

# Parallel Inverted Index for Large-Scale, Dynamic Digital Libraries

Ohm Sornil

Dissertation submitted to the Faculty of the  
Virginia Polytechnic Institute and State University  
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy  
in  
Computer Science

Edward A. Fox, Chair

Stephen H. Edwards  
Naren Ramakrishnan

C. Patrick Koelling  
Srinidhi Varadarajan

January 25, 2001  
Blacksburg, Virginia

keywords:

digital library, information retrieval, incremental update, hybrid partitioning, parallel  
inverted index, performance, simulation, terabyte text collection

Copyright 2001, Ohm Sornil

# ABSTRACT

## PARALLEL INVERTED INDEX FOR LARGE-SCALE, DYNAMIC DIGITAL LIBRARIES

JANUARY 2001

OHM SORNIL

B.Eng., KASETSART UNIVERSITY

M.S., SYRACUSE UNIVERSITY

Ph.D., VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Directed by: Professor Edward A. Fox

The dramatic increase in the amount of content available in digital forms gives rise to large-scale digital libraries, targeted to support millions of users and terabytes of data. Retrieving information from a system of this scale in an efficient manner is a challenging task due to the size of the collection as well as the index. This research deals with the design and implementation of an inverted index that supports searching for information in a large-scale digital library, implemented atop a massively parallel storage system. Inverted index partitioning is studied in a simulation environment, aiming at a terabyte of text. As a result, a high performance partitioning scheme is proposed. It combines the best qualities of the term and document partitioning approaches in a new *Hybrid Partitioning Scheme*. Simulation experiments show that this organization provides good performance over a wide range of conditions. Further, the issues of creation and incremental updates of the index are considered. A disk-based inversion algorithm and an extensible inverted index architecture are described, and experimental results with actual collections are presented. Finally, distributed algorithms to create a parallel inverted index partitioned according to the hybrid scheme are proposed, and performance is measured on a portion of the equipment that normally makes up the 100 node Virginia Tech PetaPlex <sup>TM</sup> system.

# Dedication

To my parents and the spirit of my grandfather

# Acknowledgements

First, I wish to express my sincere thanks to Dr. Edward A. Fox, my advisor, for being very patient and flexible with me, and for his continuing encouragement and guidance. Working with Professor Fox is the reason that I came to Virginia Tech.

I would like to thank my committee members: Dr. Stephen H. Edward, Dr. C. Patrick Koelling, Dr. Naren Ramakrishnan, and Dr. Srinidhi Varadarajan for their helpful comments on various aspects of the research and for serving on my committee.

My special thanks go to Dr. Paul Mather; without him I would probably never completed this thesis. Paul has always sat down calmly and discussed every issue with me.

Mr. Robert M. Askcyn set aside the PetaPlex nanoservers for my research and took great care of the machines. Dr. James D. Arthur has always been a great source of advices regarding aspects of operating systems and programming languages.

I also wish to thank my fellows in the Digital Library Research Laboratory past and present (Robert K. France, Hussein Suleman, Neill A. Kipp, Dr. Ghaleb Abdulla, Mir Farooq Ali, Marcos A. Goncalves, Omar Vasnaik, and Debra K. Dudley, to name a few), for their friendship and support. I have really had a great time working and chatting with you guys. We all make DLRL one of the best working environments.

I will never be able to express enough how grateful I am to my parents Associate Professor Banleng and Chitra Sornil for their love and support for me. They are my inspiration.

Since I was born, my grandfather Chot Sornil raised me, took care of me, and always taught me how to be a good person. All the strengths I have in me were grown by this farmer. I would like to thank my aunt Wattana Sinarkorn for her love and support.

I would like to thank Warangkana Punrattanasin for her enormous support and help during this challenging period. I would also like to thank Chonrada Rumjaunkiet, Suchada Chanprateep, and Jirada Kuntraruk for their care and incessant support. Also great friends who always are by my side, Suniti Kerndnoonwong and Jaturont Boon La-Or, to name a few. Last, by not least, I wish to thank my teachers for giving me knowledge.

The road toward my Ph.D. was long and challenging, but with these people I have really had a great time.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	2
1.1.1	Digital Libraries . . . . .	2
1.1.2	Information Retrieval . . . . .	4
1.1.3	Index Organizations . . . . .	6
1.1.4	Query Processing in Multiprocessor Systems . . . . .	10
1.2	The Problem . . . . .	12
1.2.1	System Configuration . . . . .	12
1.2.2	Inverted Index Partitioning Problem . . . . .	13
1.3	Research Contributions . . . . .	14
1.4	Outline of the Dissertation . . . . .	15
<b>2</b>	<b>Text Collection Characteristics</b>	<b>16</b>
2.1	Term-Frequency Distribution . . . . .	17
2.1.1	Term-Frequency Model . . . . .	19
2.2	Query Characteristic . . . . .	22
2.3	Query Length . . . . .	25
2.4	Conclusion . . . . .	27
<b>3</b>	<b>Inverted Index Partitioning Schemes</b>	<b>28</b>
3.1	Partitioning Schemes . . . . .	30
3.1.1	Term Partitioning . . . . .	30

3.1.2	Document Partitioning . . . . .	30
3.1.3	Hybrid Partitioning . . . . .	31
3.2	Simulation Models . . . . .	33
3.2.1	System Parameters . . . . .	35
3.2.2	Simulation Methodology . . . . .	35
3.3	Simulation Results . . . . .	36
3.3.1	Effect of Stopwords . . . . .	36
3.3.2	Load Balancing Characteristics . . . . .	37
3.3.3	Effect of Multiprogramming . . . . .	40
3.3.4	Effects of Other Parameters . . . . .	41
3.4	Conclusion . . . . .	44
<b>4</b>	<b>Inverted Index Implementation</b>	<b>48</b>
4.1	Document Inversion Algorithm . . . . .	51
4.1.1	Inverted Index Storage Manager . . . . .	54
4.1.2	Experimental Results . . . . .	59
4.2	Distributed Inversion . . . . .	65
4.2.1	Inversion Policy . . . . .	65
4.2.2	Posting Transfer Policy . . . . .	68
4.3	Distributed Inversion Algorithms . . . . .	69
4.3.1	Server-Based Inversion with Complete Parsing (SC) . . . . .	69
4.3.2	Server-Based Inversion with Partial Parsing (SP) . . . . .	70
4.3.3	Nonserver Inversion with Complete Parsing (NC) . . . . .	71
4.3.4	Nonserver Inversion with Partial Parsing (NP) . . . . .	72
4.4	Querying and Incrementally Updating Hybrid Partitioned Inverted Indices .	73
4.5	Experimental Results . . . . .	74
4.6	Conclusions . . . . .	78
<b>5</b>	<b>Conclusions and Future Work</b>	<b>79</b>

**A Similarity Between the Inverted File Partitioning Problem and the Partial  
Parallel Searchability Problem**

**89**

# List of Figures

1.1	Digital library architecture (adapted from [2]) . . . . .	3
1.2	Information retrieval process . . . . .	5
1.3	An inverted file . . . . .	7
1.4	A suffix array . . . . .	8
1.5	Superimposed coding for signature files . . . . .	9
1.6	Conceptual system configuration . . . . .	12
1.7	Physical system configuration . . . . .	13
2.1	Frequency of occurrence and document frequency distributions: (a) frequency distributions and (b) cumulative percentage distributions . . . . .	18
2.2	Term-frequency distribution of a combined collection and its parts . . . . .	19
2.3	Growths of 16 inverted lists . . . . .	20
2.4	Fitting of Zipf and Johnson SB models to the term frequency distribution . . . . .	21
2.5	Term frequency distribution model evaluation . . . . .	22
2.6	Term frequency distribution of a synthetic collection, generated using $V = 400,000$ and $n = 100,000$ . . . . .	23
2.7	Query characteristics . . . . .	24
2.8	Zipf-like probability functions . . . . .	25
2.9	Query length distribution . . . . .	26
3.1	System architecture . . . . .	29
3.2	Inverted index partitioning schemes . . . . .	31
3.3	Total time processing a batch of queries . . . . .	33



3.4	Effect of stopwords . . . . .	38
3.5	Node utilization at multiprogramming level 200 . . . . .	39
3.6	Node utilization at multiprogramming level 3000 . . . . .	40
3.7	Effect of multiprogramming levels at 80-20 and 70-30 query skew . . . . .	42
3.8	Effect of multiprogramming levels at 60-40 and uniform query skew . . . . .	43
3.9	Effect of the number of nodes in the system on throughput . . . . .	44
3.10	Effect of the number of nodes in the system on response time . . . . .	45
3.11	Effect of the query length . . . . .	46
3.12	Effect of LAN bandwidth . . . . .	47
4.1	Structure of the document buffer (adapted from [6]) . . . . .	53
4.2	Structure of the batch buffer . . . . .	54
4.3	Structure of a list file . . . . .	56
4.4	List Allocation Table (LAT) . . . . .	57
4.5	Block utilization and the average I/O operations versus the maximum block size . . . . .	60
4.6	Elapsed time of inversion at various collection sizes . . . . .	62
4.7	Performance of incremental updates . . . . .	63
4.8	Server-Based Inversion Policy . . . . .	67
4.9	Nonserver Inversion Policy . . . . .	68
4.10	Execution times of the SC, SP, NC, and NP algorithms for various sizes of the collection in each processor . . . . .	76
4.11	Execution times of parallel algorithms for various numbers of nodes for a collection of size 1 GB in each node . . . . .	77

# List of Tables

2.1	Collections used in the study of term-frequency distribution . . . . .	17
2.2	Term frequency model fitting (ranked by ExpertFit) . . . . .	19
2.3	Collections used in the study of query characteristics . . . . .	23
2.4	Query length model fitting (ranked by ExpertFit) . . . . .	26
3.1	System parameters . . . . .	35
3.2	Factors and levels in ANOVA . . . . .	36
3.3	ANOVA results of the preliminary experiment . . . . .	37
4.1	Parsing results . . . . .	61
4.2	Block utilization . . . . .	64
4.3	Cost distribution in parallel inversion algorithms . . . . .	75

# Chapter 1

## Introduction

Digital libraries (DLs) are among the most complex and advanced forms of information systems. Their early development is inspired mainly by the human need for information, the growing volume of information accumulated in various communities, and the advances in technology, such as networking, information storage and retrieval, hypermedia, and human-computer interaction [21].

The flexibility of digital technology allows DLs to handle various kinds of objects, ranging from text to hypermedia. Their contents, because of the evolution of multimedia technology, tend to grow in size at a very fast pace. The World Wide Web (WWW) illustrates the range of objects that are included in DLs. Its content is collectively huge and is still growing rapidly. Digital libraries whose contents are very large and growing rapidly include those related to national security and newsgroups. Many of these systems are at the terabyte scale. Storing and retrieving documents in collections at this scale is nontrivial and can be very slow. The PetaPlex project offers a solution to this problem by providing a massively parallel storage architecture to efficiently store and retrieve documents, and implements an abstraction of a digital library on top of this architecture.

In addition to storing and retrieving documents, another issue essential to the success of digital libraries is retrieving useful information from the system according to the user's need. This process is known as *information retrieval* (IR). With large amounts of data, identifying documents which are relevant to the need can be very inefficient. Operations in IR can be characterized by a small amount of computation applied to a large volume of data. Parallel processing is an interesting option to distribute the computation. Since IR relies more on input-output (I/O) and large amounts of secondary storage, much of the problem of partitioning the computation can be reduced to the problem of data partitioning [7].

Key to efficient IR, *indices* are mechanisms to facilitate the examination of relevancy of documents with respect to the user's need. Their organization greatly affects the performance of the retrieval process. In order to efficiently exploit parallelism in query processing, the

indices are partitioned across the processing elements of a parallel computing system. This approach is treated as our main research theme.

In addition to the scale of digital libraries, their dynamic nature also should be taken into account in IR applications. The contents of DLs (also referred to as *collections*) can be updated, i.e., documents are inserted, deleted, or modified. To reflect these changes, the indices must be adjusted, allowing users to receive up-to-date information. In early information systems, rebuilding the entire index was common because the collections were small and static. As they grow and evolve into digital libraries, the amount of data changing in the systems has been increasing. This makes frequent rebuilding infeasible both in terms of time to build (affecting the availability of the systems) and space required during the process.

This research addresses the problem of providing an index organization that allows efficient textual information retrieval in a large-scale, dynamic digital library, implemented on top of the PetaPlex architecture. The focus is on the design and implementation of an inverted file in a distributed environment, such as that underlying PetaPlex. In the next section, we provide background knowledge to introduce key concepts of our work and further reading.

## 1.1 Background

In this section, we provide background knowledge for the research. We begin with the discussion of digital libraries, followed by information retrieval, indices, and query processing in multiprocessor systems. Previous work related to other, more specific, issues addressed in this research will be discussed in the corresponding chapters.

### 1.1.1 Digital Libraries

A digital library is a collection of digital objects (repository), descriptions of those objects (metadata), a set of users, and systems that offer a variety of services such as capture, indexing, cataloging, search, browsing, retrieval, delivery, archiving, and preservation. It can be considered an advanced form of information retrieval system in that it includes more types of media, and additional functionality and services. DLs address issues across the information life cycle from creation through use, involving a wide range of disciplines from computer and information science (system architectures, services and infrastructures, performance, usability) to economics, law, and social concerns.

The design of DLs aims at providing flexible organization of information. To provide this flexibility, DLs are built from three building blocks: digital objects, handles, and repositories. See Figure 1.1. A digital object is the main building block to organize the material. It consists of content (bits), associated metadata (information about an object used to manage the object, including a handle – a type of name or identifier), and also properties, a signature,

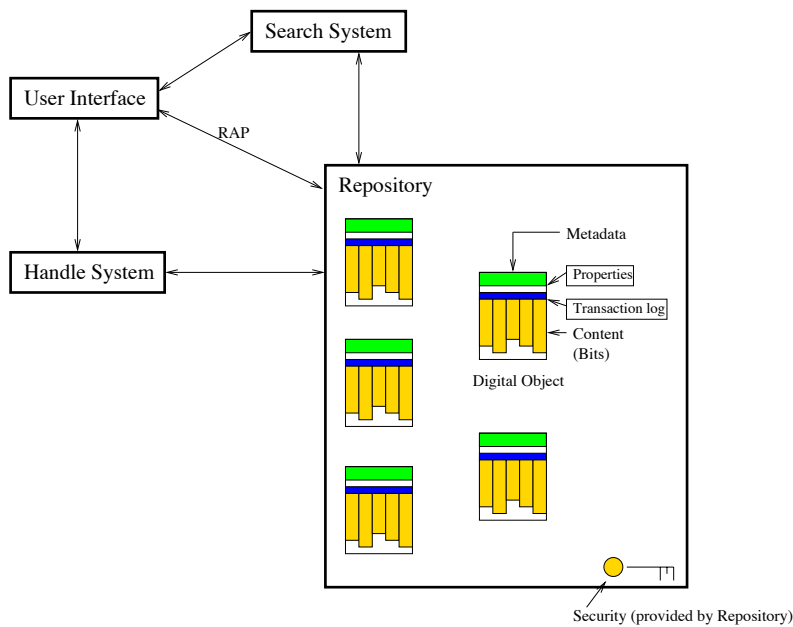


Figure 1.1: Digital library architecture (adapted from [2])

and a log of transactions that involve it.

Handles are unique identifiers of resources, such as digital objects and repositories, sometimes called “Uniform Resource Names (URN).” Handles persist for long periods of time although the resource they identify may change its form, be stored in many locations, or be moved to other locations. Significant progress has been made toward adopting a scheme of such identifiers, for example, Persistent URLs (PURLs) [67] and a standard for Digital Object Identifier (DOIs) [17]. Repositories are systems for storage and management of digital objects, providing a well-defined interface to the digital objects.

To describe DLs in operational forms that can lead to specification of protocols when various components are involved, a framework proposed by Kahn and Wilensky [31] and later revised by Arms [2] is shown in Figure 1.1, illustrating the components in the system and the relationship among these components. The system consists of four key components: user interface, repository, handle system, and search system.

A user interface is the gateway for users to interact with a DL. It allows the user to locate and manipulate digital objects contained in repositories; interprets and displays information structured as digital objects; negotiates terms and conditions; and manages relationships between digital objects.

A repository is a system (or network of systems) for storage and management of digital objects. Every interaction with the repository is through a simple protocol, known as Reposi-

tory Access Protocol (RAP). RAP provides a small set of fundamental operations, such as deposit an object in the repository and access an object. With this interface, RAP provides a clearly defined, open interface for the repository that allows others to build clients and higher level interfaces. The repository does not understand the internal structure of any digital object. The operations with the object or presentation are external to the repository. A repository of digital objects can provide security and can respond to a repository access protocol. Features of RAP are explicit recognition of rights and permissions that need to be satisfied before a user interface (UI) can access a digital object, support for a variety of disseminations of digital objects, and an open architecture with well defined interfaces.

The handle system is a computer system that provides a directory service for identifiers (handles) for resources. A search system provides retrieval functionality to the system. There can be many indices and catalogs for different kinds of information. These indices also may be independently managed and support a wide range of protocols. To the IR community a DL can be viewed as an extended IR system, in the context of federation and media variations [22]. However, the scope of DL is wider covering other issues: economic, property rights, and social implications in which IR is only part of the DL. Yet, from any perspective, information retrieval capability is an indispensable part of DLs.

### 1.1.2 Information Retrieval

Information retrieval (IR) is a core functionality of digital libraries. It deals with representation, storage, organization, and access issues related to information items [4], with the goal of retrieving information relevant to the user's *information need*, typically presented to the IR system in the form of *queries*.

A general view of the retrieval process is shown in Figure 1.2. Before an IR system is in operation, an index is built from documents in the collection. This process is called *indexing*. Users use the interface provided by the system to specify their information need, as a query. Queries are processed according to the query operation, e.g., stemmed and further transformed into the representation that can be used by the system, normally into the same representation as of the index. Then the searching process employs the index to identify a set of documents that potentially are relevant to the query. During searching, the system may give a matching score for each document. Once the set of documents has been identified and scored, they are ranked upon their relevancy to the query (using the scores computed) and presented to the user through the interface. After, the result set is returned to the users. Some systems provide ways for the user to refine the query further through relevance feedback. This dissertation addresses issues inside the dashed box in Figure 1.2.

Traditional IR systems use a set of keywords to represent the content of a document. Retrieval based on keywords is simple but raises questions regarding the accuracy of the results because of its crucial assumption that the semantics of a document can be captured by a small set of keywords [5].

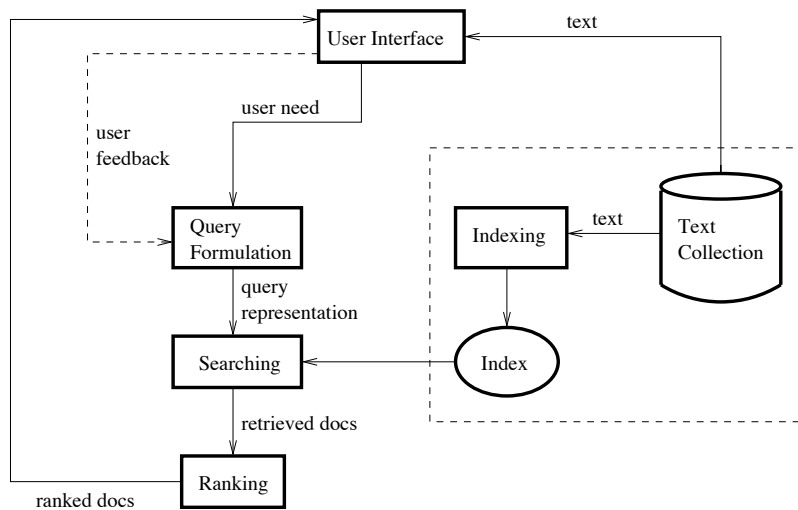


Figure 1.2: Information retrieval process

A major problem in IR systems is the issue of predicting which documents are relevant, and which are not, to the user's information need [5]. This decision usually depends upon ranking strategies intrinsic to the IR models which attempt to establish a simple ordering of the documents retrieved.

Such an IR model can be characterized by three components: the document representation, the query representation, and the ranking algorithm. As background knowledge, we discuss three classical models: the *Boolean*, *vector-space*, and *probabilistic* models. In these models, a document is represented by a set of *index terms*, normally the terms occurring in the document. Note, however, that these models actually are rather general, since concepts, citations, or other features can be represented by index terms.

The Boolean model is a simple retrieval model based upon set theory and Boolean algebra. It was adopted by many early commercial bibliographic systems. In this model, a document,  $d_i$ , is represented by a set of binary digits,  $t_j \in \{0, 1\}$ , reflecting that the term,  $t_j$ , is absent or present in a document, that is  $d_i = \{t_1, t_2, \dots, t_n\}$ . A query is represented as a Boolean expression of index terms. A document is considered relevant to the query if the representation of the document satisfies the Boolean expression. Advantages of this model are its simplicity and theoretical foundation. A major disadvantage, in some sense lying in its simplicity, is that exact matching may overlook some documents.

The vector-space model [55, 59] offers an ability to sort the retrieved documents, based upon their *degrees of similarity* to the query. The results produced usually are broader than those of the Boolean model. A document is represented as a vector of *term weights*. The dimension of the vector space is the number of terms in the *vocabulary* or *thesaurus*. A query is represented in the same way as the document. A popular term weighting scheme

is of the form  $tf \times idf$  [56, 57] where  $tf$  is a function of the number of term occurrences in the document and  $idf$  is a function (inverse) of the number of documents containing this term. Central to the model is the degree of similarity of a document, which is computed from the (query) term weights. One approach involves the *cosine similarity measure* which, as the name suggests, measures the cosine of the angle between the document and query vectors. Documents are ranked according to the similarity measure and presented to the user in decreasing order of *retrieval status value*.

The probabilistic model [53] attempts to capture the IR problem within a probabilistic framework. The algorithms typically rank the documents according to the ratio of the probability that document  $d$  is relevant to the query  $q$  over the probability that the document is not relevant to the query. That is,  $similar(d, q) = \frac{P(relevant|d)}{P(nonrelevant|d)}$ . After applying Bayes' rule, these probabilities can be estimated from the query and the collection statistics and can be improved recursively at each step of a relevance feedback process.

The three IR models discussed so far represent documents and queries by a set of index terms, and similarity between documents and queries is determined based upon the existence of query terms in the documents. This may lead to poor retrieval effectiveness because of two properties of language inherent in this type of representation. One property is that there are multiple meanings of a term in different contexts (known as *polysemy*), and the other property is that multiple terms can be used to represent a particular meaning (known as *synonymy*). These effects cause nonrelevant documents to be included and relevant documents to be excluded from the results, respectively.

*Latent semantic indexing* (LSI) [25] is regarded as a method to get around these problems. This approach represents documents and queries by a set of concepts, numerically derived from the set of index terms. The idea is to map the document-index term vector space to a lower dimensioned space using the singular value decomposition. Every new dimension in this space is termed a *concept*. Queries are transformed into the same space, and similarity between documents and queries, such as the cosine similarity, is measured in this space.

Over the years, a variety of alternative IR models have been proposed, such as the fuzzy set model [42], the extended Boolean model [58], the neural network model [76], the inference network model [73], the belief network model [50], and many more. For further information, those interested can refer to [5].

### 1.1.3 Index Organizations

In an IR system, a collection has to be searched for documents that are relevant to the query terms. This raises another major issue, i.e., the type of file structure to allow fast searching over the collection. A traditional searching technique is to scan the text sequentially to identify occurrences of the pattern. A more popular approach involves constructing index structures over the text to speed up the search. This approach incurs time and space overhead



Document 1 = Information retrieval is searching and indexing  
 Document 2 = Indexing is building an index  
 Document 3 = An inverted file is an index  
 Document 4 = Building an inverted file is indexing

Vocabulary	Inverted List (document; position)
an	(2;4), (3;1), (3;5), (4;2)
and	(1;5)
building	(2;3), (4;1)
file	(3;3), (4;4)
index	(2;5), (3;6)
indexing	(1;6), (2;1), (4;6)
information	(1;1)
inverted	(3;2), (4;3)
is	(1;3), (2;2), (3;4), (4;5)
retrieval	(1;2)
searching	(1;4)

Figure 1.3: An inverted file

in order to build, store, and maintain the index. However, the overhead is usually amortized over large numbers of searches, thus indexing is adopted in almost every IR system. Three major indexing techniques used in information retrieval systems are inverted files, suffix arrays, and signature files.

## Inverted Index

An inverted file is a term-oriented indexing mechanism. An inverted file contains an *inverted file entry* for every term in the lexicon. The entry stores a list of pointers to occurrences of that term in the collection. An inverted file entry is also known as an *inverted list*, and the pointers are referred to as *postings* [28, 5], as is illustrated in Figure 1.3. We will refer to these definitions often in this document.

An important characteristic of inverted files is the *addressing granularity* which is the accuracy to which they define the location of a term. It can range from precise positions in a document to presence in a document, document section, or other logical block. In the former approach, the space required for an inverted file is rather large. In practice, the space overhead of occurrences is 30% to 40% of the original text size [39]. For the latter approach, a two-level approach called *block addressing* reduces space requirements. That is, the text is divided into logical blocks, and a pointer in an inverted list points to a block. This can reduce the space overhead to 5% of the original text, but block scanning is necessary when the precise position of an occurrence is required.

Searching on an inverted file consists of three steps. Firstly, each term in a query is used to find its inverted file entry. Secondly, the inverted list of that term is retrieved. Thirdly, the

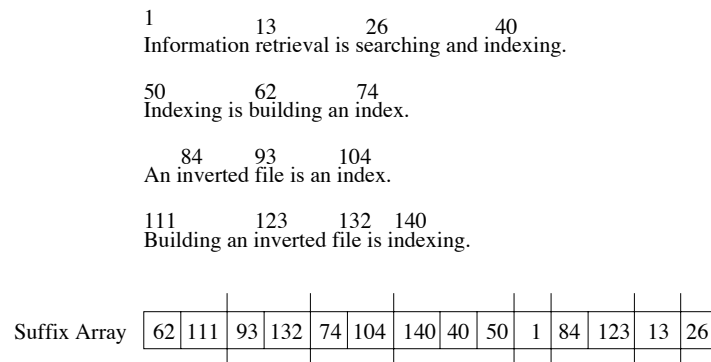


Figure 1.4: A suffix array

occurrences from the several query terms are manipulated to solve for phrases, proximity, Boolean operations, or ranking.

Practical performance achieved by inverted files illustrates that the space requirement and amount of text traversed to retrieve occurrences of a term can be close to  $O(n^{0.85})$  in a uniprocessor machine which is better than other organizations [39].

The design of an IF is a compromise between the complexity of the inversion process and the work to be done during the searching. Following similar reasoning, typically we sacrifice the resources spent during inversion to make the searching as quick as possible, especially when serving thousands of users at a time.

### Suffix Array

A suffix array is a set of pointers which are collected in ascending text order and then sorted by the text they point to. The index views the entire collection as one long string. Every position in the collection is considered as a text *suffix*, i.e., a suffix is a string that begins from that position to the end of the collection. Thus, each suffix is uniquely identified by its position. For information retrieval purpose, the index may consider only some text positions. These selected text positions are called *index points* which point to the beginning of the text positions, for example, the beginning of terms. Figure 1.4 shows an example of a suffix array. The numbers shown above of some terms indicate their starting positions in the collection. The occurrences of each term in a suffix array are sorted lexicographically by the string that follows the term, as illustrated by term “indexing” in Figure 1.4. The string that begins at position 140 is lexicographically less than the one beginning at position 40 and 50, respectively.

In order to search for a suffix  $S$ , binary search with respect to the contents of each pointer is used to find the locations of limiting patterns  $P_1$  and  $P_2$  such that  $P_1 \leq S \leq P_2$ . The result

is a sequence of terms lying between both patterns where each of them points to suffixes that start like the original pattern  $S$ . For instance, to search for term “file,” the two limiting patterns can be “file” and “flf,” and the resulting pointers point to the beginning of the third and fifth records in the suffix array. Some modifications to improve the performance of the suffix array have been proposed in the literature [3]. The positions of a term in the interval, as discussed earlier, are ordered lexicographically by the string following that term in each position, thus, some applications may require reorganization which is a costly process.

The suffix array relies on the notion of position in the text rather than term. Its applications span a wider range, for example, genetic databases and string matching, and it is natural to answer simple phrasal queries. The construction of a suffix array is expensive due to a large amount of time required to sort the pointers based on their positions as well as space required to hold the entire collection which is needed as reference points during the construction. To support ranking schemes, costly adjustments are required. In general, building suffix arrays is five to ten times slower than the construction of inverted files. For term-based applications, an inverted file outperforms a suffix array [7].

### Signature File

The signature file is the most widely used alternative to the inverted index [14]. It is a term-oriented index structure based on hashing [14, 39]. To form a signature for a document, a document is divided into logical blocks containing a constant number of distinct terms. A hash function is used to map a term to a bit-pattern (called a *term signature*) of  $F$  bits, with  $m$  bits set to 1’s, and the rest 0’s. The term signatures are *OR*’ed together to form a *block signature*, and the block signatures are concatenated to form a *document signature*. Illustrated in Figure 1.5, an example assumes that  $F = 16$  and  $m = 4$ .

indexing	0001 0000 1100 0100
is	0100 0100 0001 0010
building	0101 0011 0000 0000
an	0000 0100 0100 1100
index	1100 1000 0010 0000
Block Signature	1101 1111 1111 1110

Figure 1.5: Superimposed coding for signature files

Searching for a term is carried out by hashing the term to form its signature and examining each block signature for 1’s in those bit positions where the query term signature has a 1. The main idea is to provide a quick test to discard many of the nonqualifying documents – whenever a bit is set in the query term signature but not in a block signature, the term is not present in that block. Signature files incur low overhead (10% to 20% in size of the original text). Their search complexity is linear with a low constant. This is suitable for a moderate-size text [39].

A popular organization of signature files is the *bit-slice* format. The  $n$ th bit-slice contains the  $n$ th bits of all the signatures, stored as a sequential string. During query processing, only the slices determined by the query term signature are examined. This greatly reduces the amount of data that must be read from the signature file. This type of organization also is amenable to parallel processing [6].

With hashing, signature files allow all corresponding bits to be set even though the term is not present. For example, in a query there can be a term “signature” which has a signature 1000 1000 1000 1000, and examining this term’s signature against the block signature suggests that the term is included in this block while in reality it is not. This problem is known as a *false drop*. A major effort in the design of a signature file is ensuring that the probability of a false drop is low while keeping signatures as short as possible. In addition, editing and deleting documents indexed by signature files are costly because every slice involved with those documents has to be updated.

Signature files are suitable for the Boolean model. To support other models, costly modification must be made. Croft and Savino [12] modify signature files to support document ranking. They also find that the same performance can be achieved with inverted files. Zobel *et al.* [82] conduct a detailed comparison study of inverted files and signature files regarding various aspects. The results are conclusive that inverted files are more flexible and outperform signature files for most applications [39].

#### 1.1.4 Query Processing in Multiprocessor Systems

Information retrieval can be characterized by a small amount of computation applied to a large volume of data, relying heavily on I/O and large amounts of secondary storage. It has been seen as a natural application for parallel processing because of the obvious *data parallelism* in many major algorithms [48]. Data parallelism is the use of multiple functional units to apply the same operation simultaneously to elements of a data set [47]. Ideally, the time to complete an operation on a  $p$ -processor machine is about  $1/p$  of the time taken to complete the operation on a uniprocessor machine.

There has been a significant body of work dealing with multiprocessor systems in information retrieval. Rasmussen [48] outlines four main reasons for applying parallel computation to IR, which are: to improve response times, to search larger collections, to invent superior algorithms (hopefully providing extra retrieval effectiveness), and to reduce search cost. Because of the wide variety of architectures, parallel processing opens opportunities for research on implementation and how to make the best use of processing elements and topologies of machines, as well as new techniques that are not feasible in uniprocessor machines. The research issues range from indexing and searching techniques to new models and operations.

In terms of query processing, many indexing and searching techniques are implemented in multiprocessor systems. Pattern matching was parallelized, e.g., search considering prox-

imity or pre-computed patterns [60]. Parallel signature files were pioneered by Stanfill and Kahle [62] on the Connection Machine. An alternative method where integer values of bit positions are broadcast one by one to the processors was proposed by Pogue and Willett [45]. A bit-sliced signature file method was described by Panagopoulos and Faloutsos [43]. Frame-sliced partitioned parallel signature files were proposed by Grandi *et al.* [27]. The two-phase search technique, proposed to overcome the low effectiveness of signature files by searching the resulting list of the signature files against the query patterns to eliminate the false drops before presenting to the user, is implemented on SIMD machines [45, 44, 49] and on an MIMD transputer network [11, 10].

Other work has involved parallel inverted indices; we defer discussion to Chapter 3. All these works demonstrate the importance of data organization to the performance of the IR applications. Those interested in parallel IR technology are encouraged to refer especially to [36, 7, 48].

From the query processing perspective, the three most important classes of multiprocessor architectures consist of: *shared memory*, *shared disk*, and *shared nothing* [26].

A shared memory architecture has a single physical address space for the entire machine. That is, a processor has access to the main memory and disks of every processor. Each processor can have its own private memory as a cache buffer. The difference between the time to access data residing locally and the time to access data physically stored in other machines is not significant in most implementations. This kind of architecture has an advantage over the others in its ready adaptability to processor load-balancing. In the shared disk architecture, main memory is private for a processor, not accessible directly from outside. However, disks are shared and available for any processor through the network. The disk controllers manage the requests to prevent collisions and conflicts.

In the shared nothing architecture, every processor has its own private main memory and disks, not accessible from others. All processor to processor communications are through message-passing over the network. This type of architecture is the most commonly used with databases. It is scalable and inexpensive to build [29]. However, since every external disk access is requested through message-passing, one pays the price of communication overhead and runs the risk of network contention.

In order to support a large-scale digital library, a high-performance and high-capacity storage subsystem is required. Currently, in most cases, a hierarchy of disk arrays [9] is employed, but it has drawbacks in terms of cost/performance, availability, and scalability [37]. This calls for an alternative storage system.

The rapid drop in the cost of disks, relative to other kinds of devices, and the advance in networked commodity hardware interconnection motivate the development of parallel storage systems [65, 1]. This type of system has better cost/performance than custom-built systems, and its bandwidth is scalable while keeping cost/capacity relatively constant. Another advantage is higher mean time to failure than a single disk because redundancies

can be exploited for fault-tolerance.

Because of these advantages, the *VT-PetaPlex-1* project was initiated to investigate how to provide an abstraction of a digital library on top of a massively parallel storage system, known as *PetaPlex* [1], developed according to these trends. The architecture used as the basis for this research is a generalization of the actual architecture underlying PetaPlex, to provide more flexibility for applications, with the goal of investigating alternatives to the current approach.

## 1.2 The Problem

### 1.2.1 System Configuration

The design of a parallel algorithm is highly dependent on the system architecture and configuration. The computing environment used in this research consists of a cluster of *disk nodes* interconnected together through a high-speed network. Every disk node contains one CPU and one disk. Figure 1.6 illustrates the conceptual model of the system.

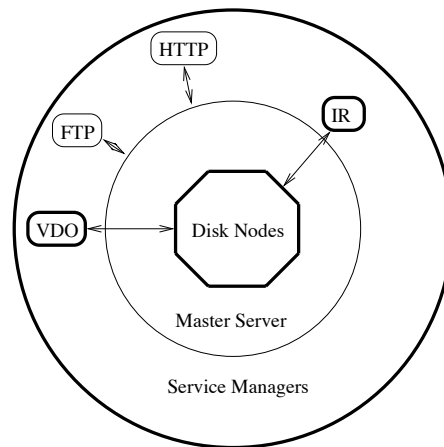


Figure 1.6: Conceptual system configuration

Content of the digital library (i.e., digital objects) is stored across disk nodes with a *master server* behaving as an interface between *application servers* and the disk nodes. It stores objects into and retrieves objects from the disk nodes in response to requests of application servers. However, some trusted application servers (such as the *IR server*) are allowed to interact directly with the disk nodes to manage their special files. Thus, the IR server, on the one hand, interacts with users and, on the other hand, with the system. It collaborates directly with disk nodes in order to store, retrieve, and manage inverted files. However, when

a user wants to look at a particular document, the IR server will request that document (object) through the master server.

In this research, we concentrate on the index layer, that is independent of the IR models and operations, which are implemented in higher layers. We focus only on performance of the index organization in query processing. The distributed environment used in this research is shown in Figure 1.7. During query processing, the disk nodes are only responsible for returning their local parts of inverted lists requested by the IR server. The computations, e.g., ranking documents, are performed at the IR server. Only during index construction do the disk nodes engage in a computation, i.e., processing their local collections and perhaps merging partial inverted lists.

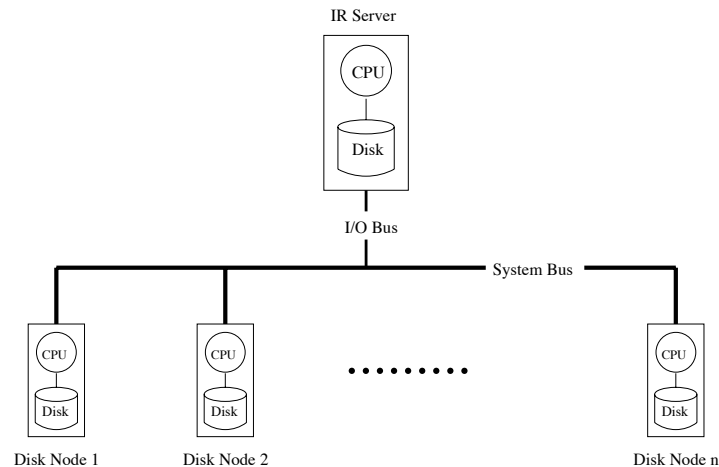


Figure 1.7: Physical system configuration

A query consists of a set of terms, i.e.,  $\{Term_1, Term_2, \dots, Term_n\}$ . Given that the inverted file is already distributed according to a partitioning scheme, once a query enters the system, the IR server extracts terms from the query and sends requests for parts of the inverted list of every term to appropriate disk nodes. Once the inverted list of every term in the query is completely received at the IR server, the query is considered complete from the index subsystem (this research) standpoint.

## 1.2.2 Inverted Index Partitioning Problem

The main issue with the design of a parallel inverted index concerns the partitioning scheme. Its function is to distribute postings of the index across the nodes in the system. In data-intensive applications like IR, effective data partitioning across multiple nodes is crucial to the performance of an IR system since partitioning critically affects the distribution of load in the system. The problem of partitioning data to each disk so that the I/O load is equal

across the disks is difficult because the data usage pattern cannot always be predicted and varies dynamically [29]. An optimization criterion for the distribution is the query response time, which should be minimized with respect to all possible queries.

Given  $N$  disk nodes, the response time to a query is defined as  $\max(T_1, T_2, \dots, T_N)$  where  $T_i$  ( $1 \leq i \leq N$ ) is the time to retrieve (partial) inverted lists from the  $i$ th node. This is because, as discussed earlier, retrieving an inverted list relies mainly on the I/O, and thus the node where the largest amount of data is retrieved determines the response time of the whole query.

#### INVERTED FILE PARTITIONING PROBLEM

The inverted file partitioning problem can be stated as follows: Given a set of postings  $P = \{p_1, p_2, \dots, p_n\}$  and a set of queries  $Q = \{q_1, q_2, \dots, q_m\}$ , find an optimal partitioning of  $P$  across  $N$  disk nodes such that the number of postings accessed to process all the queries is minimum.

Let  $P_p = \{P_{p1}, P_{p2}, \dots, P_{pN}\}$ , where  $P_{pi} \subseteq P$ , is one of the possible partitions of  $P$  into  $N$  parts. The time required to process a query  $q_j$  is

$$\max_{1 \leq i \leq N} |(P)_j \cap P_{pi}|$$

where  $(P)_j$  represents the set of postings required by the  $j$ th query. Assume that  $p_j$  is the probability of occurrence of the  $j$ th query, i.e.,  $\sum_{j=1}^m p_j = 1$ . The total time required to answer all queries in  $Q$  is

$$\sum_{j=1}^m p_j \cdot \max_{1 \leq i \leq N} |(P)_j \cap P_{pi}|$$

and the optimal partitioning scheme is the one for which this amount is minimum. This problem is equivalent to the *Partial Parallel Searchability* (PPS) problem [61] which is NP-complete, as discussed in Appendix A. Therefore, finding an optimal inverted file partitioning scheme is, from a computational point of view, at least as hard as solving an NP-complete problem. That is, no optimal inverted file partitioning can be achieved reliably in polynomial time.

### 1.3 Research Contributions

The objectives of this dissertation are: to develop an inverted index partitioning scheme suitable for large-scale digital libraries; to develop an inversion technique that supports incremental updates; and to develop distributed inversion techniques to create parallel inverted



indices, partitioned according to the scheme concluded in the first objective. In particular, the contributions include the following:

- a characterization of new standard collections that leads to the creation of simulation models of text collection and query and a design guideline for inverted indices;
- the hybrid inverted index partitioning scheme that performs well across a wide range of conditions in large-scale digital libraries;
- design and implementation of an inverted index architecture that supports incremental updates;
- development and evaluation of an incremental inversion strategy enabled by the above architecture;
- development and evaluation of distributed inversion techniques to create hybrid partitioned inverted indices.

## 1.4 Outline of the Dissertation

This chapter presents the motivation to this work, the problems addressed, and necessary background knowledge for the readers. In the remainder of this dissertation, we begin with a study of text collection characteristics (Chapter 2) whose results will be used as the basis for studying inverted index partitioning schemes and for designing the index architecture. In Chapter 3, we address the problem of partitioning an inverted index across the nodes in the system, propose a new partitioning scheme, and present results from simulation. Chapter 4 is divided into two parts; the first part focuses on uniprocessor inversion and incrementally updatable index structure. The second part extends the uniprocessor indexing algorithm to build a parallel inverted index which is partitioned according to the hybrid partitioning scheme. In both parts, results from experiments with actual collections are presented. Finally, we summarize the conclusions drawn from the research and discuss future work in Chapter 5.

## Chapter 2

# Text Collection Characteristics

In this chapter, quantitative properties contained in full-text collections are studied. The analysis will provide insight for designing partitioning schemes as well as inverted files. In addition, these characteristics will be used as the basis for simulation of hypothetical systems to study the performance of inverted index partitioning schemes in the next chapter.

The work in this part is related to the efforts to model IR systems, in the area called *Informetrics*, with its grand goal of determining general relationships among the variables of a retrieval system. One way to represent these relationships is to use computer simulation to represent both the collection and search process [64]. Tague *et al.* [63] suggests two areas that may benefit from studying text properties: (1) studies of efficient ways to store and retrieve data and (2) studies of effective ways to represent documents and queries. This research advocates the first type of study. In the first part of this research, the IR system components and their relationships are represented for simulation and the performance of different inverted index partitioning schemes are studied under a variety of conditions. In the second part, the knowledge from the analysis is used to design an efficient way to store and retrieve data.

There have been a variety of studies exploring several aspects related to modeling of IR systems, ranging from simple descriptive statistics to complex model development using term occurrence and usage distributions, up to complete system models [40]. A number of characteristics may be used to represent the functioning of an IR system. As suggested in [78], for the present study the knowledge of three main characteristics is necessary, which include the term-frequency distribution, the distribution of query terms (query selection distribution), and the query length distribution. Term-frequency distribution determines how terms are distributed in the collection; term selection distribution determines how terms are selected for use in queries with respect to their frequency of occurrence in the collection; and query length distribution provides information related to the size of user queries.

Different studies have given different representations for each of these properties. Wolfram

[78] suggests that these characteristics are areas that merit further investigation. With the results of further studies, it may be possible to determine the range of existing distributions and whether similarities exist between collections thus allowing for broader generalizations to be made. From this point of view, our contribution to Informetrics in this part is presenting the results of characterizing new collections. In addition to these three properties, the growths of term occurrences are studied. Furthermore, in this research we apply the Informetrics information to index design in Chapter 4.

## 2.1 Term-Frequency Distribution

Collection	Size (MBytes)	Number of Documents	Total Term Occurrences	Vocabulary Size (Terms)
FBIS	470	130,471	63,230,529	205,392
FR	395	55,630	33,028,747	85,288
FT	564	210,158	75,478,860	170,278
LA	475	131,896	60,491,035	169,508
COMB	1,904	528,155	232,229,171	425,353

Table 2.1: Collections used in the study of term-frequency distribution

The size of an inverted list depends on the number of occurrences of the term in the document collection. A collection of size approximately 1.9 GB (COMB) is examined in this study. It consists of four smaller collections used in the ad hoc task of the Eighth Text REtrieval Conference (TREC-8) [74] which are: Financial Times from 1991 to 1994 (FT), Foreign Broadcast Information Services from March to May of 1994 (FBIS), the LA Times from 1989 to 1990 (LA), and Federal Register of 1994 (FR). Documents in the collections are parsed without applying a stoplist. Terms are stemmed according to the Porter algorithm [18]. Proper nouns are included in the vocabulary set. Some characteristics of these collections are shown in Table 2.1.

Figure 2.1(a) shows the distributions of occurrences and document frequencies of terms in the COMB collection. Terms extracted from the collection are ordered nondecreasingly upon their frequencies of occurrence and ranked accordingly. In Figure 2.1(a), each term is plotted against its frequency of occurrence in the collection as well as the number of documents the term appears in. Figure 2.1(b) shows the corresponding cumulative percentage distributions.

Zipf [80] observed that if the terms in a document collection are ranked by decreasing number of occurrences (i.e., starting with the term that occurs most frequently), there is a constant for the collection that is approximately equal to the product of any given term's frequency and rank order number. The implication of this is that most of the terms will occur a relatively small number of times, while a few terms will occur very many times.

A similar characteristic is observed in COMB. The majority of terms rarely occur in the collection — approximately 98% of the vocabulary accounts for only 5% of the total occur-

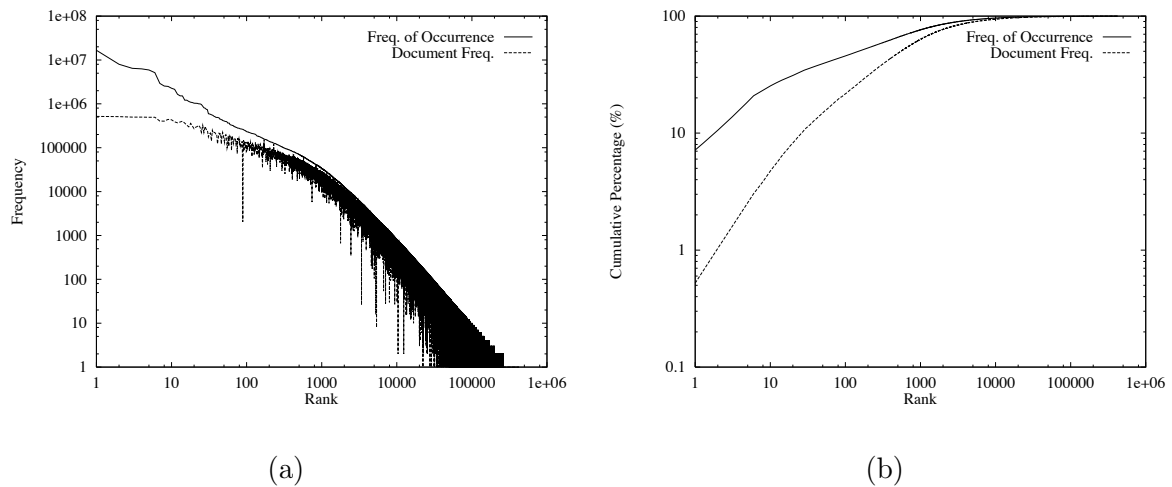


Figure 2.1: Frequency of occurrence and document frequency distributions: (a) frequency distributions and (b) cumulative percentage distributions

rences in the collection. Each term in this group has no more than 1437 occurrences. The remaining 2% of the vocabulary are used very frequently in the collection. These are terms commonly used in documents, for example, “the”, “a”, “of”, etc. The most frequently used term (“the”) occurs 16.6 million times in this 1.9 GB collection.

Figure 2.2 shows the term-frequency plots of FBIS, FT, and LA against the aggregate of all three collections. We can see that the distributions of collections which are close in size (i.e., FBIS and LA) are almost overlapping. Furthermore, each individual collection is similarly distributed as is the larger collection which it is a part of. Thus, we will use the model discussed in Section 2.1.1 to represent the collection when it scales up from its original size to larger sizes. From the document frequency characteristic, we observe that there is a complex correspondence between term frequency of occurrence and its document frequency, for example, a term with the frequency of occurrence of 100 is not likely to appear in 100 documents but is very likely to be in less than 100 documents.

To study the growth of term occurrence in a collection, occurrences of 16 terms from the LA Times collection are tracked for 365 days, as shown in Figure 2.3. The results show that term frequency of occurrence increases in stepwise fashion, which confirms the finding that there is non-trivial use of terms in documents, but the overall trend is relatively linear. One possible explanation of the stepwise behavior is related to the notion of context of a document. A document is likely to be written with respect to some contexts; in that case terms related to those contexts are used many times in the document but used rarely in documents about other contexts. Another explanation is related to the style of writing, in that, an author may have preferences on using some terms often in his writing. Overall, more frequently used terms grow with faster rates than less frequently used ones.

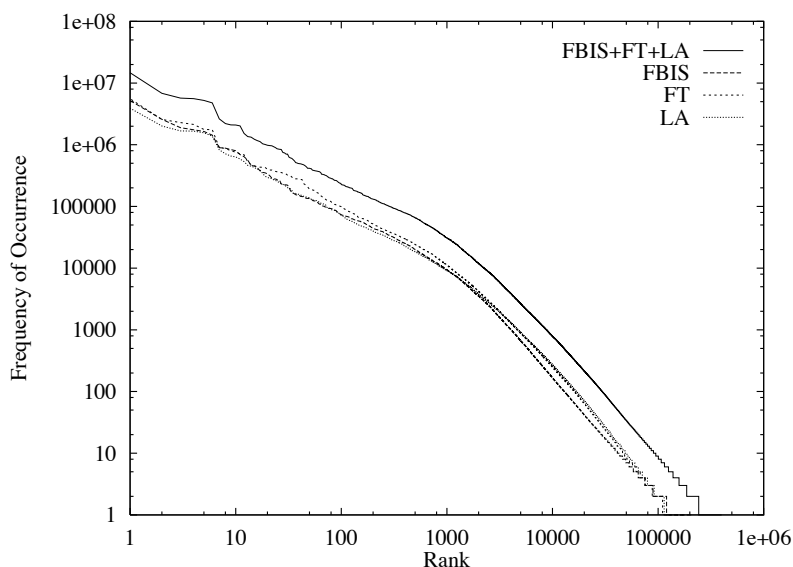


Figure 2.2: Term-frequency distribution of a combined collection and its parts

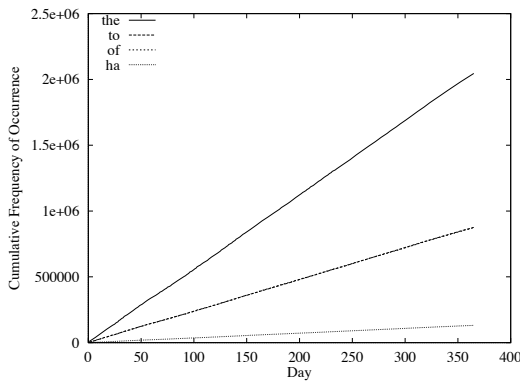
Model	Parameters	Anderson-Darling	DF
Johnson $S_B$	$a = 0, b = 425,353$ $\alpha_1 = 2.9081, \alpha_2 = 0.3534$	37.0133	4,999
Weibull	$\gamma = 0, \beta = 448.0458$ $\alpha = 0.3805$	39.2102	4,999
Gamma	$\gamma = 1, \beta = 13,704.23078$ $\alpha = 0.1819$	118.0279	4,999

Table 2.2: Term frequency model fitting (ranked by ExpertFit)

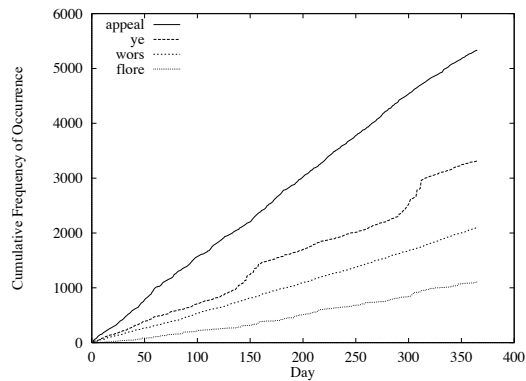
### 2.1.1 Term-Frequency Model

This distribution has been examined in many studies, reviewed in [78]. In general, the generalized Zipf's law, i.e.,  $r^a x_r = \text{constant}$ ,  $a > 0$ , where  $r$  is the rank of a term and  $x_r$  is the frequency of occurrence of the term of rank  $r$ , is the most widely used model. To construct a specific probability distribution  $Z(i)$ , that fits the rank/frequency distribution shown in Figure 2.1(a), we use Mathematica [79] to estimate the parameters of the Zipf distribution and also use ExpertFit [34] to suggest theoretical distributions and their parameters. Mathematica fits the curve according to the minimum Chi-square measure and returns the estimated parameters of the Zipf distribution, which are:  $a = 1.8485$  and  $\text{constant} = 0.551293$ .

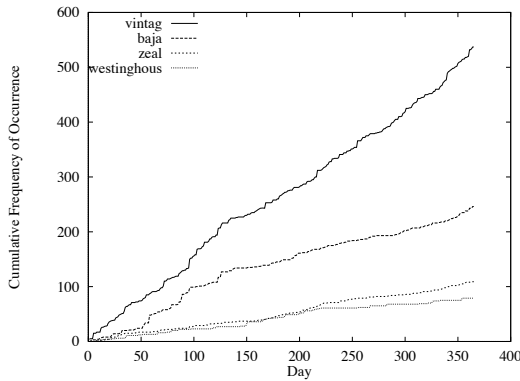
ExpertFit determines and ranks tentative representations of the data by some heuristics, as well as the Anderson-Darling test at level 0.1. The result is shown in Table 2.2. Johnson  $S_B$  is found to be the best fit model. Figure 2.4 shows the log-log graph of the fitted Zipf and Johnson  $S_B$  distributions, superimposed on the observed data. Visually examined, neither



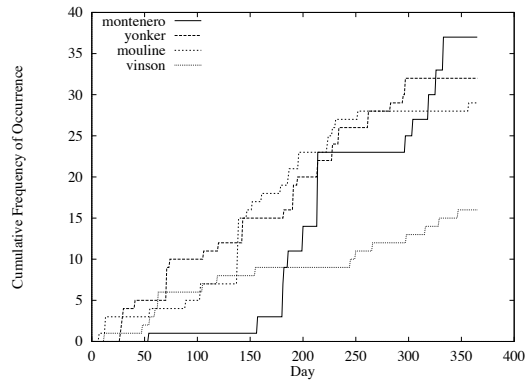
(a) The highest occurring terms



(b) The second highest occurring terms



(c) The third highest occurring terms



(d) The least occurring terms

Figure 2.3: Growths of 16 inverted lists

curve fits the data well. Thus, we use a combination of two exponential curves:

$$Z(i) = \begin{cases} e^{16.60174041 + \ln i(0.00987730 \cdot \ln i - 0.94778428)} & \text{if } i \leq 1000 \\ e^{21.93348202 - \ln i(0.01359533 \cdot \ln i + 1.55040832)} & \text{otherwise} \end{cases}$$

as fitted by Mathematica, to represent the data, as shown in Figure 2.5. We can see that this curve matches the actual distribution better and is regarded as the basis for the scalable term-frequency distribution that will be used further in this research.

The term-frequency distribution can be used as a basis to generate simple synthetic text collections. The relationship among its parameters can be expressed as follows:

Let

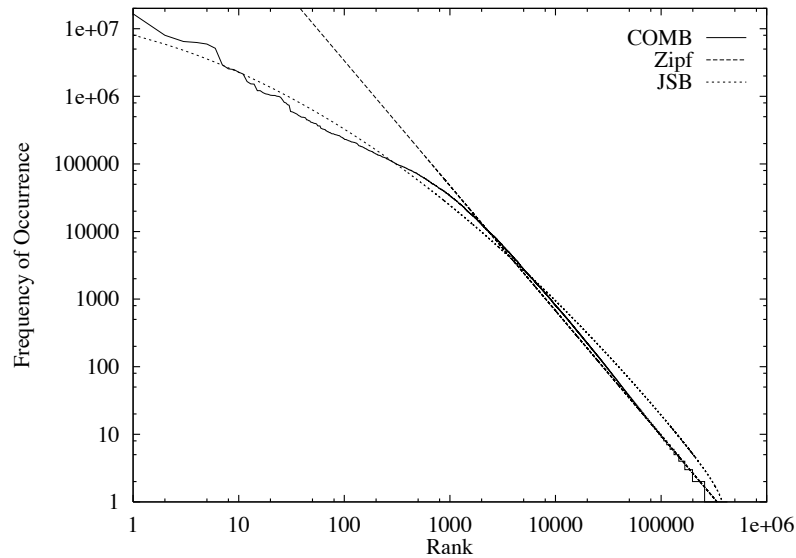


Figure 2.4: Fitting of Zipf and Johnson SB models to the term frequency distribution

- $W$  the total number of possible terms  
 $V$  the actual vocabulary size of the generated collection ( $V \in W$ )  
 $z(i)$  prob[term of rank  $i$  is selected] (the model)  
 $C$  the size of collection generated (bytes)  
 $D$  the number of documents generated  
 $T$  the size of a document (terms)  
 $t$  the size of a term (bytes)  
 $n$  the number of terms we need to generate  
 $N(i)$  the number of times that term  $i$  appears in  $n$  trials

That is,

$$V = \sum_{i=1}^W I(i) \quad \text{where} \quad I(i) = \begin{cases} 1 & \text{if } N(i) > 0 \\ 0 & \text{if } N(i) = 0 \end{cases}$$

and  $n = \lceil \frac{C}{t} \rceil$  and  $D = \lceil \frac{n}{T} \rceil$ .

The vocabulary size of a generated collection,  $E(V)$ , can be estimated by:

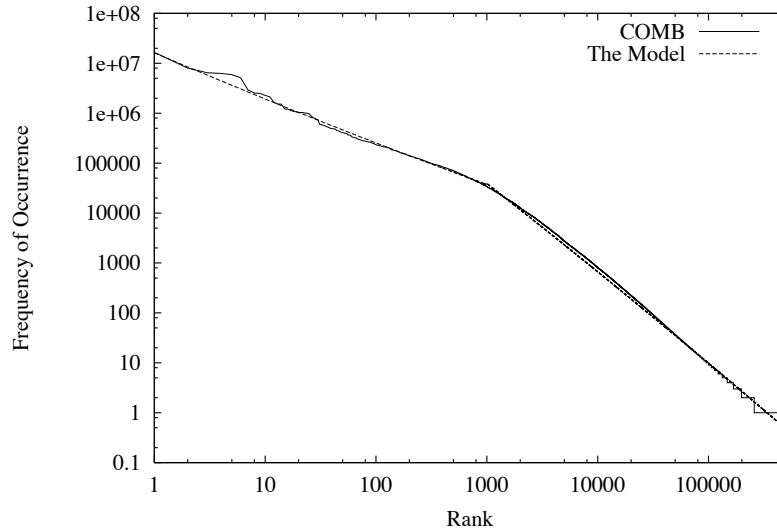


Figure 2.5: Term frequency distribution model evaluation

$$\begin{aligned}
 E(V) &= E(\sum_{i=1}^W I(i)) \\
 &= \sum_{i=1}^W E(I(i)) \\
 &= \sum_{i=1}^W P(N(i) > 0) \\
 &= \sum_{i=1}^W [1 - P(N(i) = 0)] \\
 &= \sum_{i=1}^W [1 - (1 - z(i))^n] \\
 &= W - \sum_{i=1}^W (1 - z(i))^n
 \end{aligned}$$

The rank/frequency distribution of a sample generated collection is illustrated in Figure 2.6. This collection is generated from using  $W = 400,000$  and  $n = 100,000$ . The estimated size of vocabulary in this synthetic collection is 9,919.33 terms, and the observed vocabulary size is 9,931 terms.

## 2.2 Query Characteristic

The relationship between frequency of term occurrence and frequency of term use in queries plays an important role in determining the probability of a term being used relative to the frequency of occurrence of the term which is highly skewed. The performance of indices will be highly dependent on the type of this relationship. The role of this characteristic will be shown clearly in Chapter 3.

A difficulty with studying query characteristics is to find collections with large query sets. The construction of a query set with a complete result set for reference is difficult especially



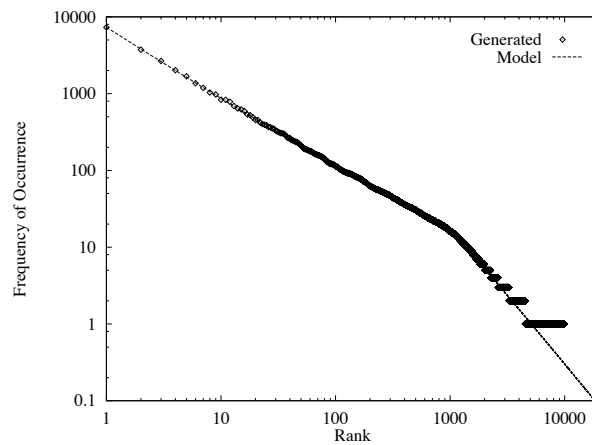


Figure 2.6: Term frequency distribution of a synthetic collection, generated using  $V = 400,000$  and  $n = 100,000$

regarding the process of constructing coherent result sets. The main purpose of information retrieval test collections is to evaluate the accuracy of returned documents which does not require large query sets. For example, the ad hoc task in TREC-8 program evaluates IR systems using a 1.9 GB collection with only 50 queries.

In this study, three information retrieval test collections of different sizes and their query sets are studied which are:

- COMB: It is the aggregate of four collections, an approximately 1.9 GB collection, discussed above.
- Cystic Fibrosis Database (CF): It contains 1239 documents published from 1974 to 1979 discussing Cystic Fibrosis aspects.
- Cranfield: It is an aeronautics collection used in many early retrieval studies.

Collection	Size	Number of Queries	Vocabulary Size (terms)	Number of Terms in Query Set
COMB	1.9 GB	50	425,353	343
CF	4.81 MB	100	17724	288
Cranfield	1.65 MB	365	4820	730

Table 2.3: Collections used in the study of query characteristics

Some characteristics of these collections are shown in Table 2.3.

The resulting distributions are shown in Figure 2.7. The cumulative data reveals that in CF, terms that occur most frequently in the collection are used in the query set most frequently,

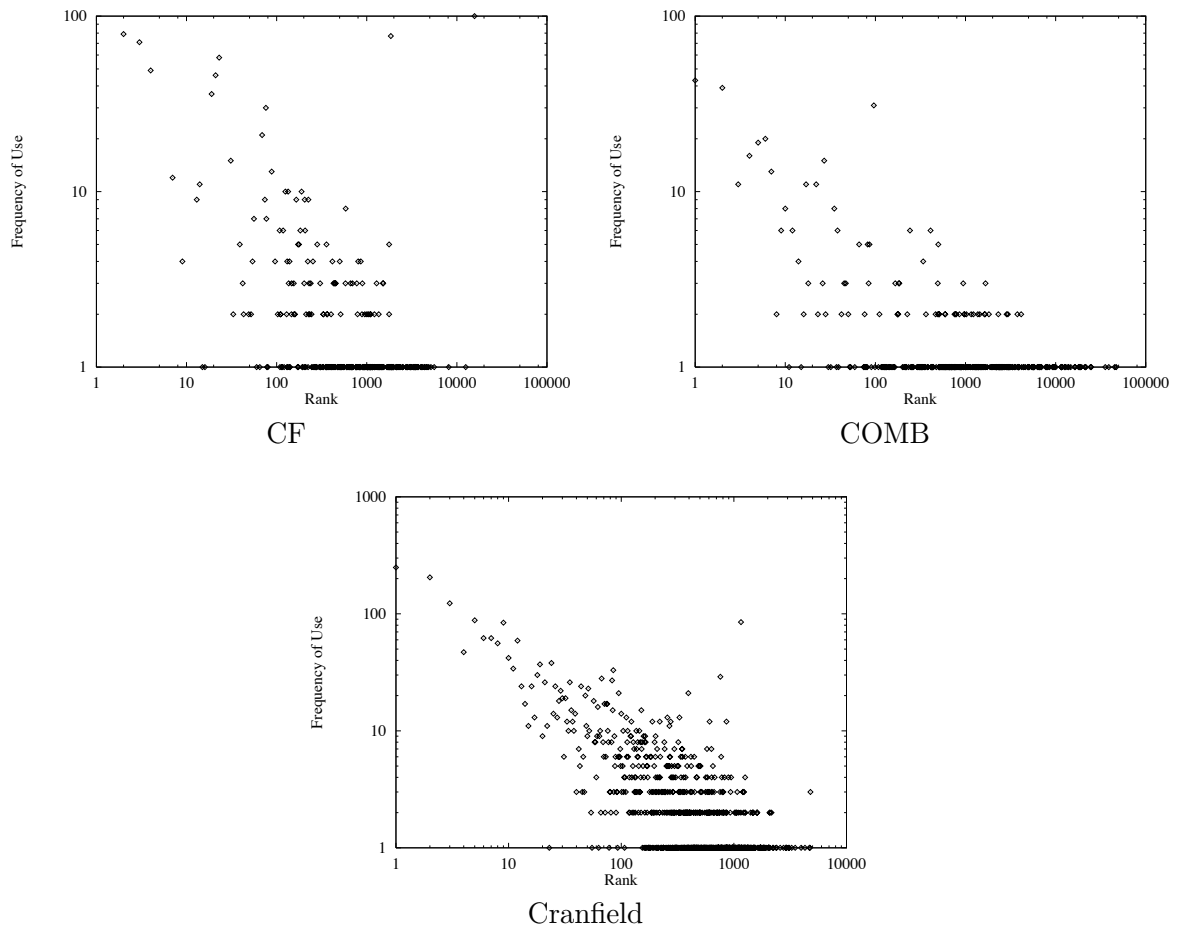


Figure 2.7: Query characteristics

followed by mid-frequency and then low-frequency terms. In COMB, a similar pattern is observed, but mid-frequency terms are used relatively more frequently than those in CF. In Cranfield, terms with high frequency are used most frequently, terms that occur few times are used with mid-range frequency, and terms that occur with mid-frequency are, on average, used least frequently.

Our results show different patterns of term usage. Several studies discover different relationships, e.g., uniform, linear, mid-frequency terms used the most, or no relationship at all. This distribution reflects the behavior of users. Salton [56] hypothesizes that the mid-frequency terms should be used for the best retrieval results because of having the greatest discrimination power. Nelson [41] observes a significant correlation with high frequency terms being chosen most frequently. Tague [63] finds that probability of a term occurring in a query is proportional to that term's frequency in the database as a whole.

There are many types of this relationship conceivable in actual information systems. An observed distribution may not be representative for all systems. However, in our study high frequency terms are generally used most frequently in queries. Thus, in our experiments we assume that the relationship follows a Zipf-like function with probability mass function,  $q(i)$  [32]:

$$q(i) = c/i^{1-\theta}$$

where  $V$  is the vocabulary size,  $c = 1/\sum_{j=1}^V(1/j^{1-\theta})$ ,  $1 \leq i \leq V$ , and  $\theta = \log(\text{Fraction of query terms}) / \log(\text{Fraction of vocabulary})$

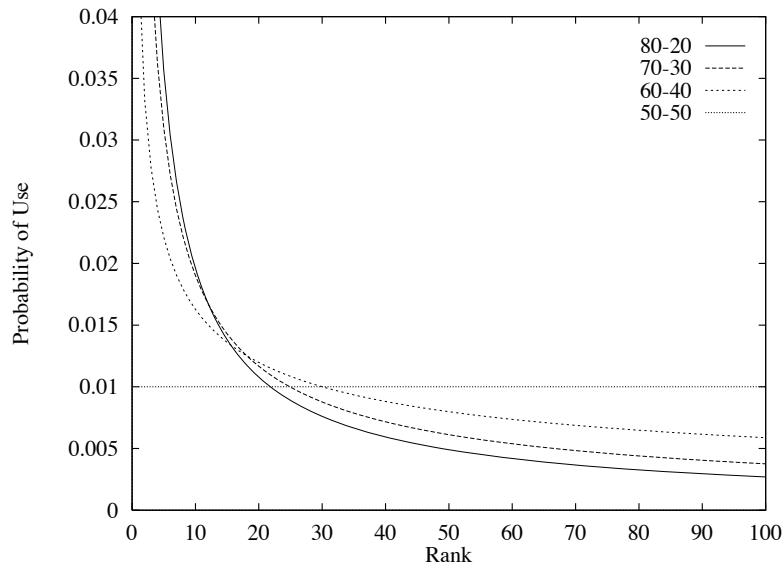


Figure 2.8: Zipf-like probability functions

For example, the  $\theta$  value for the 80-20 degree of data skew is  $\theta = \log(0.8)/\log(0.2)$  which means that 80% of the terms in a query will be selected from the top 20% of the vocabulary. Varying the value of  $\theta$  from 1 to 0, the probabilities vary from a uniform situation to the pure Zipf distribution. Figure 2.8 shows the effect of varying the values of  $\theta$ .

## 2.3 Query Length

In order to study the distribution of query length, actual queries are required. Most information retrieval test collections provide queries in forms of description which in turn are

translated into user queries by searchers. Thus descriptions cannot be used automatically to study query length.

A transaction log of the MARIAN system [19, 8], a digital library and information retrieval system developed at Virginia Tech, collected from January 26 to February 14, 1999 is used in this study. The transaction consists of 33,042 queries ranging from 1 to 18 terms in length. MARIAN queries are likely to be longer than Web queries, since the query windows are very large for MARIAN applications. This should lead to conservative estimates in the rest of this study regarding performance, which should strengthen conclusions reached.

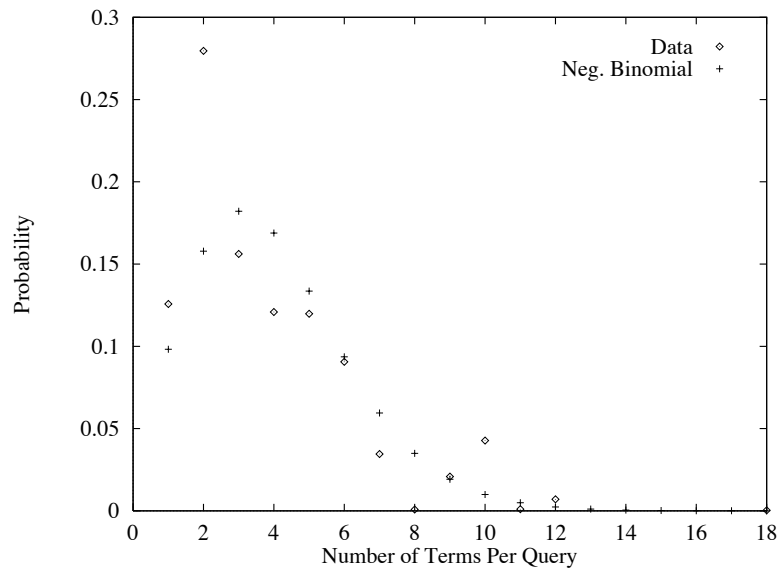


Figure 2.9: Query length distribution

Model	Parameters	Chi-square	DF
Negative Binomial	$p = 0.7527, s = 12$	6,263.8596	17
Poisson	$\lambda = 3.9418$	99,740.825	17
Geometric	$p = 0.2024$	8,185.4646	17

Table 2.4: Query length model fitting (ranked by ExpertFit)

The observed data has the sample mean of 3.74, median of 3, mode of 2, and standard deviation of 2.40. ExpertFit is used to suggest and rank sample distributions for the data. The result, shown in Table 2.4, suggests that the Negative Binomial distribution best fits the data although the test statistic is large, compared to the critical values of 18.252 and 30.578 at 0.25 and 0.01 levels of significance, respectively. It indicates that the distribution does not perfectly represent the data. See Figure 2.9. In his study, Wolfram [78] predetermines from the collected data to model the query length with selected Poisson-like distributions. His results also show that the Negative Binomial distribution provides the best fit to his data

although it does not perfectly fit based on the test statistic. In the same light, the Negative Binomial will be used as the model of query length when needed.

## 2.4 Conclusion

This study has attempted to develop models of IR system collection characteristics to be used in the simulation in the next chapter. A two-part polynomial curve is employed to represent the term-frequency distribution. Complex relationships are observed for the term selection distribution. Thus a Zipf-like function will be used as its representation.

It is found that the term frequency of occurrence spans a wide range, from one to millions of occurrences in the collection. Although the vast majority (98%) of the terms occur very few times, the remaining portion of terms in the vocabulary account for 95% of the total occurrences. The effect of this skewness during query processing is aggravated when high-frequency terms also are frequently used in queries. This calls for storage organization approaches for the index that treat terms differently based on how frequently they occur in the collection and how frequently they are used in queries.

During incremental updates, the inverted lists of more commonly used terms expand more quickly. The index has to take care of not only how to efficiently store the inverted lists for retrieval purpose but also how to quickly reorganize its structure to incorporate new postings with different rates of expansion. We will return to all these issues in subsequent chapters as we explore, design, simulate, and implement parallel inverted file systems.

## Chapter 3

# Inverted Index Partitioning Schemes

A network of workstations has been an attractive approach to meet high I/O demand in data intensive applications, such as information retrieval. The main issue with the design of a parallel inverted index concerns the partitioning scheme. Its function is to distribute postings of the inverted index across the nodes in the system. Effective data partitioning across multiple nodes is crucial to the performance of an IR system since the partitioning critically affects the distribution of load in the system. In this chapter, we propose a new inverted index partitioning scheme and examine the performance of this scheme relative to the document and the term partitioning methods on a shared nothing multiprocessor system.

The data representation in inverted indices reveals two main methods for partitioning the data: *document partitioning* and *term partitioning* [7, 29]. In document partitioning, an inverted file is decomposed into sub-files according to the physical location of the documents. That is, all term occurrences in a document are placed on a single node. In term partitioning, an inverted file is partitioned into a disjoint set of inverted lists. All postings of a term, regardless of their physical locations, are stored on a single node.

A query term in the document partitioning scheme is processed in parallel on every node (*intraquery parallelism*). On the other hand, in the term partitioning scheme only the node holding the inverted list of that term processes it. Thus, the system can process multiple queries in parallel (*interquery parallelism*).

On systems where searching can be carried out at the processing node, document partitioning allows searching to be performed locally at each node and produces globally consistent document scores that can be merged directly at brokers, i.e., designated nodes merging results from various nodes together (assuming that the global term statistics are available to a node, at the cost of requiring a more expensive indexing process). On the other hand, with the term partitioning scheme each node returns its local results to the brokers. The brokers then combine and compute globally consistent document scores. Both of these two methods have been studied in various environments; we discuss some of them next.

Jeong and Omiecinski [29] study the document and term partitioning schemes in a multiple disk, shared everything environment. Every node performs Boolean search against local documents and returns the result set to the master process. They find that query processing performance depends on term distributions. When the queries tend to have only high or low frequency terms (*skewed*), the document partitioning scheme performs better. When the distribution is uniform, the term partitioning scheme performs better.

Tomasic and Garcia-Molina [68, 70] explore the effects of four allocation schemes in a distributed shared nothing environment, assuming multiple disks attached to a single I/O bus and multiple I/O buses attached to a single CPU. They classify the schemes as: host, disk, I/O bus, and system organizations. The disk and system organizations are equivalent to the document and term partitioning, respectively. In the host organization, documents are distributed to CPUs according to the document partitioning scheme, and the inverted file within each CPU is distributed according to the term partitioning. The I/O bus organization follows the same distribution principle as the host organization, except it begins by partitioning documents with respect to I/O buses instead of CPUs. The processing is the same as in [29]. That is, every node involving in query processing performs Boolean search locally and returns only matched documents to the original node where the query arrives. The results with a full-text IR system show that the host organization performs the best. In another work [69], they conduct the same study with a database of abstracts. The results are different from the full-text case. The system (i.e., term partitioning) organization is found to perform the best over a wide range of parameter values, although it utilizes the network heavily.

Recently, Ribeiro-Neto *et al.* [51] study the performance of document and term partitioning in a shared nothing environment, implemented on a high bandwidth ATM network of high end workstations. A difference from other works is that, instead of the Boolean model, the vector space model is used for ranking which also is done at processing nodes, and the higher overhead of the document ranking process also is included in the response time. The results indicate that term partitioning outperforms document partitioning in this environment.

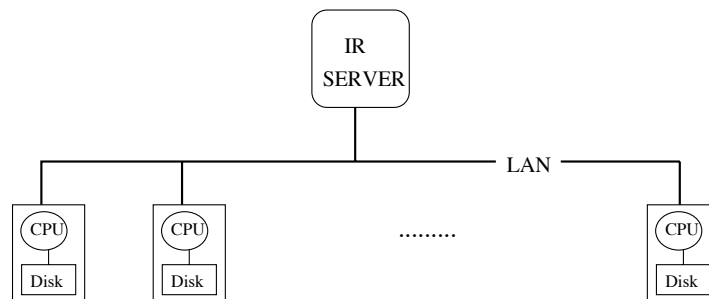


Figure 3.1: System architecture

The system architecture is shown in Figure 3.1. It consists of a number of (processing) nodes

and the IR server connected through an interconnection network. Each node contains one disk and one CPU. The interconnection network is a 100BaseT switched Ethernet.

Given that the inverted file is already distributed across the nodes according to a partitioning scheme, once a query enters the system, the IR server extracts query terms from the query and sends requests for the inverted list of every term to appropriate nodes. Each node fetches the partial list requested from its local disk and returns it to the server. The returning of the list to the IR server is carried out in a nonblocking fashion. That is, the node can go back to serve another request, without waiting for the transfer to complete. Thus, the communication cost is overlapped with the cost of retrieving lists. After the list of every term in the query is completely received at the server, the query processing is completed from the index system standpoint.

The differences among research works in this topic are mainly from the variety of possible system architectures and configurations which impose requirements and constraints on the processing. In the environment of this work, we deal only with returning the entire inverted lists to the IR server, regardless of the partitioning scheme being used. Further processes, e.g., searching and ranking, can be carried out inside the IR server. The network of processing nodes can be regarded as “one big disk” supplying inverted lists to the IR server. The second difference is that this research focuses on much larger collections and much higher workloads. The last feature that makes this work unique is using a richer set of parameters in the study (five major parameters in this work).

## 3.1 Partitioning Schemes

We explore three partitioning schemes, each illustrated for the data in Figure 3.2(a).

### 3.1.1 Term Partitioning

In the term partitioning approach, the inverted list of a term regardless of its length and physical locations of documents is entirely stored in one node. Figure 3.2(b) shows an example of this scheme. Suppose that term  $a$  is assigned to Node 1,  $b$  to Node 2, and so on, then every posting of term  $a$  regardless of its original location (i.e., in position 1 and 3 of document  $d_i$ ) is stored in Node 1.

### 3.1.2 Document Partitioning

In the document partitioning approach, an inverted file is decomposed into sub-files according to physical locations of the documents. That is, all term occurrences in a document are placed in a single node. An example of this scheme is shown in Figure 3.2(c). Suppose that



**Given:**

4 Disks  
 Collection (4 docs):  
 d1: <a, b, a, c, b>  
 d2: <a, d, e, a>  
 d3: <b, c, a, b>  
 d4: <b>

(a)

**Term Partitioning**

Node 1: a = (d1;1),(d1;3),(d2;1),(d2;4),(d3;3)  
 Node 2: b = (d1;2),(d1;5),(d3;1),(d3;4),(d4;1)  
 Node 3: c = (d1;4),(d3;2)  
 Node 4: d = (d2;2) e = (d2;3)

(b)

**Document Partitioning**

Node 1: a = (d1;1),(d1;3)  
 b = (d1;2),(d1;5)  
 c = (d1;4)  
 Node 2: a = (d2;1),(d2;4)  
 d = (d2;2)  
 e = (d2;3)  
 Node 3: a = (d3;3) c=(d3;2)  
 b = (d3;1),(d3;4)  
 Node 4: b = (d4;1)

(c)

**Hybrid Partitioning**

Assume: Chunk Size = 4 postings

Node 1: a = (d1;1),(d1;3),(d2;1),(d2;4)

Node 2: b = (d1;2),(d1;5),(d3;1),(d3;4)

Node 3: a = (d3;3) c = (d1;4),(d3;2)

Node 4: b = (d4;1)

d = (d2;2)

e = (d2;3)

(d)

Figure 3.2: Inverted index partitioning schemes

document 1 is stored in Node 1, document 2 is in Node 2, etc., then postings from documents stored in Node 1 are stored only in Node 1.

### 3.1.3 Hybrid Partitioning

The document and term partitioning schemes use two radically different approaches. The document partitioning scheme balances the load by using as many nodes as possible to process a query term. The term partitioning scheme takes another route by not trying to balance the load even in highly skewed environments.

In the hybrid scheme, we divide an inverted list into a number of *chunks*. Chunks are equal in size, except for the last chunk which can be less full. They are allocated to different nodes. The hybrid partitioning scheme takes a moderate approach to balancing load among the nodes in the system but employs a larger unit than that of the document partitioning. It is a middle-ground between these two extremes, so deserves its name “Hybrid.”

As shown in Figure 3.2(d), term *a* has 5 postings. Suppose that the chunk size is 4 postings, then the inverted list of term *a* is broken into two chunks of size 4 and 1, respectively. Each chunk is distributed pseudo-randomly to a node in the system. Term *c* has an inverted list of length 2 which is less than the chunk size, so the entire list is stored as a chunk.

The chunk size can be anywhere from 1 posting to even greater than the largest inverted list in the collection. If the chunk size is 1, the hybrid scheme imitates the document partitioning

scheme, and if the chunk size is at least equal to the biggest inverted list size, the hybrid scheme imitates the term partitioning scheme.

### Hybrid Partitioning Scheme's Chunk Size Selection

We propose a simple heuristic chunk size selection scheme for the hybrid partitioning approach based on the time to complete processing of a batch of queries. Since the returning of the list is overlapped with the retrieval operation, the communication can be ignored from the model, as is confirmed by our initial investigation.

The model consists of five major parameters: term-frequency distribution  $Z(i)$  (implicitly including collection size and stoplist), query selection distribution  $q(i)$ , chunk size  $c$ , number of query terms  $T$ , and number of nodes in the system  $N$ . The average number of chunks required from a node to retrieve an average term,  $C$ , is:

$$C = \frac{\left\lceil \frac{\sum_{i=1}^V q(i) \cdot Z(i)}{c} \right\rceil}{N}$$

where  $i$  is the rank of a term and  $V$  is the vocabulary size.

If processing an average query term involves no more than  $N$  nodes ( $C \leq 1$ ), at least one term can be processed at the same time at multiple nodes. If the processing involves

more than  $N$  nodes ( $C > 1$ ), only one term is processed at every node for  $\left\lceil \frac{\left\lceil \frac{\sum_{i=1}^V q(i) \cdot Z(i)}{c} \right\rceil}{N} \right\rceil$

chunks and, during fetching the last  $\left( \left\lceil \frac{\sum_{i=1}^V q(i) \cdot Z(i)}{c} \right\rceil - N \cdot \left\lceil \frac{\left\lceil \frac{\sum_{i=1}^V q(i) \cdot Z(i)}{c} \right\rceil}{N} \right\rceil \right)$  chunks of a

term,  $1 / \left( C - \left\lceil \frac{\left\lceil \frac{\sum_{i=1}^V q(i) \cdot Z(i)}{c} \right\rceil}{N} \right\rceil \right)$  terms can be processed concurrently across the system.

The number of nodes involved in retrieving an average list,  $U$ , is:

$$U = \begin{cases} \sum_{i=1}^V q(i) \cdot N & \text{if } \left\lceil \frac{Z(i)}{c} \right\rceil \geq N \\ \sum_{i=1}^V q(i) \cdot \left\lceil \frac{Z(i)}{c} \right\rceil & \text{otherwise} \end{cases}$$

Thus, the total amount of time (in milliseconds) required for completely processing a batch of  $T$  query terms entering the system at the same time can be computed by:

$$Total\ time = T \cdot c \cdot C \cdot (Size_{posting}/BW_{disk}) + \frac{Overhead_{disk} \cdot T}{N/U}$$

where  $Size_{posting}$ ,  $Overhead_{disk}$ , and  $BW_{disk}$  represent the size of a posting, the disk overhead, and the disk bandwidth, respectively.

This equation is used to compute the total time of a batch of queries ranging in size from 100 to 9000 terms against the results from 5 random replications of simulation. Figure 3.3 shows the results. The example case shown here is specified as follows: collection size is 200 GB, query skew is 80-20, stoplist consists of 512 terms, system has 100 nodes, and the chunk size is 16384 postings. We can see that, although there are small variations among experimental results, due to the fact that different replications use different query sets during the simulation, the analytical and experimental results correlate very well.

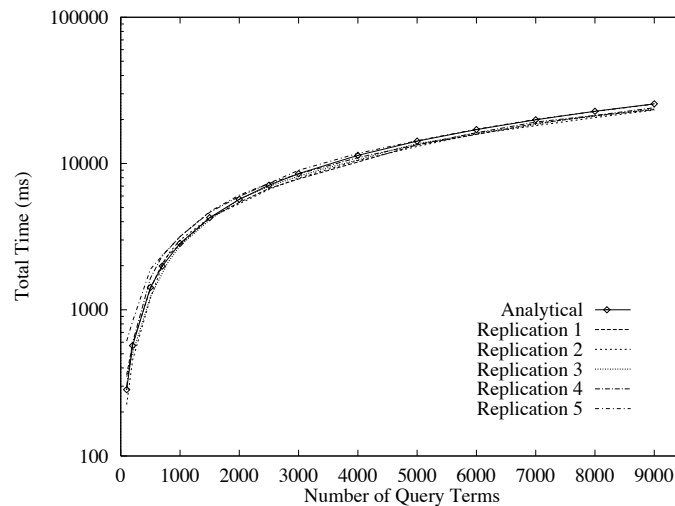


Figure 3.3: Total time processing a batch of queries

## 3.2 Simulation Models

In order to simulate results we model collections, query term selection, disk nodes, and the network.

### COLLECTION MODEL

The collection model is the rank-frequency of occurrence distribution constructed in Chapter 2, from the collection used in the ad hoc task of TREC-8, which is:

$$Z(i) = \begin{cases} \frac{|C|}{1.9GB} \cdot e^{16.60174041 + \ln i(0.00987730 \cdot \ln i - 0.94778428)}, & \text{if } stoplist < i \leq \max(stoplist, 1000) \\ \frac{|C|}{1.9GB} \cdot e^{21.93348202 - \ln i(0.01359533 \cdot \ln i + 1.55040832)}, & \text{if } \max(stoplist, 1000) < i \leq V \\ 0 & \text{otherwise} \end{cases}$$

where  $Z(i)$  is the frequency of occurrence of the term of rank  $i$ ,  $|C|$  is the collection size, and  $V$  is the vocabulary size.

#### QUERY SELECTION MODEL

The query selection model  $q(i)$  is the probability that the term of rank  $i$  is used in a query. We model the query selection based on a Zipf-like probability distribution function, proposed by Knuth [32], as follows:

$$q(i) = c/i^{1-\theta}$$

where  $c = 1/\sum_{i=1}^V (1/i^{1-\theta})$ ,  $1 \leq i \leq V$ ,

and  $\theta = \log(\text{Fraction of query terms}) / \log(\text{Fraction of vocabulary})$

Our experiments use four different  $\theta$  values, i.e.,  $\theta = 0.1386, 0.2962, 0.5575$ , and  $1.0$  which represent four different levels of data skew: 80-20, 70-30, 60-40, and 50-50 (uniform), respectively.

#### DISK NODE MODEL

A node fetches the partial inverted list upon request from the IR server and sends the list back to the server. Each request for term  $i$  has a length of  $ListLength(i)$ . The node is an FCFS with unlimited capacity queue. The node service time for a request is determined by:

$$Overhead_{disk} + ListLength(i) \cdot Size_{posting} / BW_{disk}$$

#### NETWORK MODEL

The network is a switched Ethernet. Every communication between the IR server and a particular node has a full dedicated bandwidth. A communication channel is modeled as an unlimited capacity FCFS queue. The cost of delivering a partial list of term  $i$  of size  $ListLength(i)$  is determined by:

$$Overhead_{network} + ListLength(i) \cdot Size_{posting} / BW_{network}$$

### 3.2.1 System Parameters

Parameters of the system are listed in Table 3.1. In the simulation, a posting (a tuple  $\langle docID, location \rangle$ ) is assumed to be of size 8 bytes. A (partial) inverted list stored in a disk node is compressed with 50% compression rate. After the query is parsed at the IR server, requests are sent to every node involved, assuming that a request takes 0.1 ms from the server to reach a disk node; this cost includes the cost of parsing the query. Network overhead includes all protocol startup times and packet assembly costs. The overhead of the disk includes seek time and rotational delay.

Parameter	Value
Posting size	8 bytes
Compression rate	50%
Request delay	0.1 ms
Network (Overhead / BW)	0.1 ms / 100 Mbps
Disk (Overhead / BW)	11 ms / 5662 bytes per ms

Table 3.1: System parameters

### 3.2.2 Simulation Methodology

The simulation is written in YACSIM [30]. A simulation run proceeds for 500,000 milliseconds. Results in this paper are generated from using the system parameter values in Table 3.1. For each different condition, the chunk size of the hybrid partitioning is determined by using the value that gives the smallest time from the selection scheme, presented earlier.

It is desirable to remove aspects of the simulation that are system dependent. The main emphasis of the present study is comparative, so removal of operations common to all systems simplifies the process to be modeled. Since the focus of this research is on the partitioning scheme and since the only task of the IR server with this setup other than parsing the query is copying data from kernel buffer to user buffer, an assumption of this research is that the IR server is modeled by an unlimited capacity queue with zero service time.

For each run, the program proceeds to generate query terms, represented by ranks which are generated as random variates according to the query model by the inverse transform method [33]. The length of the inverted list of each term is then determined by the term-frequency model.

To distribute postings according to the hybrid partitioning scheme, the following distribution function is employed to randomly allocate the corresponding chunks of an inverted list to disk nodes.

$$destination = (termID \text{ XOR } chunkID) \text{ MOD } totalDiskNodes$$

where termID and chunkID are 32-bit integers.

### 3.3 Simulation Results

In this section, we study the effects of some key parameters on the performance of the inverted index partitioning schemes. Note that results in this paper, except when studying the effect of query length, assume a query consists of 4 terms. Multiprogramming level refers to the number of queries concurrently processed in the system at a time. Initial examination of throughput from simulation in each cell of the initial ANOVA run (under combinations of the parameters as shown in Table 3.2) with five replications reveals widely ranging variances. Such differences could affect the outcome of the ANOVA runs; a natural logarithmic transformation was performed on all the observed data prior to the analysis to reduce these differences.

Factor	Level
Partitioning scheme	document, term, hybrid
Multiprogramming level	100, 1000
Query selection	uniform, 80-20 skew
Collection size	10 GB, 100 GB
Number of stopwords	0, 512

Table 3.2: Factors and levels in ANOVA

The ANOVA results (with  $r^2 = 0.999073$  at  $\alpha = 0.05$ ) are shown in Table 3.3. Overall, every factor in the study is significant and in the following order: query selection, stoplist, collection size, partitioning scheme, and multiprogramming level. Note that the significance of collection size highlights the distinction of the present study from previous studies which focus on much smaller collections. There exist complex relationships among factors. Some interactions of interest will be broken down into smaller parts for coherent interpretation in subsequent sections.

#### 3.3.1 Effect of Stopwords

The effect of stoplist size on response time is shown in Figure 3.4. As we might expect, the more terms are included in the stoplist, the less response time a query has. This is due to the fact that more terms with very long inverted lists are ignored during the query processing. This clearly is indicated in the 80-20 skewed query environment which prefers terms with high frequency of occurrence. However, in the uniform query environment, where terms in the vocabulary set are equally likely to be selected in a query, the benefit of stopwords is less than that in the skewed environment.

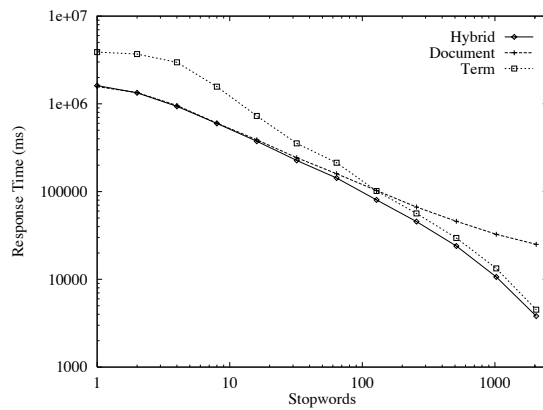
Source of Variation	DF	SS	MS	F	Sig. of F
scheme	2	128.50	64.25	7647.70	<.0001
multprog	1	12.68	12.68	1509.85	<.0001
query	1	605.13	605.13	72028.7	<.0001
size	1	128.72	128.72	15321.6	<.0001
stoplist	1	410.31	410.31	48838.8	<.0001
scheme*multprog	2	18.48	9.249	1100.27	<.0001
scheme*query	2	55.57	27.78	3307.79	<.0001
scheme*size	2	3.55	1.77	211.55	<.0001
scheme*stoplist	2	60.23	30.11	3584.83	<.0001
multprog*query	1	0.97	0.97	116.63	<.0001
multprog*size	1	0.17	0.17	21.40	<.0001
multprog*stoplist	1	1.41	1.41	168.12	<.0001
query*size	1	7.99	7.99	951.94	<.0001
query*stoplist	1	256.15	256.15	30489.1	<.0001
size*stoplist	1	6.20	6.20	738.63	<.0001
scheme*multpro*query	2	2.24	1.12	133.55	<.0001
scheme*multprog*size	2	0.96	0.48	57.44	<.0001
scheme*multpr*stopli	2	4.11	2.05	245.14	<.0001
scheme*query*size	2	2.64	1.32	157.54	<.0001
scheme*query*stoplis	2	26.15	13.07	1556.87	<.0001
scheme*size*stoplist	2	0.43	0.21	25.86	<.0001
multprog*query*size	1	0.21	0.21	26.08	<.0001
multpr*query*stoplis	1	0.44	0.44	52.63	<.0001
multpro*size*stoplis	1	0.002	0.002	0.24	0.6221
query*size*stoplist	1	0.09	0.09	11.03	0.0011
sche*mult*query*size	2	0.34	0.17	20.56	<.0001
sche*mult*quer*stopl	2	1.49	0.75	89.12	<.0001
sche*mult*size*stopl	2	0.02	0.008	0.98	0.3772
sche*quer*size*stopl	2	3.85	1.93	229.27	<.0001
mult*quer*size*stopl	1	0.0001	0.0001	0.02	0.8854
sch*mul*que*siz*stop	2	0.093	0.05	5.58	0.0044

Table 3.3: ANOVA results of the preliminary experiment

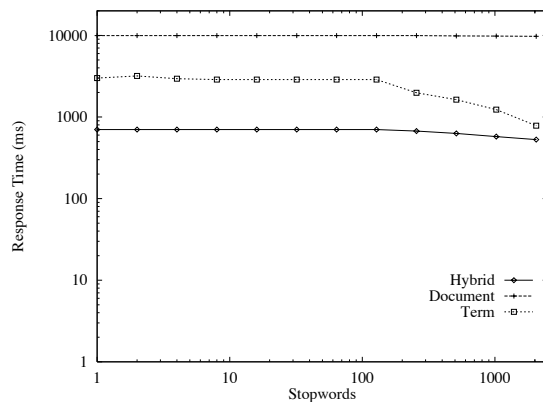
### 3.3.2 Load Balancing Characteristics

Load balancing is a crucial characteristic for parallel inverted indices. The node utilization gives us a sense of how well the load is distributed across the nodes in the system and how heavily the nodes are used. Since the size of an inverted list is skewed, the performance of the index is highly dependent on the query term preference. Skewed query environments prefer frequently occurring terms to the less frequently occurring ones causing the size of an average inverted list to be large, and thus the nodes are heavily utilized. At the same time, multiprocessing level makes use of idle resources, thus is expected to increase the node utilization.

We here examine, under different degrees of query skew, the load balancing characteristic of the partitioning schemes: document partitioning (most balancing), term partitioning (no balancing), and hybrid partitioning (balancing with a larger partitioning unit than document partitioning). Figure 3.5(a) and Figure 3.5(b) show the utilization of nodes in the system under highly skewed and uniform query environments at multiprocessing level of 200. Figures 3.6(a) and 3.6(b) show the node utilization in both query environments at multiprocessing level of 3000.



(a) 80-20 skew



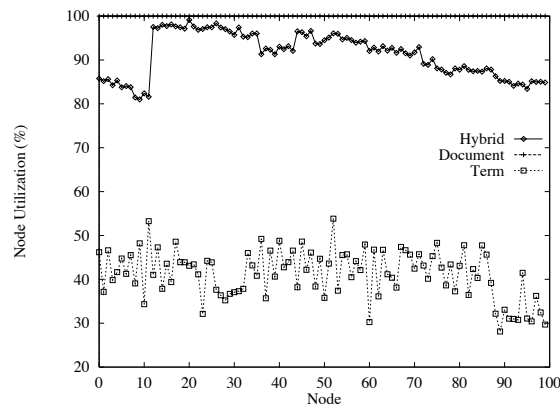
(b) Uniform

Figure 3.4: Effect of stopwords

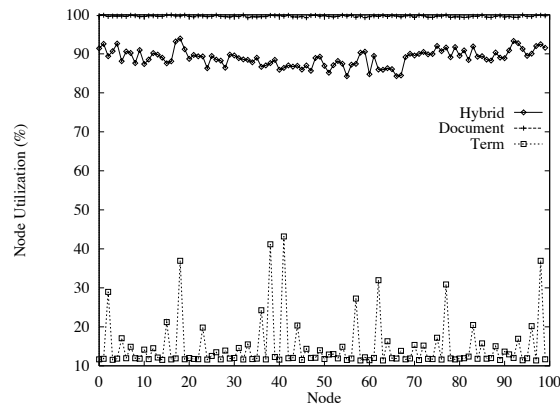
The system consists of two parts (i.e., nodes and communication channels) in which, as discussed earlier, processing proceeds concurrently. We find that communication channel utilization between a node and the IR server always is less than the utilization of the node itself. This is because the overhead and transfer costs of disk are higher than those of communication. Thus the nodes are expected to be the critical part of the system. Small fluctuations in utilization among nodes indicates good load balancing.

From Figures 3.5 and 3.6, document partitioning always has a near perfect load balancing where all nodes have relatively equal utilization in every condition. However, since this approach requires a larger amount of I/O operations at a node, their utilizations are almost always 100%, indicating that the scheme heavily utilizes the nodes, and nodes become saturated. In the term partitioning, nodes in the system are unequally utilized. Some nodes are much more utilized than others. The load imbalance of this scheme is aggravated when the query is less skewed because the range of possible query terms is wider. For example, the





(a) 80-20 skew

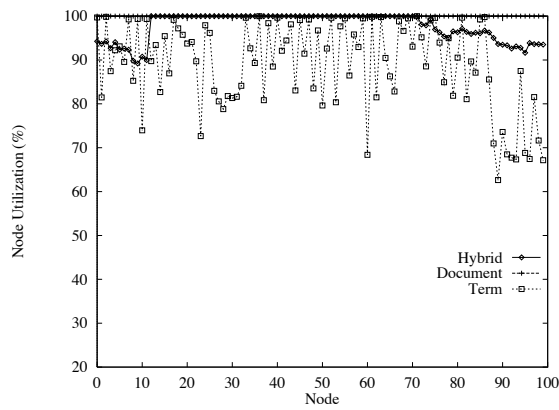


(b) Uniform

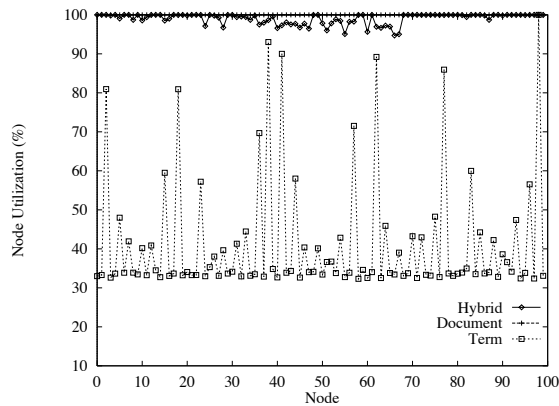
Figure 3.5: Node utilization at multiprogramming level 200

difference between the highest and the lowest utilization of the term partitioning approach at multiprogramming level of 3000 with 80-20 skewed query is 37% and with uniform query is 67%. The results also indicate that this scheme does not saturate the nodes, thus has more spare resources to consume as multiprogramming level rises. The hybrid partitioning scheme has more balanced load across the nodes than does the term partitioning scheme but less than does the document partitioning scheme. There is minor fluctuation among the utilization of nodes. This scheme utilizes the nodes in the range of 90-100% in every condition. Although utilizing nodes more heavily than does the term partitioning, this scheme still has some resources available to benefit from multiprogramming and so is the best of the three.

When the multiprogramming level rises, the nodes are more heavily utilized in every scheme, as we expected. The effect of multiprogramming will be discussed in more detail in the next section.



(a) 80-20 skew



(b) Uniform

Figure 3.6: Node utilization at multiprogramming level 3000

### 3.3.3 Effect of Multiprogramming

With multiprogramming, multiple queries can be processed concurrently. The multiprogramming level indicates the number of queries in the system at a particular time. It accelerates the throughput of the system by utilizing nodes which are idle. Results are shown in Figures 3.7 and 3.8.

In skewed query environments, the document partitioning performs better than the term partitioning at low multiprogramming levels. However, after the multiprogramming is beyond a certain level (approximately 300 in 80-20 skewed query), the term partitioning performs better than does the document partitioning. At low multiprogramming levels, the document partitioning which balances the load by distributing the I/O load equally among the disks performs well while the term partitioning with its load imbalanced characteristic performs

worse because some nodes are used more heavily than others while other nodes finish their tasks earlier and lie idle. As multiprogramming level increases, the term partitioning which has more spare resources (i.e., lightly utilized nodes) performs better because spare resources can process other queries which are in the system. The document partitioning because of its saturated nodes (due to the amount of I/O incurred by the scheme) does not have improved performance when the multiprogramming level increases.

The results also show that the hybrid partitioning performs better than the other two schemes in this high query skew environment. One reason is because the partitioning (unit) of the inverted file takes into account the query condition; the result is a suitable distribution unit for each condition. Although the nodes in this scheme are more heavily loaded than in the term partitioning, there still are spare resources available to process more queries currently in the system as the multiprogramming increases. However, since the amount of these resources is less than that of the term partitioning, this scheme is saturated earlier than the term partitioning.

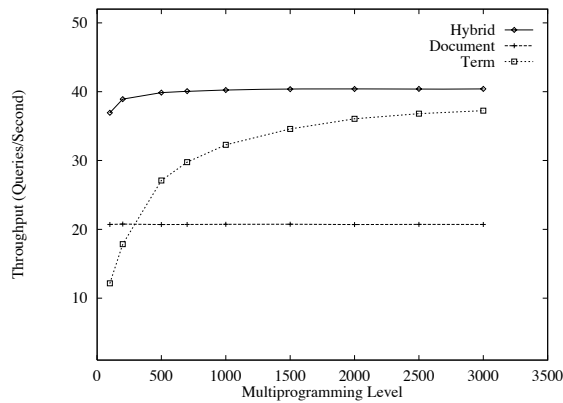
The throughput of term partitioning tends to approach that achieved by the hybrid partitioning scheme at high multiprogramming level because of the amount of its spare resources. However, we can see that its trend is slower as the multiprogramming rises because of the counter effect of its load imbalance property. Thus its performance may not approach the performance of the hybrid partitioning, or if it does, the multiprogramming level might have to be very high.

When query terms are less skewed, the average size of an inverted list is smaller, and every scheme has increased throughput. The term partitioning begins to perform better than the document partitioning at lower levels of multiprogramming because the sizes of lists are smaller causing more queries to finish more quickly even when the load is unbalanced, while the document partitioning always saturates its nodes. The difference among the performance of these schemes is wider as query terms become less skewed – i.e., at 1000 multiprogramming level, the ratios of throughput (hybrid/term, term/document) are (1.25, 1.56) at 80-20, (1.46, 1.89) at 70-30, (2.50, 2.48) at 60-40, and (2.80, 5.90) at 50-50 multiprogramming levels – because of the decrease in the average size of an inverted list.

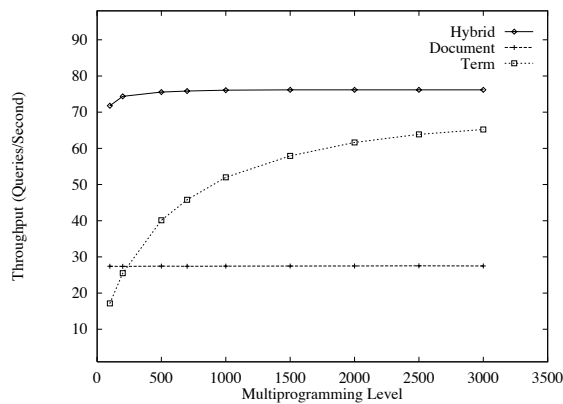
When the range of possible query terms is wider, the node utilization in the term partitioning approach fluctuates more, causing higher differences in its node utilization, while the hybrid partitioning still maintains its load balancing, and the document partitioning is saturated. The saturation of every scheme when the multiprogramming level increases also is observed in every query condition.

### 3.3.4 Effects of Other Parameters

Figure 3.9 (a) and (b) show the effect of the number of nodes in the system on throughput of different partitioning schemes while maintaining the multiprogramming level at 1000. During



(a) 80-20 skew

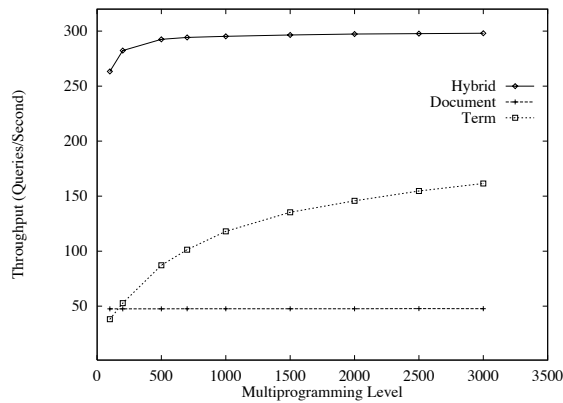


(b) 70-30 skew

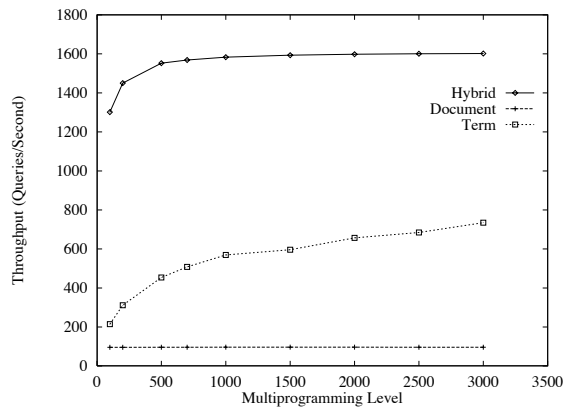
Figure 3.7: Effect of multiprogramming levels at 80-20 and 70-30 query skew

this experiment, the chunk size of the hybrid partitioning scheme is adjusted according to the number of nodes currently studied. The graphs show that every scheme scales with the system in both highly skewed and uniform environments. As the number of disk nodes increases, the amount of work per query decreases though with different rates. The results also indicate that the hybrid partitioning scheme scales with the system better than do the other two schemes. The effect on response time is similar as shown in Figure 3.10 (a) and (b).

In Figure 3.11 we graph the query response time as the query length varies while keeping the multiprogramming level fixed at 1000. As we might expect, for every scheme the response time increases as the number of terms in a query increases. The relative performance of the schemes remains the same. The hybrid partitioning performs the best followed by the term partitioning and the document partitioning approaches, respectively.



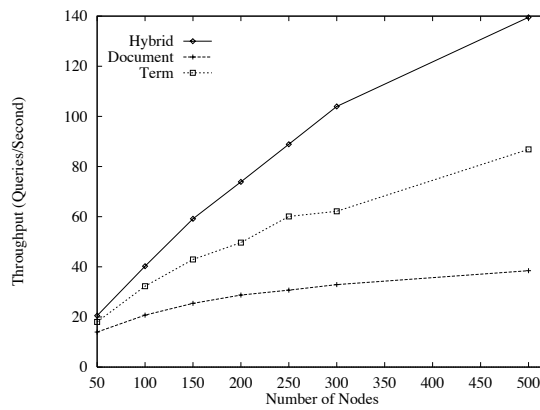
(a) 60-40 skew



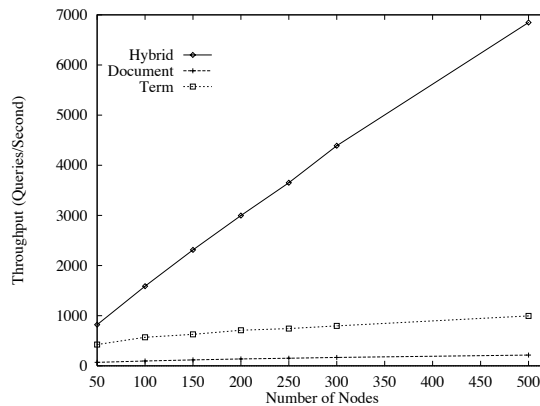
(b) Uniform

Figure 3.8: Effect of multiprogramming levels at 60-40 and uniform query skew

In Figure 3.12 the impact of network bandwidth on response time is shown. Since in the hybrid and term partitioning schemes disk nodes transmit large amounts of data to the server relative to the document partitioning, at low bandwidth the document partitioning has better performance than these two. However, as the network bandwidth increases, the response time of these schemes is improved more than with the document partitioning scheme. Once the bandwidth is large enough, the overhead of fetching data from disk is larger than the time taken to transmit the data, so increasing the bandwidth no longer improves the response time.



(a) Throughput, 80-20 skew

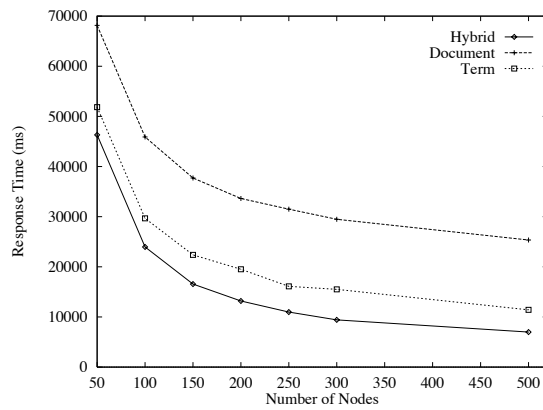


(b) Throughput, Uniform

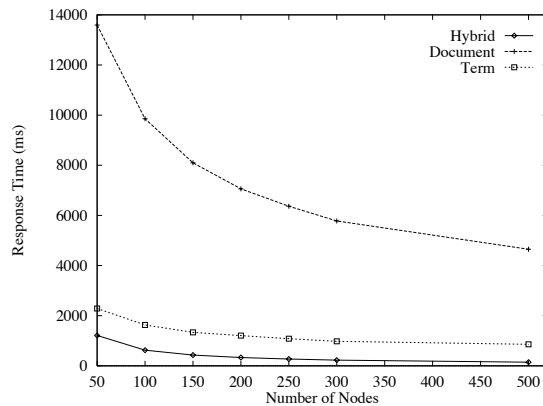
Figure 3.9: Effect of the number of nodes in the system on throughput

### 3.4 Conclusion

In this chapter, the hybrid inverted index partitioning scheme is proposed to support searching for information in a large-scale digital library, implemented atop a network of workstations. It combines the document and term approaches to partitioning inverted indices. The organization, based on this scheme is called *Hybrid Partitioned Inverted Index*. A heuristic to select good chunk sizes for a specific operating environment, based on the time to complete processing of a batch of queries, is presented which is flexible in six dimensions: term-frequency distribution, collection size, stopword list, query selection distribution, number of query terms, and number of nodes in the system. As results from simulation with a terabyte of text have indicated, this method outperforms the document and term partitioning schemes over a wide range of conditions.



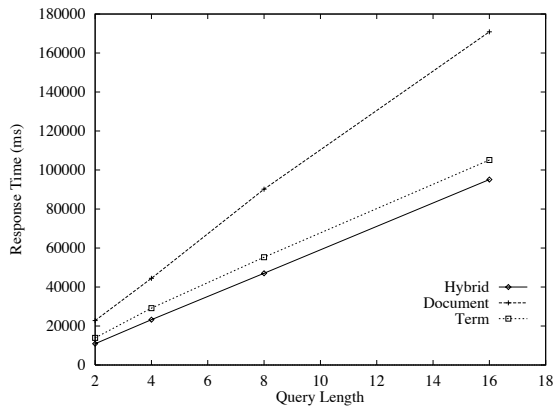
(a) Response Time, 80-20 skew



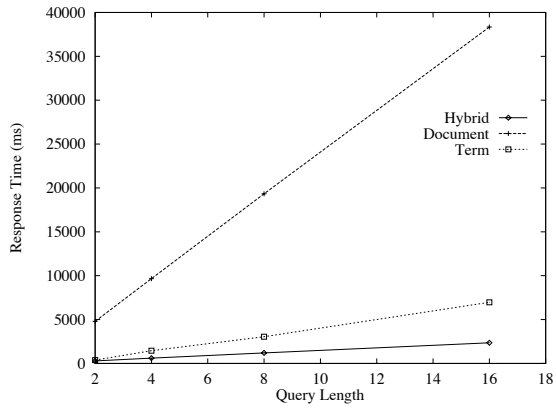
(b) Response Time, Uniform

Figure 3.10: Effect of the number of nodes in the system on response time

The main advantage of the document partitioning scheme is that it is simple to implement. No communication is required to build the index. The term partitioning performs better than the document partitioning in most conditions. An implementation of the term partitioning scheme has been proposed in the literature [52]. However, when the collection scales up to terabytes or larger, storing an entire very long list in one node may not be efficient because a large amount of storage and superior performance are required at the nodes containing very long lists during both inversion and query processing. The resources required by the hybrid scheme are divided across the nodes in the system. Nevertheless, due to the requirement of dividing an inverted list into chunks its implementation is expected to be more complex than the document and the term partitioning. In Chapter 4, the implementation of hybrid partitioned inverted indices will be presented.



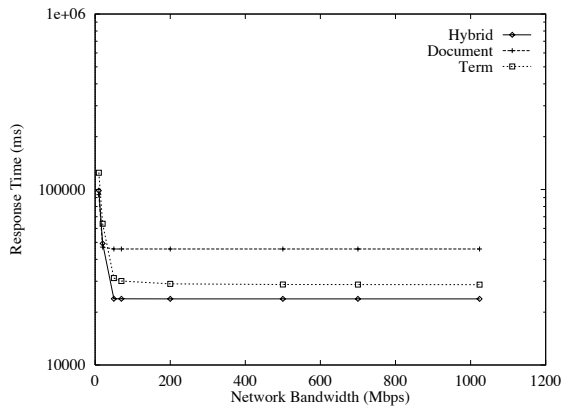
(a) 80-20 skew



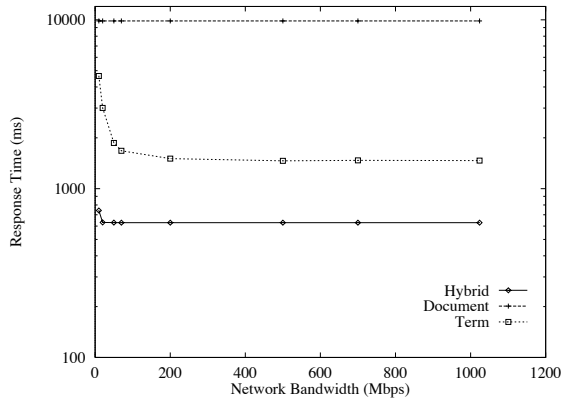
(b) Uniform

Figure 3.11: Effect of the query length





(a) 80-20 skew



(b) Uniform

Figure 3.12: Effect of LAN bandwidth

## Chapter 4

# Inverted Index Implementation

The process of constructing an inverted file is known as *inversion* of the text. In a common approach, the production of an inverted file consists of two steps. In the first step, documents are parsed, and terms are recorded onto an in-memory trie data structure. Each distinct term has an entry in the trie. Once the collection parsing is completed, the trie is traversed and written to disk in the form of an inverted file. Although the performance of computers and the amount of main memory continually increases, collection sizes also grow rapidly. So, it is likely that memory will never overtake the demand [77]. Virtual memory inversion is a way to get around this problem, however, swapping pages in and out of the main memory can degrade inversion performance. Thus it is not practical for large collections.

It is recognized that coordination between disk and memory is required to cope with large collections. We here discuss three approaches which differ in the way disk and memory interact, but where each is shown to have good performance.

The first algorithm, called *multiway merging* [77], subdivides the collection into multiple runs such that the inverted index built from each run fits in memory. A run is a partial in-memory inverted index whose terms are sorted based on term strings. The sorted run is then written to the disk. Once the collection is exhausted, and all the runs are written to the disk, the runs are merged together in memory to produce the final inverted index. In order to merge N runs together from the disk, the memory is divided into N roughly equal parts. The data from each run is read from the disk to fill the part of memory allocated for it at that time. We can use a priority-queue with N entries which selects the list of the smallest or largest term (depending on the order we use to sort terms in the runs before writing to the disk) from each run and merges lists of the same term together.

The next algorithm, called *FAST-INV* [20], is a technique that involves partitioning the vocabulary. It subdivides the inversion into small tasks such that each task can be carried out in main memory. The first task processes only the first part of the vocabulary and writes the partial inverted file to the disk. Later tasks proceed in the same way with different parts

of the vocabulary and append their outputs to the inverted file in the disk. This algorithm requires at least two passes over the text. The first pass collects statistics and determines the number of tasks and their boundary terms. Then multiple second passes proceed as described.

The last approach, called *text-based partitioning* [77], distributes the collection into subcollections of documents, called chunks, and processes them one at a time. The first pass over the text collects statistics and predetermines the exact location and length of each inverted list in the final inverted index. After a chunk is processed in main memory, the resulting lists are transferred to the terms' entries at the exact positions within the final index file.

Witten *et al.* [77] discuss a number of inversion algorithms, including some described above, and the interesting problem of on how to use compression to enhance the performance of inversion.

With the rapid growth of digital libraries (contents and users), completely rebuilding an entire inverted file may not be feasible. Therefore, the inverted file itself has to be dynamic. It must support *in-place updates*. The contents of digital libraries are updated in three ways: *incremental update*, document *modification*, and document *deletion*. Document deletion can simply be achieved by tagging the documents and their corresponding postings as deleted. Document modification can be treated as a deletion followed by an incremental update. The most problematic operation is the incremental update since it requires the expansion and readjustment of the index currently in use. This may disrupt the index availability during the update process. The problem is aggravated in situations where insertions are frequent. In the rest of this section, we survey previous work related to the issue of incremental updates of inverted files in uniprocessor systems.

Central to the expansion of an index to accommodate new postings is the use of disk-based structures that can cope with the expansion without consuming inordinate amounts of extra resources. In general a document contains many terms; the structures must support *multi-point* expansion rather than a simple append mode of expansion. Since inverted lists vary widely in size, the structures must support a collection of dynamically changing variable-length records [77]. The general design is to treat individual lists separately and provide spare space at the end of a list for new postings. The key question arises when the lists expanded no longer fit in the current structure. How this is addressed is the main difference among research solutions in this area. In this section, we will look at different structures that support collection expansion.

There are two main camps of research regarding these issues. The first camp builds a custom-made storage subsystem from scratch using, for example a UNIX raw file. The second camp chooses to build structures on top of a file system. The former approach avoids the limitations and overheads of file systems. It can treat the disk as a contiguous sequence of bytes and have full control of the data layouts on the disk. This way, the specific functionality and performance requirements can be satisfied exactly. On the other hand, the benefits of file systems are missing, such as delayed write, concurrency control, and recovery mechanisms.

The latter approach benefits from advances in file systems. At the same time, it relies on the file system regarding the placement of data on the disk, and accepts the overheads and limitations of the file systems, such as file size limitation and sensitivity to disk fragmentation. Next we discuss two organizations that are built atop a file system. Then we discuss another four organizations that directly manipulate data layouts on the disk.

In an early work, Cutting and Pedersen [13] present three scalable inverted file organizations, implemented as a disk-resident B-tree. In the first technique, called *merge update*, the main memory is used to buffer and sort the incoming postings, which are merged with the B-tree when full. This technique eliminates many disk accesses by batching updates in a buffer before flushing to the disk. The next technique, called *pulsing*, writes the postings directly in the B-tree and overflows (pulses) to a heap file when the specified number of postings to be stored immediately is reached. The last method, called *delta encoding*, stores differences between consecutive locations, instead of actual locations, in the file. The pulsing and delta encoding methods are shown to reduce the size of the index. However, deletion in the delta encoding technique is costly. To delete a single entry the entire inverted list may have to be read into the memory.

Zobel *et al.* [83] propose a structure, called *disk file*, that is organized into fixed length blocks. A block contains a block address table, records, and free space. The address table stores, for each list, the address of that list in the block. To locate a list, a record address table is maintained in memory, containing the block number containing the record needed. To retrieve or expand a list, the entire block is read into memory. In cases that a list needs to be expanded, if there is not enough space in the current record and if the block still contains sufficient free space such that the extended record can be accommodated, records in the block are linearly shifted to make the correct space available. If, on the other hand, there is insufficient free space within the original block, the smallest record whose removal leaves enough space is deleted and treated as an insertion to another block after consulting the free list which is maintained in memory.

Now, consider the work that is not built on top of file systems. Faloutsos and Jagadish [16], motivated by the skewness of the term-frequency distribution, propose a geometric growth technique to accommodate the space for postings in an inverted file, implemented as a B-tree. In another work [15], they divide inverted lists into two classes (i.e., long and short lists) by a fixed criteria and present a hybrid index structure in which long lists are stored in a signature file, and short lists are stored in an inverted file.

Tomasic *et al.* [71, 72] dynamically divide lists into long and short lists. Short lists are placed in fixed-size regions, called *buckets*. Every list begins as a short list. Once a bucket fills up, the longest list in the bucket becomes a long list. Long lists are stored as contiguous sequences of blocks on the disk. Since updating the index is costly and can disrupt normal operation of the system, they batch together a small number of incoming documents and provide that the batch space can be searched in parallel with the inverted file.

Brown [6] proposes a method to apply the Mnome persistent object store [38] which is a

system for storing and retrieving objects, where an object is a chunk of contiguous bytes, identified by a unique identifier. The idea of this method is creating an object for a list and submitting it to Mneme which will store and return the objects, as requested. This way, an inverted index is built as an abstraction (transparently managed by Mneme) with a lookup table containing object identifier as an interface. The interesting aspect of this method is the use of objects of various sizes to manage not only inverted lists but also the lookup table.

## 4.1 Document Inversion Algorithm

In this section, we describe a document inversion algorithm in a single node which will be the basis for our distributed inversion algorithms. The first step in building an inverted index is scanning the collection and extracting *words* from each document. Each word can further be checked against the *stoplist* and/or stemmed into *terms* by a *stemmer* [23].

A stoplist (or a negative dictionary) is a list of many of the most frequently occurring words in English (like “the,” “of,” “and,” etc.). A search using one of these terms is likely to retrieve almost every document in a collection regardless of its relevance. These terms are considered to have low discrimination powers, thus be worthless as index terms. Terms in a stoplist are usually filtered out during automatic indexing which eliminates potential indexing terms which in turn saves huge amounts of time to build and space to store the index without sacrificing retrieval effectiveness.

Stemming is the automated *conflation* (the process of matching morphological term variants). Conflation can be done manually or automatically through programs called *stemmers*. The most commonly used technique is affix removal, removing suffixes and/or prefixes from a word, leaving its stem. The process may include the transformation of the stem to other forms so that the stemmed term will match multiple words. For instance, a truncated term “retriev” will match terms such as retrieval, retriever, and retrieving. Other alternatives include successor variety which bases the stemming on the frequencies of letter sequences in the text and the n-gram method which conflates terms based on the number of n-grams they share.

The steps (and rules) adopted during indexing not only affect the effectiveness of an IR system but also the efficiency of indexing and query processing. A stoplist can be very general or specific to the collection and features of the system. A more specific stoplist leads to better performance. However, it is difficult to predict what terms should be included in the stoplist without going through the collection before indexing. In addition, a more fine-tuned stoplist is limited to a particular instance of the collection and may not be applicable when the collection expands.

Stemming is intended to improve *recall* — the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in the collection. It also decreases the size of the index, sometimes by as much as 50 percent or more. However,

the benefit of stemming is equivocal [23]. Incorrect stemming can degrade the effectiveness of an IR system. When too much of a term is removed, unrelated terms can be conflated which causes nonrelevant documents to be retrieved. This is called *overstemming* which hurts *precision* — the ratio of the number of relevant documents retrieved over the total number of documents retrieved. On the other hand, when the removal is too little, relevant documents will not be retrieved. This is called *understemming* which hurts recall.

In order to make the results as general as possible, in this research we use the stoplist and the stemming algorithm described in [23], regardless of the collections at hand. The stoplist consists of 425 words derived from the Brown corpus [24] of 1,014,000 words drawn from a broad range of literature in English. The stemmer is Porter's algorithm [46] which employs a type of affix removal. The parsing process we use is adapted from the double buffer scheme, proposed by Brown [6], and is shown below:

```

Foreach (document  $d$  in the collection)
  Parse  $d$ ;
  Foreach (term  $t$  parsed from  $d$ );
    if ( $t$  is not in stoplist)
       $t' = \text{Stem}(t)$ ;
      Merge  $t'$  with Document Buffer;
    end;
  end;
  Flush Document Buffer to Batch Buffer;

  If (Batch Buffer fills up)
    Write a run to Temporary File;
  end;
end;

While (Merge Buffers are not all empty)
  Generate an inverted list  $l$ ;
  Submit  $l$  to Storage Manager
end;

```

The parser reads a document from the disk, extracts words and their exact locations within the document, and checks each word against the stoplist. Words that are not filtered out by the stoplist are stemmed into their reduced forms and stored in the *document buffer*. The document buffer is an in-memory binary search tree, shown in Figure 4.1. Nodes in the tree are ordered by term string. Each node is equivalent to a term and contains the term string, the term frequency in that document, and a pointer to the list of locations. In each node, there also is a pointer to the last location of the location list to facilitate the addition of a new location node to the list.

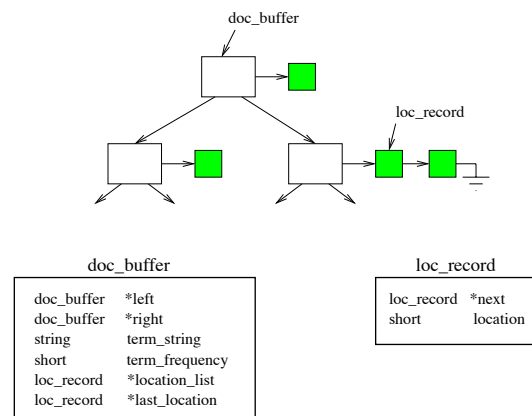


Figure 4.1: Structure of the document buffer (adapted from [6])

When a document is completely parsed, the document buffer is flushed to the *batch buffer*. The batch buffer also is constructed as an in-memory binary search tree, shown in Figure 4.2. Its function is to batch as many documents as possible before writing to the temporary file to reduce the amount of disk I/O required. A node in the batch buffer contains the term string, the total term frequency in this batch, a pointer to the list of documents that this term appears in, and a pointer to the last document record of the document list.

Once the batch buffer is full, every node in the buffer is written to a temporary file in an inorder tree traversal fashion so that a block of the temporary file on the disk is lexicographically ordered. A temporary file block (also called a *run*) is in fact a partial inverted index, generated from the documents that are held in the batch buffer. The parser proceeds this way until the collection is exhausted, and all the blocks are completely written to the temporary file.

The temporary file now contains one or more blocks of partial inverted indices. All these partial indices must be merged together in memory to build the final inverted index. We adopt the multiway-merging method [77]. Suppose that there are  $N$  blocks in the temporary file of approximately  $B$  bytes each. If we allocate  $M$  bytes of memory for merging, to perform an  $N$ -way merge each block will have at most  $\frac{M}{N}$  memory space to hold its data at a time. In total, to read the temporary file of size  $N \cdot B$  as a sequence of  $\frac{M}{N}$  bytes in random requires  $\frac{N^2 \cdot B}{M}$  disk operations. Merging requires the use of a priority query such as a heap to allow the lexicographically minimal elements to be found efficiently. Matching elements from other blocks are found by extracting elements from the heap until a non-matching element appears at the top of the heap. A final inverted list is built by concatenating the matching partial lists as they are extracted from the heap. Each block that contributes the minimal element for the current round is advanced to its next element. The process repeats until all the partial lists in the temporary file blocks have been consumed. Merging is accomplished because the partial inverted lists in each block are stored on the disk in sorted order. As the final

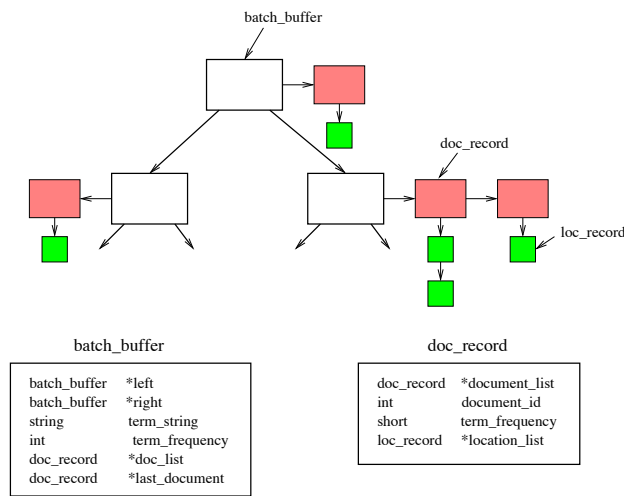


Figure 4.2: Structure of the batch buffer

inverted lists are produced, they are sent to the Storage Manager (described in Section 4.1.2) to make them available for future access.

The process described above divides parsing into four steps: storing terms in the document buffer, flushing the document buffer to the batch buffer after each document, flushing the batch buffer to the temporary file, and finally merging postings in the temporary files to produce final inverted lists. In Section 4.1.2 we consider, after partial inverted lists are merged to create final inverted lists, how they are stored in order to support incremental updates. We will see in Section 4.2 that this division also can be adapted naturally to build parallel inverted indices.

### 4.1.1 Inverted Index Storage Manager

The inverted index storage manager, whose task is to store inverted lists and make them available for query processing, is crucial for the performance and functionality of inverted indices. Incremental updates can in large part be supported by this system. The characteristics of collections and queries studied in Chapter 2 provide insights into the design of the storage organization. We revisit some of them here. Inverted lists are accessed in a sequential fashion, so it is best to store a list as a contiguous sequence of bytes for efficient access. Each inverted list expands independently upon the arrival of new documents, thus we should store each list independently to isolate the effect of list expansion in the index. The sizes of the inverted lists vary widely, ranging from 1 to millions of postings. Very frequently occurring terms (2% of the vocabulary) account for 95% of the total amount of storage allocated for the index. Because of this observation, inverted lists should be treated differently depending on their frequencies of occurrence. In addition, these frequently occurring terms tend to be



used more frequently and expand more rapidly with the arrival of new documents than less frequently occurring ones. Therefore, we must provide efficient access and expansion to a very large portion of the postings. The access performance also may benefit from caching based on frequency of occurrence. Furthermore, new documents will contain previously unseen terms. Thus not only the existing lists will expand but new lists also will be added into the index.

In our approach, most of the terms will be stored with precise positions in documents. We call inverted lists in this class, *normal lists*. Some terms with low discrimination power will be stored only with document level granularity using *long lists* (i.e., for each term, only the document identifier and its term frequency in the document are stored for each document). Some of these terms may be filtered out during the parsing stage where a stoplist is applied. However, using a stoplist suggested for general collections may overlook some of these terms. So we keep records of these terms in the index but only with document granularity because the benefit of storing full location information is much less compared to the effectiveness of the retrieval system gained with this information. In addition, the effects of ignoring the location information for very frequently occurring terms can be alleviated by taking advantage of the nature of the operations. For example, in proximity search we may use locations of less frequently occurring terms as reference points to search in the overlapped documents regarding the proximity specified in the query.

A question arises of when a list will be classified as a long list. This issue is one of the major differences among research in this area. We revisit some previous work, already described above, in connection with this issue. In [72], the scheme is dynamic; whichever list the removal from the bucket causes to have sufficient space for what is currently needed becomes a long list. In [6], the scheme is static; a list larger than 8 KB is considered a long list. Our criteria is to specify based upon the number of documents in the collection: how many documents a term must appear in to be considered worthless for retrieval purpose. We specify in terms of the percentage of the total number of documents. For example, in this work we specify that if a term appears in more than 80% of the total documents, it becomes a long list.

The basic element of our scheme is the *list file* which is a structure used to store inverted lists on the disk. An inverted list (which may widely vary in size) is allocated in a *disk-based block*, whose size is the smallest power of 2 large enough (but not larger than the largest size) to contain the list. A list that is larger than the largest block will be stored in multiple blocks of the largest size. The index consists of blocks of many sizes. Storing these blocks together in a file incurs lots of overhead to manage the space, such as shifting data to make appropriate space and managing free space. To simplify space management within the file, each list file will manage only blocks of the same size. The structure of a list file is illustrated in Figure 4.3. The metadata contains the intrinsic data of the file, i.e., the size of blocks stored in this list file and the current number of blocks in the file. A free list indicates whether the space for a particular block is free or being occupied by another list. The entries in the free list and blocks in the file are matched positionwise.

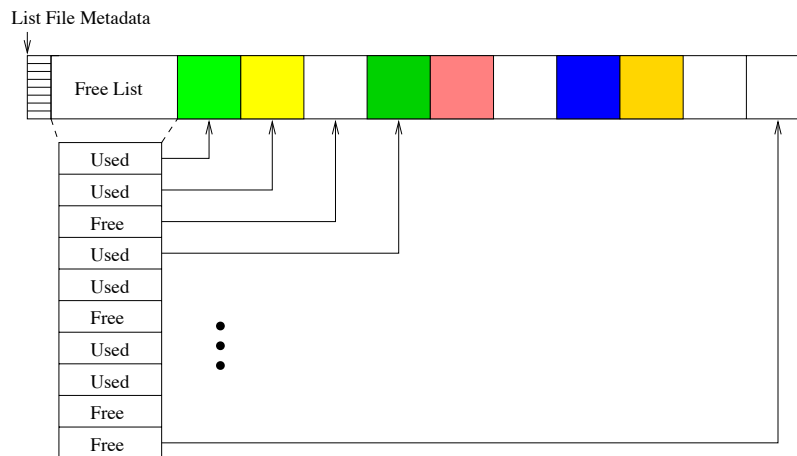


Figure 4.3: Structure of a list file

A list file is implemented as a file. The exact locations of inverted lists on the disk are determined transparently by the allocation algorithms of the file system. Most general purpose file systems are optimized for sequential read. Thus, it is likely that issuing a sequential write through the file system, the file system will make the best effort to store the data to allow best retrieval performance, based on the allocation algorithms and the condition of the disk.

The location of a term in the index is recorded in a structure, called the *List Allocation Table* (LAT), shown in Figure 4.4. LAT will be kept in memory during indexing (and updating). It also is a good practice to maintain LAT in memory during query processing. Nevertheless, if memory is scarce, LAT has to be accessed from the disk. With this possibility in mind, LAT is built as a disk-based hash table and written to disk at the end of indexing and updating.

A slot of LAT points to a list of buckets. Each bucket contains a fixed number of records all of which have the same size, so that bucket management is simplified. A record in LAT contains the following information: term string, total frequency of occurrence of the term in the index, list class, current size of the list, and offset to the beginning of the inverted list within the list file in which the list is stored. The structure of a bucket also is shown in Figure 4.4. The header of a bucket contains the number of records currently in the bucket and a pointer to the next bucket in the list. In the first bucket of a list, the current number of buckets for this slot is maintained. In each bucket, term strings are clustered at the front of the bucket for fast lookup, and the matching records (stripped of term strings) are clustered at the other end, which are matched positionwise.

Given a term string, the LAT slot that the record of this term will be stored is determined by:  $slotID = termNumeric \text{ MOD } numLATSlot$ , where  $numLATSlot$  is the number of slots in LAT, and  $termNumeric$  is a numeric value converted from a term string, as suggested by Wiederhold [75]. In this work, the bucket size is 2 KB, and the length of LAT is fixed

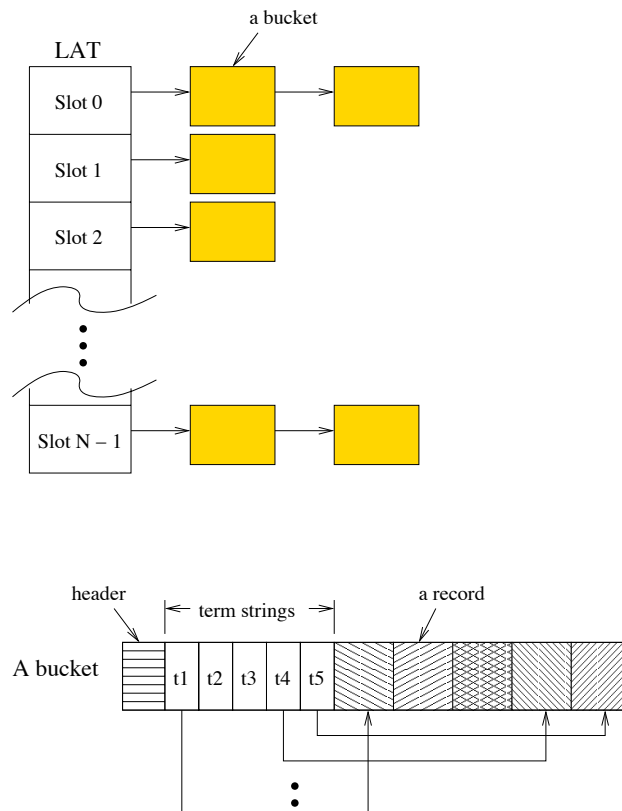


Figure 4.4: List Allocation Table (LAT)

at 2,047 slots. When an inverted list (partial or complete) is created, LAT is examined to determine whether there is a record for the term. If the record of the term is not found in LAT, first it will establish a record in LAT, next the free list of the appropriate list file (computed from the size of the incoming list) is consulted to find a space for the list, then the list is written to the disk and location information is recorded in the LAT record.

If the record of the term is found in LAT, it means that the inverted list of the term has already existed in the index. The appropriate list file can be determined from the current size of the list (stored in the LAT record). Before proceeding further, we need to check whether the current block contains sufficient space to store the expanded list or not. If the current block is large enough to hold the list, the postings are appended at the end of the existing list, and the LAT record is modified accordingly. The exact location to append new postings can be found by a shift from the offset equal to the current list size. However, if the current block is too small, the content of the old block is migrated to the appropriate file (determined by the extended size), and the new list is treated as an insertion to the new file. When an inverted list is moved to the list file of a larger block size, the corresponding free list entry in the old file is marked free so that a subsequent insertion to this file can reuse

the block. As we can see, from the information in LAT, storing a list requires at most 3 I/O operations (in the case of switching files — read old data, write old data to a new list file, and append new data). An improvement can be achieved if the old inverted list is combined with incoming postings in memory before writing. In this case, moving an inverted list to a new list file requires 2 I/O operations. However, for systems where memory is limited, this approach may not be applicable, especially when the list is very long.

If the incoming list is a long list, the location information will be stripped from the list before proceeding as described. Yet if the existing list is a normal list but incoming postings cause it to become a long list, we can catch this case ahead of time with the information in LAT, so the location information in the old content and the new postings are removed before being written to the new list file.

Now we consider the issue of structure expansion. Once a bucket in a list of LAT fills up, the next bucket is allocated in memory and linked to the bucket list. A list file is on the disk all the time; it expands to twice its original size in the way a file is expanded (lseek and write a null character).

The free list is read from a list file and kept resident in memory during indexing and incremental updates. After the operations are completed, it is written back to its place in the list file. There at least as many list files as there are sizes of blocks. Due to file size limitation of a file system (for example, 2 GB in the Second Extended File System (Ext2 fs)), the list file identifier field for each block size may be maintained in the record of LAT.

Having discussed the design, the remaining issue is how many different block sizes should we have. The smallest size is clearly the smallest block which can contain smallest size lists. In this work, it is 8 bytes. Now what should the maximum block size be? In order to make this decision, we use two measures to guide the decision: the space utilization of the blocks and the I/O operations required to retrieve an inverted list. The definitions of these measures are as follows:

Let

$d$	the amount of storage required for document identifier plus term frequency
$l$	the amount of storage required for a location
$L$	the largest rank for which a term will be considered a long list
$D(i)$	the number of documents the term of rank $i$ appears in
$C(i)$	the amount of storage required for the term of rank $i$
$B_{max}$	the maximum block size
$A(i)$	the block size allocated for the list of the term of rank $i$
$V$	the vocabulary size
$S(i)$	the number of I/O operations required to retrieve the list of the term of rank $i$
$Z(i)$	the frequency of occurrence of the term of rank $i$
$z(i)$	the probability that a randomly selected term is the term of rank $i$

Then

$$C(i) = \begin{cases} l \cdot Z(i) + d \cdot D(i) & \text{if } i > L \\ d \cdot D(i) & \text{if } i \leq L \end{cases}$$

$$A(i) = \begin{cases} 2^{\lceil \log_2(C(i)) \rceil} & \text{if } C(i) \leq 2^{B_{max}} \\ \left\lceil \frac{C(i)}{2^{B_{max}}} \right\rceil \cdot 2^{B_{max}} & \text{if } C(i) > 2^{B_{max}} \end{cases}$$

$$\text{Overall Block Utilization}(\%) = 100 \times \frac{\sum_{i=1}^V C(i)}{\sum_{i=1}^V A(i)}$$

$$S(i) = \begin{cases} 1 & \text{if } C(i) \leq 2^{B_{max}} \\ \left\lceil \frac{C(i)}{2^{B_{max}}} \right\rceil & \text{if } C(i) > 2^{B_{max}} \end{cases}$$

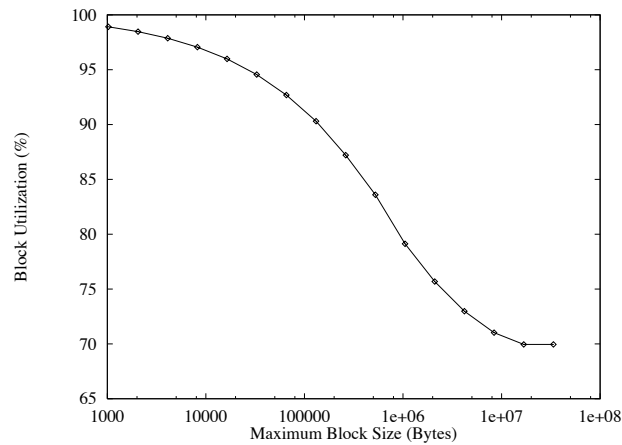
$$\text{Avg. I/O Operation} = \sum_{i=1}^V z(i) \cdot S(i)$$

Ideally, it is preferred that space utilization is high while the number of I/O operations is low. The graphs of the space utilization and I/O operations at different maximum block sizes from 8 bytes to 32 MB are shown in Figure 4.5.

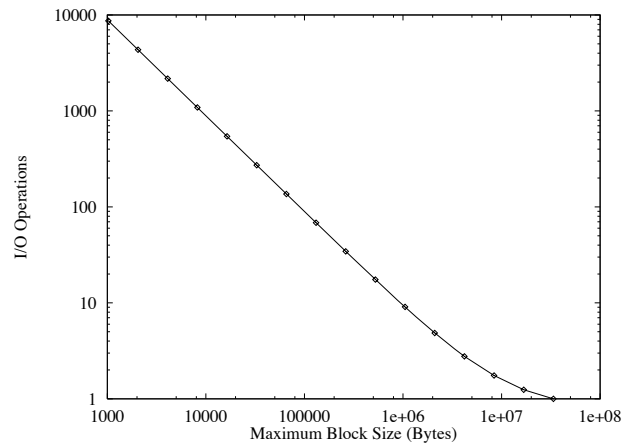
The results show a classic tradeoff between space and time. To have high utilization, the largest block has to be small; doing that the retrieval of many terms (especially very frequently occurring terms) requires a lot of I/O operations, and vice versa. The experiments in the next section are based on a maximum block size of 1 MB which, from the graphs, gives 79.13% block utilization and requires 9.07 I/O operations on average.

### 4.1.2 Experimental Results

The experiments consist of two parts. In the first part, we measure the performance of indexing static collections. In the second part, we measure incremental update performance. All of the experiments are run on an IBM PC compatible with Pentium II processor (clock speed at 233 MHz) running the RedHat Linux operating system, version 6.1, with 96 MB of memory and 25 GB IDE disk drive. The program is written in the C language. The text used in the experiments is the test collection used in The Eighth Text REtrieval Conference (TREC-8). The collection is broken down into 18 individual collections. These collections vary from newspaper to patent records and have different characteristics, such as number of terms, number of unique terms, number of documents, and document length.



(a) Block utilization at various maximum block sizes



(b) Average I/O operations at various maximum block sizes

Figure 4.5: Block utilization and the average I/O operations versus the maximum block size

## Bulk Indexing

In this section, we measure the elapsed time required to index the entire 5.3 GB TREC combined collection. The program allocates 30 MB for the batch buffer. All structures are preallocated before run time. If more structures than the preallocated amount are needed, they will be dynamically allocated. Each experiment is run twice, and the elapsed time averaged across the runs is reported.

Table 4.1 shows elapsed time of the parser, number of temporary file blocks used, and the aggregate amount of storage occupied by the the temporary blocks for each collection. Note that the notation AP(1) designates the AP collection in Disk 1. The total time required to parse the entire collection is 38,655.2 seconds (or 10 hours and 44 minutes). The total

Collection	Size (MB)	Parsing Times (sec)	Temp Blocks	Temp Size (MB)
AP(1)	254	2153.3	19	123.4
DOE(1)	184	1299.3	13	85.7
FR(1)	260	1553.0	13	73.7
WSJ(1)	267	2078.5	19	122.9
ZIFF(1)	242	1858.8	16	100.2
AP(2)	237	2036.6	18	117.1
FR(2)	209	1252.1	9	50.6
WSJ(2)	242	2066.1	18	115.8
ZIFF(2)	175	1339.9	11	69.0
AP(3)	237	1987.1	18	117.0
PATENTS(3)	243	1381.9	10	54.1
SJM(3)	287	2192.9	20	130.3
ZIFF(3)	345	2644.7	24	151.9
CR(4)	235	1558.3	12	69.2
FR(4)	395	1842.2	14	83.7
FT(4)	564	4452.3	42	275.8
FB(5)	470	3370.4	11	63.3
LA(5)	475	3581.0	31	207.9
Total	5321	38655.2	319	2015.0

Table 4.1: Parsing results

storage space required for temporary file blocks is 2.01 GB, approximately 38% of the size of the collection.

The results also show that the parsing performance varies with the size of the collection. Some fluctuations occur among different collections of roughly the same size due to the differences in their characteristics, such as the number of documents and the document length. Overhead relates to events such as the document buffer being flushed to the batch buffer after a document is completely parsed. A collection with longer documents and/or smaller number of documents tends to have better performance than another of around the same size. For example, AP(3) which has 78,321 documents with average size of 487 words takes more time to parse than does PATENTS(3) which has 6,711 documents with an average of 5,391 words in each.

After the entire collection has been parsed and the temporary file blocks have been written to the disk, the merging process begins. In this experiment, 50 MB of memory is allocated for the merger buffer. Using the model, developed in Chapter 2, the vocabulary size of a 5.3 GB collection is estimated to be 734,687 terms. Approximately 20 MB of memory is set aside to maintain LAT and other structures in memory. The rest of the memory will be used for dynamically allocated memory, the program code, the operating system, and other processes. Merging and completely storing 734,687 terms in list files requires 12,646.0 seconds (or approximately  $3\frac{1}{2}$  hours).

Now we investigate how well our technique scales with the collection size. Five collections

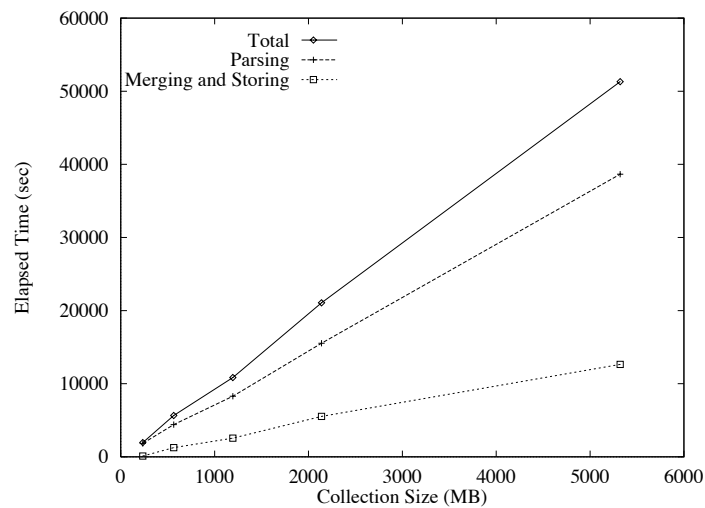


Figure 4.6: Elapsed time of inversion at various collection sizes

are indexed: CR (235 MB), FT (564 MB), Disk 4 (1,104 MB), Disk 4 and 5 (2,139 MB), and the entire TREC-8 (5.3 GB). The results suggest that parsing time, merging and storing time, and the total time scale approximately linearly with the collection being indexed. The parsing process dominates the total time.

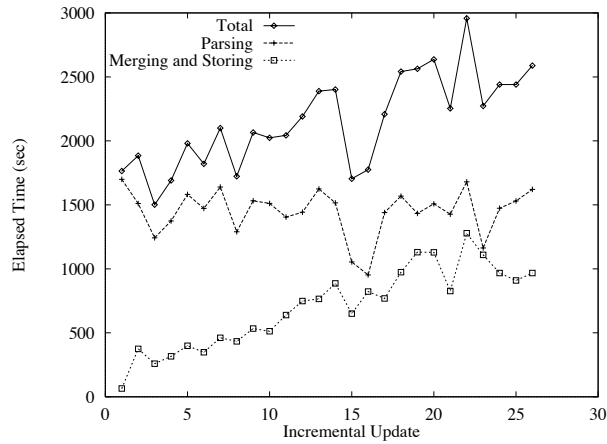
The relative performance of different algorithms depends highly on the components adopted in each. For example, the vocabulary sizes in AP(3) and PATENTS(3) found in our algorithm are 120,705 and 99,759 terms while [6] reports 42,228 and 76,986 terms, respectively. Having larger vocabulary requires more space for structures and incurs more overhead during parsing, as well more time and space to merge and store the lists to the disk. This results from differences in the stoplist, the stemming algorithms, and other rules employed in each of the algorithms.

### Incremental Updates

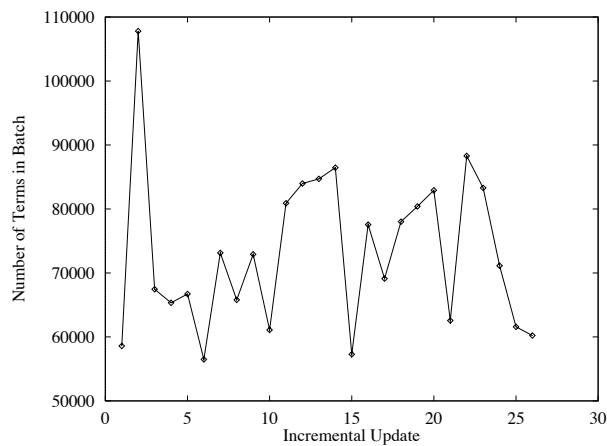
To evaluate the performance of incremental updates, we invert the TREC collection as a series of batch updates, using 26 batches of 200 MB each (the last batch is ignored because it is less than 200 MB, thus the total size is 5.2 GB). The results, shown in Figure 4.7, indicate that parsing time is relatively constant except for some variations. The lowest parsing times (i.e., the fifteenth and sixteenth batches) correspond to FR(4) which as shown in Table 4.1 has exceptionally low parsing time. This collection has only 85,288 unique terms at 395 MB. Having fewer terms in the vocabulary implies lower overhead during growing the document buffer, flushing it to the batch buffer, and writing the batch buffer to the temporary file, thus lower time. The total time to invert the 5.2 GB of text in batch updates is 15 hours and 33 minutes, which already is longer than the time used in bulk indexing of a 5.3 GB



text.



(a) Elapsed time of incremental updates



(b) The number of inverted lists updated

Figure 4.7: Performance of incremental updates

The results also show that the updating cost is increasing with every update. It suggests that the incremental update performance may benefit from using larger block sizes because it batches up updates to a term from smaller batches, thus in effect it reduces the number of updates. From Figure 4.7(a) and (b), the merging and storing cost is in part dependent on the number of lists it needs to store. We can see the same pattern of fluctuations in time follows exactly the fluctuations in the number of lists. In addition to the number of lists to be stored, the performance of merging and storing is dependent more on the number of lists currently in the index. The fluctuations in the number of lists to be stored is more clearly reflected in the elapsed time in later batches than earlier batches. This behavior results from the following reasons: (1) the overhead of searching LAT for the existence of the term in

the index grows with every update, (2) more lists are moved to the list files of larger blocks in later updates, and (3) the range of addresses within a list file is wider with every update due to the insertion of data into the file (especially in the list files of relative large blocks), thus increasing the distances that the head of the disk drive has to move. An investigation into this issue indicates that the behavior is mainly influenced by the last reason.

### Efficiency of Inverted Index Storage Architecture

Next we measure the efficiency of the storage architecture. Space utilization of different list files for the 5.3 GB TREC collection is shown in Table 4.2. We measure only the utilization of actual space allocated for inverted lists because a list file can be compacted to fit the actual amount of space used. For each list file, the table shows the number of inverted lists in each type, the number of blocks allocated in each list file, the actual space used by inverted lists, the total space allocated for blocks in the file, and the space utilization.

Block Size (bytes)	Number of Blocks Allocated	Number of Lists		Space Occupied (bytes)	Space Allocated (bytes)	Block Utilization (%)
		Short	Long			
8	363813	363813	0	2910504	2910504	100.00
16	122355	122355	0	1681784	1957680	85.91
32	91617	91617	0	2243336	2931744	76.52
64	64601	64601	0	3006452	4134464	72.72
128	47243	47243	0	4286672	6047104	70.89
256	30101	30101	0	5476754	7705856	71.07
512	20091	20091	0	7284156	10286592	70.81
1024	14131	14131	0	10233184	14470144	70.72
2048	10281	10281	0	15005918	21055488	71.27
4096	7176	7176	0	20727794	29392896	70.52
8192	5181	5181	0	30077230	42442752	70.87
16384	3812	3812	0	44031528	62455808	70.50
32768	2727	2727	0	63347456	89358336	70.89
65536	1845	1845	0	86043312	120913920	71.16
131072	1247	1247	0	115601232	163446784	70.73
262144	914	914	0	168688336	239599616	70.40
524288	684	684	0	252435344	358612992	70.39
1048576	748	748	0	835135808	1167065088	71.56
Total	788567	788567	0	1668216832	2344787712	71.15

Table 4.2: Block utilization

The block utilization is 71.15%, and the average I/O operations to access a list is 1.50. Both the utilization and I/O operations observed are less than what we expected from the model (i.e., 79.13% utilization and 9.07 I/O operations). Recall that an inverted list larger than the largest block size will be broken down into smaller parts which are stored using many of the largest blocks. The fact that a list is larger than the block size implies that it fully utilizes

some blocks, which tends to improve the overall utilization of the list file of the largest block. Considering only the list file of the largest block size, we know that on average every block is half full, thus for a smaller block size the space utilization is better than that of a larger size. This indirectly improves the overall utilization of the entire storage architecture.

We can see from the table that there is no inverted list which is larger than the largest block size in the index (i.e., the number of blocks allocated is the same as the number of lists). It indicates that the criteria to select the maximum block size is too strict because the criteria we use is based upon the model which does not consider a stoplist while our experiments employ a stoplist during parsing. Thus, many potential long lists are eliminated at an early stage. In the model we use, there are a number of lists which are longer than 1 MB. Without these lists, the utilization is lower, as we can observe from the results. However, an advantage of this is a lower number of I/O operations.

## 4.2 Distributed Inversion

The structure of a parallel inverted file is dependent upon the distributed organization, governed by the partitioning scheme. Since the subcollection stored at each node is independently inverted, the document partitioned organization follows naturally. However, it is not trivial to build a term partitioning inverted file. The main problem is how to merge partial inverted lists, stored across multiple nodes, together. A solution to this problem is presented by Ribeiro-Neto *et al.* [52]. We will discuss two techniques proposed in their paper. The first method, called *local buffer and remote list* (LR), sends the partial lists to their destination nodes when the main memory buffer fills up. The other technique, called *remote buffer and remote lists* (RR), assembles postings in small messages and sends them out very soon to avoid storing them at the local nodes. Analysis with document granularity inverted indices shows that the time to invert a 100 GB collection by the LR algorithm with 8 processors is roughly 8 hours.

The hybrid partitioning scheme requires that inverted lists are divided into chunks, and for many inverted lists their chunks are allocated to different nodes. This posts a major problem on how to keep track of the distribution of chunks of an inverted list across the nodes in the system. In this section, we examine four distributed algorithms for creating hybrid partitioned inverted indices. These algorithms are based on two design policy decisions: the inversion policy and the posting transfer policy. Next, we discuss each of these policies in detail.

### 4.2.1 Inversion Policy

The main issue regarding the creation of hybrid partitioned inverted indices is how a node knows where to forward a posting from its local collection. We refer to this issue as the

*Inversion Policy.* There are two alternatives for the inversion policy: Server-Based Inversion and Nonserver Inversion.

### Server-Based Inversion

In order for a node (called an *origin node*) to know where to forward a posting generated from its local collection, it must first know, with respect to the current state of the distribution which is a global notion (consistent with the entire system) the chunk that the posting belongs to. To acquire this knowledge, a component is needed to maintain the current state of the distribution and provide this information to every node, upon request. We call this component, *Distribution Server (DS)*. DS contains a record of every term in the vocabulary. A record is a tuple  $\langle t, p \rangle$  where  $t$  is the term string, and  $p$  is the global number of postings allocated by DS so far.

Assume that the chunk size is known to every node before the indexing time. What an origin node needs to know to determine the chunk that a posting belongs to is the number of postings of term  $t$  globally allocated so far,  $p$ . A node  $i$  sends a request in the form  $\langle t, p_i \rangle$  where  $p_i$  is the number of postings of term  $t$  that will be distributed from the node  $i$  at that time. When DS receives a request for  $p$ , it returns the value of  $p$  to node  $i$  and updates its record ( $p := p + p_i$ ) for this term. At an origin node, the chunk identifier can be determined by:

$$chunkID = \left\lfloor \frac{p}{chunkSize} \right\rfloor$$

and the remaining space in that chunk is determined by:

$$remainingSpace = \left\lceil \frac{p}{chunkSize} \right\rceil \cdot chunkSize - p$$

After obtaining this information, an origin node can determine to which chunk the posting belongs. The *destination node* of a chunk of a term is:

$$destination(termID, chunkID) = (termID \text{ XOR } chunkID) \text{ MOD } totalDiskNodes$$

where  $termID$  and  $chunkID$  are 32-bit integers, converted from the term string [75] and the chunk identifier, respectively.

To spread the load and increase the performance for the DS lookup, the task of DS is distributed across many nodes. In this work, we implement a set of DSs, each running in a node and maintaining information on a set of terms. The vocabulary is partitioned into  $N$  parts (where  $N$  is the number of nodes in the system); each part is taken care of by a DS.

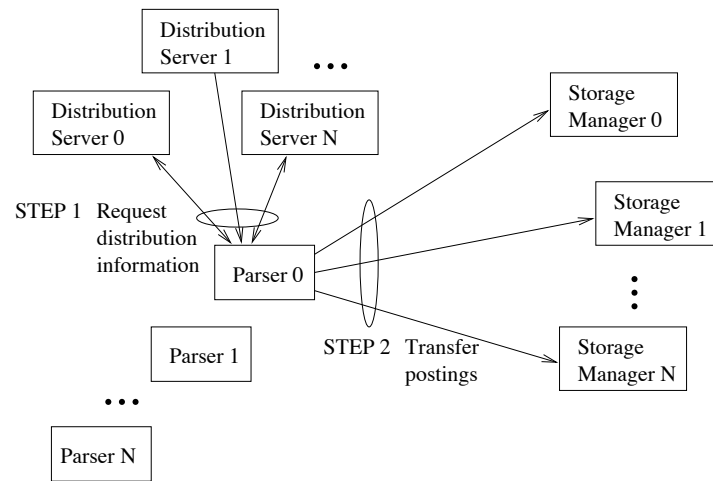


Figure 4.8: Server-Based Inversion Policy

An origin node can send out a query message for a term to a particular DS running at the appropriate node, which is determined by:  $destination(termID, 0)$ .

The approach described above is based on the distribution knowledge maintained at DSs. The process consists of two steps. An origin node sends out a query to appropriate DSs for the current distribution state of each term and, after receiving the information needed, sends postings to their destinations. We call this inversion approach *Server-Based Inversion*, as shown in Figure 4.8.

### Nonserver Inversion

An alternative approach is to collect all partial lists of a particular term created from every node somewhere called an *intermediate node* of that term to build a globally unified list before dividing it into chunks and transferring them to their destinations. The process at an intermediate node assigned for this task is called a *mediator*. At any node, partial lists from an origin node are collected by the mediator and stored in an *intermediate file*, assigned for each origin node. After a mediator receives every list that should be stored at its node, it merges all intermediate files, generates inverted lists, breaks those lists into chunks, and distributes chunks to store at their appropriate destination nodes. The intermediate node (having a mediator running on) for a term also can be determined by:  $intermediateNode = destination(termID, 0)$ .

This approach does not require DS because a list collected by the mediator at this moment is the global inverted list of the entire system. The distribution of chunks can be managed completely by the mediator at an intermediate node. We call this inversion policy *Nonserver Inversion*, as shown in Figure 4.9. In other words, nonserver inversion is an indirect inversion,

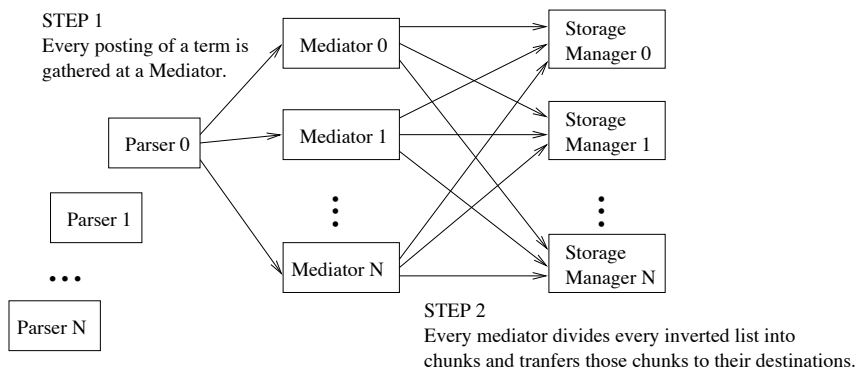


Figure 4.9: NonsERVER Inversion Policy

i.e., reorganization after building a term partitioned inverted index. The idea behind this approach is that a vast majority of inverted lists are small (thus fit in one chunk); most lists will not be chunked or transferred again after being stored at their intermediate nodes.

## 4.2.2 Posting Transfer Policy

Another issue in the creation of hybrid partitioned inverted indices is the *Posting Transfer Approach*. That is when an origin node sends postings out during the inversion. The posting transfer policy also consists of two alternatives: Complete Parsing and Partial Parsing.

### Complete Parsing

From the discussion on uniprocessor inversion, when the batch buffer fills up, the partial inverted lists are written to the temporary file. Next those inverted lists are merged and finally stored in the disk. We adapt this algorithm. Instead of storing lists to the local disk, we send them out, regardless of the inversion approach (server-based or nonsERVER). We call this posting transfer policy *Complete Parsing*.

### Partial Parsing

An alternative to the above approach is instead of writing partial inverted lists to the temporary file, we send them out right away when the batch buffer is flushed, regardless of the inversion style. Since many lists or parts of a list will not be stored at the origin node, the I/O costs of writing the temporary file and merging runs at the origin node are wasted. On the other hand, if we store the partial inverted lists that will not be stored locally directly at other nodes, these I/O operations are saved. We call this posting transfer policy *Partial Parsing*.

To make an efficient use of the network bandwidth, we batch together a set of requests targeted to a particular DS before sending them out as a message when a buffer is full. Replies from a DS to a particular node also are batched up and sent together as a message. In very much the same way, postings targeted to a particular node are batched up before sending. Next we discuss four distributed algorithms for generating hybrid partitioned inverted indices.

## 4.3 Distributed Inversion Algorithms

### 4.3.1 Server-Based Inversion with Complete Parsing (SC)

In this algorithm, within an origin node the local collection is parsed (by *parser*), and postings are written to the local temporary file. Then merging begins to create locally complete inverted lists. When these lists are generated, they are broken into chunks and transferred to their respective destination nodes with the help of DSs.

Within every node, there is a process called *storage manager* to receive chunks from every origin node (sent by its parser process) and store postings in *storage buffers*. Each buffer is assigned specifically for an origin node. When the storage manager receives all the lists from every node, it performs N-way merge (where  $N$  is the number of nodes, thus the number of storage buffers) and stores the lists to its local disk organized in the form of list files (as described in Section 4.1.1). A list at this stage is the globally final inverted list that must be stored in this node.

In this algorithm, every node consists of three components: a parser, a DS, and a storage manager. Since every parser process proceeds independently and concurrently communicates with DSs and the storage managers at every node, to overlap the processing within a node, all of these components are implemented to run concurrently. Next we discuss the parser and the storage manager components of this algorithm.

The parser of the SC algorithm works as follows:

- (1) parse local text and build the temporary file
- (2) transfer local inverted lists
  - (2.1) request current distribution information from DSs
  - (2.2) break each inverted list into chunks
  - (2.3) send all chunks out to their destinations

The storage manager of the SC algorithm works as follows:

- (1) while (NOT done)
  - (1.1) block waiting for an incoming posting from any node
  - (1.2) receive partial inverted lists from node  $j$

- (1.3) store in a temporary file  $j$
- end
- (2) N-way merge the  $N$  temporary files (each collects postings from an origin node) and store the final lists to the disk

Note that Step 1 of the storage manager is done when it receives all postings from every origin node that must finally be stored at this node. To implement this functionality, after the parser running at an origin node completely sends its postings out, a completion signal is sent to the storage manager running on every node indicating the completion of posting transfer to it from this node. After the storage manager receives all completion signals from every node, it terminates the while-loop and begins the merging and storing process.

In this study, the data transfers are all in blocking mode in which the program will wait for the requested action to complete before returning control to the calling process. The times a node spends sending or receiving data can be measured at those primitives. Since in the server-based algorithms DSs running at every node are used by the parsers to obtain distribution information, the processing of the parser will complete after its last access to DSs. Thus the execution time of this algorithm can be determined by the parser time, plus list transfer time, plus the storage manager time. We define the following costs for the SC algorithm:

- $P_{SC}$  the time spent in parser (parsing till writing the temporary file)
- $MRT_{SC}$  the time spent sending lists to their destinations (merging, requesting distribution information, and transferring lists)
- $MS_{SC}$  the time spent in storage manager (N-way merging and storing lists)

The execution time of the SC algorithm is:

$$Execution\ time_{SC} = P_{SC} + MRT_{SC} + MS_{SC}$$

### 4.3.2 Server-Based Inversion with Partial Parsing (SP)

In this algorithm, the posting transfer takes place every time the batch buffer fills up. Thus the temporary file is not created. The algorithm consists of three components at a node. DS proceeds similarly to that in the SC algorithm. The storage manager proceeds according to what is described for the SC algorithm, with a crucial exception.

Since the posting transfer takes place every time the batch buffer is flushed, postings of a term in an origin node's local vocabulary will be sent out many times during the course of parsing (once every time the batch buffer fills up and is flushed). Therefore, in a temporary file assigned for an origin node  $i$  there will be many runs in it. Each run in a temporary file is a partial inverted index of terms sent to this node after the batch buffer at node  $i$  is



flushed. If we assume that during the course of parsing at a node  $i$ , the parser in that node generates  $r_i$  runs (i.e., the batch buffer fills up and is flushed  $r_i$  times), the temporary file at any node  $j$  assigned to hold postings from node  $i$  will consist of  $r_i$  runs. To produce the unified list from inverted lists that are stored in various runs of  $N$  temporary files requires R-way merge (where  $R = \sum_{k=1}^N r_k$ ), which is costlier than N-way merge. Another difference from the SC algorithm is the operation inside the parser of the SP algorithm, which can be described as:

```

foreach (document in the local collection)
  (1) foreach (word in the document)
    (1.1) check against the stoplist and stem
    (1.2) store term information in the document buffer
    end
  (2) flush the document buffer to the batch buffer
  (3) if (batch buffer fills up)
    (3.1) request current distribution information from DSs
    (3.2) break each inverted list into chunks
    (3.3) send chunks out to their appropriate destinations
    end
end

```

The costs of the SP algorithm are:

$P_{SP}$  the time spent in parser (parsing and building the batch buffer)  
 $RT_{SP}$  the time spent sending lists to their destinations (requesting distribution information and transferring chunks)  
 $MS_{SP}$  the time spent in storage manager (R-way merging and storing lists)

The execution time of the SP algorithm is:

$$Execution\ time_{SP} = P_{SP} + RT_{SP} + MS_{SP}$$

### 4.3.3 NonsERVER Inversion with Complete Parsing (NC)

In this algorithm, the temporary file is created, partial inverted lists from every run are merged, and locally final inverted lists are produced, as described in SC. The distinction from SC is that in NC an entire list at an origin node is sent to an intermediate node of that term. After the mediator at that node completely receives partial lists from every node, it breaks the lists into chunks and distributes them according to the term string and the chunk identifier. Three components are running within a node: the parser, the mediator, and the storage manager. The function of the storage manager is the same as those in SC.

The parser of this algorithm can be described as follows:

- (1) parse local text and build the temporary file
- (2) transfer an entire locally complete inverted list to its mediator

The operation of a mediator can be described as follows:

- (1) while (NOT done)
  - (1.1) block waiting for incoming inverted lists from any node
  - (1.2) receive partial inverted lists from node  $j$
  - (1.3) store them in a temporary file  $j$
 end
- (2) perform N-way merge on the intermediate files
- (3) foreach (inverted list)
  - (3.1) break the list into chunks
  - (3.2) send chunks out to their appropriate destinations
 end

Since in the nonserver algorithms, the receiving part of the mediator is running concurrently at every node, and lists are sent to it from origin nodes, the processing of parsers will overlap with the receiving part of mediators.

The execution time of this algorithm can be determined by the parser time, plus the time the mediator spends after it completely receives lists from every node, plus the time spent by the storage manager. The costs of the NC algorithm are:

$P_{NC}$	the time spent in the parser (parsing till writing the temporary file)
$MT_{NC}$	the time spent sending lists to mediators (merging and sending lists to mediators)
$MM_{NC}$	the time spent in the mediator after receiving is completed (N-way merging, chunking, and sending chunks to their destinations)
$MS_{NC}$	the time spent in storage manager (N-way merging and storing list)

The execution time of the NC algorithm is:

$$Execution\ time_{NC} = P_{NC} + MT_{NC} + MM_{NC} + MS_{NC}$$

#### 4.3.4 Nonserver Inversion with Partial Parsing (NP)

In this algorithm, the posting transfer takes place every time the batch buffer fills up, as in SP. However, the algorithm uses mediators instead of DSs. When the batch buffer at an origin node is flushed, partial inverted lists from this batch are sent to mediators. Once a mediator receives all lists that are targeted to it from all of the parsers, it performs R-way merge, as explained earlier. The storage manager proceeds as in the NC algorithm.

The parser of the NP algorithm can be described as follows:

```

foreach (document in the local collection)
  (1) foreach (word)
    (1.1) check against the stoplist and stem
    (1.2) store document information in the document buffer
    end
  (2) flush the document buffer to the batch buffer
  (3) if (batch buffer fills up)
    (3.1) transfer every list to its mediator
    end
end
end

```

$P_{NP}$  the time spent in the parser (parsing and building the batch buffer)

$T_{NP}$  the time spent sending lists to mediators

$MM_{NP}$  the time spent in mediators after receiving is completed (R-way merging, chunking, and sending chunks to their destinations)

$MS_{NP}$  the time spent in storage manager (N-way merging and storing list)

The execution time of the NP algorithm is:

$$Execution\ time_{NP} = P_{NP} + T_{NP} + MM_{NP} + MS_{NP}$$

## 4.4 Querying and Incrementally Updating Hybrid Partitioned Inverted Indices

In this section, we discuss how hybrid partitioned inverted indices we developed can be queried, incrementally updated, and searched. In each node there is a list allocation table, maintaining the information on where an inverted list is stored on the disk. Note that queries enter the system only through the IR server.

In order for the IR server to know which nodes it should contact to obtain the inverted list of a query term, there is an indexing structure in the IR server, called the *Index Manager* (IM), maintaining information for every term in the vocabulary. The information maintained in a record is a tuple  $\langle t, c \rangle$  where  $t$  represents term string, and  $c$  represents the number of chunks in the inverted list of term  $t$ . When a query enters the system, the IR server breaks the query into terms, looks up the number of chunks  $c$  for each term from IM, and sends requests to every node, storing chunks of this list. The node which stores the  $i$ th chunks ( $0 \leq i < c$ ) of term  $t$  can be determined from  $destination(\text{termID}, i)$ .

To build IM, the IR server needs term frequency information for each term from every node. One technique is, after the inversion as described in any of the four algorithms is completed,

every node sends information to the IR server for each term, in the form of a tuple  $\langle t, tf_i \rangle$  where  $t$  is term string, and  $tf_i$  is the total term frequency of term  $t$  in the local index of node  $i$ . The IR server then sums up the  $tf$ 's from every node and computes the number of chunks in the list. Another technique that can be used to poll this information is pairing up nodes and summing their values in pairs until reaching the IR server, which takes  $\log_2 N$  steps. Also the task of IM can be spread across the system, in the same way as DS.

The incremental updates of the hybrid partitioned inverted indices proceed in the same way as does the inversion. That is, every node parses its newly added collection and depending on the algorithm adopted transfers data to each term's destination node. Then the storage manager at each node will merge new lists to its local list files. The implication of this technique is that the number of postings allocated so far has to be maintained in DSs (in server-based algorithms) or mediators (in nonservers algorithms) at all time, or stored on the disk and retrieved when needed.

## 4.5 Experimental Results

In this section we discuss experimental results of four parallel inversion algorithms for creating hybrid partitioned inverted indices. The elapsed time of the algorithm is determined by the slowest node. In order to reduce the variation of the time and simplify the study, we use the same collection in every node. Each experiment is run twice, and the time averaged across the runs is reported. Note that the time measured includes only the time to organize the index according to the hybrid partitioning scheme which does not include the process described in Section 4.4. The collections used in this study are from Disks 4 and 5 of TREC-8 [74]. In the experiments that require the use of a collection of size  $x$ , the program inverts documents and increments the size of text it has processed. When the size is greater than or equal to  $x$ , the parser terminates. The environment used in the study consists of 8 IBM PC compatibles with Pentium II 233 processor, 96 MB of memory, and a 25 GB IDE disk drive. Every machine runs RedHat Linux 6.1 and is interconnected through a 10BaseT Switch. The inversion programs are written in the C language and employ the LAM implementation of Message Passing Interface (MPI), version 6.3.2 [66], as the networking programming interface. Every process starts at the same time and then the measured time begins.

Table 4.3 shows the distribution of time spent in different parts of each algorithm, using 8 nodes with a 1 GB collection in each (for a total of 8 GB). Reported here we use the values of times averaged across every node. The results indicate that the parsing step dominates the execution time of every algorithm.

As we might expect, with regard to the parsing policy, complete parsing algorithms (SC and NC) take more time during the parsing step than partial parsing algorithms (SP and NP), i.e., SC takes 10356.13 seconds while SP takes 8108.50 seconds, and NC takes 10428.99 seconds while NP takes 8727.27 seconds. This is due to the I/O cost of writing the temporary

Algorithm	Part	Time Spent (sec)
SC	$P_{SC}$	10356.13
	$MRT_{SC}$	963.75
	$MS_{SC}$	406.46
SP	$P_{SP}$	8108.50
	$RT_{SP}$	4521.48
	$MS_{SP}$	628.88
NC	$P_{NC}$	10428.99
	$MT_{NC}$	700.79
	$MM_{NC}$	426.27
	$MS_{NC}$	524.74
NP	$P_{NP}$	8727.27
	$T_{NP}$	3371.06
	$MM_{NP}$	1311.79
	$MS_{NP}$	685.40

Table 4.3: Cost distribution in parallel inversion algorithms

file and merging before sending lists out from the origin nodes, which are avoided in partial parsing algorithms.

At the same time, the complete parsing algorithms spend less time for chunks of inverted lists to reach their destinations, i.e., SC spends 963.75 seconds while SP spends 4521.48 seconds, and NC spends 1127.06 while NP spends 4682.85 seconds. This is due to more transfers per term during parsing, and the higher cost of R-way merging at destination nodes in SP or R-merging plus distributing lists at intermediate nodes in NP, respectively.

With regard to the inversion approach, the results show that data transfer costs to send data out of origin nodes are higher in server-based algorithms than in nonserver algorithms, i.e.,  $MRT_{SC}$  is 963.75 seconds while  $MT_{NC}$  is 700.79 seconds, and  $RT_{SP}$  is 4521.48 seconds while  $T_{NP}$  is 3371.06 seconds. This is due to the cost when server-based algorithms request the distribution information from DSs.

However, when including extra costs at the intermediate nodes of nonserver algorithms, the results show that the overhead of transferring inverted lists from original nodes to destination nodes of nonserver algorithms are higher than those of server-based algorithms, i.e.,  $MRT_{SC}$  is 963.75 seconds while  $MT_{NC} + MM_{NC}$  is 1127.06 seconds, and  $RT_{SP}$  is 4521.48 seconds while  $T_{NP} + MM_{NP}$  is 4682.85 seconds. This is due to additional costs at intermediate nodes for merging and transferring chunks of some inverted lists to their destinations, which are represented by  $MM_{NC}$  in NC and  $MM_{NP}$  in NP, respectively.

Partial parsing algorithms trade the I/O costs to write and merge at origin nodes with more transfers and higher cost of R-way merging at later stages (i.e., at destination nodes in SP or at intermediate nodes in NP). On the other hand, complete parsing algorithms spend more time at the origin node (to write and merge runs in the temporary file) but less during

list transfer and merging at later stages. Server-based algorithms identify destination nodes before transferring partial inverted lists, but it pays the price to obtain the distribution information. On the other hand, nonserver algorithms avoid this cost by transferring and merging (if applicable) lists in two steps (i.e., first from origin nodes to intermediate nodes and then from intermediate nodes to destination nodes).

Overall, in this environment we find that: server-based algorithms are faster than their corresponding nonserver algorithms, given the same posting transfer approach; and complete parsing algorithms are faster than their corresponding partial parsing algorithms, given the same inversion approach. The results also indicate that the algorithms can be ordered from the fastest to slowest in this environment as follow: SC, NC, SP, and NP.

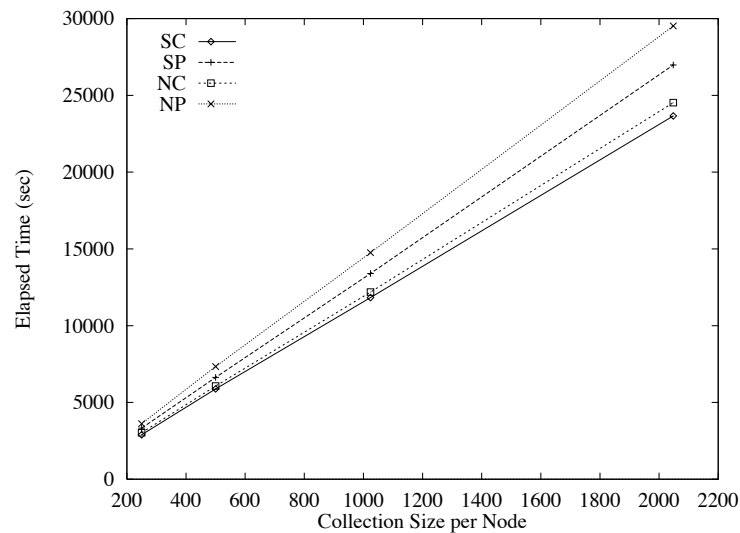


Figure 4.10: Execution times of the SC, SP, NC, and NP algorithms for various sizes of the collection in each processor

In Figure 4.10 we compare the execution time of the four algorithms with collections per node of sizes: 250 MB, 500 MB, 1 GB, and 2 GB. The results are similar to what we observe from Table 4.3: that SC is fastest, followed by NC, SP, and NP, in this order. Using 8 nodes with 2 GB of text per node (for a total of 16 GB), SC takes 23,662 seconds, NC takes 24,526.73 seconds, SP takes 26,981.65 seconds, and NP takes 29,516.79 seconds to index this collection, or the running time of (SC : NC : SP : NP) is (1 : 1.04 : 1.14 : 1.25). As the collection scales up, the effects discussed above are aggravated, and the differences in performance are wider.

Figure 4.11 shows the execution times of the four algorithms for a constant collection per node of size 1 GB, when the number of nodes is varied (i.e., the total collection size increases with the number of nodes). We observe that every algorithm has a minor increase in execution time when the number of nodes increases. The overhead is due to more data being transferred

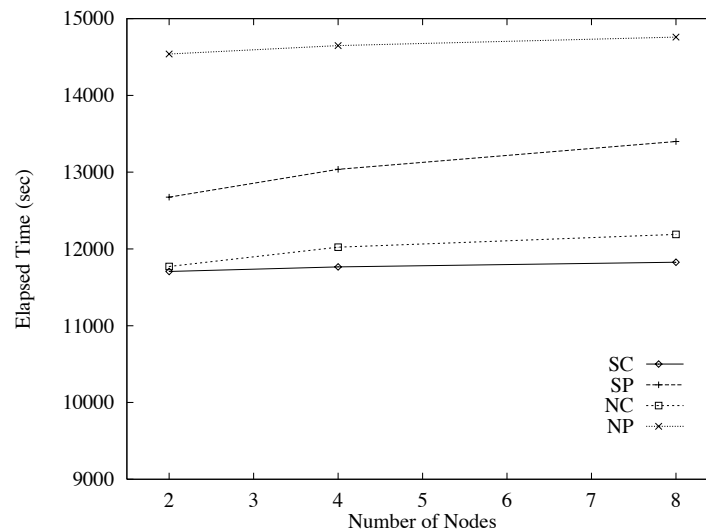


Figure 4.11: Execution times of parallel algorithms for various numbers of nodes for a collection of size 1 GB in each node

and merged in parts of the algorithms; the congestion at DSs (in server-based algorithms) or the receiving ends of mediators (in nonservers algorithms); and congestion at the receiving ends of storage managers. Overall, the results show that in every algorithm larger collections can be inverted with little additional cost in time.

The discussion above centers around the execution time of algorithms. Now we consider other issues related to the adoption of an algorithm. Systems where disk space is limited or where the processor or the I/O system is slow may prefer partial parsing to complete parsing because the space for the temporary file and the processing at origin nodes is saved, and prefer server-based algorithms to nonservers algorithms because the space and the processing at the intermediate nodes are saved. For example, SP may be more suitable than SC in this environment. Systems where network bandwidth is high may benefit from partial parsing because the overhead of writing the temporary file and merging runs at origin nodes can be avoided, and may benefit from server-based algorithms because the distribution information can be obtained quickly, and the overhead at intermediate nodes is avoided.

Improvements can be made to partial parsing algorithms if we transfer inverted lists in nonblocking fashion. From the experiments, we find that the cost of forming the batch buffer of size 30 MB, until flushing it, is approximately 280 seconds while the cost of transferring data from the batch varies between 60 and 97 seconds. As we can see, with the batch buffer of size 30 MB, the time the parser uses to form the buffer is greater than the transfer time. This gives an opportunity to improve the performance by hiding the cost of transfer in the parsing cost.

## 4.6 Conclusions

In this chapter, we propose an inversion algorithm for inverting large text collections where the index does not fit in main memory. A storage organization based on list files is proposed to support in-place updates of the index. This technique is evaluated by indexing a 5.3 GB collection as a whole, versus as a sequence of batch updates, to study the incremental update performance. The results show that indexing a collection of this scale as a whole takes 14 hours and 14 minutes with 2 GB of disk space required for the temporary file.

Next, we investigate four scalable algorithms (SC, SP, NC, and NP) for building hybrid partitioned inverted indices. These distributed algorithms are extended from the inversion algorithm developed earlier. They also can operate in incremental update mode. Each algorithm is a unique combination of two design policy decisions: the inversion policy and the posting transfer policy. Based on experimentation, we discuss the performance of these algorithms and the tradeoffs among them. We are able to show that the SC algorithm is the fastest algorithm in this environment, followed by NC, SP, and NP respectively. The SC algorithm inverts a 16 GB collection in about six and a half hours with 8 nodes (Pentium II, 233 MHz) and a 10BaseT Switched Ethernet network. The results also show that every algorithm scales well. They can invert larger collections with performance scaling roughly linearly in the number of nodes.



# Chapter 5

## Conclusions and Future Work

There are three major objectives behind the work described in this dissertation:

- develop an inverted index partitioning scheme suitable for large-scale digital libraries;
- develop an inversion technique which supports incremental updates;
- develop distributed inversion techniques to create parallel inverted indices, partitioned according to the scheme satisfying the first objective.

Guidelines for designing inverted indices were developed from the study of characteristics of text collections and queries, reported in Chapter 2. The study was conducted with a standard information retrieval test collection (TREC-8). The properties examined include: the term-frequency distribution, the query selection distribution, and the query length. It was found that an inverted list is accessed in a sequential fashion. The frequencies of occurrence of terms in the vocabulary widely vary, from one to millions of occurrences. The majority of terms in the vocabulary occur very few times, and only a very small portion of terms accounts for most of the occurrences in the collection. It also was found that the inverted lists of more frequently occurring terms tend to expand more rapidly with the arrival of new documents than those of less frequently occurring terms.

The relationship between the frequency of occurrence of a term and its frequency of use in queries is not simple. Many patterns were observed in this study. However, the overall pattern suggests that inverted lists of more frequently occurring terms are accessed more frequently than are those of less frequently occurring terms. Query transaction logs from MARIAN, a digital library system, were used for studying the query length. The study indicates that a query spans from 1 to 18 terms in length, and on average a query consists of 3.74 terms.

From the findings described above, a design guideline for inverted indices can be summarized as follows: (1) it is best to store a list as a contiguous sequence of bytes; (2) we should provide

efficient access and incremental updates for terms based on their frequency of occurrence; (3) the performance improvement may benefit from caching inverted lists based upon term frequency, and (4) upon the arrival of new documents, not only the existing inverted lists in the index will expand, but new lists also will be added into the index. In addition to studying the properties of the collection, the mathematical models for these properties are discussed.

For parallel inverted indices, effective data partitioning across the nodes in the system is crucial to the performance of the indices. It heavily affects the distribution of load across the system. Inspired by the skewness of the length of inverted lists and the variety of their patterns of use, we realize that document and term partitioning approaches perform well in different conditions. In Chapter 3, the hybrid inverted index partitioning scheme which combines the document and term partitioning approaches to partitioning inverted indices is purposed. A simulation environment is built, driven by the mathematical models developed in Chapter 2. Simulation experiments with a terabyte of text indicate the superiority of the hybrid partitioning method over the document and term partitioning methods.

Having shown the superiority of the hybrid partitioning scheme, in Chapter 4, we present algorithms to generate inverted indices, partitioned according to this scheme. A requirement of indices for digital libraries is the ability to expand to accommodate the arrival of new documents. A scalable inversion algorithm which will function as the basis for the parallel algorithms is developed. This technique is examined by indexing 5.3 GB standard test collection as a whole and as a sequence of batch updates to study the incremental update performance. The results show that indexing 5.3 GB of text as a whole takes 14 hours and 14 minutes and 2 GB of space required for the temporary file.

Four parallel inversion algorithms are proposed, extending the inversion technique, described above. Each of them is a unique combination of two design decisions: (1) the inversion approach (i.e., whether to use distribution server), and (2) the posting transfer approach (i.e., when the posting transfer takes place during the inversion). Each algorithm has distinct tradeoffs of how much time is spent in each step of the algorithm. A design call Server-Based Inversion with Complete Parsing (or SC) is shown to have the best performance, as well as scalability. It indices a 16 GB collection in  $6 \frac{1}{2}$  hours with 8 nodes in a 10 Mbps Switched Ethernet network.

As seen from the summary given above, all the objectives have been achieved. Next we discuss future work that extends the capability or improves the performance of the indices.

A custom performance model should be developed for PetaPlex in order to predict more accurately how applications, not just indices, will perform and where the bottlenecks are in this environment which can lead to early detection of problems and recommendations to modify either software (e.g., Boolean retrieval) or hardware components.

Improvements can be made to parts of the index. In the current design, operations (such as accessing and updating inverted lists) are applied to the portion of index stored at a node in

a sequential manner. A mechanism can be provided to improve the performance by allowing lists to be retrieved and/or updated concurrently while maintaining the consistency of the index, which is partitioned across the system. MacFarlane *et al.* [35] present some work in this issue on a uniprocessor system.

As indicated by the query selection characteristic, larger inverted lists tend to be accessed more often than smaller ones. Caching techniques can be applied to parts of the index to improve the performance according to this trend. However, it may not be feasible to cache inverted lists of very frequently used terms because of their sizes. Lookup structures are prominent candidates to apply caching. For example, the records in the list allocation table of a few very frequently used terms can be cached for fast look up, or the table itself can be ordered nonincreasingly by term frequency.

Compression can be applied to improve the time to access inverted lists and invert documents, as well as to reduce the amount of memory and disk space required by the index. A difficulty for applying compression to dynamic collections is how to include newly added postings to existing lists which already have been compressed in an efficient manner. Work by Ziviani *et al.* [81] is related to this issue.

Software systems built atop a distributed memory environment are vulnerable to the non-availability of nodes in the system. While a node is inaccessible, data (local or shared across the system) that is maintained at that node becomes unavailable. In the case that the node crashes, the data in memory at that node that has not been flushed to the disk can be lost. Extending the index to be fault-tolerant is another crucial improvement.

Another type of future work that can be carried out is evaluating the performance of this index organization against various requirements of digital libraries. A benchmark of this kind is proposed by Esler and Nelson [54].

A key result of this dissertation is an inverted index which employs the parallelism inherent in the cluster of workstations for efficient access to inverted lists and supports the dynamicity of the collection. It can be used as an index for an information retrieval system. Its design is extensible to support various IR models with minor modification. This research not only proposes techniques that can be used in general distributed, shared-nothing environments but also lays the ground work for further migration of other components of digital libraries onto the PetaPlex environment.

# Bibliography

- [1] R. M. Akscyn. The PetaPlex project, January 2001. <http://www.ks.com/vt/50.html>.
- [2] W. Y. Arms, C. Blanchi, and E. A. Overly. An Architecture for Information in Digital Libraries. *D-Lib Magazine*, February 1997. <http://www.dlib.org/dlib/february97/cnri/02arms1.html>.
- [3] R. Baeza-Yates, E. Barbosa, and Z. Ziviani. Hierarchies of indices for text searching. In *Proceedings of RIAO'94 Intelligent Multimedia Information Retrieval Systems and Management*, pages 11 – 13, 1994.
- [4] R. Baeza-Yates and B. Ribeiro-Neto. Introduction (chapter 1). In R. Baeza-Yates and B. Ribeiro-Neto, editors, *Modern Information Retrieval*. AWL England, 1999.
- [5] R. Baeza-Yates and B. Ribeiro-Neto. Modeling (chapter 2). In R. Baeza-Yates and B. Ribeiro-Neto, editors, *Modern Information Retrieval*. AWL England, 1999.
- [6] E. W. Brown. *Execution Performance Issues in Full-Text Information Retrieval*. PhD thesis, University of Massachusetts at Amherst, October 1995.
- [7] E. W. Brown. Parallel and distributed IR (Chapter 9). In R. Baeza-Yates and B. Ribeiro-Neto, editors, *Modern Information Retrieval*. AWL England, 1999.
- [8] F. Can, E. Fox, C. Snavely, and R. France. Incremental clustering for very large document databases: Initial MARIAN experience. *Information Systems*, 84:101–114, 1995.
- [9] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [10] J. K. Cringean, R. England, G. A. Manson, and P. Willett. Network design for the implementation of text searching using a multicomputer. *Information Processing & Management*, 27(4):265 – 283, 1991.
- [11] J. K. Cringean, R. England, G. A. Manson, and P. Willett. Parallel text searching in serial files using a processor farm. In *Proceedings of the 13th Annual International ACM*

- SIGIR Conference on Research and Development in Information Retrieval*, pages 429 – 453, Chicago, IL USA, 1991.
- [12] W. B. Croft and P. Savino. Implementing ranking strategies using text signatures. *ACM Transactions on Office Information Systems*, 6(1):42–62, January 1988.
- [13] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the Thirteen International Conference on Research and Development in Information Retrieval (SIGIR'90)*, pages 405–412, Brussels Belgium, 1990.
- [14] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, October 1984.
- [15] C. Faloutsos and H. V. Jagadish. Hybrid index organizations for text databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings of the Third International Conference on Extending Database Technology – EDBT'92*, pages 23 – 27, Vienna, 1992. Springer-Verlag.
- [16] C. Faloutsos and H. V. Jagadish. On B-tree indices for skewed distributions. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 363–374, Vancouver, British Columbia, Canada, 1992.
- [17] The International DOI Foundation. The Digital Object Identifier System, January 2001. <http://www.doi.org>.
- [18] C. Fox. Lexical analysis and stoplists (chapter 7). In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.
- [19] E. A. Fox, R. K. France, E. Sahle, A. Daoud, and B. E. Cline. Development of a modern OPAC: from REVTOLC to MARIAN. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 248–259, Pittsburgh, PA, 1993.
- [20] E. A. Fox and W. C. Lee. FAST-INV: A fast algorithm for building large inverted files. Technical Report TR-91-10, VPI&SU Department of Computer Science, March 1991.
- [21] E. A. Fox and L. F. Lunin. Introduction and overview to perspectives on digital libraries. *Journal of the American Society for Information Science*, 44(8):480 – 491, September 1993.
- [22] E. A. Fox and O. Sornil. Digital libraries (Chapter 15). In R. Baeza-Yates and B. Ribeiro-Neto, editors, *Modern Information Retrieval*. AWL England, 1999.
- [23] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

- [24] W. Francis and H. Kucera. *Frequency Analysis of English Usage*. New York: Houghton Mifflin, 1982.
- [25] G. W. Furnas, S. Deerwester, S. T. Dumais, T. K. Landauer, R. A. Harshman, L. A. Streeter, and K. E. Lochbaum. Information retrieval using a singular value decomposition model of latent semantic structure. In *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 465 – 480, Grenoble France, 1988.
- [26] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 1999.
- [27] F. Grandi, P. Tiberio, and P. Zezula. Frame-sliced partitioned parallel signature files. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 286–297, Copenhagen Denmark, 1992.
- [28] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee. Inverted files (chapter 3). In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.
- [29] B.-S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.
- [30] J. R. Jump. *YACSIM: Reference Manual*. Rice University, version 2.1 edition, March 1993. <http://www.crpc.rice.edu/softlib/rppt.html>.
- [31] R. Kahn and R. Wilensky. A Framework for Distributed Digital Object Services. Technical Report cnri.dlib/tn95-01, CNRI, May 1995. <http://www.cnri.reston.va.us/k-w.html>.
- [32] D. E. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley Publishing Company, Inc., 1973.
- [33] A. M. Law and W. D. Kelton. *Simulation Modeling & Analysis*. New York: McGraw-Hill, 2 edition, 1991.
- [34] A. M. Law and S. Vincent. *ExpertFit: User's Guide*. Averill M. Law & Associates, Tucson, AZ USA, 1995.
- [35] A. Macfarlane, S. E. Robertson, and J. A. McCann. On concurrency control for inverted files. In *Proceedings of the 18th Annual BCS Colloquium on Information Retrieval*, Manchester Metropolitan University, 1996.
- [36] A. Macfarlane, S.E. Robertson, and J.A. McCann. Parallel computing in information retrieval - an updated review. *Journal of Documentation*, 53(3):274–315, June 1997.
- [37] E. L. Miller. *Hierarchy Management for Scientific Computing*. PhD thesis, University of California, Berkeley, 1995.

- [38] J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.
- [39] G. Navarro. Indexing and searching (chapter 8). In R. Baeza-Yates and B. Ribeiro-Neto, editors, *Modern Information Retrieval*. AWL England, 1999.
- [40] M. J. Nelson. *Probabilistic Models for the Simulation of Bibliographic Retrieval Systems*. PhD thesis, University of Western Ontario, 1982.
- [41] M. J. Nelson. Correlation of term usage and term indexing frequencies. *Information Processing & Management*, 24(5):541–547, 1988.
- [42] Y. Ogawa, T. Morita, and K. Kobayashi. A fuzzy document retrieval system using the keyword connection matrix and a learning method. *Fuzzy Sets and Systems*, 39:163 – 179, 1991.
- [43] G. Panagopoulos and C. Faloutsos. Bit-sliced signature files for very large text databases on a parallel machine architecture. In *Proceedings of the 4th International Conference on Extending Database Technology (EDBT)*, pages 379–392, Cambridge, UK, March 1994.
- [44] C. Pogue, E. Rasmussen, and P. Willett. Searching and clustering of database using the ICL distributed array processor. *Parallel Computing*, 8:399–407, October 1988.
- [45] C. Pogue and P. Willett. Use of text signatures for document retrieval in a highly parallel environment. *Parallel Computing*, 4:259–268, June 1987.
- [46] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
- [47] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill Publishing Company, 1994.
- [48] E. Rasmussen. Parallel information processing. In *Annual Review of Information Science and Technology, Volume 27*, pages 99–130. Medford, NJ: American Society for Information Science, 1992.
- [49] S. F. Reddaway. High speed text retrieval from large databases on a massively parallel processor. *Information Processing & Management*, 27(4):311 – 316, 1991.
- [50] B. Ribeiro-Neto and R. Muntz. A belief network model for IR. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 253 – 260, Zurich, Switzerland, 1996.
- [51] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the 3rd ACM Conference on Digital Libraries*, pages 182–190, 1998.

- [52] B. A. Robeiro-Neto, E. S. Moura, M. S. Neubert, and N. Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of ACM SIGIR'99*, pages 105–112, Berkeley, CA USA, 1999.
- [53] S. E. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Sciences*, 27(3):129 – 146, 1976.
- [54] M. L. Nelson S. L. Esler. NASA indexing benchmarks: Evaluating text search engines, 1997. <http://techreports.larc.nasa.gov/ltrs/PDF/1997/jp/NASA-97-jnca-sle.pdf>.
- [55] G. Salton. *The SMART Retrieval System – Experiments in Automatic Document Processing*. Prentice Hall Inc., Englewood Cliffs, NJ, 1971.
- [56] G. Salton. *A theory of indexing*. Philadelphia: Society for Industrial and Applied Mathematics, 1975.
- [57] G. Salton and C. Buckley. Term-weighting approaches in automatic retrieval. *Information Processing & Management*, 24(5):513–523, 1988.
- [58] G. Salton, E. A. Fox, and H. Wu. Extended boolean information retrieval. *Communications of the ACM*, 26(11):1022 – 1036, 1983.
- [59] G. Salton and M. E. Lesk. Computer evaluation of indexing and text processing. *Journal of the ACM*, 15(1):8 – 36, January 1968.
- [60] D. B. Skillicorn. A generalization of indexing for parallel document search. Technical report, Queen’s University, Ontario Canada, 1995.
- [61] R. Srinivasan. Parallel searching in distributed databases. *Computer Networks*, 4:157 – 166, 1980.
- [62] C. Stanfill and B. Kahle. Parallel free-text search on the Connection Machine system. *Communications of the ACM*, 29(12):1229 – 1239, 1986.
- [63] J. M. Tague and M. J. Nelson. Problems in the simulation of bibliographic retrieval systems. In R. N. Oddy et al., editor, *Information Retrieval Research*, pages 236–255. London: Butterworths, 1981.
- [64] J. M. Tague and M. J. Nelson. Simulation of bibliographic retrieval databases using hyperterms. In G. Salton and H. J. Schneider, editors, *Research and Development in Information Retrieval*, pages 194–208. Berlin: Springer-Verlag, 1983.
- [65] N. Talagala, S. Asami, T. Anderson, K. Lutz, and D. Patterson. Tertiary disk: Large scale distributed storage. Technical Report UCB/CSD-98-989, University of California at Berkeley, 1996.



- [66] LAM Team. LAM/MPI parallel computing. <http://www.mpi.nd.edu/lam/> (January 2001).
- [67] The PURL Team. Persistent Uniform Resource Locator (PURL), February 2001. <http://purl.oclc.org>.
- [68] A. S. Tomasic. *Distributed Queries and Incremental Updates in Information Retrieval Systems*. PhD thesis, Princeton University, June 1994.
- [69] A. S. Tomasic and H. Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proceedings of SIGMOD'93*, Washington, D.C., May 1993.
- [70] A. S. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of PDIS'93*, San Diego, CA USA, 1993.
- [71] A. S. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. Technical Report STAN-CS-TN-93-1, Stanford University, 1993.
- [72] A. S. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 1994.
- [73] H. Turtle and W. B. Croft. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems*, 9(3):187 – 222, July 1991.
- [74] E. M. Voorhees and D. K. Harman. NIST special publication: The 8th Text REtrieval Conference (TREC-8), November 1999. [http://trec.nist.gov/pubs/trec7/t8\\_proceedings.html](http://trec.nist.gov/pubs/trec7/t8_proceedings.html).
- [75] G. Wiederhold. *File Organization For Database Design*. New York: McGraw-Hill, 1987.
- [76] R. Wilkinson and P. Hingston. Using the cosine measure in a neural network for document retrieval. In *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 202 – 210, Chicago, IL USA, October 1991.
- [77] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.
- [78] D. Wolfram. Applying informetric characteristics of databases to IR system file design, part I: Informetric models. *Information Processing & Management*, 28(1):121–133, 1992.
- [79] S. Wolfram. *The Mathematica Book*. New York : Cambridge University Press, 3rd edition, 1996.

- [80] G. K. Zipf. *Human behavior and the principle of least effort*. Addison-Wesley, Reading MA, 1949.
- [81] N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, November 2000.
- [82] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, December 1998.
- [83] J. Zobel, A. Moffat, and R. Sack-Davis. Storage management for files of dynamic records. In *Proceedings of 4th Australian Database Conference*, pages 26 – 38, Brisbane, Australia, 1993.

## Appendix A

# Similarity Between the Inverted File Partitioning Problem and the Partial Parallel Searchability Problem

In the Partial Parallel Searchability (PPS) problem [61], the database consists of segment types  $S = \{s_1, s_2, \dots, s_n\}$ . Given a set of queries  $Q$ , where each query requires different segment types of data from  $S$ , find an optimal distribution of  $S$  in a network of  $m$  nodes such that the number of searches to answer all queries once is minimum.

By transformation, the PPS problem can be shown equivalent to the Inverted Index Partitioning problem, defined in Section 4, as follows: We can view the set of segment types  $S$  as the set of postings  $P$  and view the network nodes as disk nodes. Then we can easily see that the PPS problem is equivalent to the Inverted Index Partitioning problem.

# Vita

Ohm Sornil was born in Bangkok, Thailand. He received a B.Eng. in Electrical Engineering (Second Class Honours) from Kasetsart University in March 1993 and an M.S. in Computer Science from Syracuse University in January 1997. He completed his Ph.D. in Computer Science at Virginia Polytechnic Institute and State University in January 2001. He has accepted a teaching position with Information Systems Education Center, The National Institute of Development Administration in Bangkok, Thailand.