conditions. In this chapter, we present the details of these systems.

In Section 5.1, we describe the architecture of the Skynet parallel text retrieval system. In Section 5.2, we present the parallel text retrieval simulator. Section 5.3 provides the performance results of Skynet in parallel query processing with different inverted index organizations as well as some simulation results obtained with our simulator. Finally, in Section 5.4, we stress the limitations of Skynet and point at some further work.

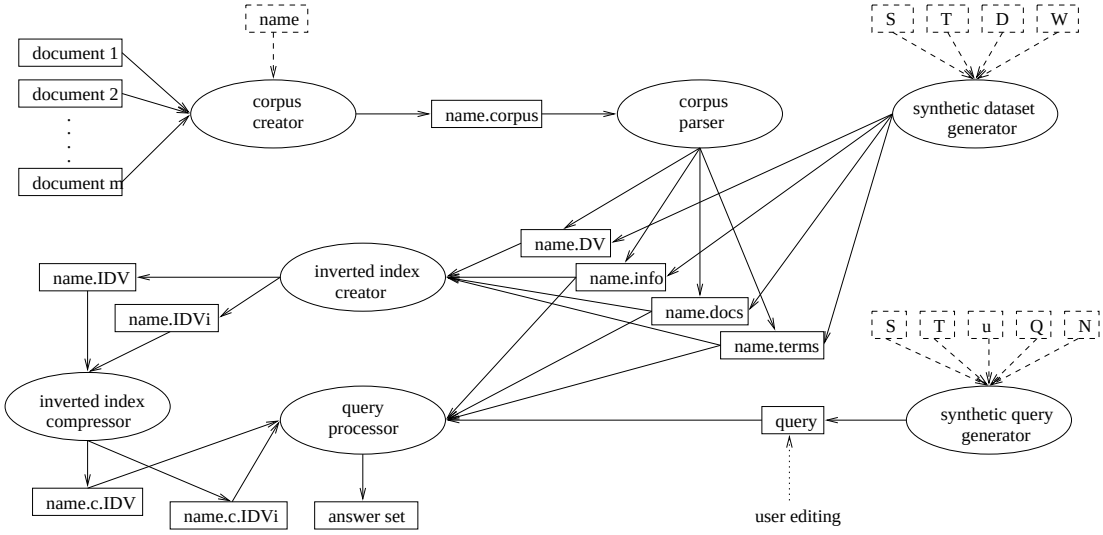


Figure 5.1: The sequential text retrieval system.

5.1 Architecture of Skynet

5.1.1 Sequential Text Retrieval System

Before the parallel text retrieval system is implemented, a sequential text retrieval system is developed as a basis for the parallel system. The architecture of this sequential system is shown in Figure 5.1. Although not mentioned here, the system contains several modules for automated query generation and collecting statistical information in a given document collection. The functions of the modules in this software system are described below.

Corpus creator: The aim of the corpus creator is to transform a given document collection into a common and standard format. This module performs all text filtering tasks on the given collection, i.e., it eliminates white spaces and removes punctuation from the text. The extracted alphanumeric character groups are converted into upper case and written into a single, formatted corpus file. Since the collection of input documents can be unformatted and vary in size and structure, it may be necessary to modify the I/O routines of this module depending on the input's properties. Hence, a separate corpus creator module must be used for each document collection at hand.

Corpus parser: Once the collection is converted into a standard corpus format, the corpus parser module is used to generate the files that keep detailed information about the corpus. The corpus parser module reads a single corpus file and produces four output files. These newly generated files are all in ASCII format and keep information about the document collection. The .terms file keeps the names of the terms their ids and document frequencies; the .docs file stores the names of the documents and their term count; the .DV file keeps the document vectors, each vector containing the term ids of the terms appearing in the document with their frequencies; and .info file stores general information about the collection such as the total number of terms and documents. The corpus parser module is able to apply some cleansing procedures on the input document corpus. The stop-words are eliminated from the corpus by supplying a stop-word file, which contains the list of words to be removed. In this module, it is also possible to remove very short or long terms, discard completely numeric terms, and apply stemming on the terms.

In addition to the preprocessing modules above, two modules exist for synthetic document and query generation. These modules, described below, are developed for experimental purposes.

Synthetic dataset generator: This module randomly generates document collections. The skewness of term distribution (S), average document size (W), and other parameters such as the total number of documents (D) to be generated and the total distinct term count in the collection (T) can be passed as user arguments. The probability distribution followed is similar to Zipf's distribution and is adapted from [73].

Synthetic query generator: This module functions similar to the synthetic dataset generator. It generates random user queries. Number of queries to be generated (N), term skewness of the queries (Q), and a cutoff value for the term frequencies (u) can be passed as argument to the module.

Inverted index creator: For efficient query processing, the document vectors in the .DV file are converted into inverted lists. The inverted index creator module performs this task. The outputs of the inverted index creator are two binary files.

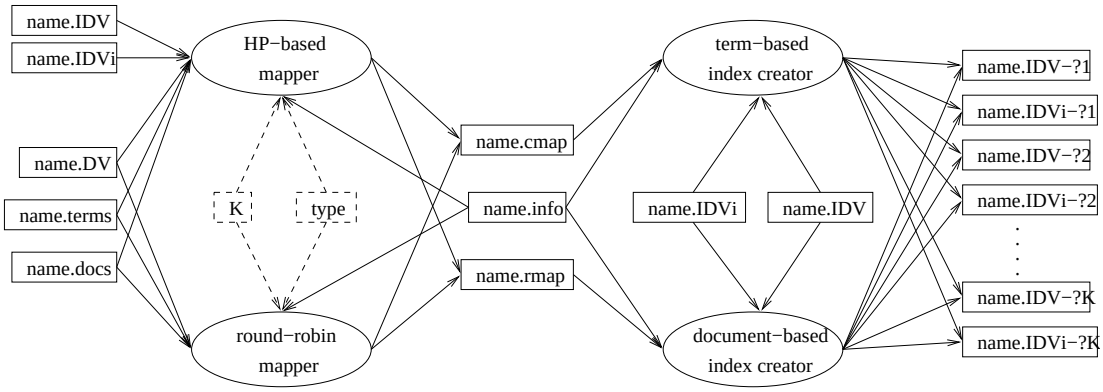


Figure 5.2: The inverted index partitioning system in Skynet.

The first file, .IDVi, is the index file, which keeps pointers to the start addresses of inverted lists. The second file, .IDV, keeps the inverted lists as a contiguous array. In determining the weights in the postings, several schemes, including the tf-idf weighting, can be used.

Inverted index compressor: This recently added module supports several implementations for data compression. As the name suggests, the module is used to compress the inverted index. The compressed inverted index is stored with the extensions .c.IDVi and .c.IDV.

Query processor: The final and the most important module of the developed sequential query processing system is the query processor. This module reads the user queries from a file and returns the set of most similar documents as answer. Our query processing module employs the ranking-based retrieval strategy and supports 11 different query processing implementations.

5.1.2 Inverted Index Partitioning System

Inverted index partitioning system includes the modules for partitioning the inverted index among a number of index servers. Figure 5.2 displays this system. The details of the modules are presented below.

HP-based mapper: By this module, a inverted index is transformed into a

hypergraph (see Chapter 3 for details). The constructed hypergraph is then K-way partitioned by the K-PaToH hypergraph partitioning toolkit (see Chapter 8 for the details of this toolkit) according to the partitioning type, which may be term- or document-based and the number K of index servers. The resulting partition induces a mapping from the term ids (the .cmap file in case of term-based partitioning) or documents ids (the .rmap file in case of document-based partitioning) to the set of index servers.

Round-robin mapper: In this module, the mapping between the terms (or document) and the index servers is created via round-robin assignment. This module also supports load-balanced and bin-packing-based assignment.

Term-based index creator: This module reads the term-based mapping generated by one of the mappers, described above, and distributes the inverted index among a number of index servers according to the mapping between the terms and index servers.

Document-based index creator: This module basically performs the same task for document-based inverted index partitioning.

5.1.3 Parallel Text Retrieval System

The Skynet parallel text retrieval system is implemented in C using the LAM/MPI [17] library. It currently runs on a 48-node PC cluster, located in the Computer Engineering Department of Bilkent University. The Skynet has 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 (PASs) to the received queries, using the local inverted indices stored in their disk. The generated PASs are later merged into a global answer set at the central broker, forming an answer to the query. Figure 5.3 displays the Skynet architecture. The functions of the central broker and the index servers are described below:

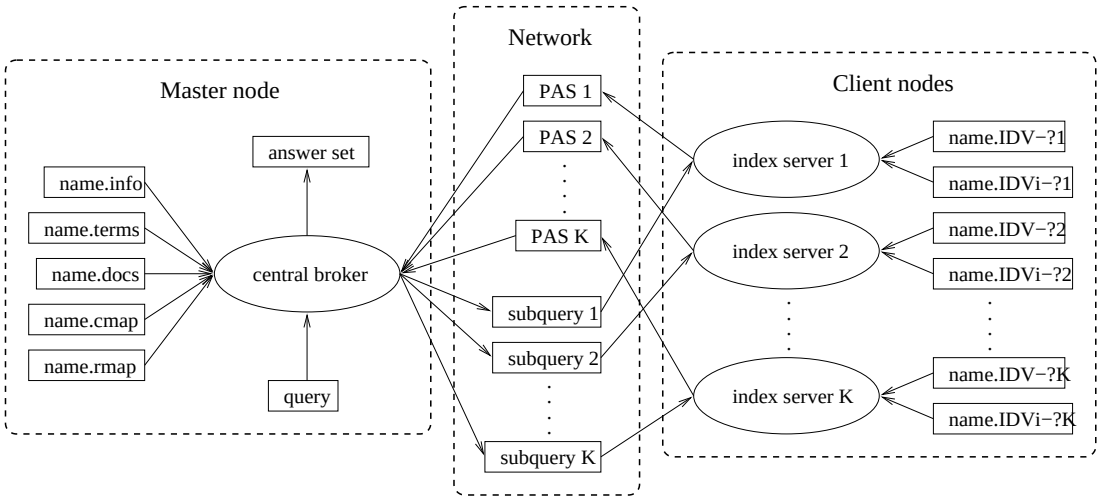


Figure 5.3: The architecture of the Skynet parallel text retrieval system.

Central broker: The central broker is responsible from a number of tasks. First, the submitted queries are converted into subqueries, one per index server, taking the partitioning of the inverted index into consideration. These subqueries are transmitted to the index servers through the local area network. Second, the central broker listens to the network for packets sent by the index servers. The incoming packets, which contain PASs, are merged into a final answer set at the central broker. Finally, the central broker is responsible from returning the final answer set to the user. For this purpose, a search interface, implemented as a CGI script, is provided. The interface to the central broker of Skynet is available via `http://skynet.cs.bilkent.edu.tr`. Appendix A contains some screenshots of this interface.

Index servers: Index servers are responsible from evaluating the incoming subqueries over their local inverted indices. Each inverted index independently runs a sequential query processor, and the PASs for a query are concurrently constructed. PASs are transmitted to the central broker over the local area network, where they will be merged.

Table 5.1: Values used for the cost components in the simulator

Cost type	Hardware	Symbol	Cost
Packing a byte of a packet	CPU	t_{pa}	5 ns
Unpacking a byte of a packet	CPU	t_{un}	5 ns
Mapping a query term to a server	CPU	t_{ma}	25 ns
Updating an accumulator	CPU	t_{up}	5 ns
Propagation delay	Network	t_{pd}	40 ns
Transmitting a byte	Network	t_{tb}	76 ns
Seek time	Disk	t_{ds}	8.5 ms
Rotational latency	Disk	t_{rl}	4.2 ms
Reading a 512-byte disk block	Disk	t_{io}	13 μs

5.2 Parallel Text Retrieval System Simulator

Response time to a query is affected by many factors, including query-dependent factors (e.g., query size and frequencies of query terms), collection-dependent factors (e.g., the number of documents in the collection and the vocabulary size), and several system-dependent factors (e.g., disk, memory, and CPU performance). Additional factors are involved in parallel query processing. These include the number of processors, network parameters, and the inverted index organization employed.

To encapsulate and observe the effect of all these factors, we simulated the working of a parallel text retrieval system by a discrete, event-based simulator implemented in C. The simulator models the three typical hardware components: disk, network, and CPU. Also, the concurrent execution of a parallel system and network queues are simulated. The abbreviations used in the equations and the cost parameters for a typical PC are provided in Table 5.1. These are the default values, and unless stated otherwise, they are used in the simulations.

There are four types of objects in the simulator: user, central broker, network packet, and index server. There are one user, one central broker, and K index server objects. The number of packet objects varies depending on the current state of the simulation. Each object has a single, time-stamped event associated

Table 5.2: Objects and events in the parallel text retrieval system simulator

ID	Event	Object	ID	Event	Object
0	Idle	All	5	Process subquery	Index server
1	Insert query	User	6	Read inverted list	Index server
2	Process query	Central broker	7	Update PAS	Index server
3	Prepare subquery packet	Central broker	8	Prepare PAS packet	Index server
4	Merge PAS	Central broker	9	Transmit packet	Packet

with it. The simulator always picks the object having the event with the smallest time-stamp and simulates its event. The simulator clock is incremented by an amount equal to the estimated duration of the event. After its current event is simulated, an object is associated with a new event of which time-stamp is set to the current simulator clock. Events with which objects can be associated are given in Table 5.2. Figure 5.4 shows the event transition diagram. Arcs represent the rules for changing events. A rule of the form $x : y \rightarrow \{z\}$ means, if the source object has event x and the destination object has event y, then the new event of the destination object will be z. For example, the rule $6 : 6 \rightarrow \{7\}$ changes the event of an index server from reading an inverted list to updating a partial answer set (PAS).

5.2.1 Disk Simulation

Our main assumption in disk simulation is that each access to an inverted list requires a disk access. That is, disk and memory caches are not simulated. Each index server is assumed to have a single disk. We consider three main components in retrieval of an inverted list from the disk: disk seek, rotational latency, and block transfer. The formula for computing the time T_{R} to read the inverted list $\mathcal{I}_i$ of a term t_i is estimated as

$$T_{\mathrm{R}} = t_{\mathrm{ds}} + t_{\mathrm{rl}} + \left\lceil \frac{|\mathcal{I}_i|}{B} \right\rceil \times t_{\mathrm{io}}, \tag{5.1}$$

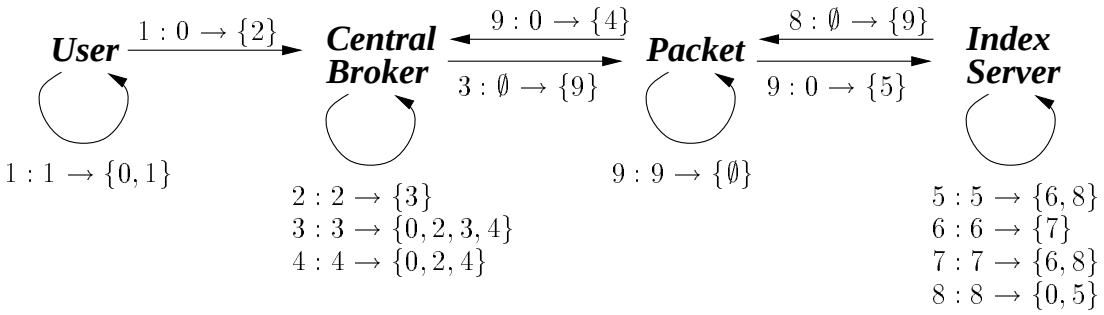


Figure 5.4: The event transition diagram for the parallel text retrieval simulator.

where B stands for the blocking factor of the disk and $|\mathcal{I}_i|$ is the size of the inverted index in bytes. In our simulations, we assumed 512-byte disk blocks.

5.2.2 Network Simulation

For network simulation, i.e., simulating transfer of subqueries and PASs between the central broker and index servers, we assumed a Fast Ethernet connection with the theoretical 100 Mbps transfer rate and negligible propagation delay. We did not model congestion at any network layer. Each network packet is assumed to have an 18-byte-long header h. The time T_{T} of transmitting a PAS over the network is estimated as

$$T_{\mathrm{T}} = t_{\mathrm{pd}} + (h + |\mathcal{P}_i|) \times t_{\mathrm{tb}}, \tag{5.2}$$

where $|\mathcal{P}_i|$ is the size (in bytes) of a PAS produced by index server S_i.

5.2.3 CPU Simulation

For CPU simulation, typical values in today's PCs are used. We simulated both subquery creation and packing/unpacking of network packets as well as updating and merging PASs. The time T_{M} for merging a PAS is estimated as

$$T_{\mathrm{M}} = |\mathcal{P}_i| \times t_{\mathrm{up}}. \tag{5.3}$$

Although, we modeled and simulated, the overheads for subquery creation and packing/unpacking of network packets may be neglected.

5.2.4 Queue Simulation

It is assumed that the network queues in both the central broker and index servers have infinite storage capability. Hence, no subquery or PAS is dropped. The network queues in index servers keep only incoming subqueries. The queue of the central broker contains both user queries and PASs sent by index servers.

5.3 Performance Results

In this section, we report the performance results obtained at the Skynet parallel text retrieval system in parallel query processing on term-based and document-based inverted index organizations. Moreover, we provide simulation results obtained on our parallel text retrieval systems simulator.

5.3.1 Experiments on Skynet

The hardware platform used in the experiments is a 32-node PC cluster interconnected by a Gigabit Ethernet switch. Each node contains an Intel Pentium IV 3.0 GHz processor, 1 GB of RAM, and runs Mandrake Linux, version 10.1. The sequential query processing algorithm is a term-ordered algorithm with static accumulator allocation [23].

As the document collection, results of a large crawl performed over the '.edu' domain (i.e., the educational US Web sites) is used. The entire collection is 30 GB and contains 1,883,037 Web pages. After cleansing and stop-word elimination,

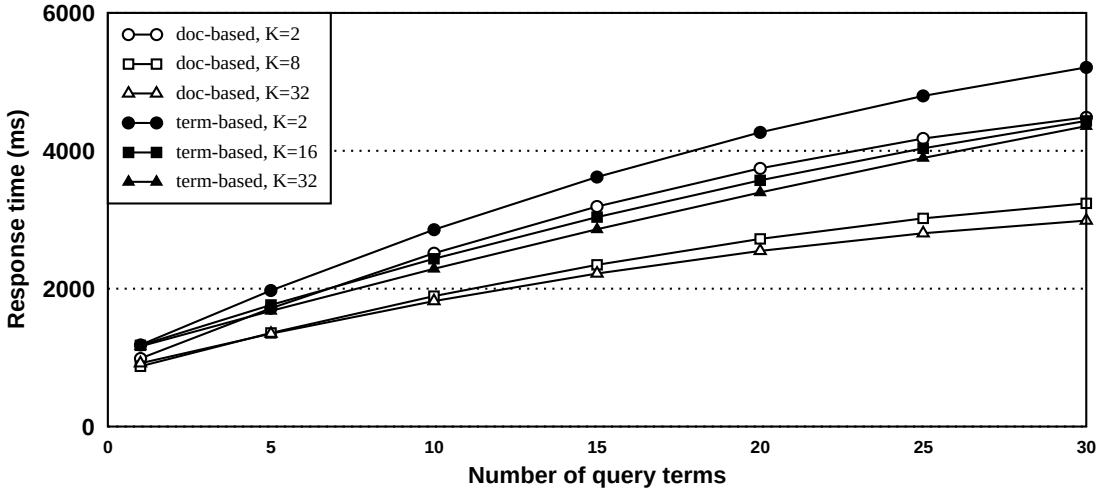


Figure 5.5: Response times for varying number of query terms.

3,325,075 distinct index terms remain. The size of the inverted index constructed using this collection is around 2.7 GB. The best-fit-decreasing heuristic used in solving the K-feasible bin-packing problem [70] is adapted to obtain the inverted index partitions over the index servers. In term-based (document-based) partitioning, terms (documents) are assigned to K index servers in decreasing number of postings, where best-fit criterion corresponds to assigning a term (document) to an index server which currently has the minimum total amount of postings.

Figure 5.5 shows the query processing performance with increasing number of query terms for different partitioning techniques and number K of index servers. In this experiment, the central broker submits a single query to the index server and waits for completion of the answer set before submitting the next query. According to the figure, document-based partitioning leads to better response times compared to term-ordered partitioning. This is due to the more balanced distribution of the query processing load on index servers in the case of document-based partitioning. The results show that term-based partitioning is not appropriate for text retrieval systems, where queries arrive to the system infrequently. The poor performance of term-based partitioning is due to the imbalance in the number of disk accesses as well as communication volumes of index servers.

Figure 5.6 presents the performance of the system with batch query processing.

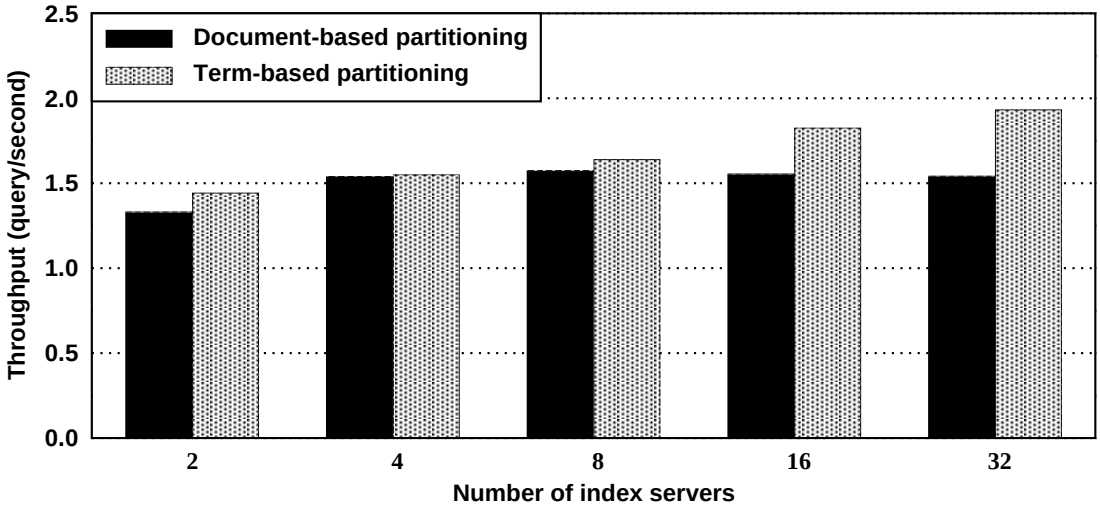


Figure 5.6: Throughput with varying number of index servers.

In these experiments, a batch of 100 queries, each containing between 1 and 5 query terms, was submitted to the system at the same time. The results indicate that term-based partitioning results in better throughput, especially as the number of index servers increases. This is mainly due to the better utilization of index servers and the capability to concurrently process query terms belonging to different queries. For document-based partitioning case, the number of disk accesses becomes a dominating overhead. In our case, after 8 index servers, the throughput starts to degrade.

These results indicate that, for batch query processing, term-ordered partitioning produces superior throughput. However, for the case where queries are infrequently submitted, document-based partitioning should be preferred.

5.3.2 Simulation Results

Simulations are conducted on the Google Web repository (see Section 2.6.1), using two query sets, one with shorter and one with longer queries. Each query set has 50 queries, whose terms are selected from the documents in the repository.

Table 5.3: Response times (in seconds) for varying number of index servers

Query size	Type	$K = 4$	$K = 8$	$K = 12$	$K = 16$	$K = 20$	$K = 24$
$1 \leq \|\mathcal{Q}\| \leq 10$	term-id	5.296	2.933	2.650	2.293	2.255	2.243
	doc-id	4.999	4.288	4.050	3.932	3.860	3.812
$11 \leq \|\mathcal{Q}\| \leq 50$	term-id	15.248	8.964	7.007	6.921	6.552	6.380
	doc-id	25.614	22.558	21.491	21.002	20.637	20.435

Table 5.3 shows the response times for the two index organizations with varying number of index servers. Except for the case with short queries and $K = 4$, term-id partitioning always outperforms document-id partitioning. The performance begins to saturate in both models as the number of index servers increases. For term-id partitioning, this is basically due to the imbalance in distribution of inverted lists and the network overhead due to the duplicate score entries in PASs. For document-id partitioning, this is because of the number of disk seeks growing linearly with K.

5.4 Limitations and Future Work

The Skynet parallel text retrieval system is by no means a complete, fully-functional search engine although it offers most functionality a search engine could offer. It is rather developed as a prototype test-bed, where the proposed models and algorithms will be evaluated. Hence, it is developed in an extensible fashion, in which new modules could be easily integrated into the system.

The limitations of the Skynet system are as follows. The system currently supports search over static document collections. That is, incremental updates are not supported on neither the document collection nor the inverted index. Although the modules are fully pipelined, the process is not automated, i.e., user intervention is necessary to convert a document collection into a queryable form. Finally, a B^+ tree implementation is required for the inverted index.

Integrating a crawling component into the system is among our future plans.

We think, when coupled with incremental updates of the inverted index, this could turn the system into a complete search engine with little effort. We have an ongoing work on document-id reassignment for efficient inverted index compression. Hence, we recently started to integrate compression/decompression modules into the system.

Chapter 6

Search Engine for South-East Europe

Based on our experience in parallel and distributed text retrieval, we developed the Search Engine for South-East Europe (SE4SEE) [19, 24]. SE4SEE is a socio-cultural search engine running on the grid infrastructure. It offers a personalized, on-demand, country-specific, category-based Web search facility. The main goal of SE4SEE is to attack the page freshness problem by performing the search on the original pages residing in the Web, rather than on the previously fetched copies as done in the traditional search engines. SE4SEE also aims to obtain high crawling rates in Web crawling by making use of the geographically distributed nature of the grid. In this work, we present the architectural design issues and implementation details of this search engine. We conduct various experiments to illustrate performance results obtained on a grid infrastructure and justify the use of the search strategy specific to SE4SEE.

The organization of this chapter is as follows. In Section 6.1, we make an introduction to issues in Web search. In Section 6.2, we provide some background information on Web crawling and text classification, which are the basic building blocks of SE4SEE, while justifying the use of the grid. In Section 6.3, we give a brief survey of the previous work on distributed/gridified Web search. Section 6.4

presents the architecture of SE4SEE and its implementation details. We report the results of the conducted experiments in Section 6.5. Finally, in Section 6.6, we conclude and discuss some future work.

6.1 Introduction

The effectiveness problem in Web search appears in both Web crawling and query processing. In Web crawling, effectiveness is related with the freshness of the indexed pages [40], which is highly correlated with the crawling efficiency, i.e., if pages are more frequently downloaded, it is more probable that the pages' cached copies are fresh. In query processing, effectiveness refers to the classical precision and recall measures, which respectively evaluate the accuracy and coverage of the results [30, 42, 123].

In addition to the effectiveness problem, both Web crawling and query processing have an efficiency problem. The efficiency problem in Web crawling [21] is due to the large scale of the Web as well as the Web's constantly evolving nature, which require pages to be downloaded and indexed frequently. According to the results reported by Google, on the average, it takes around a month to recrawl the same page again. The efficiency problem in query processing is due to the need to quickly evaluate a query over a rather large index [18, 30, 91], in the presence of many user queries being submitted concurrently. The state-of-the-art search engines attack this second problem using some algorithmic optimizations that may trade effectiveness for improved efficiency [96, 118, 126] (e.g., short-circuit evaluation) or programming improvements (e.g., trying to keep the whole Web index in the main memory). But, in general, the primary method to cope with both problems is to employ parallel/distributed computing systems, which execute multiple crawler agents to crawl the Web [39] and multiple query engines to evaluate queries over replicated/partitioned copies of the Web index [9, 103], thus increasing both page download rates and query processing throughput.

In this chapter, we present the design and implementation details of a grid-enabled search engine, Search Engine for South-East Europe[1] (SE4SEE), which somewhat differs from the above-mentioned, traditional search engines in both its design philosophy and functionality. In short, SE4SEE is a personalized, on-demand, country-specific, category-based search engine running on the grid infrastructure. It provides a Web search facility which combines crawling and classification. SE4SEE primarily addresses the page freshness and efficiency problems in Web crawling by utilizing the computational power inherently available in the grid and the grid's geographically distributed nature. In this work, we also conduct experiments to illustrate the performance of grid-enabled Web search and justify the features specific to SE4SEE.

6.2 Preliminaries

6.2.1 Web Crawling

Although it seems to be a simple task, there exist many challenges in Web crawling. The two important issues are coverage and freshness. The coverage refers to the size of the set of pages retrieved within a certain period of time. A successful crawler tries to maximize its coverage in order to provide a larger, searchable collection to the users. Similarly, the freshness of the collection is important to minimize the difference between the cached copies of pages and the originals on the Web, thus keeping the served information up-to-date.

Another important issue in Web crawling is the need for a large amount of computational resources. First, a high amount of processing power is necessary to parse the crawled pages, extract the hyperlinks, and index the pages' content. Second, large amounts of main memory is required to store and manage the data structures, which quickly and continuously grow during the crawl. The final and most important resource requirement is a high network bandwidth. The network bandwidth determines the page download rate and hence indirectly affects the

[1] SE4SEE homepage, `http://www.grid.org.tr/~SE4SEE`

crawler's coverage as well as the page freshness.

We believe that all these computational requirements make Web crawling a suitable target for grid computing [56]. In general terms, the grid can be defined as "a type of a parallel and distributed system that enables sharing, selection, and aggregation of geographically distributed autonomous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements"[2]. The grids contain computationally powerful nodes, which have the resources necessary for running a Web crawling application. Furthermore, in cases where the spatial locality of the pages is important, the geographically distributed nature of the grid can be utilized to increase page download rates, as is the case in the SE4SEE architecture.

6.2.2 Text Classification

Informally, text classification is the problem of assigning a category to a document from a predefined set of categories. In the literature, various machine learning techniques are employed to solve this problem. Most of these techniques are based on the supervised learning approach, where the classifier is trained by a set of previously labeled set of documents and then is used to predict categories for unseen test documents. The accuracy of the classification depends on the choice of the underlying machine learning algorithm as well as the quality of the documents used for training the classifier.

Most search engines rely on keyword-based search, where a query, consisting of a number of keywords, is evaluated over an inverted index, and the top k documents are returned to the user in decreasing order of their similarity to the query [86]. However, there are also approaches employing text classification in querying of document collections and/or presentation of the results. The use of text classification in search engines is mainly in the form of pre-classification (e.g., engines providing topic directories manually created by human experts) or post-classification (e.g., engines providing automated classification of the query

[2]Grid Computing Info Centre, `http://www.gridcomputing.com/gridfaq.html`

results). While the former of these increases precision, the latter enhances the presentation of the results. SE4SEE adopts the post-classification approach, where the crawled pages are classified under several topic categories before being presented to the user.

6.3 Related Work

Although many different Web search engines exist[3], the market is dominated by three major engines[4]. These engines have huge multi-processor computing infrastructures consisting of thousands of PCs. However, they are mostly centralized systems, not suitable for crawling geographically distributed Web sites. There exist quite a few information retrieval works on peer to peer environments [10], distributed systems [94], and the grid [105].

MINERVA [10] is a peer to peer Web search engine, in which each peer independently executes a Web crawler. This peer to peer system lacks a central coordinator, and hence there is no control over the coverage of each peer. Consequently, the same pages may be crawled multiple times by different peers, resulting in an overlap of pages. This overlap is a crucial problem in peer to peer Web search. MINERVA offers techniques that aim to solve this overlap problem and tries to aggregate the results of independent crawls to generate a global result.

Melnik et al. [94] proposes a distributed architecture for a Web search engine. The described search engine employs a 3-tier architecture, where each computing node is either a crawler, an indexer or a query server. Computing nodes do not use shared repositories and connected by a local area network. The crawler nodes collect the pages to be indexed from the Web and store them in local repositories. The accumulated pages are then divided into disjoint subsets and sent to the indexers. Each indexer node parses the textual information within the pages and generate local indexes. These local indexes are then merged into a global index structure and sent to the query servers. Upon receiving a search

[3] `http://www.searchenginewatch.com`
[4] `http://www.google.com`, `http://www.yahoo.com`, `http://search.msn.com`

query, one or more query servers, depending on the portion of the index residing in their local storage, answer the query according to the global index. Since crawlers, indexers and query servers share no information, crawling, indexing, and querying operations can be done concurrently in this architecture.

The use of the grid for information retrieval is relatively new. To the best of our knowledge, GRACE[5] is the only attempt to develop a grid-enabled search engine [105]. The aim of GRACE is to build a search and categorization tool over the grid. As a knowledge repository, GRACE can use both local directories or rely on the query results of other search engines. The main objective of GRACE is to analyze the search results and categorize them via linguistic analysis. In this perspective, GRACE is an unsupervised categorization tool rather than a search engine. In GRACE, the utilization of the grid resources is achieved via parallelism based on the distributed nature of the grid. A user can concurrently run multiple queries over the grid. GRACE, in turn, analyzes the query results, categorizes them, and aggregates the results of multiple queries.

Although GRACE and SE4SEE architectures both aim to utilize the grid resources, their motivations are quite different. While GRACE categorizes the results retrieved using the results obtained from other search engines, SE4SEE does not depend on the results of other search engines. Instead, the query results are retrieved directly from the Web utilizing geographical closeness in country-specific search. Furthermore, GRACE does not provide a facility for category-specific search, whereas SE4SEE allows users to select and search in a specific category as well as perform a keyword-based search.

[5]Grace Project Homepage, `http://www.grace-ist.org`

6.4 The SE4SEE Architecture

6.4.1 Features

Search Engine for South-East Europe (SE4SEE) is an attempt towards developing a grid-enabled search engine that specifically targets the countries in the South-East Europe. It is one of the two selected regional applications implemented in the EU-funded SEE-GRID FP6 project[6]. As stated in Section 6.1, SE4SEE is a personalized, on-demand, country-specific, category-based, grid-enabled search engine. Below, we briefly describe these distinguishing features of SE4SEE.

- *Personalized crawling:* In traditional search engines, the entire Web is crawled, and the pages are indexed for public search. In SE4SEE, a different crawling approach is taken. For each user query, an individual crawl is started over the Web, and hence the relevant pages are picked from the original pages. This way, since up-to-date versions of the pages are evaluated, accuracy of the resulting answer set of pages is enforced.

- *On-demand crawling:* Unlike traditional search engines, which crawl the Web continuously, in SE4SEE, the crawling task is initiated upon arrival of a user query. The users have the options to determine the stopping conditions of the crawl. This use is more appropriate for long-term query evaluation, where the user has relaxed time constraints and the Web is searched for a period of minutes or hours.

- *Category-based search:* As well as keyword-based search, SE4SEE has support for category-based search. In this approach, pages downloaded by the crawler are categorized using a previously trained text classifier. At the completion of the crawl, only the set of pages relevant to the category selected by the user is presented to her.

[6]SEE-GRID project homepage, `http://www.see-grid.org`

- *Country-specific search:* Since one of the initial motivations behind SE4SEE is to develop a socio-cultural search engine, SE4SEE provides country-specific search. In general, country-specific search can be performed based on the language of the page, the country domain of the page URL, or the geographical locality of the hosting site. Currently, in SE4SEE, the pages are resolved according to the URL extensions, e.g., the user may request only the links in the ".tr" domain to be downloaded during the crawl.

- Gridification: SE4SEE is fully enabled to the grid. The computational burden of Web crawling to an individual user is tried to be alleviated by the utilization of resources (computational power, storage capacity, and the network bandwidth) available in the grid. In particular, SE4SEE runs on the grid infrastructure established as a part of the SEE-GRID project. By submitting country-specific queries to the servers residing in the corresponding country, SE4SEE aims to exploit the geographical locality of Web pages and grid sites, thus increasing the page crawling throughput.

6.4.2 Overview of Query Processing

Basically, there are two alternatives for parallelism in grid-enabled Web crawling: intra-query or inter-query parallelism. In intra-query parallelism, a query is submitted to multiple grid nodes, and a crawling task is started at the nodes, each crawling a portion of the Web. The crawled pages are than merged into a global answer set. Although this approach offers good performance in reducing the crawling time, issues such as avoiding overlap in local answer sets or communicating inter-node links between crawlers must be addressed [39]. Inter-query parallelism, on the other hand, is a coarse-grain parallel approach, targeting high throughput in query processing. In this approach, each computing node completes the whole crawling task on its own. Although we have an on-going work on intra-query parallelism, the inter-query parallelism approach is currently employed in SE4SEE.

The deployment diagram of the SE4SEE application is given in Figure 6.1. A

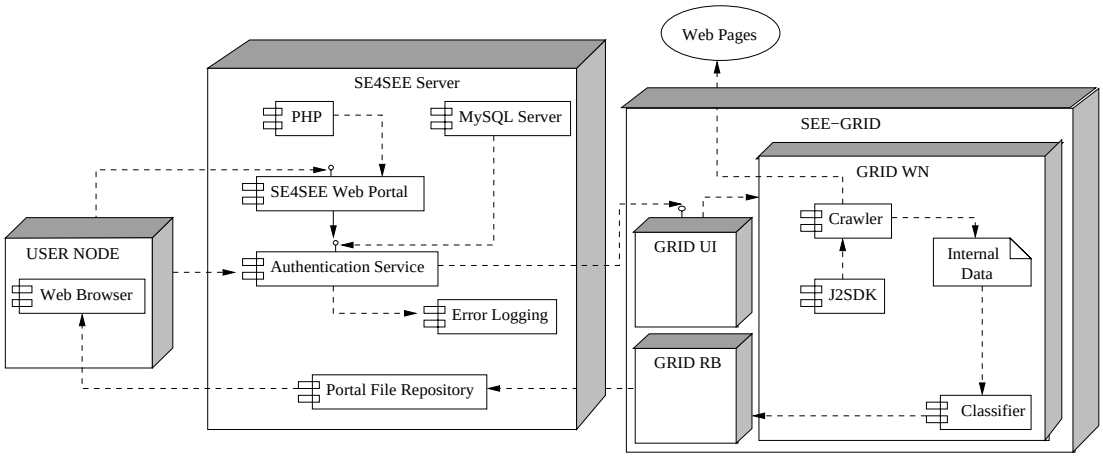


Figure 6.1: Deployment diagram of SE4SEE describing the relationship between the software and hardware components.

user requires a computer with a browser to connect to the Web portal running on the SE4SEE server. In order to prevent the overuse of grid resources, the user is expected to have a valid SE4SEE account, which is verified by the authentication module in the server. The Web portal acts as a mediator between the user and the grid. That is, it converts the user query into a grid job and submits it through the user interface (UI) to a grid node. The crawler and the classification tasks are executed on the node and the generated crawling/classification output is stored at the resource broker (RB). After a time period, the user may retrieve the output from the resource broker to the result repository in the SE4SEE server so that the results can be visualized.

In Figure 6.2, we exemplify the job execution in SE4SEE. In the figure, directed edges show the data flow over the network between different computing systems. In our sample scenario (indicated by bold edges), a user living in Romania performs a search over the Hotels located in Croatia. The user connects to the SE4SEE portal located in Ankara through her Web browser and submits the query. The portal transforms the query into an executable grid job and submits the job to an available computing node located in Zagreb, which is highly likely to be geographically close to the target Web pages. A number of hotel pages in the Croatian Web space are located, fetched, and stored in the grid node. When the crawling and classification jobs terminate, the resulting set of pages are retrieved

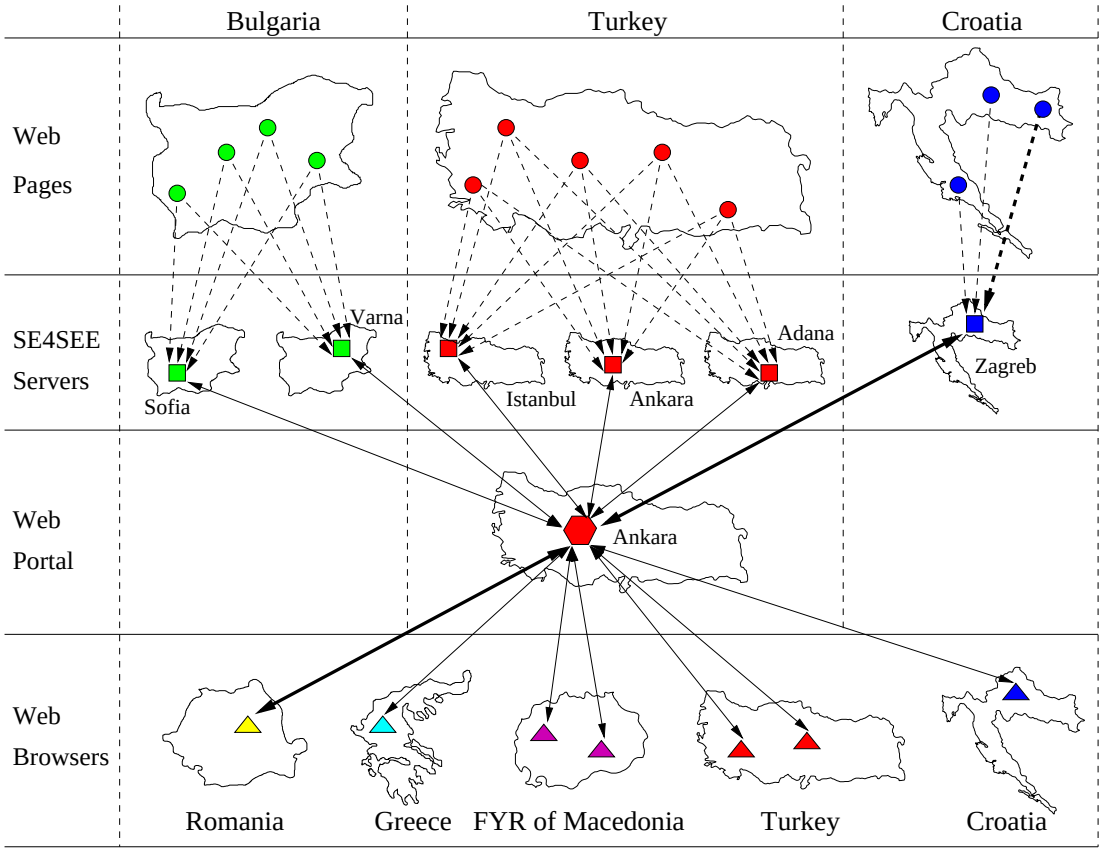


Figure 6.2: A sample search scenario over the SE4SEE architecture.

back to the portal. At any time, the user can connect to the Web portal and access the results.

6.4.3 Components

SE4SEE is composed of three main components: a crawling component, a text classification component, and a Web portal. We provide the details of these components in the following sections.

6.4.3.1 Web Crawler

Since SE4SEE is a "personal" search engine, which serves to a large number of users each with specific, personal crawling needs, an easily customizable crawler is required. Furthermore, in order to be able to adapt to the heterogeneous nature of the grid infrastructure, a platform independent crawler should be preferred. Such a crawler is capable of executing on different architectures, thus preventing the recompilation overhead and compatibility issues.

The Web crawling component of SE4SEE is implemented in Java utilizing the SPHINX[7] [95] interactive development environment for Web crawlers. SPHINX is designed to enable and ease the development of personally customized, Web-site-specific, relocatable crawlers and also provides libraries for HTML parsing, pattern matching, and common Web transformations.

The crawler in SE4SEE retrieves the pages in a breadth-first manner. This approach is more suitable for processing category-based queries, compared to depth-first traversal of pages. Unless a seed URL is provided by the user, the crawls are started from seed pages which contain links to relevant pages for each topic category. Seed pages are selected by human experts from the sites that provide up-to-date links to pages specific to each topic category. The stopping conditions for the crawls are determined by the user, who may specify either the duration of the download or the maximum number of pages crawled.

6.4.3.2 Text Classifier

The Harbinger machine learning toolkit[8] [20] is used as the text classifier in SE4SEE (see Chapter 7). This toolkit provides implementations for a number of machine learning algorithms, readily available for use in text classification. There is also built-in support for instance selection, feature selection and class balancing, which all help in improving the accuracy of classification. In particular, SE4SEE

[7]Websphinx homepage, `http://www.cs.cmu.edu/~rcm/websphinx`
[8]Harbinger homepage, `http://www.cs.bilkent.edu.tr/~berkant/coding/HMLT`

uses the naive Bayesian classifier in this toolkit for Web page classification.

The searchable categories in SE4SEE are mostly socio-cultural in nature. The currently provided categories are Banks, Dining, Festivals, Hotels, Politics, Sports, Transportation, and Universities. An important issue in successful classification is the selection of high quality Web pages. These pages should be good representatives of their categories for better classification accuracy. In SE4SEE, the training pages are manually collected from the Web by human experts. Currently, the training pages are only available for Turkey, but the training sets for several other countries are expected to be added to the system.

The execution of the classifier is pipelined with the crawler. The crawled pages are passed to the classifier for classification. The classifier is concurrently executed as a separate process, which wakes up regularly and checks if there are pages to be classified. The classifier terminates if there are no new pages for a period of time. The concurrent execution allows the network-bound operation of the crawler to be overlapped by the CPU-bound execution of the classifier, thus reducing the total query execution times.

6.4.3.3 Web Portal

As the only interaction point between the user and the SE4SEE back-end, the Web portal is a major component of the search engine. It has to be user-friendly, even though it requires a more complex interface than classic search engines due to the application's increased capabilities. There are several, SE4SEE-specific issues that are addressed in the design of the Web portal. The concept of multiple users and jobs led to implementation of an authentication system. The inherent batch-like behavior of the crawling task resulted in addition of a result maintenance mechanism. Finally, the nature of the grid environment led to the introduction of error checking and logging mechanisms.

The long execution times of a typical crawling session, especially combined with the high task initiation costs of the grid environment, prevent creation of a

real-time search engine. A significant amount of time passes between the submission of a query and the availability of the result, making it impractical for a user to wait for that amount of time. Furthermore, since crawling is a time-consuming task which requires a significant amount of network resources, the retrieved results should be stored for later access. To address this issues, SE4SEE implements a job management system.

After a user query is submitted to the portal, the job management system creates an appropriate JDL (Job Description Language) file and a shell script containing the statements to be executed. A copy of the query parameters are saved for future reference. Then, the system locates a computing node where the query can be processed. In country-specific queries, the closest grid nodes are tried to be selected by the system. Once a grid node is determined, the executables of the crawler and text classifier are transferred to the target node. The crawler and text classifier binaries are executed at the target grid node until the user-specified stopping criterion is met. When the job execution completes, the crawled pages are automatically retrieved from the resource broker to the Web portal. The user can then view the results of the search. As the results of a crawl can only be deleted explicitly, the user can save a result set and recall it multiple times later on, thereby preventing the waste of grid resources by re-querying.

To prevent the extensive use of grid resources, a user-based system is implemented. Users need to log on to the system before any grid interaction takes place. A user, once authorized, has the ability to submit queries, manage the crawling tasks and view the results of completed crawls. Queries can be submitted in two forms: a category-based search – which crawls pages, classifies them, and returns only relevant results – and a keyword search that crawls pages starting form a given seed page and returns only those that contain certain, user-given, keywords. Both types of queries result in the submission of grid jobs that can be examined and, if desired, aborted. The results for completed crawls are presented in a manner similar to common search engines, along with an option to view the page in the form it was retrieved by the crawler, effectively forming a time-stamped local cache of the results. A keyword search can also be performed in the crawled results, allowing the refinement of presented results without having to resort to

additional searches.

Finally, to ensure the durability and security of the system, additional considerations are made. A robust authentication mechanism is implemented, preventing the unprotected storage of passwords. All queries and database accesses are logged. Errors due to the underlying grid architecture are caught and interpreted. Constraints are placed on certain parameters of the application to prevent misuse of resources and to make the application behave like a "good citizen" of the grid community.

The pages of the web portal are prepared using PHP, actions from these pages invoke external applications that perform the desired tasks. All grid-interaction is over command-line utilities, relying on the robustness of these utilities in unforeseen circumstances. This method also provides a layer of abstraction between the grid and the application code, preventing any changes on grid side having an immediate effect on the application. Any data used in the invocation of these utilities is stored in a regularly backed-up MySQL database, again providing a robust solution for critical information.

6.5 Experiments

6.5.1 Platform

As the hardware platform, SE4SEE utilizes the resources available in the grid infrastructure established throughout the SEE-GRID project. These resources, in conformance with the grid philosophy, is composed of a variety of heterogeneous, geographically distributed computational resources. The SEE-GRID infrastructure is essentially a large network of computers that, although located in different regions of South-East Europe, work together to perform a common task. All of our experiments presented in this section are conducted utilizing this infrastructure.

Table 6.1: Characteristics of the grid sites used in the experiments.

Tag	Grid site	CPU (GHz)	RAM (GB)	Disk (TB)	Middleware	OS
BA	grid01.pmf.unsa.ba	Intel P4 2.4	0.5	0.036	SL 3.0.5	LCG-2.6.0
HR	grid1.irb.hr	Intel Xeon 2x2.8	2	0.03	SL 3.0.3	LCG-2.4.0
MK	grid-ce.ii.edu.mk	Intel P4 3.0	0.5	0.12	SL 3.0.3	LCG-2.4.0
BG	ce001.grid.bas.bg	Intel P4 2.4	0.5	0.1	SL 3.0.3	LCG-2.6.0
TR	grid2.cs.bilkent.edu.tr	Intel P4 3.0	1	0.08	SL 3.0.3	LCG-2.3.0
UI	ce.ulakbim.gov.tr	Intel P4 3.0	1	0.2	SL 3.0.3	LCG-2.6.0

Table 6.1 summarizes hardware/software characteristics of the grid sites available in the SEE-GRID infrastructure, which are used in our experiments. In general, it is hard to mention a typical configuration as the individual sites that form the grid have a variety of hardware resources, sometimes even having different configurations within a site. However, broadly speaking, we can say that experiments are conducted computers with an x86 processor clocked at 2.4 GHz or higher, and having at least 512 MB RAM. Although reported in the table, disk capacity is not much of a concern in the experiments since all nodes met the minimum requirements. Network connectivity of the grid sites was uncertain and had to be measured through experiments. The grid site at the last row of the table is tagged as UI since this site provides the primary interface to the SEE-GRID infrastructure. All other sites are tagged according to their geographical locality.

6.5.2 Setup

The experiments were performed using the application's command-line back-end. The typical approach of letting the grid infrastructure decide at which site the application runs is bypassed. Instead, specific sites were chosen manually and jobs are directly submitted to them. Running times for the crawler and classifier were measured by utilizing the executing system's measurement mechanisms and are typically accurate to the millisecond. Scheduling times for the task were derived from the timestamps found on the execution logs provided by the grid middleware. As the nodes on the grid are synchronized using the Network Time Protocol, the derived times are accurate to the order of seconds.

6.5.3 Results

Five sets of experiments are conducted, where each experiment tries to justify or investigate one of the search features provided by SE4SEE (Section 6.4.1). First, efficiency of personalized crawling is investigated via experiments to have a knowledge of the overhead that crawling introduces. Second, experiments are carried out on page freshness to justify the on-demand crawling strategy employed in SE4SEE. Third, we conducted experiments to reveal the benefits of geographically distributed Web crawling. Fourth, we experimented on the overheads introduced by grid-enabled Web search. Finally, we investigated the effectiveness of the category-based search provided by SE4SEE. The following sections present these experiments.

6.5.3.1 Efficiency

Personalized Web search requires a different crawling/classification task to be initiated over the Web. This is a computationally costly and time-consuming task. In this set of experiments, we try to investigate the efficiency of personalized Web crawling. For this purpose, we crawled and classified varying numbers of pages from the ".edu" domain (U.S. educational sites) and Stanford University Web server. In the experiments, the classifier is executed separately after the crawler finished downloading pages, thus enabling us to measure the relative overheads of the two components more accurately.

Figure 6.3 displays the times obtained in crawling and classifying varying number of pages using the grid site denoted with tag UI. The times for archiving/compressing the resulting set of pages are relatively negligible and hence not displayed. According to the figure, although the crawling and classification components have similar overheads at low number of pages, the crawling overhead dominates as the number of pages increases. The results show that personalized search is practical for crawling a fair number of pages. Moreover, in SE4SEE, since the crawler and classifier are concurrently executed in a pipelined fashion,

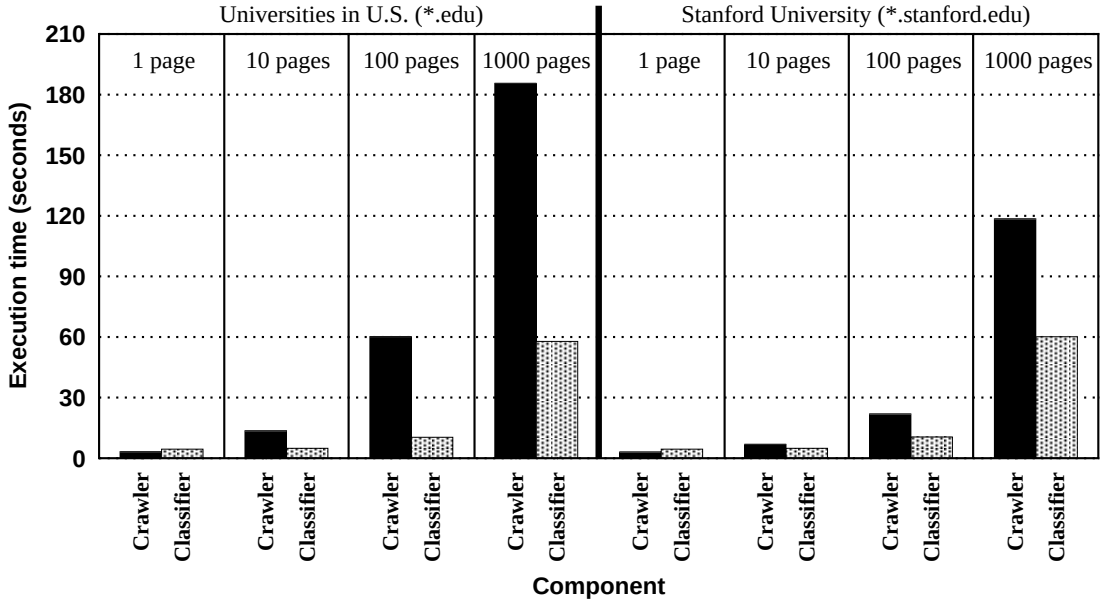


Figure 6.3: Performance of Web crawling/classification with increasing number of pages.

the classification is overlapped with network transfer and the actual total execution time is less than the sum of the reported execution times of these two components.

6.5.3.2 Page Freshness

Since obtaining high page freshness is the one of the motivations behind SE4SEE, we tried to figure out the importance of page freshness via experiments and observed the rate of change in the textual material found in the Web pages (ignoring the HTML content and other information). For this purpose, we first made an initial large crawl over a set of Web sites to obtain an initial collection. Throughout a week, the pages in the initial collection were daily recrawled. The freshness $F(t)$ of a crawl at time t is measured by the $F(t) = 100 \times (I - M(t))/I$ formula, where I is the number of pages in the initial collection and $M(t)$ is the number of pages whose content is modified (i.e., updated or deleted) and hence differs from the initial download.

Figure 6.4 displays the change of page freshness after $t = 1$ and $t = 7$ days. At

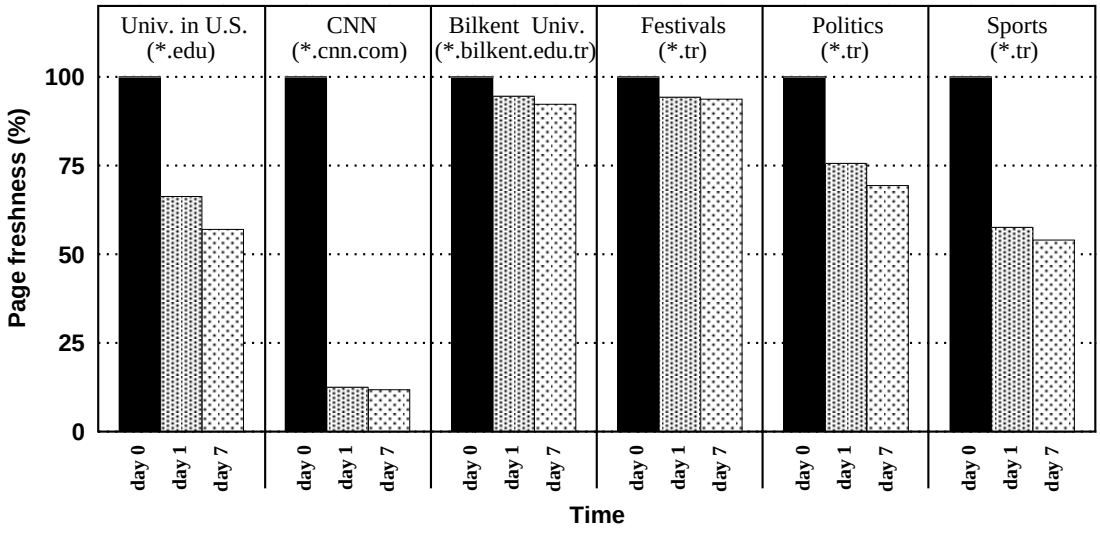


Figure 6.4: The variation of page freshness in time for different sites or topic categories.

the top of the figure, the sites or topic categories are given. The topic categories include sites picked from the training set of pages we manually created. According to Figure 6.4, a considerable portion of the pages seems to be modified frequently. Especially, in the CNN Web site, only 12.50% of the pages remain the same after a day. Similarly, after a week, almost half of the educational pages are modified. A similar behavior is not observed in the crawl made over the Bilkent University since this crawl includes pages deep in the directory hierarchy, which have a tendency to be modified less frequently.

Page freshness also shows variation among the topic categories, i.e., while pages belonging to a category remain untouched, pages in some other category may be modified frequently. For example, according to our experiments, the festival pages remain rather static, whereas sports pages are updated more frequently. Overall, we believe that these experiments justify the need for the on-demand crawling strategy employed in SE4SEE, but not available in the traditional search engines.

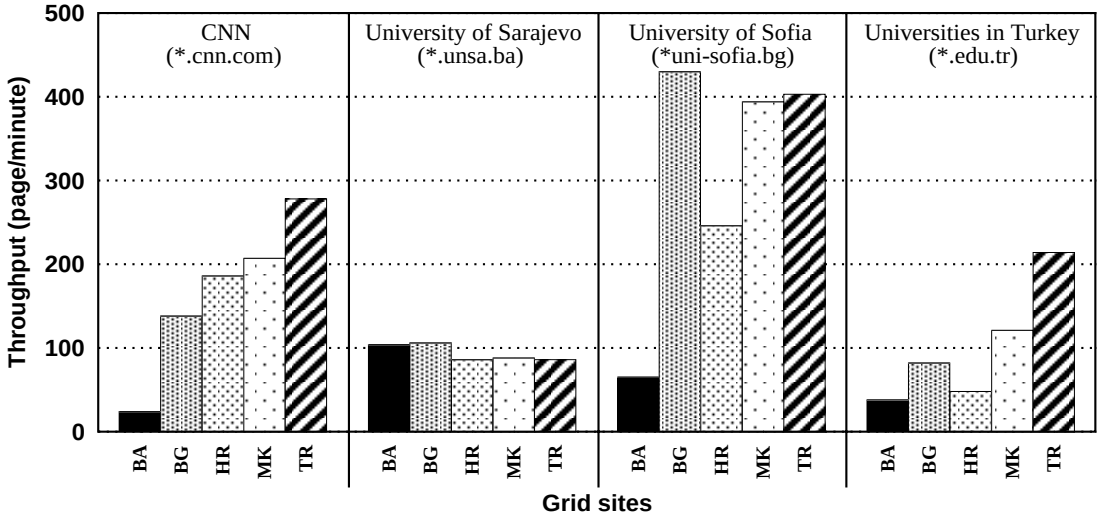


Figure 6.5: Effect of geographical locality on crawling throughput.

6.5.3.3 Geographical Locality

A primary benefit of the use of the grid infrastructure in SE4SEE is the geographically distributed nature of the grid sites. Hence, experiments are conducted to investigate the effect of utilizing the grid for geographically distributed Web crawling, where pages are tried to be downloaded by geographically closer servers. Specific sites were chosen as test sites based on their location, and jobs were directly submitted to them. In the experiments, crawling tasks were initiated at five different grid sites, located in Bosnia-Herzegovina (BA), Bulgaria (BG), Croatia (HR), FYROM (MK), and Turkey (TR).

Figure 6.5 displays the page crawling throughput (number of pages crawled per minute) achieved by the grid sites for different sets of pages. In this experiment, we first aimed to figure out the typical bandwidth of the individual sites. Note that a closer site with a low network bandwidth might perform worse than a site that is geographically far to the pages, even though the latter has a higher latency with respect to the crawled pages. To avoid misinterpretation of the other results due to the differences in the bandwidth, an approximation of the bandwidth is required. To obtain such a value, a crawl was performed on a website geographically distant to all sites, far enough to make any advantages due to the proximity negligible. For this purpose, the CNN site, located in U.S., is chosen

and crawled by all grid sites. This experiment shows that the network capacity of the grid site BA is problematic, whereas the TR site performs relatively better than the rest.

According to Figure 6.5, as expected, each grid site performs quite good in downloading the pages geographically nearby. Even the BA site, which has a limited bandwidth, achieves a fair throughput in crawling pages from the Web server of the University of Sarajevo. Similarly, the BG and TR sites achieve the highest throughput in crawling pages located Bulgaria and Turkey, respectively. Note that, if the throughputs were normalized with respect to the estimated site bandwidths, in the third experiment (the University of Sofia), the throughput gap between the BG site and the others would be more significant in favor of the BG site. These experimental results indicate that the spatial proximity between the crawling sites and the target pages plays an important role in the crawling throughput, thus justifying the geographically distributed crawling approach of SE4SEE.

6.5.3.4 Gridification

The overhead of the grid architecture had to be determined to be able to make time-comparisons to classic search engines. To this effect, several crawls of different sizes were made from the same grid site. Four times were extracted from the grid logs: the ready, scheduled running, and fetching times. The ready time is the time it takes for a job to be assigned to a site once it has been submitted to the system. The scheduled time is the duration of how long the job waits at the grid node. The running time is the execution time of the application, and the fetching time is the time it takes for the output to be retrieved form the resource broker. Note that the time it takes for the output to be transmitted from the grid node to the resource broker could not be timed.

The results in Figure 6.6 demonstrate the high start-up costs of the grid infrastructure. The startup overhead of the jobs take a dominating amount of time for smaller crawls and are still a significant source of delay even for the

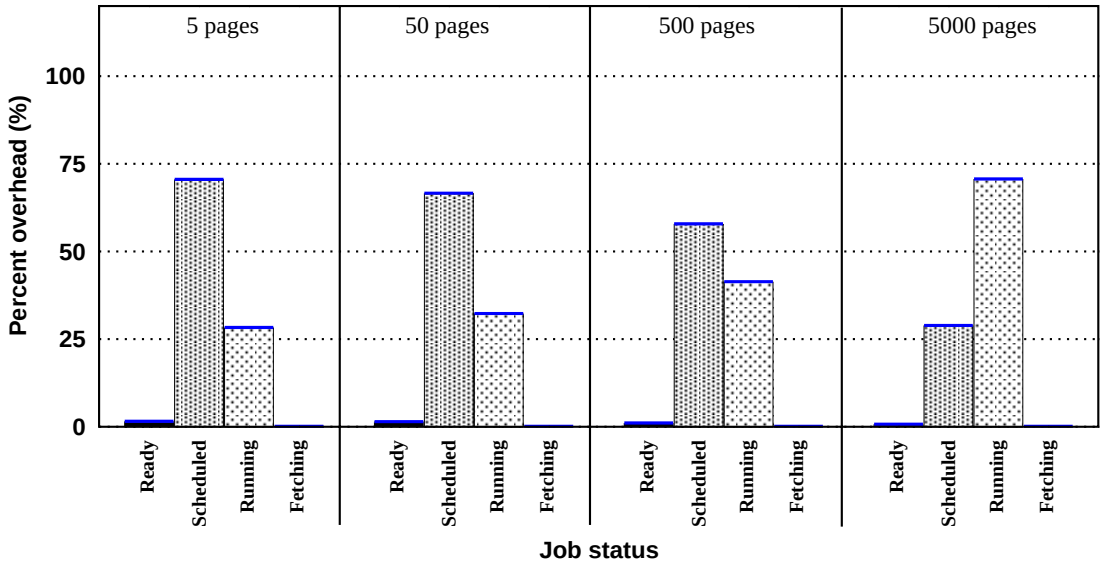


Figure 6.6: The percent dissection of duration for different phases of query execution on the grid.

larger crawls sizes. Most of this overhead comes from the delays introduced at the crawling nodes. The time to fetch the results form the resource broker is negligible, but increases linearly with the number of fetched pages, as expected.

6.5.3.5 Effectiveness

One of the benefits provided by the SE4SEE application is that it assigns categories to the retrieved pages. Selection of good seed pages for topic categories is important, as the crawling task is started from these pages and continued in a breadth-first manner. In this set of experiments, we try to investigate the quality of seed page selection and the behavior of classification. For this purpose, 100-page and 1000-page crawls are initiated for two different topic categories (banks and sports) and the distribution of pages into categories are investigated.

Figure 6.7 shows the results obtained in these experiments. As expected, as the pages are more distant in the link structure from the starting set of seed pages, the probability of classifying pages into categories other than the target category increases. This is because either the classification accuracy degrades

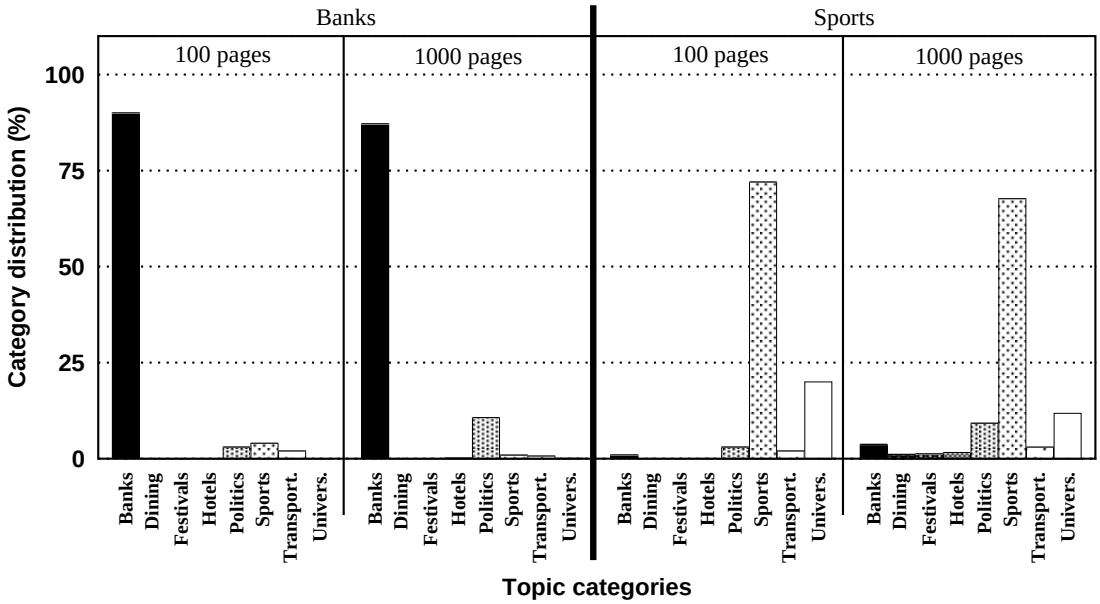


Figure 6.7: Effect of seed page selection in classification of crawled pages.

or pages belonging to irrelevant categories are crawled. For example, in the 100-page crawl performed over the sports pages, 72.0% of the to pages are classified as sports pages, whereas the rate is 67.7% in the 1000-page crawl case. The behavior of the classification also depends on the characteristics of the topic category. For example, the bank pages are more easily distinguished (a similar behavior is also observed for the politics and universities categories) even though some portion of them are classified as politics pages. Accurately classifying sports pages seem to be harder (similar to the transportation category), probably because textual features identifying sports pages overlap with the features identifying other categories.

6.6 Conclusion and Future Work

In the current version of SE4SEE, the usage of grid resources is via an inter-query-parallel approach. One other perspective could be to use an intra-query-parallel approach where each query is decomposed into subqueries running on multiple machines. As an improvement over the current SE4SEE architecture, the future

direction of the SE4SEE infrastructure is to support intra-query-parallelism to make a better use of the grid resources.

One of the assets of the SE4SEE is its socio-cultural value. Grid, by its very nature is a domain of cultural integration. As a part of the grid infrastructure, SE4SEE aims to promote the establishment of the cultural foundations of the grid infrastructure and serve as a basis for socio-cultural interaction and integration. In order to achieve it's goal, SE4SEE provides the grid community with tools for country- and category-specific search options. Hence, the categories selected so far are picked according to their emphasis on the cultural variations within the grid community. We hope this to be a good opportunity to enhance the inter-cultural relations in South-East European region.

Chapter 7

Harbinger Text Classification System

The use of text classification in the SE4SEE search engine [19, 24], presented in Section 6, required a text classification system to be designed and developed. For this purpose, we implemented the Harbinger text classification system [20], features of which are presented in this chapter. In general, the system is implemented in an extendible and modular fashion. Hence, we believe that further research in text classification can easily be built upon this prototype. We plan to take this system as a basis and use it in our future studies, as we have already started to use it in some applications [28, 29, 116] other than the SE4SEE search engine.

The outline of the chapter is as follows. In Section 7.1, we give some background information on text classification. In Section 7.2, we provide pointers to the related work in text classification and machine learning. In Section 7.3, we describe the architecture of the Harbinger text classification system. Section 7.4 gives a description of the Harbinger machine learning toolkit, utilized by the text classification system. In Section 7.5, we underline the limitations of the current system and point at some future work.

7.1 Introduction

Text classification [84, 60, 128] is the task of assigning a category to a given text document by analyzing the attributes of the textual material contained in the document. In other words, it can be defined as the process of choosing a type or topic for a piece of text among a predefined set of types or topics. Currently, text classification is considered as a hot research topic. The exponential increase in the number of documents on the Web, and the need to classify the huge amounts of textual material stored in digital libraries are the basic reasons for this research interest on text classification.

Text classification, like other text processing problems such as topic identification, text summarization, and text clustering, is a difficult problem to be solved. At the extreme case, a mixture of natural language processing and artificial intelligence techniques, which perform semantic and contextual analysis, is necessary for accurate classification of text documents. However, in our work, the focus is on statistical techniques [127], which rely solely on syntactical analysis and were abundantly used for classification in the past.

The main reasons which prevent further improvement on both efficiency and accuracy of the text classification algorithms stem from the nature of the textual data. As pointed out in many works, the main reasons are the high dimensionality of the attribute space of documents and the high amount of sparsity in documents' attribute spaces. In other words, the number of distinct terms that may occur in a document dataset is in the order of ten thousands, but only a small fraction of these terms occur in a single document. In most works, this high-dimensionality problem is attacked and eliminated within a special preprocessing step. This preprocessing step mainly contains either all or some of the following techniques: stop-word elimination, stemming, word grouping, and feature selection [90]. We integrated some of these preprocessing steps into the Harbinger text classification system.

7.2 Related Work

Applications of text classification lie on a wide range including automated document indexing [54, 62], organization of document collections [85], author detection [28, 55], text filtering [4, 49], natural language processing [58, 67], and Web page classification [3, 36, 74, 111, 116]. In the literature, different techniques are proposed as solution to the text classification problem. These solutions are mainly based on application of different machine learning algorithms on text classification [84]. Abundance of machine learning algorithms [107, 127] such as k-nearest neighbor [61], naive Bayesian [93], neural networks [98], decision trees [89], and support vector machines [111] are used in the literature. A number of machine learning tools such as Weka [124], Grid Weka [82], and Harbinger [20] are readily available for use in text classification. For an excellent survey about machine learning techniques in text classification, the reader may refer to [107].

7.3 Harbinger Text Classification System

Figure 7.1 depicts a general picture of the input-output relations among the modules of the Harbinger text classification system, using its involvement in the SE4SEE search engine for illustration purposes. In the figure, an ellipse corresponds to a module or more specifically a piece of code, which can be compiled and executed independently. Solid oblong boxes represent the files stored on the disk. The arrows on the arcs between the files and the modules indicate whether the file is supplied as input to a module or generated by it. Dashed boxes are the inputs passed as parameters to a module during the initial module startup. Bold lines indicate user interaction.

As we stated, Figure 7.1 illustrates the use of our text classification system in SE4SEE. The corpus creator and corpus parser modules are the two preprocessing modules used in generating the necessary input (in the sample case, information extracted from the training Web pages) for training the text classifier. These modules are shortly described below. In the figure, the crawled Web repository

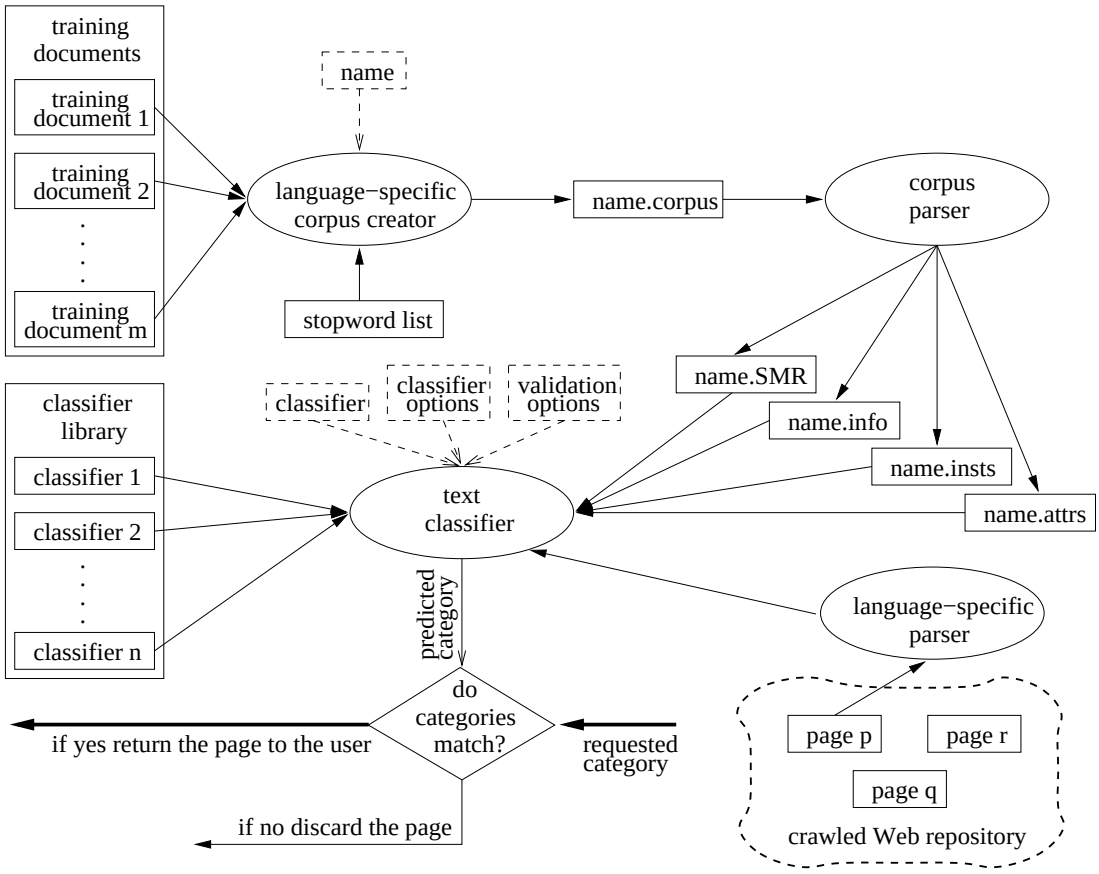


Figure 7.1: The use of the Harbinger text classification system in SE4SEE.

corresponds to the collection of test documents, whose categories are to be predicted. The test documents (i.e., Web pages) are passed from a language-specific parser, and converted into a format acceptable by the text classifier. The text classifier, using a classifier picked from the Harbinger machine learning toolkit, predicts a category for the test document. If the predicted category matches the user-requested category, the document is returned to the user, or discarded otherwise. The details of the modules in the Harbinger text classification system are summarized below.

Corpus creator: The text filtering tasks mentioned in Section 5.1.1 are performed by this module. For each document collection, a separate corpus creator must be implemented to convert the collection into a standard corpus format.

Corpus parser: This module basically has the same duty with the corpus

parser employed in Skynet (see Section 5.1.1), i.e., it generates the files that contain information about the corpus. For details of these files, please refer to Appendix B.1).

Text classifier: The core of the classification system is the text classifier module. This module calls the appropriate machine learning classifiers for the text classification task. It is also possible to run each classifier as a stand-alone application. This module offers several validation techniques and hides the details of partitioning the training document set. The classifier to be used and its options are passed as input to the module. The module first reads the document collection for training purposes and generates a classification model. It then consecutively reads the documents whose categories are to be predicted from the disk and tries to guess a category for each document using the classifier chosen from the classifier library.

Classifier library: As the classifier library, Harbinger machine learning toolkit [20] is used. The following section, Section 7.4 is dedicated to this toolkit. A more thorough discussion of the classifier options, file formats, and several examples can be found in the Harbinger machine learning toolkit manual provided in the Appendix B.

7.4 Harbinger Machine Learning Toolkit

7.4.1 Features

The Harbinger machine learning toolkit (HMLT) is a general-purpose toolkit, providing implementations for some well-known and frequently used machine learning classifiers. The primary concerns in development of HMLT are correctness, effectiveness, transparency, modularity, and re-usability. At the moment, efficiency is not claimed to be a primary concern in any part of the toolkit. This is basically due to the fact that all supported classifier implementations use common representations and data structures, preventing further utilization and employment

of some classifier-specific optimizations. However, we believe that the toolkit is quite robust and supposed to successfully execute under most circumstances.

HMLT currently supports ten different machine learning classifiers including the naive Bayesian and k-nearest neighbor classifiers. The toolkit uses a common format for storing and reading the datasets. In this format, aliases can be defined for attribute values preventing repetition of long strings. Moreover, this format allows using both dense matrix and sparse matrix representations for the datasets. Hence, for problems with high attribute dimensions (like text classification), sparse matrix format can be used to save storage space and reduce the I/O time. As well as classifiers, the toolkit offers a wrapper program to ease the validation process. By means of this wrapper, the user can easily perform cross-validation, leave-1-out validation, and some other validation techniques over the dataset. This eliminates the need for writing an extra piece of code to partition the instance set into train and test sets for each dataset at hand.

Furthermore, HMLT contains software modules for instance filtering, class balancing, and feature selection. The instance filtering module allows instances to be filtered out from the training depending on the their features. The class balancing module establishes a balance on the number of instances in each class by undersampling some classes (i.e., omitting a portion of the instances in the class). By this module, the classes as a whole can be eliminated from the dataset. Finally, the feature selection module provides support for selecting the highly representative features of a dataset. The currently available feature selection methods are document frequency thresholding and Chi-square [129]. The feature selection module is also integrated into the HMLT library.

7.4.2 Supported Classifiers

The classifiers supported by HMLT can be collected under four main headings: instance-based classifiers, probabilistic classifiers, symbolic learning classifiers, and neural network classifiers. At this point, the reader is assumed to be aware of the theoretical and practical details of these classifiers. Hence, here we present

only a very brief and rough description of the classifiers in HMLT.

- *Instance-based classifiers*: K-nearest neighbor (k-NN) [61], k-nearest neighbor with feature projections (k-NN-FP) [130] and k-means classifiers are the supported implementations of this type. K-NN and its derivations (like weight adjusted k-NN) are among the most frequently used classifiers. This type of classifiers are able to capture the local properties in the data, but fail to capture the global features of a dataset. These classifiers perform all the work in the test phase. In the test phase, all test instances are compared with the training instances and for each test instance, the most similar N training instance is determined. Depending on the classes of these N training instances, a prediction about the class of the test instance is made.

  For k-NN classifier, our classifier library supports three different distance measures for finding the similarity of two instances: cosine similarity measure, Euclidean distance measure, and Manhattan distance measure. After the most similar K instances are found the most best-matching class can be determined using majority voting or similarity score summing. Similarly, for k-NN-FP, majority voting or similarity score summing can be used to make the final decision on class selection.

- *Probabilistic classifiers*: Currently, the only classifier under this category is the naive Bayesian classifier [93]. In contrast to instance-based classifiers, naive Bayesian classifier tries to capture the global properties of a dataset. Naive Bayesian classifier works only on categorical attributes. In the training phase, the probability that a class value will be observed when an input attribute value is observed is calculated. In the test phase, for each instance, these probabilities are multiplied depending on the attribute values of the test instance. For each class value, a probability is calculated, and the class with the highest probability is selected as the predicted class. Despite its assumption that attributes appear independent of each other, naive Bayesian performs quite well in most datasets.

- *Symbolic learning classifiers*: Classifiers of this type are one-rule, decision

trees, covering rules, and classification rules. Currently, the two supported implementations are covering rules classifier (PRIM) and one-rule classifier. The covering rules classifier aims to produce human-understandable rules for classifying the test instances. During the training phase, each class value is visited. For each class value, using the training instances, a set of rules that will cover all training instances with that class value is generated. Later, these rules are used for classification of test instances. This classifier works on categorical attributes. An instance can be classified by more than one rule as belonging to many classes. In that case, a majority voting scheme can be used, or the most frequently appearing class can be assigned as the predicted class.

One-rule classifier is a similar but simpler version of covering rules classifier. It produces its rules depending on the values of just a single attribute. Although being a rather naive classifier, for small datasets with a few important attributes, it was shown that this classifier produces surprisingly good results [69].

- *Neural network classifiers*: Supported neural network classifiers [66] include perceptron, back-propagation, Kohonen and Hopfield networks. All classifiers in this category convert their input attribute values to -1 and 1, by taking the sign of actual input attribute values. Compared to others, training phase is quite slow in neural network classifiers. It may take large number of iterations to find a local optimum. Hence, it is wise to limit the epoch counts in most cases.

  Perceptron is the simplest of neural networks. It acts as a black box, which maps a given input instance to an output class value. Back-propagation neural network is a more enhanced classifier. It is known to be outperforming perceptron neural network in many applications. However, due to massive amount of computations performed, it is relatively slower. Our implementation of back-propagation neural network constructs a three layer (input, hidden and output layers) network. This classifier can be used both for classification and regression problems.

Kohonen and Hopfield networks are examples of unsupervised classifiers. In fact, these two are clustering algorithms rather than classification algorithms. However, created clusters can be used for classification purposes. In Hopfield neural networks, the class values of the training instances is not utilized at all. In Kohonen network, they are used just calibration.

7.5 Limitations of HMLT and Future Work

Current limitations of HMLT are the following. First, there is no handling of missing data values. Second, although there is a partial support for class balancing via undersampling of instances, there is no support for instance oversampling. Finally, the parser code for the input files should be enhanced and made more flexible.

The following are among our future development plans. New classifiers are planned to be added (including C4.5 decision tree algorithm). The code will be optimized in terms of both memory and execution time. Moreover, implementation of a graphical front-end to HMLT seems to be beneficial.

Chapter 8

K-PaToH Hypergraph Partitioning Toolkit

Since the models we proposed in Chapters 2 and 3 heavily rely on hypergraph partitioning, we developed an efficient and effective hypergraph partitioning tool, called K-PaToH [6]. In the future, we plan to use this partitioner in partitioning the hypergraphs created in our parallel Web crawling model and our inverted index partitioning models, instead of the currently used PaToH toolkit [33].

In the literature, K-way hypergraph partitioning is implemented usually employing the recursive bisection paradigm. In this part of our work, we show that hypergraph partitioning with multiple constraints and fixed vertices should be implemented using direct K-way refinement since the recursive-bisection-based partitioning algorithms perform considerably worse in these domains. We report the reasons for this performance degradation. We describe a careful implementation of a multi-level direct K-way hypergraph partitioning algorithm. We also experimentally show that the proposed algorithm is rather effective in partitioning hypergraphs with medium net sizes and vertex degrees.

The chapter is organized as follows. In Section 8.2, we give an overview of the previously developed hypergraph partitioning tools and a number of problems that are modeled as a hypergraph partitioning problem in the literature.

The proposed hypergraph partitioning algorithm is presented in Section 8.3. In Section 8.4, we present an extension to this algorithm in order to encapsulate the hypergraphs with fixed vertices. In Section 8.5, we verify the validity of the proposed work by experimenting on well-known benchmark datasets. The chapter is concluded in Section 8.6.

8.1 Introduction

8.1.1 Background

In the literature, hypergraphs and hypergraph partitioning find application in a wide range of parallel computing problems such as parallel sparse-matrix vector multiplication [34], sparse matrix permutation for parallel LU and QR factorization [7], performance analysis [51], and parallel volume rendering [22] as well as in many research fields including VLSI design [75], software design [13], and spatial databases [48]. Recently, various combinatorial models, which are based on hypergraph partitioning, are proposed as solutions to some complex and irregular computing problems arising in the above-mentioned fields. In these models, which formulate the original problem as a hypergraph partitioning problem, the purpose is to optimize a certain objective function (e.g., minimizing the total volume of communication in parallel volume rendering, optimizing the placement of circuitry on a dice area, minimizing the access to disk pages in processing GIS queries) while maintaining a constraint (e.g., balancing the computational load in a parallel system, using disk page capacities as an upper bound in data allocation) imposed by the problem.

Due to the direct relation between the solution qualities of the hypergraph partitioning problem and the original problem, finding a good solution to the first problem yields a good solution for the attacked problem. Consequently, the studies on developing efficient and effective hypergraph partitioning algorithms have importance in that many prior works that utilize hypergraph partitioning can benefit from the improvements introduced.

8.1.2 Definitions

Basically, a hypergraph is a generalization of the more special graph data structure. A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$ [12]. Each net n_j in $\mathcal{N}$ connects a subset of vertices in $\mathcal{V}$, which are said to be the pins of n_j and denoted as $Pins(n_j)$. Each vertex v_i has a weight w_i, and each net n_j has a cost c_j.

$\Pi=\{\mathcal{V}_1,\mathcal{V}_2,\ldots,\mathcal{V}_K\}$ is a K-way vertex partition if each part $\mathcal{V}_k$ is non-empty, parts are pairwise disjoint, and the union of parts gives $\mathcal{V}$. In Π, a net is said to connect a part if it has at least one pin in that part. The connectivity set Λ_j of a net n_j is the set of parts connected by n_j. The connectivity $\lambda_j=|\Lambda_j|$ of a net n_j is equal to the number of parts connected by n_j. If $\lambda_j=1$, then n_j is an internal net. If $\lambda_j>1$, then n_j is an external net and is said to be at cut. In Π, the weight W_k of a part $\mathcal{V}_k$ is equal to the sum of the weights of vertices in $\mathcal{V}_k$, i.e., $W_k=\sum_{v_i\in\mathcal{V}_k} w_i$.

The K-way hypergraph partitioning problem [2] is defined as finding a vertex partition Π for a given hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ such that a partitioning constraint is maintained while a partitioning objective is optimized. Although other options are possible, typically, the partitioning constraint is to maintain the balance on the part weights, and the partitioning objective is to minimize an objective function defined over the cut nets. The frequently used objective functions include the cut-net metric

$$\chi(\Pi)=\sum_{n_j\in\mathcal{N}_{\text{cut}}} c_j, \tag{8.1}$$

where $\mathcal{N}_{\text{cut}}$ is the set of cut nets and the connectivity-1 metric [88]

$$\chi(\Pi)=\sum_{n_j\in\mathcal{N}_{\text{cut}}} c_j(\lambda_j-1), \tag{8.2}$$

in which each cut net n_j contributes $c_j(\lambda_j-1)$ to the cost $\chi(\Pi)$ of partition Π. In

this work, the connectivity-1 metric is used.

8.1.3 Issues in Hypergraph Partitioning

The hypergraph partitioning problem is a rather difficult problem to be solved. In fact, the problem is known to be NP-hard, and the algorithms employed in partitioning a hypergraph are merely heuristics. Consequently, the partitioning algorithms must be carefully designed and implemented for increasing the quality of the optimization. At the same time, the computational overhead due to the partitioning process should be minimized in case this overhead (e.g., the duration of preprocessing within the total run-time of a parallel application) is a part of the entire cost to be minimized.

The very first works (mostly in the VLSI domain) on hypergraph partitioning utilized the recursive bisection (RB) paradigm, in which a hypergraph is recursively bisected (i.e., two-way partitioned) until the desired number of parts is obtained. Since RB is applied on the top-level, flat hypergraphs, especially in cases of hypergraphs with high net sizes, the obtained solution qualities are usually far from being optimal.

The multi-level hypergraph partitioning approach emerges as a remedy to the above-mentioned problem. In multilevel bisection, a hypergraph is coarsened into a smaller hypergraph after a series of coarsening levels, in which highly coherent vertices are grouped into supervertices. After the bisection of the coarsest hypergraph, the generated coarse hypergraphs are uncoarsened back to the original, flat hypergraph. At each uncoarsening level, a refinement heuristic (e.g., FM [53] or KL [80]) is applied to minimize the partitioning objective defined over the nets while maintaining a partitioning constraint on the part weights. The multi-level approach proved to be very successful in optimizing various objective functions.

With the wide-spread use of hypergraph partitioning in modeling computational problems outside the VLSI domain, the above-mentioned approach based on the multi-level RB scheme turned out to be inadequate due to the following

reasons. First, in partitioning hypergraphs with large net sizes, if the partitioning objective depends on the connectivity of the nets (e.g., the connectivity-1 metric), good partitions cannot be obtained since the decisions made at each bisection step do not take the pin distribution of cut nets across the parts into consideration. Consequently, cut nets with high number of pins may be generated in succeeding bisections of both parts, degrading the solution qualities. Second, in problems where the number of parts is high, satisfactory load balance values may not always be guaranteed since it is not possible to enforce a tight load balance constraint. Third, several formulations that are variations of the traditional hypergraph partitioning problem (e.g., multiple balance constraints, multi-objective functions, fixed vertices), which have recently started to find application in the literature, are not appropriate for the multi-level RB paradigm.

As stated above, the RB scheme performs rather poorly in problems where a hypergraph representing the computational structure of a problem is augmented by imposing more than one constraints on vertex weights or introducing a set of fixed vertices into the hypergraph. In the multi-constraint partitioning case, the solution space is usually restricted since multiple constraints may further restrict the movement of vertices between the parts. In case of fixed vertices, each fixed vertex must be finally assigned to a part. However, during the bisections it is not possible to obtain a good assignment of fixed vertices to parts since it is not yet known in what way the two parts emerging as a result of the bisection will be partitioned and the fixed vertices will be further assigned to the vertex parts.

8.1.4 Contributions

In this work, we propose a new multi-level hypergraph partitioning algorithm with direct K-way refinement. Based on this algorithm, we develop a hypergraph partitioning tool capable of partitioning hypergraphs with multiple constraints on vertex weights. We extend the proposed algorithm and the tool in order to partition the hypergraphs with fixed vertices. This extension of the algorithm finds an optimal assignment of fixed vertices to parts prior to direct K-way refinement by using the maximal weighted bipartite graph matching algorithm.

We conduct experiments on a wide range of benchmark hypergraphs with different topological properties (i.e., numbers of vertices, average net sizes). The experimental results indicate that the proposed algorithm performs better than a state-of-the-art partitioning algorithm [33] utilizing the RB paradigm in terms of both execution time and solution quality. In the case of multiple constraints and fixed vertices, the obtained results are even more promising.

8.2 Previous Work on Hypergraph Partitioning

8.2.1 Hypergraph Partitioning Tools

Although hypergraph partitioning is widely used in both academia and industry, the number of publicly available tools is quite limited. In fact, there are only three hypergraph partitioning tools that we are aware of: hMETIS [76], PaToH [33], and Parkway [115], listed in chronological order.

hMETIS [76] is the earliest hypergraph partitioning tool, published in 1998 by Karypis and Kumar. It contains algorithms for both RB-based and direct K-way partitioning. The objective functions that can be optimized using this tool are the cut-net and sum of external degrees metrics. The tool has support for partitioning hypergraphs with fixed vertices.

PaToH [33] is published in 1999 by Catalyurek and Aykanat. It is a multi-level, RB-based partitioning tool with support for multiple constraints and fixed vertices. The built-in objective functions are the cut-net and connectivity-1 cost metrics. A high number of heuristics for coarsening, initial partitioning and refinement phases are readily available in the tool for use by the end users.

Parkway [115] is the first parallel hypergraph partitioning tool, published by Trifunovic and Knottenbelt in 2004. It is suitable for partitioning large hypergraphs in multi-processor systems. The tool supports both the cut-net and connectivity-1 cost metrics.

8.2.2 Applications of Hypergraph Partitioning

Hypergraph partitioning has been used in VLSI design since 1970s [106]. The application of hypergraph partitioning in parallel computing is started by the work of Catalyurek and Aykanat [34]. This work addresses 1D partitioning of sparse matrices for efficient parallelization of matrix vector multiplies. Later, Catalyurek and Aykanat [31, 32] and Vastenhouw and Bisseling [122] proposed hypergraph partitioning models for 2D partitioning of sparse matrices. In these models, the partitioning objective is to minimize the total volume of communication incurred due to the parallelization while avoiding computational imbalance in the processors. These matrix partitioning models are utilized in different applications that involve repeated matrix-vector multiplies, such as computation of response time densities in large Markov models [51] and restoration of blurred images [121].

In the parallel computing domain, there exist hypergraph-partitioning-based models employing objective functions other than minimizing the total volume of communication. For example, Aykanat et al. [7] develop models for permuting sparse rectangular matrices into singly-bordered block diagonal form for efficient coarse-grain parallelization of linear programming, LU factorization, and QR factorization problems. Their models try to minimize the size of the border, which corresponds to minimizing the overhead of the coordination task, while providing load balance over the diagonal block sizes and thus on the computational loads of processors. Another example is the communication hypergraph model proposed by Ucar and Aykanat [119] for considering message latency overhead in parallel sparse matrix vector multiples based on 1D matrix partitioning. In this model, partitioning objective corresponds to minimizing the total number of messages, and partitioning constraint corresponds to maintaining the balance on communication volume loads of processors.

Besides matrix partitioning, hypergraph partitioning models are also proposed for use in other parallel and distributed computing applications. These include workload partitioning in data aggregation [37], image-space-parallel direct volume rendering [23], and scheduling file-sharing tasks in heterogeneous master-slave computing environments [79, 81].

Formulations that extend the traditional hypergraph partitioning problem such as fixed vertices and multiple vertex weights also find application. For instance, multi-constraint hypergraph partitioning is used for 2D checkerboard partitioning of sparse matrices [32], and parallelizing preconditioned iterative methods [120]. Hypergraph partitioning with fixed vertices is used in image-space-parallel direct volume rendering [23]. In that work, a remapping model is proposed in order to minimize the total volume of communication in data migration while balancing the rendering loads of processors. Fixed vertices are used to incorporate the initial distribution of data over the processors into the model in order to capture the total volume of communication requirement during the remapping.

Finally, we should note that hypergraph partitioning also finds application in problems outside the parallel computing domain such as road network clustering for efficient query processing [48], pattern-based data clustering [99], reducing software development and maintenance costs [13], topic identification in text databases [43], and processing spatial join operations [108].

8.3 K-Way Hypergraph Partitioning Algorithm

The proposed algorithm follows the traditional multi-level partitioning paradigm. It includes three consecutive phases: multi-level coarsening, initial partitioning, and multi-level K-way refinement. Figure 8.1 illustrates the algorithm.

8.3.1 Multi-level Coarsening

In the coarsening phase, a given flat hypergraph $\mathcal{H}^0$ is converted into a sufficiently dense hypergraph $\mathcal{H}^m$ after m successive coarsening levels. At each level ℓ, an intermediate coarse hypergraph $\mathcal{H}^{\ell+1} = (\mathcal{V}^{\ell+1}, \mathcal{N}^{\ell+1})$ is generated by coarsening the parent hypergraph $\mathcal{H}^{\ell} = (\mathcal{V}^{\ell}, \mathcal{N}^{\ell})$. The coarsening phase results in a sequence $\{\mathcal{H}^1, \mathcal{H}^2, \ldots, \mathcal{H}^m\}$ of m coarse hypergraphs.

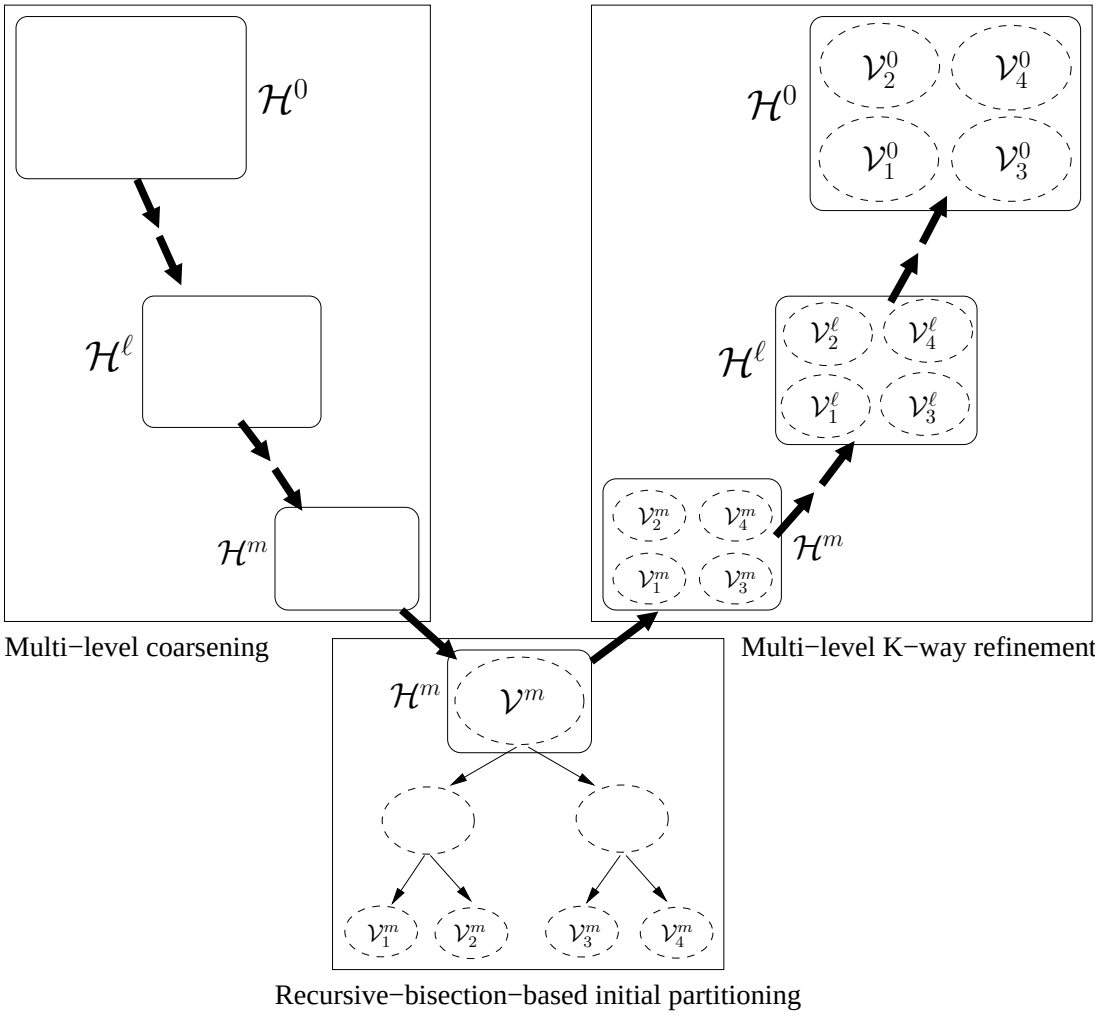


Figure 8.1: The proposed multi-level K-way hypergraph partitioning algorithm.

The coarsening at each level ℓ is performed by coalescing vertices in $\mathcal{H}^\ell$ into supervertices in $\mathcal{H}^{\ell+1}$. For vertex grouping, agglomerative or matching-based heuristics may be used. In the coarsening phase of our algorithm, we use the randomized heavy-connectivity matching heuristic. In this heuristic, vertices in vertex set $\mathcal{V}^\ell$ are visited in a random order. Each vertex $v_i^\ell \in \mathcal{V}^\ell$ is matched with a vertex $v_j^\ell \in \mathcal{V}^\ell$ if $\sum_{n_h^\ell \in \mathcal{C}} c(n_h^\ell)$, where $\mathcal{C} = \{n_h^\ell : v_i^\ell \in Pins(n_h^\ell) \wedge v_j^\ell \in Pins(n_h^\ell)\}$, is the maximum over all vertices in $\mathcal{V}^\ell$. Each matched vertex pair (v_i^ℓ, v_j^ℓ) forms a single supervertex $v_k^{\ell+1}$ in $\mathcal{V}^{\ell+1}$.

8.3.2 RB-Based Initial Partitioning

The objective in the initial partitioning phase is to obtain a K-way initial partition $\Pi^m = \{\mathcal{V}_1^m, \mathcal{V}_2^m, \ldots, \mathcal{V}_K^m\}$ of the coarsest hypergraph $\mathcal{H}^m$ before direct K-way refinement. For this purpose, we use the multi-level RB scheme of PaToH to partition $\mathcal{H}^m$. Experimental results show that, since $\mathcal{H}^m$ is already a coarse hypergraph, it is better to avoid further coarsening during the coarsening phases within PaToH. At each bisection, we use the greedy hypergraph growing heuristic to partition the intermediate hypergraphs into two parts. For two-way refinement passes over the bisected hypergraphs, we employ the tight boundary FM heuristic to obtain a viable load balance on the set $\{\mathcal{V}_1^m, \mathcal{V}_2^m, \ldots, \mathcal{V}_K^m\}$ of vertex parts.

At the end of the initial partitioning phase, if the current imbalance is over the allowed imbalance rate set by the user, a load balancer, which performs vertex moves (starting with the negative, lowest FM gains) among the K parts, is executed in order to drop the imbalance below the allowed rate. Note that, once the load imbalance is below the allowed rate, it can never rise above this rate during the direct K-way refinement.

Although possibilities other than RB exist for generating the initial set of vertex parts, RB emerges as a viable and practical method. A partition of the coarsest hypergraph $\mathcal{H}^m$ generated by RB is very amenable to FM-based refinement since $\mathcal{H}^m$ contains nets of small size and vertices of large degree.

8.3.3 Multi-level Uncoarsening with Direct K-Way Refinement

Every uncoarsening level ℓ includes a refinement step, followed by a projection step. In the refinement step, which involves a number of passes, partition Π^ℓ is refined by moving vertices among the vertex parts, trying to maintain the load balance constraint while trying to minimize the partitioning objective. In the projection step, the current coarse hypergraph $\mathcal{H}^\ell$ and partition Π^ℓ are reflected back to $\mathcal{H}^{\ell-1}$ and $\Pi^{\ell-1}$. The refinement and projection steps are iteratively

repeated until the top-level, flat hypergraph $\mathcal{H}^0$ with a partition Π^0 is obtained.

At the very beginning of the uncoarsening phase, a connectivity data structure Λ and a lookup data structure δ are created. These structures keep the connectivity of cut nets to the vertex parts in different forms. By means of Λ, which is implemented as a staggered array, the connectivity set of cut nets are obtained. That is, $\Lambda(n_i)$ returns the connectivity set Λ_i of a cut net n_i. No information is stored in Λ for internal nets. δ is an $|\mathcal{N}_{\text{cut}}|$ by K, 2D array, used to lookup the connectivity of a cut net to a part. That is, $\delta(n_i, \mathcal{V}_k)$ returns the number of pins that cut net n_i has on part $\mathcal{V}_k$ at constant time. Both Λ and δ structures are allocated once at the start of the uncoarsening phase and maintained during the projection steps. For this purpose, at each coarsening level, an inverse map of net ids is computed so that Λ and δ are modified appropriately in the corresponding projection steps. Part assignments of vertices are kept in a *part* array, where $part[v_i]$ shows the current part of vertex v_i.

During the refinement passes, only boundary vertices are considered for movement. For this purpose, a list B of boundary vertices is maintained. A vertex v_i is boundary if it is among the pins of at least one cut net n_j, i.e., $v_i \in B \Leftrightarrow v_i \in Pins(n_j) \wedge \lambda_j > 1$. B is updated at each vertex move if the move causes some vertices to become boundary or internal to a part. Each vertex v_i has a lock count b_i, indicating the number of times v_i is inserted into B. The lock counts are initially set to 0 at the beginning of each refinement pass. Every time a vertex enters B, its lock count is incremented by 1. No vertex v_i is allowed to enter B if b_i is greater than a prespecified threshold value. This way, vertices are prevented from repeatedly moving back and forth between the same pair of parts. The boundary list B is randomly shuffled at the beginning of each refinement pass.

For vertex movement, each boundary vertex $v_i \in B$ is considered in turn. The gain $gain(v_i, \mathcal{V}_k)$ of v_i is computed for each destination part $\mathcal{V}_k$ only if $v_i \notin \mathcal{V}_k \wedge v_i \in Pins(n_j) \wedge \mathcal{V}_k \in \Lambda(n_j)$ for some cut net n_j. After gains are computed, the vertex is moved to the part with the highest positive FM gain. Moves to parts with negative FM gains are ignored. Zero-gain moves are performed only if they lead

```
COMPUTE-FM-GAINS(v_i, part, Λ, δ)
    gain ← 0
    for each net n_j ∈ Nets(v_i) do
        if δ(n_j, part[v_i]) = 1 then
            gain ← gain + c_j
    if gain = 0 then
        return NO-PART-TO-MOVE
    targetParts ← {}
    for each net n_j ∈ Nets(v_i) do
        gain ← gain − c_i
        if part[v_i] ∉ Λ(n_j) then
            for each part V_k ∈ Λ(n_j) − part[v_i] do
                if V_k ∉ targetParts then
                    targetParts ← targetParts ∪ {V_k}
                gain(v_i, V_k) ← gain(v_i, V_k) + c_j
    maxGain ← −1
    for each part V_k ∈ targetParts do
        gain(v_i, V_k) ← gain(v_i, V_k) + gain
        if gain(v_i, V_k) > maxGain then
            maxGain = gain(v_i, V_k)
            maxPart ← V_k
    return maxPart
```

Figure 8.2: The algorithm for computing the K-way FM gains of a vertex v_i.

to a reduction in the load imbalance. For FM-based gain computation, we use the highly efficient algorithm given in Figure 8.2. A refinement pass terminates when all boundary vertices are considered for movement. No more refinement passes are made if a predetermined pass count is reached or improvement in the cutsize drops below a prespecified threshold.

8.3.4 Extension to Multiple Constraints

Extension to multi-constraint partitioning involves the use of multiple weights for vertices (i.e., $w_1(v_i), w_2(v_i), \ldots$). During vertex moves, each weight constraint is separately verified for load balancing. In zero-gain moves, the move is realized

only if the load balance is improved for all constraints. During the coarsening phase, in vertex matching, the maximum allowed vertex weight is set according to the constraint which has the maximum total vertex weight over all vertices. In the initial partitioning phase, the multi-constraint partitioning feature of PaToH is used with default parameters.

8.4 Extensions to Hypergraphs with Fixed Vertices

In hypergraph partitioning with fixed vertices, a number of fixed vertices are assigned to parts prior to partitioning with the condition that, at the end of the partitioning, each fixed vertex will remain in the part to which it is fixed. Our extension to partitioning hypergraphs with fixed vertices follows the multi-level paradigm, which is, in our case, composed of three phases: coarsening with modified heavy-connectivity matching, initial partitioning with maximal-weighted bipartite graph matching, and direct K-way refinement with locked fixed vertices. Throughout the presentation, we assume that, at each coarsening/uncoarsening level ℓ, f_i^ℓ is a fixed vertex in the set $\mathcal{F}^\ell$ of fixed vertices, and o_j^ℓ is an ordinary vertex in the set $\mathcal{O}^\ell$ of ordinary vertices, where $\mathcal{O}^\ell = \mathcal{V}^\ell - \mathcal{F}^\ell$. For each part $\mathcal{V}_k^0$, there is a set $\mathcal{F}_k^0$ of fixed vertices that must end up in $\mathcal{V}_k^0$ at the end of the partitioning such that $\mathcal{F}^0 = \mathcal{F}_1^0 \cup \mathcal{F}_2^0 \ldots \cup \mathcal{F}_K^0$.

In the coarsening phase of our algorithm, we modify the heavy-connectivity matching heuristic such that no two fixed vertices $f_i^\ell \in \mathcal{F}^\ell$ and $f_j^\ell \in \mathcal{F}^\ell$ are matched at any coarsening level ℓ. However, any fixed vertex f_i^ℓ in a fixed vertex set $\mathcal{F}_k^\ell$ can be matched with an ordinary vertex $o_j^\ell \in \mathcal{O}^\ell$, forming a fixed supervertex $f_i^{\ell+1} \in \mathcal{F}_k^{\ell+1}$. Ordinary vertices are matched as before. Consequently, fixed vertices are propagated throughout the coarsening such that $|\mathcal{F}_k^{\ell+1}| = |\mathcal{F}_k^\ell|$, for $k = 1, 2, \ldots, K$ and $\ell = 0, 1, \ldots, m$. Hence, in the coarsest hypergraph $\mathcal{H}^m$, there are $|\mathcal{F}^{\Updownarrow}| = |\mathcal{F}^0|$ fixed supervertices.

In the initial partitioning phase, a new hypergraph $\tilde{\mathcal{H}}^m = (\mathcal{O}^m, \tilde{\mathcal{N}}^m)$, where

$\tilde{\mathcal{N}}^m$ is a subset of nets in $\mathcal{N}^m$ whose pins contain at least two ordinary vertices, i.e., $\tilde{\mathcal{N}}^m = \{n_i^m : n_i^m \in \mathcal{N}^m \wedge |\mathcal{O}^m \cap Pins(n_i^m)| > 1\}$, is formed. Hypergraph $\tilde{\mathcal{H}}^m$, which is free from fixed vertices, is partitioned to obtain a K-way vertex partition $\tilde{\Pi}^m = \{\mathcal{O}_1^m, \mathcal{O}_2^m, \ldots, \mathcal{O}_K^m\}$ over the set $\mathcal{O}^m$ of ordinary vertices. Partition $\tilde{\Pi}^m$ induces an initial part assignment for each ordinary vertex in $\mathcal{V}^m$, i.e., $v_i^m \in \mathcal{O}_k^m \Rightarrow part(v_i^m) = \mathcal{V}_k^m$.

However, this initial assignment may not be appropriate in terms of the cutsize since the connectivity of fixed vertices are not considered at all in computation of the cutsize. At this point, a relabeling of ordinary vertex parts must be found so that the cut metric is tried to be minimized as the fixed vertices are assigned to appropriate parts. We formulate this reassignment problem as the maximum-weighted bipartite graph matching problem [38].

In this formulation, the sets of fixed supervertices and the ordinary vertex parts respectively form the two partite node sets of a bipartite graph $\mathcal{B} = (\mathcal{X}, \mathcal{Y})$. That is, in $\mathcal{B}$, for each fixed vertex set $\mathcal{F}_k^m$, there is a corresponding node $x_k \in \mathcal{X}$, and for each ordinary vertex part $\mathcal{O}_\ell^m$ there is a corresponding node $y_\ell \in \mathcal{Y}$. An edge exists between nodes x_k and y_ℓ if there is a net $n_h^m \in \mathcal{N}^m$ with $f_i^m \in Pins(n_h^m)$ and $o_j^m \in Pins(n_h^m)$ such that $f_i^m \in \mathcal{F}_k^m$ and $o_j^m \in \mathcal{O}_\ell^m$. The weight of the (x_k, y_ℓ) edge is assigned as the cost of $c(n_h^m)$ of net n_h^m. Multiple edges between the same pair of nodes are contracted into a single edge, whose weight is equal to the sum of the weights of the contracted edges.

In this setting, finding the maximum-weighted matching in bipartite graph $\mathcal{B}$ corresponds to finding a matching between fixed vertex sets and ordinary vertex parts, which has the minimum increase in the cutsize. Each edge (x_k, y_ℓ) in the resulting maximum-weighted matching $\mathcal{M}$ matches a fixed vertex set to an ordinary vertex part. By using matching $\mathcal{M}$, the ordinary vertices are reassigned to parts. An ordinary vertex $o_i^m \in \mathcal{O}_k^m$ is reassigned to a vertex part $\mathcal{V}_\ell^m$ if and only if edge (x_k, y_ℓ) is in the matching found, i.e., $part(o_i^m) = \mathcal{V}_\ell^m \Leftrightarrow (x_k, y_\ell) \in \mathcal{M}$. Each fixed vertex $f_j^m \in \mathcal{F}_\ell$ is also assigned to the corresponding vertex part, i.e., $part(f_j^m) = \mathcal{V}_\ell^m$. This reassignment induces an initial partition $\Pi^m = \{\mathcal{V}_1^m, \mathcal{V}_2^m, \ldots, \mathcal{V}_K^m\}$, which is an optimum solution in terms of the cutsize for this

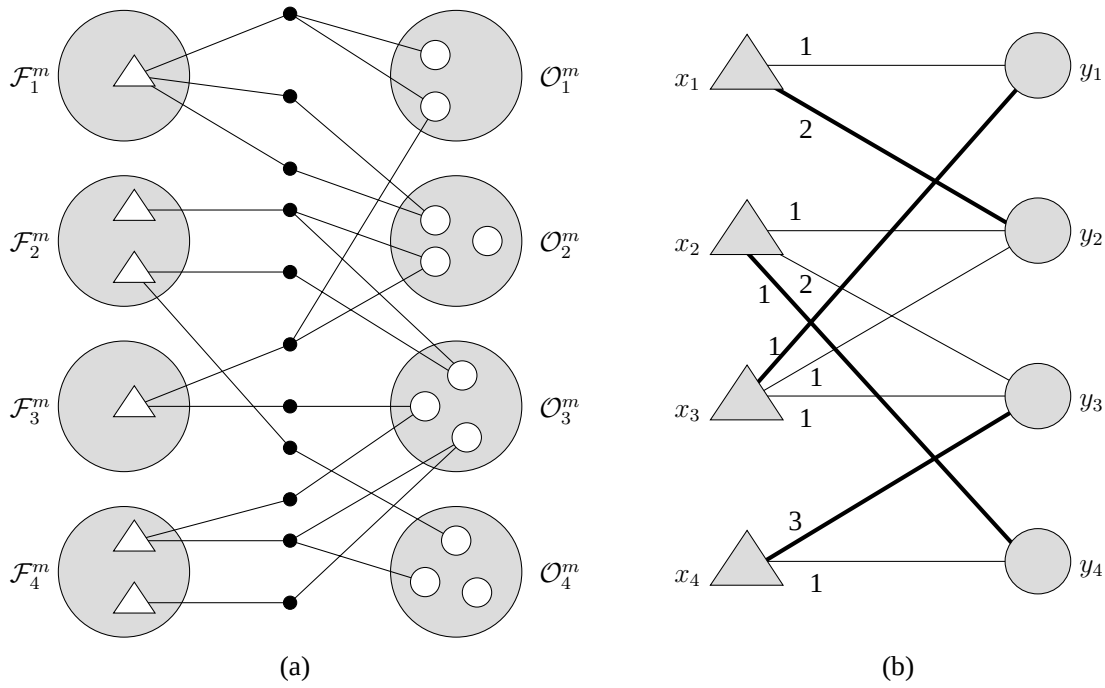


Figure 8.3: (a) A sample coarse hypergraph. (b) Bipartite graph representing the sample hypergraph in Figure 8.3(a) and assignment of parts to fixed vertex sets via maximal-weighted matching.

particular reassignment problem.

Figure 8.3(a) represents a sample, coarse hypergraph $\mathcal{H}^m$, where fixed and ordinary vertices are respectively represented as triangles and circles. For ease of presentation unit net costs are assumed. Only the nets between the fixed vertices and ordinary vertices are displayed since all cost contribution on the edges of the constructed bipartite graph are due to these nets. Note that in this hypergraph an arbitrary assignment of ordinary vertex parts to fixed vertex sets (e.g., $\mathcal{O}_k^m$ matched with $\mathcal{F}_k^m$, for $k = 1, 2, \ldots, K$) has a cost saving of $1+1+1+1 = 4$ from the cutsize. Figure 8.3(b) displays the bipartite graph constructed for the sample hypergraph in Figure 8.3(a). In the figure, triangles and circles denote the sets of fixed vertices and ordinary vertex parts, respectively. The bold edges show the maximum-weighted matching, which obtains the highest cost saving $2+1+1+3 = 7$ on the cutsize.

During the K-way refinement phase, Π^m is refined using a modified version of

the algorithm described in Section 8.3.3. Throughout the uncoarsening, the fixed vertices are locked to their parts and are not allowed to move between the parts. Hence, each fixed vertex f_i^0 of which ancestor supervertex in the m-th level is f_i^m ends up in part $\mathcal{V}_k^0 = \mathcal{V}k^m$ if and only if $f_i^m \in \mathcal{F}_k$.

8.5 Experiments

8.5.1 Experimental Platform

In the experiments, a Pentium IV 3.00 GHz PC, which has 1 GB of main memory, 512 KB of L2 cache, and 8 KB of L1 cache, is used. As the operating system, Mandrake Linux, version 13 is installed. All algorithms are implemented in C and are compiled in gcc with -O3 optimization option. Due to the randomized nature of some of the heuristics, the results are reported by averaging the values obtained in 20 different runs, each randomly seeded.

The hypergraphs used in the experiments are obtained from the University of Florida Sparse Matrix Collection [47]. These hypergraphs are originally in the form of sparse matrices that are used in various problems, emerging in different domains of scientific computing. The properties of the hypergraphs, obtained by converting these matrices, are given in Table 8.1, where the hypergraphs are sorted in increasing order of the number of pins. In all hypergraphs, the number of nets equals the number of cells and the average cell degree equals the average net size since all matrices are square matrices. Among these datasets, *Hamrle3*, *cage13*, and *pre2* datasets are partitioned on a 2 GB PC, all other parameters remaining the same, since the internal data structures maintained during the partitioning do not fit into the main memory. In the following sections and tables, we refer to PaToH and the proposed algorithm K-PaToH as R-P and K-P, respectively.

In all tables, the minimum cutsizes ($Cost_{\min}$) and average cutsizes ($Cost_{\text{avr}}$) achieved by both partitioners are reported over all datasets together with their

Table 8.1: Properties of the datasets used in the experiments

Dataset	Number of vertices	Number of pins	Average net size
dawson5	51,537	1,010,777	19.61
language	399,130	1,216,334	3.05
Lin	256,000	1,766,400	6.90
poisson3Db	85,623	2,374,949	27.74
helm2d03	392,257	2,741,935	6.99
stomach	213,360	3,021,648	14.16
barrier2-1	113,076	3,805,068	33.65
Hamrle3	1,447,360	5,514,242	3.81
pre2	659,033	5,959,282	9.04
cage13	445,135	7,479,343	16.80
hood	220,542	10,768,436	48.83
bmw3_2	227,362	11,288,630	49.65

average partitioning times ($Time_{\text{avr}}$), for changing number K of parts, where $K \in \{32, 64, 128, 256\}$. The rightmost two columns in all tables show the percent average cutsize improvement ($\%I_{\text{cost}}$) and the speedup improvement (I_{time}) of K-P over R-P. The averages over all datasets are displayed as a separate entry at the bottom of the tables. Unless otherwise stated, the number of K-way refinement passes in K-P is set to 3 in the experiments. In single-constraint partitioning, weight $w_1(v_i)$ of a vertex v_i is set equal to its vertex degree $d(v_i)$, i.e., $w_1(v_i) = d(v_i)$. In all experiments, the allowed load imbalance threshold is set to 0.10.

8.5.2 Experiments on Partitioning Quality and Performance

Table 8.2 displays the performance comparison of R-P and K-P on a variety of hypergraphs with a single partitioning constraint and no fixed vertices. According to the averages over all datasets, as K increases, K-P begins to perform better in reducing the average cutsize. The percent average cutsize improvement of 4.60% at $K = 32$ rises to 6.23% at $K = 256$. Although not displayed in the table, on the average, a similar behavior is observed in the improvement of K-P over R-P

in the minimum cutsizes achieved. A slight decrease is observed in the speedups K-P obtains as K increases. However, even at $K = 256$ vertex parts, K-P runs almost twice faster than R-P.

According to Table 8.2, except for a single case (the *language* dataset with $K = 32$), K-P achieves cutsize values lower than those of R-P at all datasets and K values. In general, K-P performs relatively better in reducing the cutsize on hypergraphs having net sizes between 6 and 20. This is expected since R-P is known to be already very effective in partitioning hypergraphs with low net sizes (e.g., *language* and *Hamrle3*). On the other hand, in partitioning hypergraphs with large net sizes (e.g., *barrier2-1*, *bmw3_2*), the partitioners begin to display a close performance in minimizing the cutsize since the solution space of the partitioning problem is restricted as the nets are highly connected to the parts and the FM-based heuristics perform poorly.

Table 8.3 shows the behavior of K-P with increasing number of K-way refinement passes. The values reported are averages over all datasets. According to the results in this table, with number of refinement passes greater than 3, the improvement of K-P over the cutsize and hence on R-P becomes marginal compared to the performance of 3-pass refinement. A similar saturation is observed at the decrease in the speedups as the number of refinement passes go beyond 3 passes. This is basically due to fact that our FM-based refinement heuristic is trapped in a local minima after a few refinement passes and perform relatively few vertex moves as the number of passes increases.

8.5.3 Experiments on Multi-constraint Partitioning

Tables 8.4 and 8.5 show the performance of R-P and K-P in partitioning hypergraphs with multiple constraints (2 and 4 constraints, respectively). In the 2-constraint case, a unit weight of 1 is used as the second vertex weight for all vertices, i.e., $w_2(v_i) = 1$. In addition to this, in the 4-constraint case, a random weight $w_3(v_i) = \alpha_i$, where $1 \leq \alpha_i \leq w_1(v_i) - 1$, and $w_4(v_i) = w_1(v_i) - \alpha_i$ are respectively used as the third and fourth vertex weights.

Table 8.2: Performance of PaToH and K-PaToH in partitioning hypergraphs with a single partitioning constraint and no fixed vertices

		$Cost_{\min}$		$Cost_{\text{avr}}$		$Time_{\text{avr}}$		Improvements	
Dataset	K	R-P	K-P	R-P	K-P	R-P	K-P	$\%I_{\text{cost}}$	I_{time}
dawson5	32	6,959	6,432	7,468	6,761	1.524	0.611	9.46	2.50
	64	11,293	10,117	11,907	10,734	1.809	0.784	9.85	2.31
	128	19,058	17,328	19,393	17,948	2.099	1.045	7.45	2.01
	256	29,655	28,160	30,351	28,634	2.380	1.411	5.66	1.69
language	32	94,210	94,393	95,399	96,047	12.266	8.833	-0.68	1.39
	64	107,299	106,670	108,432	107,467	13.064	9.033	0.89	1.45
	128	119,636	118,406	120,234	119,398	13.835	9.098	0.70	1.52
	256	131,251	130,358	131,690	130,939	14.489	9.484	0.57	1.53
Lin	32	49,458	43,848	50,800	44,629	5.763	4.190	12.15	1.38
	64	68,994	60,022	70,645	60,827	6.632	4.927	13.90	1.35
	128	91,701	80,076	93,622	80,638	7.471	5.868	13.87	1.27
	256	119,529	105,324	121,346	106,016	8.327	7.195	12.63	1.16
poisson3Db	32	40,599	38,767	41,759	39,891	9.358	5.904	4.47	1.59
	64	59,198	56,362	60,013	58,422	10.407	6.783	2.65	1.53
	128	84,630	82,377	86,118	83,930	11.366	7.635	2.54	1.49
	256	121,733	115,931	123,051	117,988	12.240	8.509	4.11	1.44
helm2d03	32	13,016	12,350	13,591	12,904	7.689	2.713	5.06	2.83
	64	19,677	18,689	20,251	19,237	8.757	3.073	5.01	2.85
	128	29,169	27,665	29,696	28,104	9.801	3.577	5.36	2.74
	256	42,763	40,682	43,079	41,033	10.850	4.405	4.75	2.46
stomach	32	26,231	25,559	27,054	26,048	6.635	2.899	3.72	2.29
	64	37,885	36,784	38,918	37,207	7.795	3.567	4.40	2.19
	128	54,651	52,313	55,370	52,877	8.968	4.467	4.50	2.01
	256	78,289	74,556	79,143	75,540	10.156	5.832	4.55	1.74
barrier2-1	32	52,877	52,326	53,560	53,349	9.797	5.244	0.39	1.87
	64	73,864	72,411	75,037	74,212	11.135	6.016	1.10	1.85
	128	102,750	100,657	104,035	101,856	12.406	6.821	2.09	1.82
	256	142,833	137,521	143,995	138,345	13.526	7.832	3.92	1.73
Hamrle3	32	35,728	35,282	36,814	36,397	21.190	8.483	1.13	2.50
	64	52,475	51,202	53,770	52,886	24.201	9.502	1.64	2.55
	128	75,818	74,038	76,851	74,919	26.802	10.934	2.51	2.45
	256	106,555	105,708	107,983	106,746	29.187	13.150	1.15	2.22
pre2	32	82,591	76,032	85,456	81,395	24.406	12.373	4.75	1.97
	64	108,714	101,718	112,486	105,741	28.484	14.123	6.00	2.02
	128	139,605	121,934	143,879	125,652	32.250	15.309	12.67	2.11
	256	177,310	139,790	183,037	143,673	35.702	16.843	21.51	2.12
cage13	32	369,330	341,229	373,617	346,002	45.887	36.711	7.39	1.25
	64	490,789	454,279	497,744	457,420	51.035	40.941	8.10	1.25
	128	643,278	585,036	647,609	590,801	55.754	44.854	8.77	1.24
	256	824,294	749,580	829,962	754,873	59.928	48.690	9.05	1.23
hood	32	22,799	22,260	24,392	23,541	15.693	4.704	3.49	3.34
	64	37,877	37,072	39,855	38,583	18.383	5.375	3.19	3.42
	128	60,039	57,183	61,087	58,525	20.983	6.137	4.19	3.42
	256	91,007	86,758	92,367	87,794	23.515	7.384	4.95	3.18
bmw3_2	32	29,861	28,113	31,129	29,922	15.383	4.773	3.88	3.22
	64	44,208	43,433	45,376	44,897	18.150	5.409	1.06	3.36
	128	65,752	64,325	67,551	66,304	20.853	6.270	1.85	3.33
	256	100,504	98,773	102,548	100,562	23.454	7.706	1.94	3.04
average	32	68,638	64,716	70,087	66,407	14.632	8.120	4.60	2.18
	64	92,689	87,397	94,536	88,969	16.654	9.128	4.82	2.18
	128	123,841	115,112	125,454	116,746	18.549	10.168	5.54	2.12
	256	163,810	151,095	165,713	152,678	20.313	11.537	6.23	1.96

Table 8.3: Performance of PaToH and K-PaToH with increasing number of K-way refinement passes

		$Cost_{\min}$		$Cost_{\text{avr}}$		$Time_{\text{avr}}$		Improvements	
Pass	K	R-P	K-P	R-P	K-P	R-P	K-P	$\%I_{\text{cost}}$	I_{time}
1	32	68,638	66,451	70,087	68,168	14.632	5.714	2.06	2.77
	64	92,689	89,587	94,536	91,239	16.654	6.202	2.37	2.89
	128	123,841	118,088	125,454	119,685	18.549	6.783	2.94	2.91
	256	163,810	154,874	165,713	156,610	20.313	7.663	3.66	2.75
2	32	68,638	65,180	70,087	66,894	14.632	6.982	4.11	2.40
	64	92,689	87,509	94,536	89,489	16.654	7.717	4.43	2.45
	128	123,841	115,750	125,454	117,305	18.549	8.553	5.07	2.41
	256	163,810	151,833	165,713	153,719	20.313	9.716	5.63	2.25
3	32	68,638	64,716	70,087	66,407	14.632	8.120	4.60	2.18
	64	92,689	87,397	94,536	88,969	16.654	9.128	4.82	2.18
	128	123,841	115,112	125,454	116,746	18.549	10.168	5.54	2.12
	256	163,810	151,095	165,713	152,678	20.313	11.537	6.23	1.96
4	32	68,638	64,750	70,087	66,383	14.632	9.038	4.70	2.05
	64	92,689	86,834	94,536	88,674	16.654	10.251	5.12	2.03
	128	123,841	115,012	125,454	116,508	18.549	11.515	5.68	1.95
	256	163,810	150,619	165,713	152,245	20.313	13.126	6.49	1.79
5	32	68,638	64,587	70,087	66,333	14.632	9.429	4.86	2.01
	64	92,689	87,293	94,536	88,731	16.654	10.776	5.21	1.98
	128	123,841	115,075	125,454	116,402	18.549	12.279	5.77	1.88
	256	163,810	150,748	165,713	152,148	20.313	14.091	6.56	1.72

The performance results of K-P, provided in Tables 8.4 and 8.5, are quite impressive. The cutsize improvement of K-P over R-P goes up to 48.84% at the 2-constraint case and up to 65.91% at the 4-constraint case. Comparison of the two tables show that increasing number of partitioning constraints favor the K-P partitioner. In general, the performance gap between K-P and R-P in reducing the cutsize decreases with increasing K. The speedups, although being slightly smaller, are close to the speedups at the single-constraint case.

8.5.4 Experiments on Partitioning with Fixed Vertices

In experiments on partitioning hypergraphs with fixed vertices, instead of using hypergraphs with synthetically generated fixed vertices, we use hypergraphs emerging in a real-life problem [23]. The properties of the hypergraphs are given in Table 8.6. In naming the datasets, the numbers after the dash indicate the number of fixed vertices in the hypergraph, e.g., there are 32 fixed vertices in the BF-32 dataset. In CC datasets, the net sizes are rather uniform, whereas, in BF and OP datasets, net sizes show high variation.

Table 8.4: Performance of PaToH and K-PaToH in partitioning hypergraphs with two partitioning constraints

		$Cost_{\min}$		$Cost_{\text{avr}}$		$Time_{\text{avr}}$		Improvements	
Dataset	K	R-P	K-P	R-P	K-P	R-P	K-P	$\%I_{\text{cost}}$	I_{time}
dawson5	32	11,294	7,634	12,598	6,761	1.418	0.623	46.33	2.28
	64	18,342	12,497	19,446	10,734	1.673	0.805	44.80	2.08
	128	28,382	20,902	30,553	17,948	1.919	1.066	41.26	1.80
	256	45,929	34,308	48,331	28,634	2.142	1.412	40.76	1.52
language	32	110,620	100,395	114,748	96,047	10.342	8.791	16.30	1.18
	64	124,426	112,789	127,849	107,467	11.024	8.827	15.94	1.25
	128	135,843	123,819	140,173	119,398	11.742	8.806	14.82	1.33
	256	149,615	138,716	154,821	130,939	12.159	9.138	15.43	1.33
Lin	32	60,912	44,495	62,727	44,629	5.060	4.114	28.85	1.23
	64	84,861	61,314	86,483	60,827	5.805	4.818	29.67	1.20
	128	114,890	82,478	117,727	80,638	6.509	5.705	31.50	1.14
	256	151,652	108,555	153,346	106,016	7.177	6.905	30.86	1.04
poisson3Db	32	47,813	40,780	50,122	39,891	8.138	5.803	20.41	1.40
	64	71,849	58,438	74,269	58,422	9.099	6.556	21.34	1.39
	128	104,590	85,151	108,143	83,930	9.964	7.374	22.39	1.35
	256	152,908	122,024	154,651	117,988	10.703	8.174	23.71	1.31
helm2d03	32	21,292	13,162	22,531	12,904	6.491	2.734	42.73	2.37
	64	30,305	20,022	32,557	19,237	7.384	3.075	40.91	2.40
	128	44,819	29,556	46,078	28,104	8.240	3.574	39.01	2.31
	256	62,859	42,878	64,195	41,033	9.046	4.378	36.08	2.07
stomach	32	34,168	26,837	35,787	26,048	6.051	2.933	27.21	2.06
	64	48,082	38,765	49,632	37,207	7.088	3.591	25.03	1.97
	128	66,512	54,766	68,199	52,877	8.115	4.443	22.47	1.83
	256	92,662	78,182	95,056	75,540	9.128	5.751	20.53	1.59
barrier2-1	32	63,376	56,223	65,498	53,349	8.711	5.324	18.55	1.64
	64	89,650	80,323	92,626	74,212	9.876	6.131	19.88	1.61
	128	125,234	111,692	127,423	101,856	10.949	6.961	20.06	1.57
	256	171,482	154,422	177,107	138,345	11.922	7.860	21.89	1.52
Hamrle3	32	49,678	37,634	54,846	36,397	19.498	8.711	33.64	2.24
	64	66,303	53,531	74,097	52,886	22.257	9.710	28.63	2.29
	128	94,701	77,481	99,669	74,919	24.763	11.178	24.83	2.22
	256	132,449	109,189	135,964	106,746	27.072	13.269	21.49	2.04
pre2	32	106,199	85,119	114,920	81,395	22.688	12.797	29.17	1.77
	64	139,973	116,133	155,620	105,741	26.474	14.592	32.05	1.81
	128	200,692	165,845	207,614	125,652	29.936	16.547	39.48	1.81
	256	270,510	214,387	280,857	143,673	33.100	18.784	48.84	1.76
cage13	32	432,428	365,497	443,298	346,002	37.214	36.594	21.95	1.02
	64	568,292	485,559	582,279	457,420	41.490	40.624	21.44	1.02
	128	736,109	631,638	746,979	590,801	45.307	44.403	20.91	1.02
	256	942,314	825,861	957,385	754,873	48.754	48.137	21.15	1.01
hood	32	30,184	24,535	32,279	23,541	14.767	4.779	27.07	3.09
	64	48,580	40,957	50,910	38,583	17.206	5.371	24.21	3.20
	128	73,857	62,461	76,913	58,525	19.544	6.197	23.91	3.15
	256	112,224	94,017	114,197	87,794	21.818	7.458	23.12	2.93
bmw3_2	32	42,905	31,633	45,457	29,922	14.068	4.865	34.18	2.89
	64	60,947	49,431	65,546	44,897	16.512	5.556	31.50	2.97
	128	94,851	74,680	101,070	66,304	18.881	6.470	34.40	2.92
	256	148,610	115,287	157,599	100,562	21.160	7.975	36.19	2.65
average	32	84,239	69,495	87,901	66,407	12.870	8.172	21.60	1.93
	64	112,634	94,147	117,610	88,969	14.657	9.138	20.87	1.93
	128	151,707	126,706	155,878	116,746	16.322	10.227	19.48	1.87
	256	202,768	169,819	207,792	152,678	17.849	11.603	19.24	1.73

Table 8.5: Performance of PaToH and K-PaToH in partitioning hypergraphs with four partitioning constraints

		$Cost_{\min}$		$Cost_{\text{avr}}$		$Time_{\text{avr}}$		Improvements	
Dataset	K	R-P	K-P	R-P	K-P	R-P	K-P	$\%I_{\text{cost}}$	I_{time}
dawson5	32	13,737	7,479	14,781	8,595	1.439	0.634	41.85	2.27
	64	19,318	12,836	22,841	14,009	1.692	0.833	38.67	2.03
	128	33,860	22,562	36,084	24,111	1.941	1.113	33.18	1.74
	256	51,794	37,858	56,280	40,004	2.161	1.482	28.92	1.46
language	32	139,353	100,125	141,895	106,155	9.580	8.628	25.19	1.11
	64	148,714	113,142	151,633	114,905	10.229	9.077	24.22	1.13
	128	156,375	126,121	163,899	128,114	10.842	9.254	21.83	1.17
	256	165,782	141,905	175,689	145,975	11.365	10.214	16.91	1.11
Lin	32	91,234	44,258	98,966	46,483	5.019	4.190	53.03	1.20
	64	120,349	62,016	125,700	63,252	5.730	4.866	49.68	1.18
	128	152,362	83,103	157,968	84,056	6.399	5.769	46.79	1.11
	256	187,114	109,672	192,952	110,519	7.031	6.964	42.72	1.01
poisson3Db	32	64,204	40,377	72,387	43,176	8.029	5.904	40.35	1.36
	64	92,385	59,990	95,745	62,535	8.965	6.698	34.69	1.34
	128	124,979	87,268	129,528	89,103	9.775	7.473	31.21	1.31
	256	170,152	124,145	175,514	126,705	10.496	8.232	27.81	1.27
helm2d03	32	24,307	13,598	27,429	14,471	6.701	2.775	47.24	2.41
	64	37,354	20,367	38,828	21,416	7.642	3.126	44.84	2.44
	128	51,410	30,162	53,462	31,025	8.541	3.632	41.97	2.35
	256	69,835	44,232	73,373	44,884	9.420	4.456	38.83	2.11
stomach	32	47,275	26,226	51,908	27,929	6.038	2.939	46.20	2.05
	64	65,598	38,703	69,666	40,270	7.063	3.598	42.20	1.96
	128	85,852	55,648	89,528	57,133	8.092	4.488	36.18	1.80
	256	115,517	80,572	118,783	81,994	9.097	5.859	30.97	1.55
barrier2-1	32	87,700	57,049	93,946	60,022	8.633	5.379	36.11	1.60
	64	113,469	80,514	121,148	84,081	9.817	6.202	30.60	1.58
	128	150,990	113,807	159,412	117,184	10.841	7.037	26.49	1.54
	256	203,583	162,242	208,792	166,821	11.864	8.004	20.10	1.48
Hamrle3	32	105,671	37,739	115,453	39,363	20.145	8.757	65.91	2.30
	64	139,438	55,051	146,122	56,319	22.900	9.774	61.46	2.34
	128	175,186	76,982	181,742	79,888	25.409	11.309	56.04	2.25
	256	216,312	110,452	222,124	111,526	27.646	13.396	49.79	2.06
pre2	32	222,989	88,477	240,992	94,414	21.903	13.034	60.82	1.68
	64	280,190	117,452	288,496	129,519	25.308	15.029	55.11	1.68
	128	333,089	176,258	349,267	197,127	28.463	17.111	43.56	1.66
	256	407,435	253,799	418,959	261,323	31.515	19.693	37.63	1.60
cage13	32	734,084	372,888	780,736	385,673	34.957	36.751	50.60	0.95
	64	881,612	496,551	928,273	508,647	38.757	40.919	45.21	0.95
	128	1,040,360	638,553	1,073,786	654,367	42.286	44.491	39.06	0.95
	256	1,222,315	832,409	1,257,893	847,588	45.385	48.550	32.62	0.93
hood	32	46,844	25,949	50,503	27,636	14.786	4.837	45.28	3.06
	64	68,600	42,308	74,043	44,073	17.212	5.394	40.48	3.19
	128	97,104	63,014	102,604	66,595	19.536	6.226	35.09	3.14
	256	140,910	97,867	145,102	100,287	21.798	7.553	30.88	2.89
bmw3_2	32	56,881	33,912	64,026	35,698	14.105	4.884	44.24	2.89
	64	83,150	51,557	89,492	54,284	16.518	5.641	39.34	2.93
	128	116,628	75,985	127,693	82,191	18.853	6.618	35.63	2.85
	256	177,088	121,710	185,200	127,319	21.093	8.167	31.25	2.58
average	32	136,190	70,673	146,085	74,135	12.611	8.226	46.40	1.91
	64	170,848	95,874	179,332	99,443	14.319	9.263	42.21	1.90
	128	209,850	129,122	218,748	134,241	15.915	10.377	37.25	1.82
	256	260,653	176,405	269,222	180,412	17.406	11.881	32.37	1.67

Table 8.6: Properties of the hypergraphs used in the experiments on partitioning hypergraphs with fixed vertices

Dataset	Number of vertices	Number of nets	Number of pins
BF-32	28,930	4,800	688,018
BF-64	28,962	9,600	930,412
BF-128	29,026	19,200	1,335,049
CC-32	56,374	4,800	1,133,858
CC-64	56,406	9,600	1,472,295
CC-128	56,470	19,200	2,094,107
OP-32	68,190	4,800	1,276,595
OP-64	68,222	9,600	1,629,169
OP-128	68,286	19,200	1,924,807

Table 8.7 illustrates the performance results obtained in partitioning hypergraphs with fixed vertices. In general, K-P shows better performance compared to R-P as the number of parts increases and the number of fixed vertices decreases. This is due the fact that the disability of R-P to recursively bisect fixed vertices between two parts becomes more apparent if the number of fixed vertices per part is high. In general, compared to R-P, the relative performance of K-P in minimizing the cutsize is better in BF and OP datasets, which are irregular in terms of the net sizes.

8.6 Conclusions

We proposed a new multi-level hypergraph partitioning algorithm based on direct K-way refinement. We also provided extensions of this algorithm for partitioning hypergraphs with multiple constraints and fixed vertices. The experiments conducted on benchmark datasets indicate that the proposed technique is rather fast and effective in optimizing the partitioning objective compared to the existing hypergraph partitioning algorithms. Especially, in the multi-constraint and fixed vertices domain, the obtained results are quite promising in terms of both execution time and solution quality. We believe the proposed work is beneficial in that it will enable better solution qualities to be found in many research problems

Table 8.7: Performance of PaToH and K-PaToH in partitioning hypergraphs with fixed vertices

		$Cost_{\min}$		$Cost_{\text{avr}}$		$Time_{\text{avr}}$		Improvements	
Dataset	K	R-P	K-P	R-P	K-P	R-P	K-P	$\%I_{\text{cost}}$	I_{time}
BF-32	32	9,474	7,504	9,639	7,596	5.394	1.971	21.20	2.74
	64	11,343	9,487	11,799	9,611	5.906	2.110	18.54	2.80
	128	14,962	12,706	15,212	12,922	6.309	2.343	15.05	2.69
BF-64	32	17,790	13,561	18,625	13,726	5.152	2.022	26.30	2.55
	64	21,473	16,462	22,010	16,895	5.726	2.245	23.24	2.55
	128	25,548	21,261	26,406	21,642	6.284	2.483	18.04	2.53
BF-128	32	34,522	24,283	35,751	24,558	5.770	2.591	31.31	2.23
	64	39,837	28,791	41,521	29,333	6.569	2.833	29.36	2.32
	128	47,448	36,129	48,652	36,539	7.006	3.141	24.90	2.23
CC-32	32	9,534	8,530	9,668	8,595	4.865	2.173	11.10	2.24
	64	12,608	10,990	12,927	11,080	5.547	2.352	14.29	2.36
	128	17,635	14,788	17,873	14,925	6.172	2.604	16.49	2.37
CC-64	32	17,466	15,384	17,952	15,503	4.623	2.516	13.64	1.84
	64	21,397	19,107	21,740	19,255	5.344	2.893	11.43	1.85
	128	28,088	24,839	28,729	25,032	6.012	3.117	12.87	1.93
CC-128	32	33,201	28,705	34,298	29,001	5.407	3.495	15.44	1.55
	64	40,036	34,959	40,677	35,282	6.233	3.960	13.26	1.57
	128	49,454	43,973	50,315	44,232	6.965	4.275	12.09	1.63
OP-32	32	8,717	6,899	8,935	7,009	18.714	6.188	21.56	3.02
	64	10,367	8,568	10,804	8,650	19.485	6.408	19.93	3.04
	128	13,155	11,197	13,463	11,292	21.275	6.672	16.13	3.19
OP-64	32	15,693	12,529	16,402	12,659	17.462	5.881	22.82	2.97
	64	18,823	14,972	19,399	15,185	19.317	6.173	21.72	3.13
	128	22,972	18,760	23,404	19,004	20.020	6.541	18.80	3.06
OP-128	32	30,418	22,551	31,076	22,688	13.119	4.991	26.99	2.63
	64	34,735	26,117	35,157	26,519	14.981	5.296	24.57	2.83
	128	39,643	31,695	40,642	32,066	16.047	5.770	21.10	2.78

formulated as a hypergraph partitioning problem.

Chapter 9

Conclusions and Future Work

Despite the vast amount of both theoretical and practical research on information retrieval, the search problem is still far from being solved. In this thesis, we aimed to put just another small brick into the wall of research on information retrieval. In particular, we proposed models and algorithms for efficient parallel text retrieval. First, we presented a model based on hypergraph partitioning for data-parallel Web crawling. The proposed model proved to be quite successful in minimizing the inter-processor communication overheads during the link exchange in data-parallel Web crawling systems. Second, we developed two different models for inverted index partitioning on shared-nothing parallel text retrieval systems. The theoretical results indicate that the proposed inverted index partitioning models are quite successful in obtaining an effective utilization of system resources during the query processing. Third, we proposed, implemented, and evaluated a high number of query processing algorithms for ranking-based text retrieval systems. Finally, we developed four software systems as a practical outcome: the Skynet parallel text retrieval system, the SE4SEE search engine, the Harbinger text classification system, and the K-PaToH hypergraph partitioning toolkit.

We currently conduct studies to evaluate the performance of the proposed Web crawling model in practice. For this purpose, we have started a large crawl of the Turkish Web space, which will form a valuable dataset to be used in our experiments. Developing models for distributed crawling architectures is among our future plans. We have also been working on different formulations for the

inverted index partitioning problem, in which the previous query logs will be utilized in order to incorporate this information into the partitioning.

Bibliography

[1] Alpert, C. J., Caldwell, A. E., Kahng, A. B., & Markov, I. L. (2000). Hypergraph partitioning with fixed vertices. *IEEE Transactions on Computer-Aided Design*, *19*(1-2), 267–272.

[2] Alpert, C. J., & Kahng, A. B. (1995). Recent directions in netlist partitioning: a survey. *VLSI Journal*, *19*(1-2), 1–81.

[3] Altingovde, I. S., & Ulusoy, O. (2004). Exploiting interclass rules for focused crawling. *IEEE Intelligent Systems*, *19*, 66–73.

[4] Androutsopulos, I., Koutsias, J., Chandrinos, K. V., & Spyropulos, C. D. (2000). An experimental comparison of naive Bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 160–167). Athens, Greece.

[5] Arasu, A., Cho, J., Garcia-Molina, H., & Raghavan, S. (2001). Searching the web. *ACM Transactions on Internet Technologies*, *1*(1), 2–43.

[6] Aykanat, C., Cambazoglu, B. B., & Ucar B. Multi-level hypergraph partitioning with multiple constraints and fixed vertices. To be submitted to *Journal of Parallel and Distributed Computing*.

[7] Aykanat, C., Pinar, A., & Catalyurek, U. V. (2004). Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal of Scientific Computing*, *25*(6), 1860–1879.

[8] Baeza-Yates, R., Castillo, C., Marin, M., & Rodriguez, A. (2005). Crawling a country: better strategies than breadth-first for web page ordering. In

Special Interest Tracks and Posters of the 14th International World Wide Web Conference. Chiba, Japan.

[9] Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern information retrieval*. New York: Addison-Wesley.

[10] Bender, M., Michel, S., Triantafillou, P., Weikum, G., & Zimmer, C. (2005). Improving collection selection with overlap awareness in P2P search engines. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 67–74). Salvador, Brazil.

[11] Bell, T. C., Moffat, A., Nevill-Manning, C. G., Witten, I. H., & Zobel, J. (1993). Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, *44*(9), 508–531.

[12] Berge, C. (1973). Graphs and hypergraphs. North-Holland Publishing Company.

[13] Bisseling, R. H., Byrka, J., Cerav-Erbas, S., Gvozdenovic, N., Lorenz, M., Pendavingh, R., Reeves, C., Roger, M., & Verhoeven, A. (2005). Partitioning a call graph. *Second International Workshop on Combinatorial Scientific Computing*. Toulouse, France.

[14] Bohannon, P., Mcllroy, P., & Rastogi, R. (2001). Main-memory index structures with fixed-size partial keys. *ACM SIGMOD Record*, *30*(2), 163–174.

[15] Boldi, P., Codenotti, B., Santini, M., & Vigna, S. (2004). Ubicrawler: a scalable fully distributed Web crawler. *Software Practice & Experience*, *34*(8), 711–726.

[16] Buckley, C., & Lewit, A. (1985). Optimizations of inverted vector searches. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 97–110). Montreal, Canada.

[17] Burns, G., Daoud, R., & Vaigl, J. (1994). LAM: an open cluster environment for MPI. In *Proceedings of the Supercomputing Symposium* (pp. 379–386). Toronto, Canada.

[18] Cambazoglu, B. B., & Aykanat, C. (2006). Performance of query processing implementations in ranking-based text retrieval systems using inverted indices. *Information Processing & Management*, *42*(4), 875–898.

[19] Cambazoglu, B. B., Turk, A., Karaca, E., Aykanat, C., Ucar, B., & Kucukyilmaz, T. (2005). SE4SEE: a grid-enabled search engine for South-East Europe. In *Proceedings of the Hypermedia and Grid Systems Conference* (pp. 223–227). Opatija, Croatia.

[20] Cambazoglu, B. B., & Aykanat, C. (2005). Harbinger machine learning toolkit manual. Technical Report, BU-CE-0502, Bilkent University, Department of Computer Engineering. Ankara, Turkey.

[21] Cambazoglu, B. B., Turk, A, & Aykanat, C. (2004). Data-parallel Web crawling models. *Lecture Notes in Computer Science*, *3280*, 801–809.

[22] Cambazoglu, B. B., & Aykanat, C. (2003). Image-space-parallel direct volume rendering on a cluster of PCs. *Lecture Notes in Computer Science*, *2869*, 457–464.

[23] Cambazoglu, B. B., & Aykanat, C. Hypergraph-partitioning-based remapping models for image-space-parallel direct volume rendering of unstructured grids. Accepted for publication in *the IEEE Transactions on Parallel and Distributed Systems*.

[24] Cambazoglu, B. B., Karaca, E., Kucukyilmaz, T., Turk, A., & Aykanat, C. Architecture of a grid-enabled search engine. Submitted to *Information Processing & Management, the Special Issue on Heterogeneous and Distributed Information Retrieval*.

[25] Cambazoglu, B. B., & Aykanat, C. Inverted index partitioning models based on hypergraph partitioning. Unpublished manuscript.

[26] Cambazoglu, B. B., Catal, A., & Aykanat, C. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. Unpublished manuscript.

[27] Cambazoglu, B. B., & Aykanat, C. Simulating parallel text retrieval systems. Unpublished manuscript.

[28] Cambazoglu, B. B., Kucukyilmaz, T., & Aykanat, C. Dialog mining: extraction of speaker and dialog attributes from text-based human conversations. Unpublished manuscript.

[29] Cambazoglu, B. B., & Aykanat, C. Automatic text categorization on Turkish text documents. Unpublished manuscript.

[30] Can, F., Altingovde, I. S., & Demir, E. (2004). Efficiency and effectiveness of query processing in cluster-based retrieval. *Information Systems*, *29*(8), 697–717.

[31] Catalyurek, U. V., & Aykanat, C. (2001). A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium* (p. 118).

[32] Catalyurek, U. V., & Aykanat, C. (2001). A hypergraph-partitioning approach for coarse-grain decomposition. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (p. 28).

[33] Catalyurek, U. V., & Aykanat, C. (1999). PaToH: partitioning tool for hypergraphs. Technical Report, Department of Computer Engineering, Bilkent University.

[34] Catalyurek, U. V., & Aykanat, C. (1999). Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, *10*(7), 673–693.

[35] Chakrabarti, S., van den Berg, M., & Dom, B. (1999). Focused crawling: A new approach to topic-specific Web resource discovery. *Computer Networks*, *31*(11-16), 1623–1640.

[36] Chakrabarti, S., Dom, B. E., Agrawal, R., & Raghavan, P. (1998). Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies. *The VLDB Journal*, *7*(3), 163–178.

[37] Chang, C., Kurc, T. M., Sussman, A., Catalyurek, U. V., & Saltz, J. H. (2001). A hypergraph-based workload partitioning strategy for parallel data aggregation. *SIAM Conference on Parallel Processing for Scientific Computing*. Portsmouth, Virginia.

[38] Chartrand, G., & Oellermann, O. R. (1993). Applied and algorithmic graph theory. McGraw-Hill.

[39] Cho, J., & Garcia-Molina, H. (2002). Parallel Crawlers. In *Proceedings of the Eleventh International World Wide Web Conference* (pp. 124–135). Honolulu, Hawaii.

[40] Cho, J., & Garcia-Molina, H. (2000). The evolution of the web and implications for an incremental crawler. In *Proceedings of the 26th International Conference on Very Large Data Bases* (pp. 200–209). Cairo, Egypt.

[41] Cho, J., Garcia-Molina, H., & Page, L. (1998). Efficient crawling through URL ordering. In *Proceedings of the 7th International World Wide Web Conference* (pp. 161–172). Brisbane, Australia.

[42] Clarke, C. L. A., Cormack, G. V., & Tudhope, E. A. (2000). Relevance ranking for one to three term queries. *Information Processing and Management*, *36*(2), 291–311.

[43] Clifton, C., Cooley, R., & Rennie, J. (2004). TopCat: data mining for topic identification in a text corpus. *IEEE Transactions on Knowledge and Data Engineering*, *16*(8), 949–964.

[44] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). Cambridge, MA: MIT Press.

[45] Croft, W. B., & Savino, P. (1988). Implementing ranking strategies using text signatures. *ACM Transactions on Office Information Systems*, *6*(1), 42–62.

[46] Dasdan, A., & Aykanat, C. (1997). Two novel multiway circuit partitioning algorithms using relaxed locking. *IEEE Transactions Computer-Aided Design of Integrated Circuits and Systems*, *16*(2), 169–178.

[47] Davis, T. (1997). University of Florida sparse matrix collection (`http://www.cise.ufl.edu/research/sparse/matrices`. *NA Digest*, *97*(23).

[48] Demir, E., Aykanat, C., & Cambazoglu, B. B. Clustering spatial networks for aggregate query processing: a hypergraph approach. Submitted to *Information Systems*.

[49] Diao, Y., Lu, H., & Wu, D. (2000). A comparative study of classification-based personal e-mail filtering. In *Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 408–419). Kyoto, Japan.

[50] Diligenti, M., Coetzee, F., Lawrence, S., Giles, C. L., & Gori, M. (2000). Focused crawling using context graphs. In *Proceedings of the 26th International Conference on Very Large Data Bases* (pp. 527–534). Cairo, Egypt.

[51] Dingle, N. J., Harrison, P. G., & Knottenbelt, W. J. (2004). Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, *64*(8), 908–920.

[52] Elmasri, R., & Navathe, S. (2003). *Fundamentals of database systems* (4th ed.). Reading, MA: Addison-Wesley.

[53] Fiduccia, C. M., & Mattheyses, R. M. (1982). A linear-time heuristic for improving network partitions. *Proceedings of the 19th ACM/IEEE Design Automation Conference* (pp. 175–181). Piscataway, NJ.

[54] Field, B. (1975). Towards automatic indexing: automatic assignment of controlled-language indexing and classification from free indexing. *Journal of Documentation*, *31*(4), 246–265.

[55] Forsyth, R. S. (1999). New directions in text categorization. In *Causal Models and Intelligent Data Management* (pp. 151–185). Heidelberg, Germany.

[56] Foster, I., & Kesselman, C. (2003). *The grid 2: Blueprint for a new computing infrastructure*. San Francisco: Morgan Kaufmann.

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

[58] Gale, W. A., Church, K. W., & Yarowsky, D. (1993). A method for disambiguating word senses in a large corpus. *Computers and Humanities*, *26*(5), 415–439.

[59] Goldman, R., Shivakumar, N., Venkatasubramanian, S., & Garcia-Molina, H. (1998). Proximity search in databases. In *Proceedings of the 24rd International Conference on Very Large Data Bases* (pp. 26–37). New York, USA.

[60] Grobelnik, M., & Mladenic, D. (1998). Efficient text categorization. In *Text Mining Workshop on ECML-98*.

[61] Han, E., Karypis, G., & Kumar, V. (2002). Text categorization using weight adjusted k-nearest neighbor classification. In *Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 53–65). Hong Kong, China.

[62] Hamill, K. A., & Zamora, A. (1980). The use of titles for automatic document classification. *Journal of the American Society for Information Science*, *33*(6), 396–402.

[63] Harman, D. W. (1986). An experimental study of factors important in document ranking. In *Proceedings of the 9th Annual International ACM*

SIGIR Conference on Research and Development in Information Retrieval (pp. 186–193). Pisa, Italy.

[64] Harman, D., & Candela, G. (1990). Retrieving records from a gigabyte of text on a multicomputer using statistical ranking. *Journal of the American Society for Information Science*, *41*(8), 581–589.

[65] Harper, D. J. (1980). Relevance feedback in document retrieval systems: An evaluation of probabilistic strategies. *Ph.D. Thesis*. The University of Cambridge.

[66] Haykin, S. (1994). Neural networks: a comprehensive foundation. Macmillan College Publishing Company Inc.

[67] Hearst, M. A. (1991). Noun homograph disambiguation using local context in large corpora. In *Proceedings of the 7th Annual Conference of the University of Waterloo Centre for the New Oxford English Dictionary* (pp. 1–22).

[68] Heydon, A., & Najork, M. (1999). Mercator: A scalable, extensible Web crawler. *World Wide Web*, *2*(4), 219–229.

[69] Holte, R. C. (1993). Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, *11*, 63–91.

[70] Horowitz, E., & Sahni, S. (1978). *Fundamentals of computer algorithms*. Potomac, MD: Computer Science Press.

[71] Hristidis, V., Gravano, L., & Papakonstantinou, Y. (2003). Efficient IR-style keyword search over relational databases. In *Proceedings of the 29th International Conference on Very Large Data Bases* (pp. 850–861). Berlin, Germany.

[72] Ilyas, F., Aref, G., & Elmagarmid, K. (2004). Supporting top-k join queries in relational databases. *The VLDB Journal – The International Journal on Very Large Data Bases*, *13*(3), 207–221.

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

[74] Kan, M.-Y. (2004). Web page categorization without the Web page. In *Proceedings of the Thirteenth International World Wide Web Conference* (pp. 262–263). New York, NY.

[75] Karypis, G., Aggarwal, R., Kumar, V., & Shekhar, S. (1999). Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration Systems*, *7*, 69–79.

[76] Karypis, G., & Kumar, V. (1998). hMETIS: a hypergraph partitioning package. Technical Report, Department of Computer Science, University of Minnesota.

[77] Karypis, G., & Kumar, V. (1999). Multilevel k-way hypergraph partitioning. In *Proceedings of the ACM/IEEE Design Automation Conference* (pp. 343-348).

[78] Kaszkiel, M., Zobel, J., & Sacks-Davis, R. (1999). Efficient passage ranking for document databases. *ACM Transactions on Information Systems*, *17*(4), 406–439.

[79] Kaya, K., & Aykanat, C. Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave environments. Accepted for publication in *IEEE Transactions on Parallel and Distributed Systems*.

[80] Kernighan, B. W., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, *49*, 291–307.

[81] Khanna, G., Vydyanathan, N., Kurc, T. M., Catalyurek, U. V., Wyckoff, P., Saltz, J., & Sadayappan, P. (2005). A hypergraph partitioning based approach for scheduling of tasks with batch-shared I/O. In *Proceedings of Cluster Computing and Grid*. Brisbane, Australia.

[82] Khoussainov, R., Zuo, X., & Kushmerick, N. (2004). Grid-enabled Weka: A toolkit for machine learning on the grid. *ERCIM News 59*.

[83] Knuth, D. (1998). *The art of computer programming: Sorting and searching* (2nd ed., vol. 3). Reading, MA: Addison-Wesley.

[84] Lam, W., Ruiz, M. E., & Srinivasan, P. (1999). Automatic text categorization and its applications to text retrieval. *IEEE Transactions on Knowledge and Data Engineering*, *11*(6), 865–879.

[85] Larkey, L. S. (1999). A patent search and classification system. In *Proceedings of the 4th ACM Conference on Digital Libraries* (pp. 179–187).

[86] Lee, D. L., Chuang, H., & Seamons, K. (1997). Document ranking and the vector-space model. *IEEE Software*, *14*(2), 67–75.

[87] Lehman, T. J., & Carey, M. J. (1986). A study of index structures for main memory database management systems. In *Proceedings of the 12nd International Conference on Very Large Data Bases* (pp. 294–303). Kyoto, Japan.

[88] Lengauer, T. (1990). Combinatorial algorithms for integrated circuit layout. Chicester: Wiley-Teubner.

[89] Lewis, D. D., & Ringuette, M. (1994). A comparison of two learning algorithms for text categorization. In *Proceedings of the Third Annual Symposium on Document Analysis and Information Retrieval* (pp. 81–93). Las Vegas, US.

[90] Lewis, D. D. (1992). Feature selection and feature extraction for text categorization. In *Proceedings of Speech and Natural Language Workshop* (pp. 212–217). San Mateo, California.

[91] Long, X., & Suel, T. (2003). Optimized query execution in large search engines with global page ordering. In *Proceedings of the 29th International Conference on Very Large Databases*. Berlin, Germany.

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

[93] McCallum A., & Nigam, K. (1998). A comparison of event models for naive bayes text classification. In *Proceedings of AAAI-98 Workshop on Learning for Text Categorization* (pp. 137–142). Madison, Wisconsin.

[94] Melnik, S., Raghavan, S., Yang, B., & Garcia-Molina, H. (2001) Building a distributed full-text index for the web. *ACM Transactions on Information Systems*, *19*(3), 217–241.

[95] Miller, R. C., & Bharat, K. (1998). SPHINX: a framework for creating personal, site-specific Web crawlers. In *Proceedings of the 7th International World Wide Web Conference* (pp. 119–130). Brisbane, Australia.

[96] Moffat, A., Zobel, J., & Sacks-Davis, R. (1994). Memory efficient ranking. *Information Processing and Management*, *30*(6), 733–744.

[97] Najork, M., & Wiener, J. L. (2001). Breadth-first crawling yields high-quality pages. In *Proceedings of the 10th international conference on World Wide Web* (pp. 114–118). Hong Kong, Hong Kong.

[98] Ng, H. T., Goh, W. B., & Low, K. L. (1997). Feature selection, perceptron learning, and a usability case study for text categorization. In *Proceedings of the 20th International Conference on Research and Development in Information Retrieval* (pp. 67–73). Philadelphia, Pennsylvania.

[99] Ozdal, M. M., & Aykanat, C. (2004). Hypergraph models and algorithms for data-pattern-based clustering. *Data Mining and Knowledge Discovery*, *9*(1), 29–57.

[100] Page, L., & Brin, S. (1998). The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the 7th International World Wide Web Conference* (pp. 107–117). Brisbane, Australia.

[101] Persin, M. (1994). Document filtering for fast ranking. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 339–348). Dublin, Ireland.

[102] Pugh, W. (1990). Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, *33*(6), 668–676.

[103] Ribeiro-Neto, B. A., & Barbosa, R. A. (1998) Query performance for tightly coupled distributed digital libraries. In *Proceedings of the Third ACM Conference on Digital Libraries* (pp. 182–190).

[104] Salton, G., & McGill, M. J. (1983). *Introduction to modern information retrieval*. New York: McGraw-Hill.

[105] Scholze, F., Haya, G. Vigen, J., & Prazak, P. (2004). Project GRACE: A grid based search tool for the global digital library. In *The 7th International Conference on Electronic Theses and Dissertations*. Lexington, KY.

[106] Schweikert, D. G., & Kernighan, B. W. (1972). A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th Workshop on Design Automation* (pp. 57–62).

[107] Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM Computing Surveys*, *34*(1), 1–47.

[108] Shekhar, S., Lu, C-T., Chawla, S., & Ravada, S. (2002). Efficient Join-index-based spatial-join processing: a clustering approach. *IEEE Transactions on Knowledge and Data Engineering*, *14*(6), 1400–1421.

[109] Shkapenyuk, V., & Suel, T. (2002). Design and implementation of a high-performance distributed Web crawler. In *Proceedings of the 18th International Conference on Data Engineering* (pp. 357–368). San Jose, CA.

[110] Smeaton, A. F., & van Rijsbergen, C. J. (1981). The nearest neighbor problem in information retrieval: An algorithm using upperbounds. In *Proceedings of the 4th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 83–87). Oakland, California.

[111] Sun, A., Lim, E. P., & Ng, W. K. (2002). Web classification using support vector machine. In *Proceedings of the 4th International Workshop on Web Information and Data Management* (pp. 96–99).

[112] Teng, S., Lu, Q., Eichstaedt, M., Ford, D., & Lehman, T. (1999). Collaborative Web crawling: Information gathering/processing over Internet. In *32nd Hawaii International Conference on System Sciences*. Maui, Hawaii.

[113] Tomasic, A., Garcia-Molina, H., & Shoens, K. (1994). Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (pp. 289–300). Minneapolis, Minnesota.

[114] Tomasic, A., & Garcia Molina, H. (1993). 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* (pp. 8–17). San Diego, CA.

[115] Trifunovic, A., & Knottenbelt, W. J. (2004). Parkway 2.0: a parallel multilevel hypergraph partitioning tool. In *Proceedings of the 18th International Symposium on Computer and Information Sciences* (pp. 789–800). Antalya, Turkey.

[116] Turk, A., Cambazoglu, B. B., Aykanat, C., & Guvenir, H. A. Machine learning techniques for personal homepage detection. Unpublished manuscript.

[117] Turk, A., Cambazoglu, B. B., & Aykanat, C. Combinatorial models for efficient parallel Web crawling. To be submitted to the *IEEE Transactions on Parallel and Distributed Systems*.

[118] Turtle, H., & Flood, J. (1995). Query evaluation: Strategies and optimizations. *Information Processing and Management*, *31*(6), 831–850.

[119] Ucar, B., & Aykanat, C. (2004). Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, *25*(6), 1837–1859.

[120] Ucar, B., & Aykanat, C. Partitioning sparse matrices for parallel preconditioned iterative methods. Submitted to *SIAM Journal on Scientific Computing*.

[121] Ucar, B., Aykanat, C., Pinar, M. C., & Malas, T. Parallel image restoration using surrogate constraints methods. Submitted to *Journal of Parallel and Distributed Computing*.

[122] Vastenhouw, B., & Bisseling, R. H. (2005). A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, *47*(1), 67–95.

[123] Wilkinson, R., Zobel, J., & Sacks-Davis, R. (1995). Similarity measures for short queries. In *Fourth Text Retrieval Conference* (pp. 277–285). Gaithersburg, Maryland.

[124] Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques* (2nd ed.). San Francisco: Morgan Kaufmann.

[125] Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing gigabytes: Compressing and indexing documents and images* (2nd ed.). San Francisco, CA: Morgan Kaufmann.

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

[127] Yang, Y. (1999). An evaluation of statistical approaches to text categorization. *Information Retrieval*, *1*(1/2), 67–88.

[128] Yang, Y., & Liu, X. (1999). A re-examination of text categorization methods. In *Proceedings of the 22th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 42–49). Berkeley, CA.

[129] Yang, Y., & Pedersen, J. O. (1997). A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning* (pp. 412–420). Nashville, Tennessee.

[130] Yavuz, T., & Guvenir, H. A. (1998). Application of k-nearest neighbor on feature projections classifier to text categorization. In *Proceedings of*

the 13th International Symposium on Computer and Information Science (pp. 135–142). Antalya, Turkey.

[131] Zeinalipour-Yazti, D., & Dikaiakos, M. D. (2002). Design and implementation of a distributed crawler and filtering processor. In *Proceedings of the Next Generation Information Technologies and Systems* (pp. 58–74). Caesarea, Israel.

[132] Zobel, J., & Moffat, A. (1995). Adding compression to a full-text retrieval system. *Software Practice and Experience*, *25*(8), 891–903.

[133] Zobel, J., Moffat, A., & Sacks-Davis, R. (1992). An efficient indexing technique for full-text database systems. In *Proceedings of the 18th International Conference on Very Large Databases* (pp. 352–362). Vancouver, Canada.

Appendix A

Screenshots of Skynet and SE4SEE

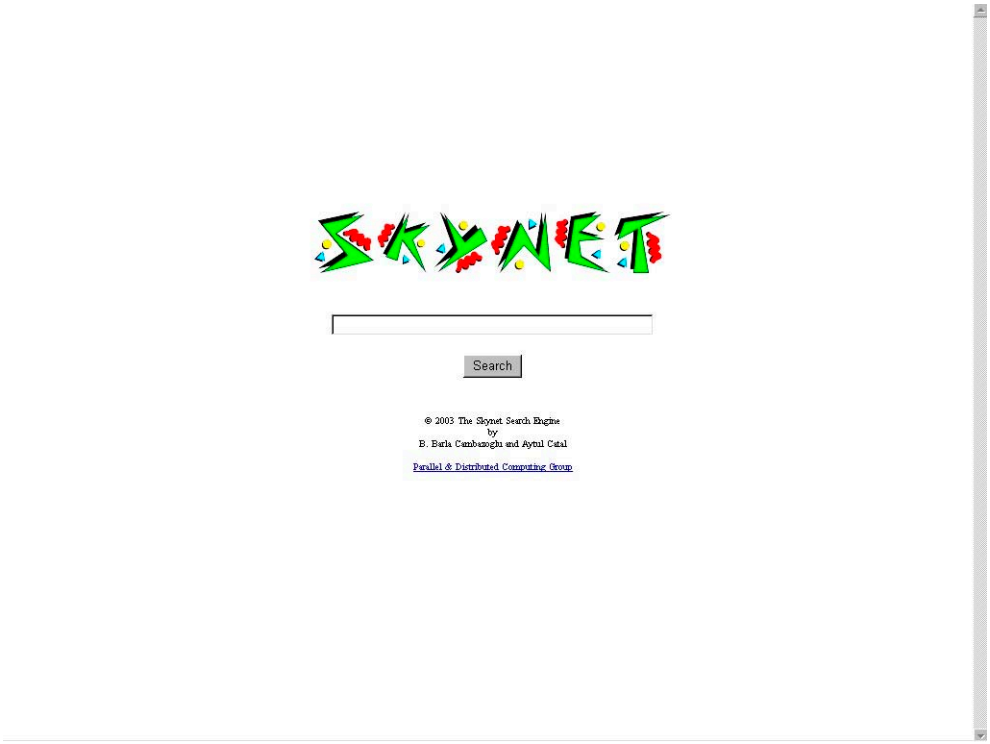


Figure A.1: Search screen of the Skynet parallel text retrieval system.

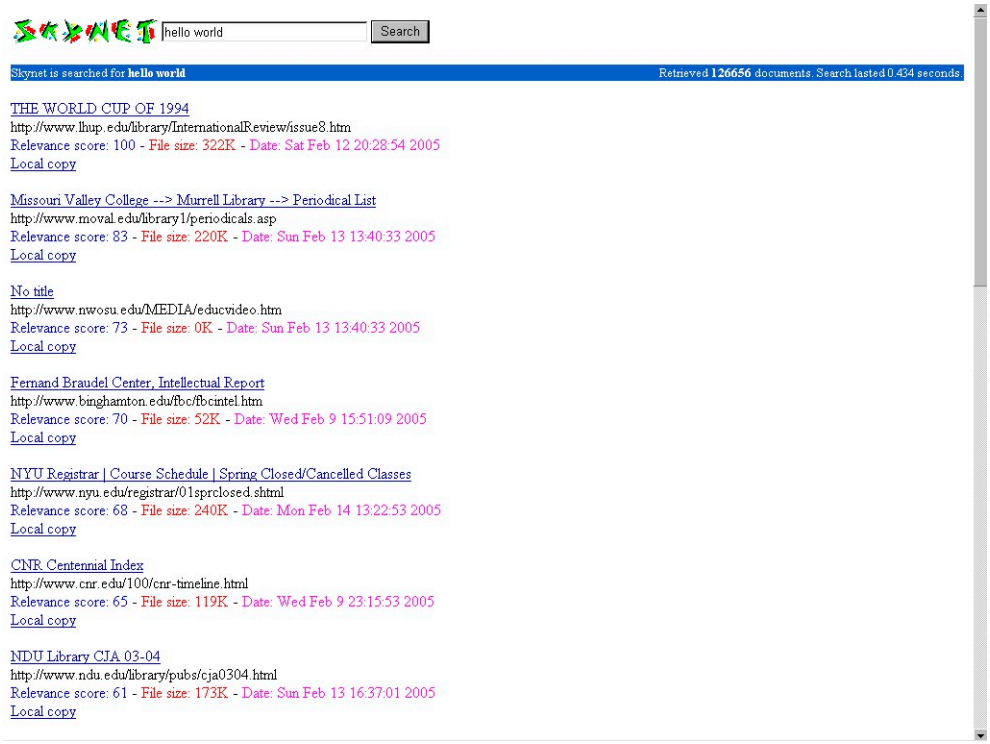


Figure A.2: Presentation of the search results in Skynet.

Figure A.3: Login screen of SE4SEE.

Figure A.4: Category-based search form in SE4SEE.

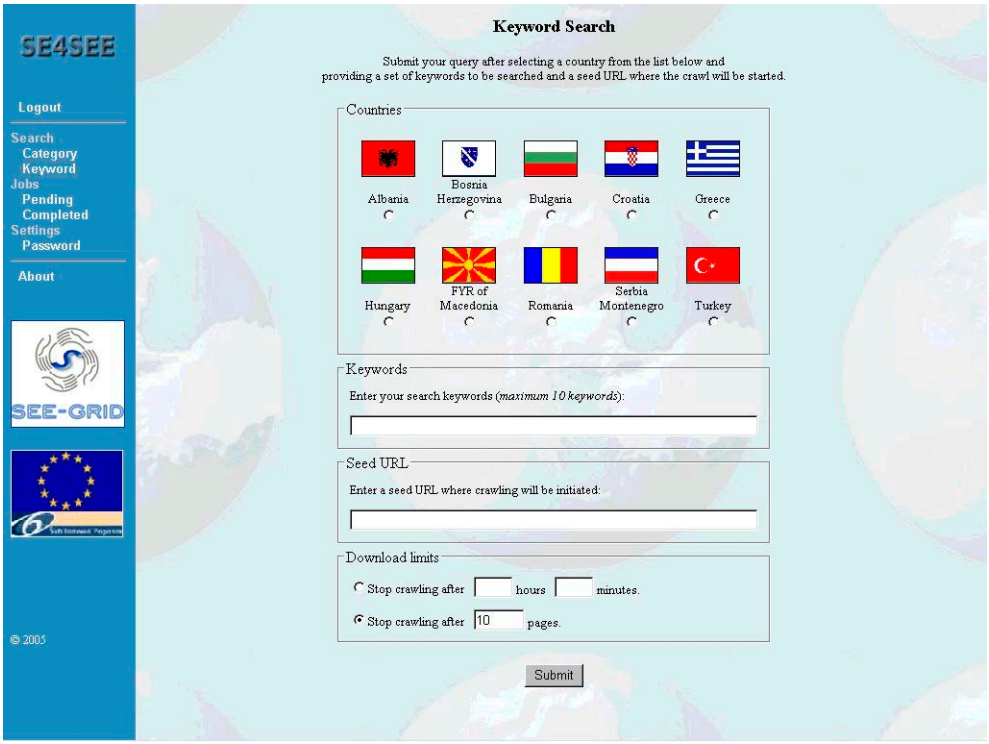


Figure A.5: Keyword-based search form in SE4SEE.

Figure A.6: Job status screen in SE4SEE.

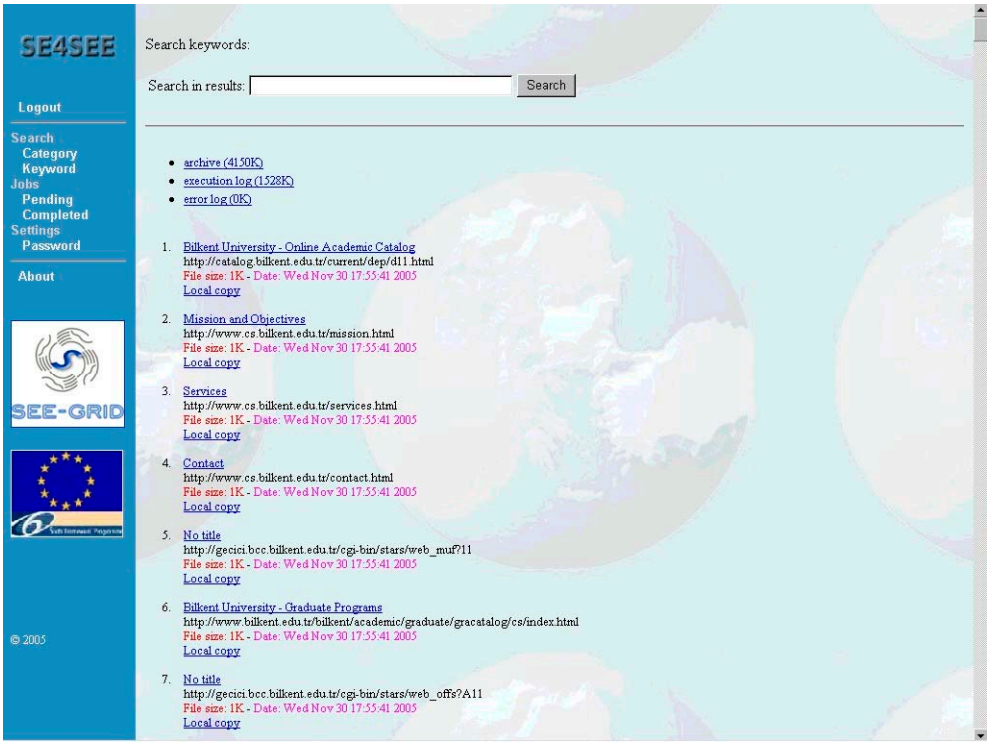


Figure A.7: Presentation of the search results in SE4SEE.

Appendix B

Harbinger Toolkit Manual

In this appendix, we provide a manual for the Harbinger machine learning toolkit (HMLT). In Section B.1, with several examples, we present the file formats utilized by HMLT. In Section B.2, we describe the installation procedure for the toolkit. A through list of the supported classifier options can be found in Section B.3. In Section B.4, the use of the wrapper module is exemplified.

B.1 Dataset Format

Throughout this discussion on the dataset format, we assume that we work on a dataset containing a total of m instances (examples), n input attributes (features), and a single output attribute (class). All classifiers expect the information about the dataset and its content to be initially distributed and stored under four separate files in the disk. These four files are pure text files, each starting with a common name, `<dataset>`, where `<dataset>` is a name representing the dataset. The file extensions for the files are fixed and are `.info`, `.insts`, `.attrs`, and `.DMR`. For example, a dataset about cancer can be stored under the files `cancer.info`, `cancer.insts`, `cancer.attrs`, and `cancer.DMR`. In all files, the lines starting with a # character are treated as comment lines and are ignored together with white spaces. All files are case-sensitive.

We describe the details of these files on an example. Assume that we have a dataset about humans. Each instance in our dataset represents a human being.

Let each human being has the input attributes skin color, age, weight, and the output attribute gender. In other words, by using the skin color, age, and weight of a human, we are trying to predict its gender. Since we have four attributes, in this particular example, $n=4$.

.insts file: `<dataset>.insts` file contains information about the labels of the instances. Each line in this file corresponds to a label identifying an instance. In our case, it contains human names:

```
# human.insts
# containing human names
# m=5

Berkant
Barla
John
Marry
Sandra
```

The use of this file is not obligatory. In the absence of the `<dataset>.insts` file, each instance is given a unique name starting from `Inst`*1* through `Inst`*m*.

.attrs file: This file keeps the labels used for the attributes and optionally the labels for the attribute values. The `<dataset>.attrs` file is also optional. If the file is not present, default attribute names `Attr`*1* through `Attr`*n* are assigned as the labels for the attributes. In our human dataset, `human.attrs` file contains something like the following:

```
# human.attrs
# containing human attributes
# n=4

eyeColor
age
weight
gender
```

In the HMLT dataset format, attributes may have three types of values: categorical, ordinal, or numeric. In our example, eye color and gender are categorical attributes, age is an ordinal attribute, and weight is a numeric attribute. We can further include this information in the `human.attrs` file as follows:

```
# human.attrs
# containing human attributes
# n=4

eyeColor C
age O
weight N
gender C
```

The letters C, O, and N indicate the attribute being categorical, ordinal, and numeric, respectively. In the absence of this information, the convention is to assume all attributes as categorical. However, even if a single numeric value is detected for an attribute, while reading the dataset content, that attribute is assumed to be of type numeric. For example, by reading the weight 54.5 for the instance Marry, the code can decide that the weight attribute is numeric.

It is possible to define aliases for categorical attribute values. This can be done by inserting `<value>:<alias>` pairs in the `<dataset>.attrs` file. This way, we can avoid repeating the same string in the `<dataset>.DMR` file and save some storage space. For example, we can use the value 0 to represent male, and 1 to represent female genders, and then define them as aliases in the `human.attrs` file. Hence, we do not repeat the strings male and female in the original data file `<dataset>.DMR`. The sample `human.attrs` file can be created like this:

```
# human.attrs
# containing human attributes
# n=4

eyeColor C 0:black 1:brown 2:green 3:blue
age O
weight N
gender C 0:male 1:female
```

.DMR and .SMR files: The attribute values are stored in the `<dataset>.DMR` file. This file contains an $m \times n$ matrix, where the rows represent the instances and the columns are the attributes. Our example `human.DMR` file is as follows:

```
# human.DMR
# containing attribute values
# mXn=5X4
```

```
1 34 67.3 0
3 48 53.2 1
0 34 78.0 0
0 78 51.2 1
2 49 55.2 1
```

For some applications, the DMR (dense matrix representation) format is not appropriate. In case the number of attributes is large and attribute values are mostly zero, using the SMR (sparse matrix representation) format and storing only the non-zero attributes may be better. Hence, as an option, it is possible to store attribute values in the SMR format in the `<dataset>.SMR` file. The above example can be stored in the `human.SMR` file as follows:

```
# human.SMR
# containing non-zero attribute values

3 1 1 2 34 3 67.3
4 1 3 2 48 3 53.2 4 1
2 2 34 3 78.0
3 2 78 3 51.2 4 1
4 1 2 2 49 3 55.2 4 1
```

In this representation, the first value at each line indicates the total number of non-zero attributes that the corresponding instance has. Note that, for the first instance, the value of the output attribute is not stored. While the data is read, it is implicitly assumed to be zero. For small datasets, the DMR format is usually the better choice and vice versa.

.info file: In the `<dataset>.info` file, some general information about the dataset is supplied. This information includes the type of the storage format used (DMR or SMR) and the total number of instances and attributes in the dataset. The `<dataset>.info` file also contains information about the partitioning of training and test instances, and selection of the attributes that will be used as input and output attributes. The sample `human.info` file is as follows:

```
# human.info

representationType DMR

totalInstanceCount 5
```

```
trainInstances 1-3 5
testInstances 4

totalAttributeCount 4
inputAttributes 1-3
outputAttribute 4
```

The tags used in this file and their meanings are as follows:

- *representationType*: The storage format used for keeping the attribute values. It can be DMR or SMR. Depending on this information, the appropriate `<dataset>.DMR` or `<dataset>.SMR` file is fetched from the disk.
- *totalInstanceCount*: Shows how many instances are expected. Instance labels beyond this count are ignored.
- *trainInstances*: Shows which instances will be used for training. "-" sign can be used to denote intervals, as in 1-3.
- *testInstances*: Shows the instances to be predicted.
- *totalAttributeCount*: Shows how many attributes are expected. Attribute labels beyond this count are ignored.
- *inputAttributes*: Shows the input attributes that will be used for prediction.
- *outputAttribute*: Shows the output attribute we are trying to predict.

Any tag other than these is accepted to be an erroneous tag. All indices in the `<dataset>.info` file start from 1. For instances and attributes, the indices beyond *totalInstanceCount* and *totalAttributeCount* are treated as errors, respectively. The output attribute need not be the last one. We can simply modify the `<dataset>.info` file to predict the eye color of a human by using its age and gender attributes as follows:

```
# human.info

representationType DMR
```

```
totalInstanceCount 5
trainInstances 1-5
testInstances 1-5

totalAttributeCount 3
inputAttributes 2 3
outputAttribute 1
```

In this example, *totalAttributeCount* is 3 since we no longer use the weight attribute. We use all instances both for training and testing purposes. No other modification is necessary in any of the remaining three files.

B.2 Installation

HMLT has been successfully installed, compiled, and executed on Linux, Unix, and Windows platforms. For Windows installation, Cygwin was used. Installation of HMLT is rather straightforward:

- Download and move `harbinger.tar.gz` file into the directory where you want to install HMLT.
- Type the following to unzip the file:

```
gunzip harbinger.tar.gz
```

- Now, extract the files by:

```
tar xvf harbinger.tar.gz
```

- This will create a directory named `Harbinger` in the current directory and extract the source files under it.
- Now, move into the `Harbinger` directory and run the installation script:

```
cd Harbinger
./install
```

- This will compile the source codes. For each classifier, there is an associated directory. Both the source codes and executables of a classifier are kept in corresponding directories. Also, a library, which is common to all classifiers is compiled under the `lib` directory. Symbolic links are created under the `bin` directory. The wrapper program is installed in the `tester` directory.
- You may need to modify some parts of the installation script depending on your system configuration (the lines at the top of the script).

B.3 Toolkit Options

In this section, we provide a list of the command line parameters that classifiers accept. Some options are common to all classifiers, whereas some are classifier-specific.

B.3.1 Options Common to All Classifiers

-h : Prints the command line options for a classifier.

-iv `<verbosity level>`: Sets instance verbosity (1:Low, 2:Medium, 3:High). If verbosity is low only the name of the instance is displayed. When medium, the class value is also displayed. If verbosity is high, the input attribute values are also displayed. But, this may be annoying if there are too many attributes in the dataset.

-cv `<verbosity level>`: Sets classifier verbosity (0:None, 1:Low, 2:Medium, 3:High). The meaning of classifier verbosity depends on the classifier used. But, in general, if verbosity is none, only the accuracy and timing information is displayed. If it is low, test instances together with predictions made for them is printed.

-M `<classification model>`: Sets classification model (1, 2, 3, 4). In the first model, all attribute values are used both for training and testing. In the second model, only the non-zero values of the test instances are used. In the third model,

only the non-zero values of the training instances are used. In the fourth model, only the non-zero values of the instances are used.

-f `<path>`: Sets the location of the dataset to be read. For example, to read the human dataset from the current directory, this option should take the parameter `./human`.

-fs `<feature selection technique>`: Sets the feature selection method (0, 1, 2) to be employed, where 0 means no feature selection, 1 means document frequency thresholding, and 2 means Chi-square method.

-fst `<feature selection threshold>`: Sets the threshold used for feature selection. The threshold can be given as a percentage of the number of features in the dataset.

B.3.2 Classifier-Specific Options

Options for the k-NN classifier:

-N `<number of neighbors>`: Sets the number of neighbors to be found.

-dm `<distance metric>`: Sets the distance metric used (c:cosine similarity, e:Euclidean distance, m:Manhattan distance).

-vm `<voting metric>`: Sets the voting metric used (m:majority voting, s:similarity voting).

Options for the k-NN-FP classifier:

-N `<number of neighbors>`: Sets the number of neighbors to be found.

-vm `<voting metric>`: Sets the voting metric used (m:majority voting, s:similarity voting).

Options for the k-means classifier:

No options.

Options for the naive Bayesian classifier:

No options.

Options for the covering rules classifier:

No options.

Options for the 1-rule classifier:

No options.

Options for the perceptron neural network classifier:

-tr: Trains the network.

-ts: Tests the network. If the -tr option is not used, the testing is performed using the initial, randomly generated weight matrix.

-lc `<learning constant>`: Sets the learning constant.

-er `<minimum error>`: Sets the minimum error before convergence. If this error is obtained over the training set, the training algorithm stops.

-ep `<maximum epoch count>`: Sets the maximum epoch count before convergence. If this epoch count is reached, the training algorithm stops.

Options for the back-propagation neural network classifier:

-tr: Trains the network. Saves the weight matrix to the disk.

-ts: Tests the network. If the -tr option is not used, the testing is performed using the weight matrix read from the disk.

-lc `<learning constant>`: Sets the learning constant.

-mc `<momentum constant>`: Sets the momentum constant.

-N `<hidden layer neuron count>`: Sets the number of hidden layer neurons.

-er `<minimum error>`: Sets the minimum error before convergence. If this error is obtained over the training set, the training algorithm stops.

-ep `<maximum epoch count>`: Sets the maximum epoch count before convergence. If this epoch count is reached, the training algorithm stops.

-tt `<test type>`: Sets the test type (r:regression, c:classification). If the test type is classification, output attribute must have categorical values. Otherwise, it must have ordinal or numeric values.

Options for the Kohonen neural network classifier:

-tr: Trains the network.

-ts: Tests the network. If the -tr option is not used, the testing is performed using the initial, randomly generated weight matrix.

-lc `<learning constant>`: Sets the learning constant.

-ep `<maximum epoch count>`: Sets the maximum epoch count before convergence. If this epoch count is reached, the training algorithm stops.

Options for the Hopfield neural network classifier:

-tr: Trains the network.

-ts: Tests the network. If the -tr option is not used, the testing is performed using the initial, randomly generated weight matrix.

B.4 The Wrapper

It is possible to run each classifier as a stand-alone application. However, HMLT also supplies a wrapper program to release the burden of modifying the `.info` file for each experiment. if the wrapper is not used, in order to perform cross-validation over a dataset, the `.info` file must be edited before each run. The wrapper offers several validation techniques and hides the details of partitioning

the instance set. The options of the wrapper are as follows:

-h : Prints the command line options for the wrapper.

-v `<verbosity level>`: Currently this option is not used.

-vt `<validation type>`: Sets the validation type (can be one of exact, cv, scv, l1o, all). If it is set to exact, the current `.info` file is used without any modification. If it is set to cv or scv, the dataset is N-fold cross validated, by partitioning the dataset into N pieces and running the classifier N times. In each run a different piece of dataset is used for testing, and the average of the results is calculated as the final result. scv is different than cv in that the instance set is shuffled before partitioning. l1o stands for leave-1-out validation. This is equivalent to m-fold cross validation. If validation type is set to all, the entire instance set is used for both training and testing.

-N `<fold count>`: Sets the fold count in cross validation.

-e `<command>`: Sets the command (i.e., the classifier) to be executed. This option must always be given as the last option to the wrapper.

The example below executes the k-NN classifier over the human dataset and performs shuffled, 10-fold cross-validation with Chi-square feature selection method, where 10% of the features are selected.

```
tester -v 3 -vt scv -N 10 -fs 2 -fst 10% -e ../knn/knn -iv 2 \
       -cv 2 -M 2 -N 5 -dm e -vm m -f ../data/human
```

Vitae

Berkant Barla Cambazoğlu graduated from Bursa Erkek Lisesi in 1990. He received his BS, MS, and PhD degrees in computer engineering from Computer Engineering Department of Bilkent University in 1997, 2000, and 2006, respectively. He has worked in two TÜBİTAK-funded research projects and an FP6 project funded by the European Union. His research interests include information retrieval, data mining, parallel computing, grid computing, and volume rendering.