DISTRIBUTED QUERIES AND INCREMENTAL UPDATES IN INFORMATION RETRIEVAL SYSTEMS

Anthony Slavko Tomasic

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1994

To Francis William Tomasic's spirit, and the Road to Nowhere

Well we know where we're goin'
But we don't know where we've been
And we know what we're knowin'
But we can't say what we've seen
And we're not little children
And we know what we want
And the future is certain
Give us time to work it out

We're on a road to nowhere
Come on inside
Takin' that ride to nowhere
We'll take that ride

Lyrics by David Byrne of The Talking Heads from the album *Little Creatures*

Abstract

The proliferation of the world's "information highways" has renewed interest in efficient document indexing techniques. This thesis explores the architecture of information retrieval systems for querying and indexing documents. Distributed queries are studied with analytical and trace-driven simulations. We focus on physical index design, inverted index caching, and database scaling in a distributed system. All three issues influence response time and throughput. Incremental updates of inverted lists are studied using a new dual-structure index data structure. This index structure separates long and short inverted lists dynamically and optimizes the retrieval, update, and storage of each type of list. To study the behavior of the index, engineering trade-offs are described that favor either update time or query performance. We explore these trade-offs quantitatively by using actual data and hardware and simulation to determine the best algorithm under a variety of criteria. Finally, implementation of our incremental update algorithms is compared to an existing information retrieval system.

Acknowledgments

Stanford Professor Hector Garcia-Molina, my thesis advisor, provided encouragement, support, and guidance. Thanks, we made a great team, hombre.

Kurt Shoens of Teknekron Software Systems collaborated on the the running of experiments described in Chapter 5 and on subsequent work described in reference [STGM94].

Stanford graduate student Luis Gravano collaborated on the distributed database selection problem described in reference [GGMT94].

Bo Parker and Norman Roth of the Stanford Data Center made the INSPEC traces available. These traces made Chapter 4 possible.

The Princeton thesis readers, Professors Kai Li and David Hanson, provided comments and criticisms. The thesis was much improved by their reading. However, any mistakes or omissions in this thesis are the author's responsibility.

Princeton University and the Department of Computer Science allowed me to complete my thesis at Stanford University. Stanford University and the Department of Computer Science made my stay at Stanford University an easy one.

Stanford students Yet-Fong Cheong, Brian Lent and John Petit participated in studies of information retrieval systems.

Thanks to Masahiko Saito, Mendel Rosenblum, Miron Livny, Sam DeFazio, Sergio Plotkin, and Tak Yan for comments and suggestions on drafts of various papers.

Benjamin (Chi-Ming) Kao, office mate of five years and fellow cross country adventurer, provided the good times and the Cantonese philosophy.

The research in Chapters 3 and 4 was partially supported by the Advanced Research Projects Agency (ARPA) of the Department of Defense under Contract No. DABT63-91-C-0025. The research in Chapters 5 and 6 was sponsored by ARPA under Grant No. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of ARPA, the U. S. Government or CNRI.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Full-text document databases of newspaper articles, journals, legal documents etc. are readily available. These databases are increasing in size rapidly as the cost of digital storage drops, as more source documents appear in electronic form, and as optical character recognition becomes commonplace. At the same time, there is a rapid increase in the number of users and queries submitted to such text retrieval systems. One reason is that more users have computers, modems, and communication networks available to reach the databases. Another is that as the volume of electronic data grows, it becomes more important to have effective search capabilities.

As the data volume and query processing loads increase, companies that provide information retrieval services are turning to distributed storage and searching. The goal of this thesis is to study distributed query processing, various distributed index organizations for information retrieval, and the incremental update of index organizations used in information retrieval.

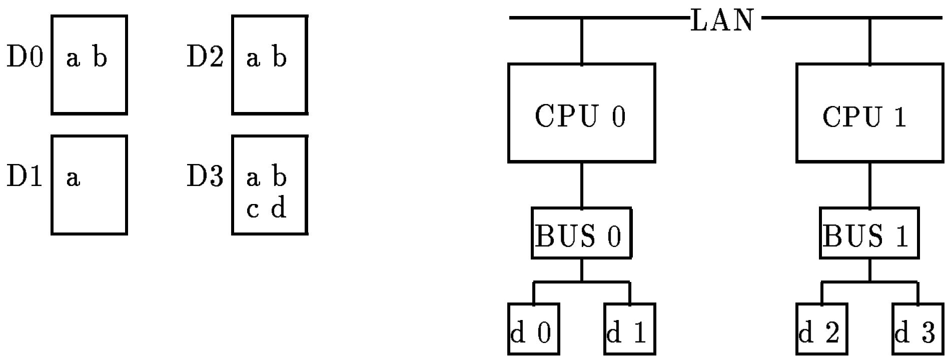

Figure 1: A example set of four documents and an example hardware configuration.

1.1 Distributed Query Processing

A simple example helps motivate the issues that are addressed. The left hand side of Figure 1 shows four sample documents, D0, D1, D2, D3, that could be stored in an information retrieval system. Each document contains a set of words (the text), and each of these words (maybe with a few exceptions) are used to index the document. In Figure 1, the words in our documents are shown within the document box, e.g., document D0 contains words a and b.

To find documents quickly, full-text document retrieval systems traditionally build *inverted lists* [Fed87, Knu73] on disk (see reference [Fal85] for a survey of access methods for text). For example, the inverted list for word b would be b: (D0,1), (D2,1), (D3,1). Each pair in the list is a *posting* and indicates an occurrence of the word (document id, position). Position can be word position or byte offset. To find documents containing word b, the system needs to retrieve only this list. To find documents containing both a and b, the system could retrieve the lists for a and b and intersect them. The position information in the list is used to answer queries involving distances, e.g., find documents where a and b occur within so many positions of each other.

Suppose that we wish to store the inverted lists on a multiprocessor like the one

Index	Disk	Inverted Lists in *word: (Document, Offset)* form
Disk	d 0	a: (D0, 0); b: (D0, 1)
	d 1	a: (D1, 0)
	d 2	a: (D2, 0); b: (D2, 1)
	d 3	a: (D3, 0); b: (D3, 1); c: (D3, 2); d: (D3, 3)
Host, I/O bus	d 0	a: (D0, 0), (D1, 0)
	d 1	b: (D0, 1)
	d 2	a: (D2, 0), (D3, 0); c: (D3, 2)
	d 3	b: (D2, 1), (D3, 1); d: (D3, 3)
System	d 0	a: (D0, 0), (D1, 0), (D2, 0), (D3, 0)
	d 1	b: (D0, 1), (D2, 1), (D3, 1)
	d 2	c: (D3, 2)
	d 3	d: (D3, 3)

Table 1: The various inverted index organizations for Figure 1.

shown on the right in Figure 1. This system has two processors (CPUs), each with a disk controller and I/O bus. (Each CPU has its own local memory.) Each bus has two disks on it. The CPUs are connected by a local area network. Table 1 shows four options for storing the lists. The host and I/O bus organizations are identical in this example because each CPU has only one I/O bus.

In the *system* index organization, the full lists are spread evenly across all the disks in the system. For example, the inverted list of word b discussed above happened to be placed on disk $d1$.

With the *disk* index organization, the documents are logically partitioned into four sets, one for each disk. In our example, we assume document D0 is assigned to disk $d0$, D1 to $d1$, and so on. In each partition, we build inverted lists for the documents that reside there. To answer the query "Find all documents with word b" we must retrieve and merge 4 lists, one from each disk. Since disk $d1$ contains no documents with word b, its b list is empty.

In the *host* index organization, documents are partitioned into two groups, one for each CPU. Here we assume that documents D0, D1 are assigned to CPU 0, and D2, D3 to CPU 1. Within each partition we again build inverted lists. The lists are then uniformly dispersed among the disks attached to the CPUs. For example, for CPU 1, the list for a is on $d2$, the list for b is on $d3$, and so on.

The *I/O bus* index organization follows the same partitioning principal as the other index organizations, except at the I/O bus level. Documents are partitioned into two groups, one for each I/O bus. Within each partition inverted lists are built and uniformly dispersed among the disks attached to the I/O buses. In our example, this organization results in the same as the host index organization since each host has exactly one I/O bus. If a host has more than one I/O bus, then the host index organizations and I/O bus index organizations would differ.

1.1.1 Query processing

Query processing under each index organization is quite different. For example, consider the query "Find documents with words a, c", and say the query initially arrives at CPU 0. Under the system index organization, CPU 0 would have to fetch the list for a, while CPU 1 would fetch the c list. CPU 1 would send its list to CPU 0, which would intersect the lists. With the host index organization, each CPU would find the matching documents within its partition. Thus, CPU 0 would get its a and c lists and intersect them. CPU 1 would do likewise. CPU 1 would send its resulting document list to CPU 0, which would merge the results. With the disk index organization, CPU 0 would retrieve the a and c lists from disk $d0$, and would also retrieve the a, c lists from disk $d1$. CPU 0 would obtain two lists of matching documents (one for each disk), would merge them, and then merge the combined list with the list from CPU 1.

There are many interesting trade-offs among these storage organizations. With the system index organization, there are fewer I/Os. That is, the a list is stored in a single place on disk. To read it, the CPU can initiate a single I/O, the disk head moves to the location, and the list is read. (this may involve the transfer of multiple blocks). In the disk index organization, on the other hand, the a list is actually stored on four different disks. To read these list fragments, 4 I/Os must be initiated, four heads must move, and four transfers occur. However, each of the transfers is roughly a fourth of the size, and they can take place in parallel. So, even though we are consuming more resources (more CPU cycles to start more I/Os, and more disk seeks), the list may.

The system index organization may save disk resources, but it consumes more resources at the network level. Notice that in our example, the entire c list is transferred from CPU 1 to CPU 0, and these inverted lists are usually much longer than the document lists exchanged under the other schemes. However, the long inverted list transfers do not occur in all cases. For example, the query "Find documents with a and b" (system index organization) does not involve any such transfers since all lists involved are within one computer. Also, it is possible to reduce the size of the transmitted inverted lists by moving the shortest list. For example, in our "Find documents with a and c", we can move the shorter list of a and c to the other computer.

The performance of each strategy depends on many factors, including the expected type of queries, the optimizations used for each query processing algorithm, whether throughput or response time is the goal, the resources available (e.g., how fast is the network, how fast are disk seeks). In this thesis, we discuss the options for index organization and parallel query processing. We also present results of detailed simulations and attempt to answer some of the key performance questions. Under what conditions are each index organization better? How does each index organization scale up to large systems (more documents, more processors)? What is the impact of key parameters? For instance, how would a system with optical disks function? How well do the algorithms and hardware scale as the database size grows? As mentioned above, current data collections are growing rapidly, and it is unclear what index organization scales best, or whether it is more important to add disk or processor or communication resources as the database grows. What is the impact of caching inverted lists in main memory? Is there enough locality of reference between queries to make caching worthwhile?

1.2 Incremental Updates

Traditional information retrieval systems, of the type used by libraries (e.g., Stanford University's Socrates or the University of California's MELVYL) or information vendors (e.g., Dialog Inc. or Mead Data Central Inc.), assume a relatively static body of documents. Given a body of documents, these systems build the inverted list index from scratch, laying out each list sequentially and contiguously to others on disk. They also built a B-tree that maps each word to the locations of its list on disk. Periodically, e.g., every weekend, new documents are added to the database and a brand new index is built. Rebuilding the index is a massive operation, but its cost is amortized over multiple days of operation.

In many of today's environments, such full index reconstruction is infeasible. One reason is that text document databases are more dynamic. For instance, if one is indexing news articles, electronic mail, or stock information, the latest information is required. Thus, one would like to update the index in place, as new documents arrive. Updating the index for each *individual* arriving document is inefficient, as we discuss in Chapter 5. Instead, the goal is to batch together small numbers of documents for each in-place index update. To maintain access to the batch, it can be searched simultaneously with the larger index.

A second reason why in-place updates are of interest is that they eliminate (or at least postpone) resource consuming reorganizations. Massive reorganizations may be acceptable in conventional systems where user load is minimal over weekends, but in today's world of 7 days a week, 24 hours a day continuous operation, degradation of service for prolonged periods is unacceptable.

A third reason why in-place updates may be desirable is that the index may simply be too massive for reorganization. As the volume of documents grows in some applications, it may be better to have a dynamic index that can grow and migrate to new disk drives without ever being fully reorganized.

In spite of the natural attractiveness of in-place index updates, little is known about their implementation options or their performance. As far as we know, systems that implement in-place updates typically use relatively naive strategies that may be inefficient. For example, any time a Wide Area Information Server (WAIS) [KMD+92] index needs to grow an inverted list, it copies the whole list to a new disk area, leaving no free space at the end for future updates. Perhaps it would be more effective to leave some space, and to make additions that fit in that space? If multiple disks are available, can we stripe large lists across multiple disks to improve performance? Inverted lists vary tremendously in size: the ones for frequently occurring words can be huge, but there may be many that have only a few postings. What is the most effective layout of these lists to make their updates efficient? Which layouts lead to less disk space utilization? To better query performance?

Although we do not answer all these questions fully, this thesis makes the following contributions.

- A new dynamic dual-structure data structure for inverted lists. Lists are initially stored in a "short list" data structure; as they grow they migrate to a "long list" data structure. Our proposed algorithm selects lists to migrate dynamically.

- A family of disk allocation policies for long lists. Each policy dictates, among other things, where to find space for a growing list, whether to try to grow a list in place or to migrate all or parts of it, how much free space to leave at the end of a list, and how to partition a list across disks.

- A detailed performance evaluation of the dual-structure lists and the various allocation policies. The evaluation is based on a collection of 64 days worth of NetNews that are indexed according to our algorithms. Our experimental system generates the exact sequence of disk block updates that each policy produces; this sequence is then executed on an IBM Risc System 6000 Model 350 computer with 3 disks to measure the update time. Based on the resulting disk layout, we also compute disk space utilization and estimate query performance.

We do not consider fault tolerance. We assume that the hardware is reliable. However, to be fair in estimating and comparing the I/O costs of various policies, we periodically flush to disk all the data and directory information for each policy. In addition, the algorithms and data structures are constructed so that the incremental update of the index can be restarted if it is aborted. Fault tolerance is an important area for future research.

Finally, we construct an implementation of the update algorithms of Chapter 5. Our implementation is termed an *alternative storage technology* (AST). The implementation modifies the underlying storage technology of an implementation of the information retrieval system WAIS [KMD+92] known as freeWAIS [FRE92]. The modifications retain the document parsing part of WAIS and replace the inverted index of WAIS with our data structures and algorithms. From the point of view of a user of WAIS, the two implementations are identical since they return the exact same set of answers and the answers take essentially (if the correct update policy is used) the same amount of time to compute. From the point of view of an administrator of a WAIS database, AST provides flexibility of update algorithms.

1.3 Thesis Organization

Chapter 2 reviews previous work. In Chapter 3, we study full-text retrieval using an analytic workload model. Section 3.1 describes our hardware scenario, query processing algorithms, and physical index organization in more detail. To study performance we need to model various key components such as the inverted lists, the queries, and the answer sets. Although there has been much work on information retrieval systems, models that are appropriate for studying parallel query execution have not been developed. Section 3.2 defines simple models for these and other critical components, Section 3.3 describes the simulation, and Section 3.4 presents our results and comparisons. Part of the work in this chapter also appears in reference [TGM93b].

Chapter 4 studies an abstracts database as opposed to a full-text system. In a full-text system, every word occurrence is indexed. In an abstracts system, only the abstract is indexed. If we compare two systems with the same *number* of documents, the index in the full-text case will be much larger. Even if the volume of raw data is equal the inverted lists for the abstracts case will still be smaller because repeated words are indexed in the full-text case only. For instance, if a word appears 10 times in a document, there will be 10 index entries (pointing to each occurrence) in the full-text case, and only one entry in the abstracts case. As we will see, the fact that inverted lists are shorter for abstracts dramatically changes the relative performance of the various organizations. Part of the work in this chapter also appears in reference [TGM93a].

Our evaluation in Chapter 4 is based on query traces from the FOLIO library information retrieval system at Stanford University, run against a detailed event-driven simulation of the hardware and query processing. This study represents the first time that an actual user query trace drives the evaluation of a distributed architecture in information retrieval. Furthermore, the traces also give the result sizes, so we can use that in our simulation. In Chapter 3, we model queries probabilistically, assuming query terms were picked at random from a vocabulary. This model gives us flexibility and a rapid method for preliminary results. Clearly, using traces yields more realistic results at the expense of flexibility and speed in determining results. It also lets us study caching issues.

In Chapter 5, we describe the dynamic dual-structure for inverted lists. We describe a model for the various allocation policies of inverted lists that reside on disk and evaluate these policies in an experimental design. Part of the work in this chapter also appears in reference [TGMS94]. In addition, we have extrapolated our results to larger synthetic text document databases and describe the results in reference [TGMS93].

In Chapter 6, we describe our implementation of an information retrieval system. The implementation is based on modifications to an existing information retrieval system. Our modifications are based on our work in Chapter 5. We compare the two implementations for various scenarios and show that incremental updates are faster than the other implementation.

Chapter 2

Previous Work

For an introduction to full-text document retrieval and information retrieval systems, the reader is referred to reference [Sal89]. An *information retrieval model* (IRM) defines the interaction with an information retrieval system and consists of three parts: a *document representation*, a *user need* and a *matching function*. For this thesis, we consider the *boolean* IRM and the *vector* IRM.

The boolean IRM is provided by most existing commercial information retrieval systems. Its document representation is the set of words that appear in each document. Typically, each word is also *typed* to indicate if it appears in the title, abstract, or some other field of the document. The boolean IRM user need is represented by a boolean *query*. A query consists of a collection of pairs of words and types structured with boolean operators. For example the query *title information and title retrieval or abstract inverted* contains three pairs and two operators. The matching function of a query in the boolean IRM is boolean satisfiability of a document representation with respect to the query.

The vector IRM is popular in academic prototypes for information retrieval systems and has recently gained commercial acceptance. Its document representation is the set of words that appear in each document and an associated *weight* with each word. The weight indicates the "relevance" of the corresponding word to the document. Thus, a document is represented as a vector. A vector IRM user need is represented by another vector (this vector can be extracted from a document or a set of words provided by a user). The matching function computes the similarity between the user need and the documents. Thus, all the documents can be ranked with respect to the similarity. Typically, the topmost similar documents are returned to the user as an answer. There is much research on the assignment of weights to words and on the effectiveness of various matching functions for information retrieval. However, both the boolean IRM and the vector IRM and associated variation of these models can be computed efficiently with inverted lists. Reference [TC92] surveys information retrieval models.

The focus of information retrieval research is to develop IRMs that provide the most *effective* interaction with the user. Our focus is to provide the most *efficient* interaction with the user in terms of response time, throughput and other measures, regardless of which IRM is used.

In the design of full-text document retrieval systems, there is a basic trade-off between the time to process the document database and the time to process queries. Broadly speaking, the more time spent processing the document database (i.e. building indexes) the less time is spent processing queries. In some scenarios (such as government monitoring of communication), a tremendous amount of information must be queried by only a few queries. In this case, time spent indexing is wasted and linear searching of documents is more efficient. Work in this area concentrates on hardware processors for speeding up the scanning of text [Hol92]. More typically, indexing the documents is worthwhile because the cost can be amortized across many queries. We consider only these latter systems in this thesis.

Emrath's thesis [Emr83] explores this trade-off between query and update time by providing a data structure that can be tuned in the amount of information indexed.

Essentially, the database is partitioned into equal sized "pages." A page is a fixed number of words located together in a document. Duplicate occurrences of words are dropped within a page. If the page is large, many duplicates are dropped from the index, speeding up indexing time. If the page is small, few duplicate words are dropped, slowing down indexing time. For certain applications this tuning of the data structure works well (we use this technique in a simple form in Chapter 4). We explore the trade-off between query and update time in Chapter 5.

Much research has gone into designing data structures for indexing text. Faloutsos [Fal85] is a survey of this issue. One approach is the use of *signature schemes* (also known as superimposed coding) [Knu73]. Here, each word is assigned a random (hashed) k-bit code of an n-bit vector – for example the word "information" might correspond to bit positions 10 and 20 of a 2 kilobyte vector. Each document is represented by an n-bit vector containing the union of all the k-bit codes of all the words in the document. Queries are constructed by producing an n-bit vector of the k-bit codes of the words in the query. Matching is performed by comparing a query against the document vectors in the database. This scheme is because the signatures of documents can be constructed in linear time. Unfortunately, the matching process produces "false drops" where different words or combinations of words are mapped into the same k-bit codes. One approach is to ignore false drops and inform the user that some additional documents may be returned. We do not consider this approach further. Otherwise, each document in the result of the matching process must be checked for false drops. While the number of false drops can be statistically controlled for the average case, the worst-case behavior of this data structure implies checking *every* document in the database for some queries, which is prohibitively expensive for large document collections. Lin [Lin91] describes a signature scheme where multiple independent signatures are used to control false drops and to improve parallel performance. In this thesis we always insure that queries can be answered without examining the text of any documents.

Another data structure is PATRICIA trees and PAT arrays [FBY92, GBYS91]. Here, the database is represented as one *database string* by placing documents end-to-end. A tree is constructed that indexes the semi-infinite strings of the database string. A semi-infinite string is a substring of the database string starting at some point and proceeding until it is a unique substring. This data structure has been used in practice for the constructed of the New Oxford English Dictionary. The query time, indexing time, and storage efficiency are approximately the same as inverted lists. The techniques described in this thesis can be applied to this data structure.

For commercial full-text retrieval systems, inverted files or inverted indexes [Fed87, Knu73] are typically used. An example of inverted lists is provided in Chapter 1. Note that the information represented in each posting (each element of an inverted list) varies depending on the type of information retrieval system. For a boolean IRM full-text information retrieval system, the posting contains the document identifier and the position (as a byte offset or word offset from the beginning of the document) of the corresponding word. For a boolean IRM abstracts text information retrieval system, the posting contains the document identifier without a positional offset (since duplicate occurrences of a word in a document are not represented in these systems). For a vector IRM full-text or abstracts information retrieval systems the posting contains the document identifier and a weight. All of the above systems can be typed. In this case, the type system can be encoded by setting aside extra bits in each posting to indicate which fields the word appears in the document. Other methods of representing the type information are also used. As the information retrieval model becomes more complicated, more information is typically placed in each posting. We accommodate various sizes of inverted lists in our models.

The inverted index for a full-text information retrieval system is very large – typically equal in size to the raw text. The original documents (minus punctuation) can be reconstructed from the inverted index. Inverted lists are a prime candidate for compression [Wei90, ZMSD92]. We accommodate compression of inverted lists in our models and study the impact of various compression ratios on the performance of query processing.

In reference [BCCM93], several experiments are reported on the performance of an information retrieval system built on top of a persistent object store. The IRM for this system is a probabilistic one based on Bayesian inference. Essentially, a standard B+-tree implementation is compared to a persistent object store. Performance improvement is due to the caching of objects. We study the caching issue in detail in Chapter 4.

A general interest in the performance of text document retrieval systems has led to a standardization effort for benchmarking of commercial systems [DeF93]. This standardization was developed independently of this thesis and offers another model for the generation of a query workload. This model is complementary to the analytic model of Chapter 3, the trace-based model of Chapter 4, and the synthetic model of Chapter 5.

2.1 Distributed Query Processing

Various distributed and parallel hardware architectures can be applied to the problem of information retrieval. A series of papers by Stanfill studies this problem for a Connection Machine. In reference [SK86], signature schemes are used. A companion paper by Stone [Sto87] argues that inverted lists on a single processor are more efficient. In reference [STW89], inverted lists are used to support parallel query processing. The algorithm presented there is similar to our system index organization in that keywords are assigned to processors. Finally, in reference [Sta90], an improvement of the previous paper based on the physical organization of inverted lists is described. The improvement essentially improves the alignment of processors to data.

An implementation of vector IRM full-text information retrieval is described in reference [AS91] for the POOMA machine. The POOMA machine is a 100-node, 2-d mesh communication network where each node has 16 MB of memory and a processor. One out of five nodes has an ethernet connection and one half of the nodes have a local disk. The implementation partitions the documents among the processors and builds a local inverted index of the partition. This approach is similar to the disk index organization of this thesis, however there are two processors per disk. In our architecture processors never outnumber disks. This paper cites a 2.098 second estimated query response time for a 191-term query on a database of 136,020 documents with a 20-node machine.

Some preliminary experimental results are reported in reference [CEMW90] for a 16 processor farm (Meiko Computing Surface). The vector IRM is used here and a signature scheme is used as the data structure. Unfortunately, the database has only 6,004 documents and the query workload only 35 queries. This experimental framework is too small to produce reliable results.

The performance of some aspects of query and update processing of an implementation of a boolean IRM full-text information retrieval is discussed in reference [DH91] for a symmetric shared-memory multiprocessor (Sequent). We use some of the figures cited in this paper, such as the number of instructions in the inner loop of an inverted list intersection algorithm, as a bases for some parameter values in our experimental work. The physical index organization of this paper corresponds to the host index organization of this thesis. In addition, we use statistics published in reference [CD90] as the basis for our analytic model of Chapter 3.

The work on document retrieval in multiprocessor shared-memory systems considers physical index organization issues for those architectures. While some issues for these systems are not considered here, the issue of physical organization is an important one and that the prefetch algorithms presented in Chapter 3 probably perform well on shared-memory architectures.

In the analysis of query processing, a query can be divided into three parts: parsing the query, matching the query against the database, and retrieving the documents in the answer. Parsing consumes few resources and is typically the same for all information retrieval systems. Retrieving of documents offers some interesting issues (such as placement of the documents) but again few resources are needed. We do not consider parsing or document retrieval in this thesis. Burkowski [Bur90] examines the performance problem of the interaction between query processing and document retrieval and studies the issue of the physical organization of documents and indices. His paper models queries and documents analytically and simulates a collection of servers on a local-area network, as we do in Chapter 3. Our work is complementary to this paper.

In reference [JO92], independently from our work, an analytic comparison of the disk index organization and system index organization is described for share-everything multiprocessors. There are several differences between this paper and the work in Chapter 3. One issue is the modeling of the B-tree look up, which maps each word in a query to its associated inverted list. The modeling of this paper analytically computes the disk cost of this lookup, whereas in this thesis we assume that the lookup is an in-memory operation. This difference can account for a difference of about 1 I/O operation per query word. A second difference is the skew in the distribution of query terms. Reference [JO92] assumes the skew is the same as the occurrence of words in documents. In some special situations in information retrieval, where documents are used as queries in the vector IRM, this assumption is valid. However, it does not apply in general to vector or boolean IRMs. A third difference is the distribution of words in documents. The referenced paper assumes a uniform distribution; even the authors admit this assumption is simplistic. We measure the distribution of terms directly in Chapter 3. Finally, this paper models the document fetching stage of processing a query, which we do not consider.

The reference [MMN86] presents a discussion of the architecture issues in implementing the IBM STAIRS information retrieval system on a network of personal computers. The index organization is the host index organization. However, this paper argues for the physical distribution of inverted lists across multiple machines when the size of a single database is larger than the storage capacity of a node on the network. This idea is essentially a special case of striping of inverted lists. We consider striping in detail in Chapter 3.

Schatz [Sch90] describes the implementation of a distributed information retrieval system. Here, performance improvements come from changing the behavior of the interface to reduce network traffic between the client interface and the backend information retrieval system. These ideas are complementary to this thesis. Three improvements are offered. First, summaries of documents (or the first page) are retrieved instead of entire documents. This scheme reduces the amount of network traffic to answer an initial query and shortens the time to present the first result of a query, but lengthens the time to present the entire answer. Second, "related" information such as document structure definitions are cached to speed up user navigation through a set of documents. Third, the contents of documents (as opposed to summaries) are prefetched while the user interface is idle.

2.2 Incremental Updates

Cutting and Pedersen [CP90] consider incremental updates of inverted lists where a B-tree is used to organize the vocabulary. Updates are optimized by storing short inverted lists directly in the B-tree. In our framework, this optimization can be represented by a very small bucket for approximately each word in the text document database. However, using fewer, larger, buckets offer better performance [TGMS93]. In addition, our scheme dynamically determines if an inverted list is stored in a fixed-size structure or a variable length one as opposed to a static division. Cutting and Pedersen also described a buddy system for the allocation of long lists. This approach deserves further experimental study.

Faloutsos and Jagadish [FJ92b] analyze the physical organization of long list. They study three methods that correspond to some schemes presented in Chapter 5. Performance comparisons between our work and the schemes presented there are difficult since updates are not batched in that paper.

In another work, Faloutsos and Jagadish [FJ92a] extensively analyze a dual-structure scheme based on signature schemes for long lists and inverted lists for short lists. The division in the structure is static as opposed to a dynamic scheme presented here. In addition, using inverted lists for short lists is computationally expensive since many I/O operations, each containing only a few postings, are required to update this structure.

Zobel, Moffat and Sacks-Davis [ZMSD92] consider several issues in inverted file indexing. The compression methods presented there complement this thesis. They also consider fixed-size buckets for storing inverted lists but do not discuss techniques for handling long lists. To compare approaches, a linear scaling of the 45 minutes update time of 132 MB of documents cited in the paper to the 686 MB text document database used here gives an update time of about 233 minutes. Halving this number to 116 minutes accounts for improvements in processor performance. Our study predicts a range of index build times from about 43 minutes to 173 minutes depending on the policy used.

Fox and Lee [FBY92] describe an interesting and entirely different approach based on preprocessing of document representations and a merge update of inverted lists.

A non-incremental update time of 1 minute 14 seconds for 8.68 MB of documents appears in this article. Harman and Candela [HC90] also describe an update method and cite an indexing time of 313 hours for 806 MB of documents on a computer with six Intel 80386 processors. Finally, our own measurements for freeWAIS version 0.202 on a DEC 5000 Model 240 (32 MB memory) with an external disk (Western Scientific) on a SCSI-I bus shows that to index 82.9 MB of our experimental text document database requires 84.1 minutes using ULTRIX V4.2A (Rev. 47) operating system.

Chapter 3

Distributed Queries – Analytic Workload

In this chapter we study full-text information retrieval. Section 3.1 describes our definitions and framework. Section 3.2 describes analytic workload, an inverted list model based on experimental data, and a probabilistic model of query processing. Section 3.3 describes the simulation of query processing and hardware. Section 3.4 describes the results.

3.1 Definitions and Framework

Documents contain *words*. The set of all words occurring in the database is the *vocabulary*. For convenience, we name words by their occurrence rank, e.g., word 0 is the most frequently occurring word, word 1 is the next most frequent, and so on. In the example of Figure 1, the vocabulary is $\{\ a,\ b,\ c,\ d\ \}$; word 0 is a, word 1 is b, etc. We use the word and the rank of the word interchangeably.

A query retrieves documents satisfying a given property. In this chapter, we concentrate on "boolean and" queries of the form $a \wedge b \wedge c \ldots$, which find the documents

Parameter	Value	Description
Hosts	4	Number of Hosts
I/OBusesPerHost	4	Number of Controllers and I/O Buses per Host
DisksPerI/OBus	2	Number of Disks for each I/O bus

Table 2: Hardware configuration parameter values and definitions.

containing all the listed words. The words appearing in a query are termed *keywords*. Given a query $a \wedge b \ldots$ the document retrieval system generates the *answer set* for the document identifiers of all the documents that match the query. A *match* is a document that contains the words appearing in the query.

We focus on boolean-and queries because they are the most primitive ones. For instance, a more complex search such as $(a \wedge b)$ OR $(c \wedge d)$ can be modeled as two simple and-queries whose answer sets are merged. A distance query "Find a and b occurring within x positions" can be modeled by the query $a \wedge b$ followed by comparing the positions of the occurrences.

Hardware Configuration We consider hardware organizations like the one in Figure 1 but we vary the number of CPUs or *hosts*, the number of I/O controllers per host, and the number of disks per controller. Table 2 lists the parameters that determine a configuration. The column "Value" in the table refers to the "base case" used in our simulation experiments (Section 3.4). That is, our experiments start from a representative configuration and explores the impact of changing the values. The base case does not represent any particular real system; it is simply a convenient starting place.

Physical Index Organization The inverted index can be partitioned and fragmented in many ways. We study a single natural division by hardware. This division does not require any unusual hardware or operating system features. The documents are uniformly distributed across all disks d in the system; $d = Hosts \cdot I/OBusesPerHost \cdot DisksPerI/OBus$. The disks are numbered from 0 to $d-1$ as in Figure 1.

The inverted index organization is compared for four mutually exclusive cases. In the *disk* index organization, an inverted index is constructed for all words in the documents residing on each disk. Thus, for a given word, there are d inverted lists containing that word. In the *I/O bus* index organization, an inverted index is constructed for all the documents on the disks attached to the same I/O bus. In the *host* index organization, an index is constructed for all the documents on a single host. Lists are distributed by host in a similar manner. Finally, in the *system* index organization a single index is constructed for all documents. Table 1 illustrated these index organizations, but note that in that example the I/O bus and host index organizations are identical because hosts have a single I/O bus. The same amount of data is stored in the system regardless of the index organizations and for any query the same amount of data is read from disk.

In any of the organizations, if an index spans x disks, we assume the lists are dispersed over the x disks. In particular, the list for word w is placed on the disk $i + (w \bmod x)$, where i is the first disk in the group. For example, for the host index organization in Table 1, one of the indices spans disks $d0$, $d1$; the second spans $d2$, $d3$. For the second index, the list for a (word 0) goes to $d2$, the list for b (word 1) goes to $d3$, the list for c (word 3) goes to $d2$, and so on.

Algorithms For all configurations except the system one, queries are processed as follows. The query $a \wedge b ...$ is initially processed at a *home* site. That site issues *subqueries* to all hosts; each subquery contains the same keywords as the original query. A subquery is processed by a host by reading all the lists involved, intersecting them, and producing a list of matching documents. The answer set of a subquery, termed the *partial answer set*, is sent to the home host, which concatenates all the partial answer sets to produce the answer set.

In the system index organization, the subquery sent to a given host contains only the keywords that are handled by that host. If a host receives a query with a single keyword, it fetches the corresponding inverted list and returns it to the home host. If the subquery contains multiple keywords, the host intersects the corresponding lists, and sends the result as the partial answer set. The home host intersects (instead of concatenates) the partial answer sets to obtain the final answer.

As mentioned in Chapter 1, the algorithm for the system index organization can be improved. Here we describe three optimizations, called *Prefetch I, II* and *III.* These are heuristics; in some cases they may not actually improve performance.

In the Prefetch I algorithm, the home host determines the query keyword k that has the shortest inverted list. We assume that hosts have information on keyword frequencies; if not, Prefetch I is not applicable. In phase 1, the home host sends a single subquery containing k to the host that handles k. When the home host receives the partial answer set, it starts phase 2, which is the same as in the un-optimized algorithm, except that the partial answer set is attached to all subqueries. Before a host returns its partial answer set, it intersects it with the partial answer set of the phase 1 subquery, which reduces the size of the partial answer sets that are returned in phase 2.

The Prefetch II algorithm is similar to Prefetch I, except that in phase 1 we send the subquery with the largest number of keywords. We expect that as the number of keywords in a subquery increases, its partial answer set decreases in size. Thus, the amount of data returned by the one host that processes the phase 1 subquery should be small. If two or more subqueries have the same number maximum of keywords, Prefetch II selects one of them at random.

Prefetch III combines the I and II optimizations. That is, the first subquery contains the largest number of keywords, but if there is a tie, the subquery with the shortest expected inverted lists is selected. Intuitively, one would expect Prefetch III to perform the best. However, we chose to study all three techniques (Section 3.4) to understand what each optimization contributes. In particular, Prefetch I and III require statistical information on inverted list sizes. Our results tell us if it is worthwhile to keep such information, i.e., if the improvement of Prefetch III over II, which does not require this information, is significant.

To illustrate these optimizations, consider the query $a \wedge b \wedge c \wedge d$ in the example of Figure 1 (system index organization). With Prefetch I, the subquery d would be sent to host CPU 1 in phase 1. (Of the four keywords, d occurs less frequently in the database, and it is stored in host CPU 1.) In phase 2, the subquery $a \wedge b$ would be sent to CPU 0, together with the list for d from phase 1. CPU 1 would receive the query c together with the d list. With Prefetch II, the first subquery would be either $a \wedge b$ (to CPU 0) or $c \wedge d$ (to CPU 1), selected at random. Prefetch III would select $c \wedge d$ as the first subquery because it involves shorter lists.

Striping Striping [PGK88] is a method to decrease the response time and increase the throughput to read an inverted list by allocating the blocks of an inverted list *horizontally* across several disks (by using modular arithmetic) and reading the blocks in parallel. For example, suppose we have four blocks $b0$, $b1$, $b2$, $b3$ that store an inverted list for a word z located on the disk $d1$ of three disks $d0$, $d1$, $d2$. In the normal case, all four blocks would be vertically allocated and would reside on disk $d1$. Striping word z across these three disk places $b0$ on $d1$, $b1$ on $d2$ (since the blocks are allocated horizontally), $b2$ on $d0$ and $b3$ on $d1$.

We can stripe an inverted list under any index organization. In the host index organization, the inverted list would be striped across all the disks on the host. In Table 1 suppose the inverted list for word a was striped with one entry per block.

(This assumption simplifies the example; in practice, many entries are stored per block.) For CPU *0*, the entry (D0,0) would be on $d0$ and the (D1,0) would be on $d1$. Similarly for CPU 1, (D2,0) would be on $d2$ and (D3,0) would be on $d3$.

In the I/O bus index organization, the inverted list would be striped across all the disks on the I/O bus. In the disk index organization, striping has essentially no effect, since there is only one disk for each index so vertical and horizontal block allocation result in the same physical allocation for any inverted list.

In the system index organization, the natural approach would be to stripe across all the disks. However, this approach complicates query processing: the blocks of an inverted list must be fetched from multiple hosts and assembled at some particular host before processing on that list can continue. Thus, we choose to stripe a system index organization inverted list only across the disks on the host that the inverted list resides. In Table 1, the inverted list for word a in the system index organization would be striped across all the disks on CPU *0*. Thus, $d0$ would hold (D0,0)(D2,0) and $d1$ would hold (D1,0)(D3,0). This method avoids the complications and still provides the advantage that the inverted list for a word is located in only one host.

Striping does not always improve response time for reading an inverted list. To understand the circumstances in which striping is an advantage, suppose s is the disk overhead time for a read and l is the time needed for the read of an inverted list. Ignoring any queuing delays or contention, the response time to read a list from disk is $s + l$. If the list is striped over k disks, the response time ranges roughly from $s + l/k$ best-case to $sk + l$ worst case when the reads are processed sequentially. Thus, under best-case conditions, striping improves response time when $s + l/k < s + l$. The additional work required for a striped read is $s(k - 1)$, which must be kept small to minimize the impact of striping on throughput. Given the range of values for these variables in our model, short inverted lists generally do not benefit from striping. Section 3.4 reports the effect of striping the longer inverted lists for all the index organizations. This case is studied by varying the fraction of the vocabulary that have striped inverted lists.

Suppose we added 100,000 documents to Figure 1. First, in the disk organization, the lengths of the inverted list for a word a would vary slightly from disk to disk, due to the variation in the number of times that the word occurs in the documents for each disk. (This variation is ignored in this study.) Second, internal fragmentation occurs for each inverted list for the word a on each disk. In the host index organization, all the inverted lists on that host for the word a are collected together and striped across the disk. Thus internal fragmentation occurs only at the end of that single inverted list.

The additional internal fragmentation that appears in the disk organization has a small impact on response time and throughput. Thus, controlling the number of striped inverted lists is similar to controlling the number of inverted lists that have a disk index organization. As the number of words with striped inverted lists approachs the entire vocabulary, performance for any index organization should approach the performance of the disk index organization.

3.2 Models

There are two choices for representing documents and queries in a simulation. One is to use a real document base and an actual query trace. The second is to generate synthetic databases and queries, from probability distributions that are based on actual statistics. Using a particular database and query trace is more realistic, but limits one to a particular application and domain. Using synthetic data offers more flexibility for studying a wide range of scenarios. Here we use synthetic data because it is better for exploring options and tradeoffs. We study a particular application in Chapter 4.

Parameter	Value	Description
D	667260	the number of documents
W	12000	words per document
V		the set of words appearing in documents, the vocabulary
T	1815322	total words in V i.e. $\mid V \mid = T$
$\mathcal{Z}(j)$	$Z(j)$	$\Pr(word = j)$, a probability distribution

Table 3: Parameters of the document model.

3.2.1 Document Model

A document is modeled by the parameters in Table 3. The database is a collection of D documents. Conceptually, each document is generated by a sequence of W independent and identically distributed trials. Each trial produces one word from the vocabulary V. Each word is represented uniquely by an integer w in the range $1 \leq w \leq T$ where $T = \mid V \mid$. The probability distribution $\mathcal{Z}$ describes the probability that any word appears. For convenience, the distribution is arranged in non-increasing order of probability i.e. $\mathcal{Z}(w) \geq \mathcal{Z}(w+i),\ \forall i > 0$. The "Value" column in Table 3 represents the base case. In this case, the values are from a legal document base described in reference [CD90].

To construct a specific probability distribution Z of $\mathcal{Z}$, we fit a curve to the rank/occurrence distribution of the vocabulary of a legal documents database [CD90] and then normalized it to a probability distribution. Figure 2 shows the log/log graph of two curves that have been fit to some of the $100,000$ most frequently occurring words. The X axis is the distinct words in the database, ranked by the number of occurrences in non-increasing order. The Y axis is the number of occurrences of each word. A diamond symbol marks the number of occurrences of a word. The curve labeled "linear" is the result of fitting a linear equation and the curve labeled "quadratic" is the fit of a quadratic equation [Wol91].

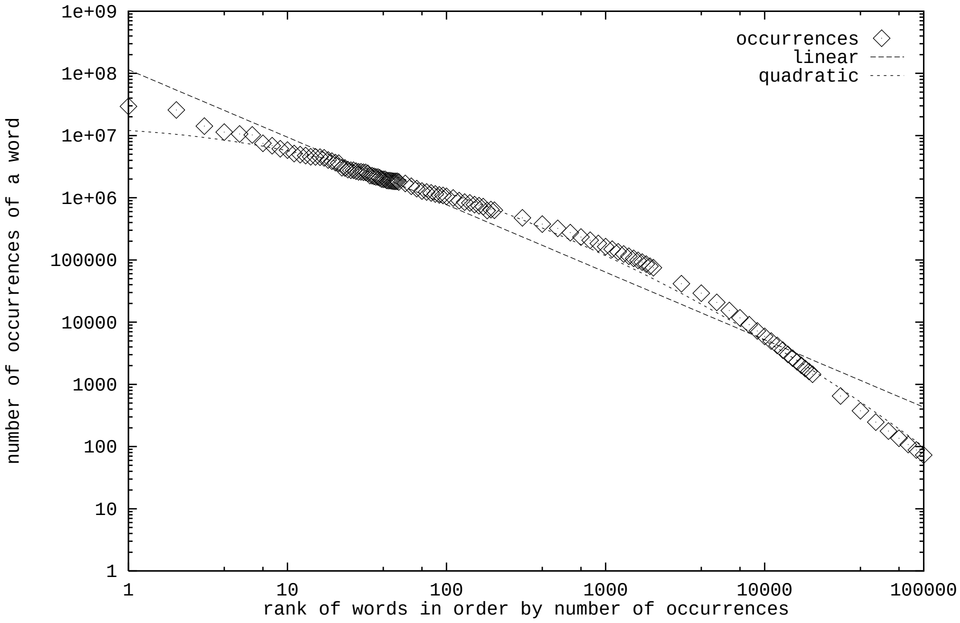

Figure 2: Curve fit to vocabulary occurrence data.

Given the quadratic fit curve, the form of the probability distribution Z is derived in Appendix A as

$$Z(j) = \frac{j^{-0.0752528 \ln j - 0.150669} e^{16.3027}}{8.47291 \times 10^{8}} \tag{1}$$

where the denominator is a normalization constant. Although our distribution is similar to Zipf's [Zip49], ours matches the actual distribution better. See Appendix A.

3.2.2 Query Model

A *query* is a sequence of words $(w_1, \ldots, w_K)$ generated from K independent and identically distributed trials from the probability distribution $Q(j)$. See Table 4 for a list of the parameters and base values chosen.

We now construct a specific probability distribution Q. There is little published data on this distribution, and there is no agreement on its shape (see reference [DeF92]

Parameter	Value	Description
K	5	number of keywords in a query
$\mathcal{Q}(j)$	$Q(j)$	$\Pr(word = j)$, a probability distribution
u	1%	fraction of T (in rank order of V) appearing in a query
V'		the u fraction of V
$\mathcal{S}$	V'^K	set of possible queries. $\mathcal{S} = V' \times \cdots \times V'$

Table 4: Parameters for the query model.

for a different model). It does not follow the same distribution as the vocabulary (Figure 2) because relatively infrequent words are often used in queries, so the uniform distribution was chosen for Q. This distribution makes it easy to understand of the impact of the distribution on performance. However, we found that the uniform distribution across the entire vocabulary gave far too much weight to the most infrequently occurring words (the tail of Figure 2). For example, these tail words are often misspellings that only appear once in the entire database and never appear in queries. Thus, in the Q distribution we cut off the most infrequent words by introducing a parameter u that determines the range of the distribution:

$$Q(k) = \begin{cases} \frac{1}{uT} & 1 \leq k \leq uT \\ 0 & \text{otherwise} \end{cases}$$

As u decreases, the probability of choosing a word of low rank in a query increases. Words of low rank occur often in the database. Thus, the expected number of documents to match a query increases since each word of the query occurs often in the database. Hence, if u is too small, queries tend to have answer sets that are a large fraction of the database. On the other hand, if u is too large, answer sets are unrealistically small. To estimate a good value for u, in Appendix B we compute the expected number of documents that match a query of length K for various values of u. The Q distribution has two other advantages. Since the distribution is simple, the

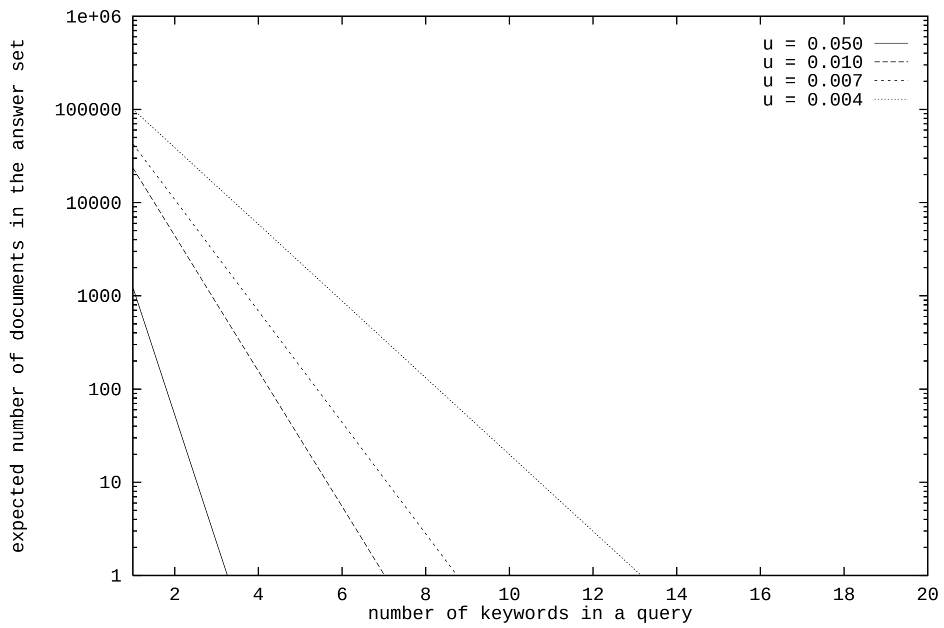

Figure 3: The Expected Number of Documents in an Answer Set for any Query.

impact of the distribution and consequently the impact of the work load on the system can be readily understood. (Section 3.4, for example, varies u and shows the impact on performance.) Secondly, this distribution may favor very long inverted lists since common words (such as "for") are part of the distribution. Thus, this simulation is a worst-case scenario.

Using the parameter values in Table 3 and Equation 1, Figure 3 show the function Z for the various values of K and u. In the base case, the number of keywords in a query is 5, so we examine the graph at the X axis value of 5. The value of $u = 0.01$ was chosen as the base value since it indicates about 25 documents on the average are found per query. In this case the fraction uT of the vocabulary includes 96.3% of the cumulative keyword occurrences in reference [CD90]. Section 3.4 details the response time sensitivity to uT of the various index organizations.

3.2.3 Answer Set Model

At various points in the simulation we need to know the expected size of a query answer set or partial answer set. Consider a particular query (or subquery) with keywords $w_1, \ldots, w_K$. Say this query is executed on a body of documents of size *Documents*. Under the system index organization $Documents = D$ where D is the total number of documents. But, for the other organizations, *Documents* is the number of documents covered by the index (or indexes) used by the particular subquery. The expected number of documents that match the query is

$$Documents \cdot [1 - e^{-WZ(w_1)}] \cdots [1 - e^{-WZ(w_K)}], \tag{2}$$

The term $[1 - e^{-WZ(w_1)}]$ is the probability that a document contains keyword w_1. Equation 2 is similar to Equation 4 in Appendix B, except that here we are looking at a specific query instead of averaging over all possible queries.

3.2.4 Inverted List Model

The inverted list contains a sequence of elements each of which describes a single appearance of the word. Each element contains a document identifier and a word offset of the word in the document. Thus, the inverted index is essentially a one-to-one mapping to the documents.

The expected number of occurrences of a word in a document is $\mathcal{Z}(w) \cdot W$. Thus, the expected number of entries of the corresponding inverted list is

$$Z(w) \cdot W \cdot Documents \tag{3}$$

where $Z(w)$ is the value of Equation 1 for the word w.

Parameter	Value	Description
DiskBandwidth	10.4	Mbits/sec Bandwidth of the disk
DiskBuff	32768	Size of a disk buffer in bytes
BlockSize	512	Number of bytes per disk block
SeekTime	6.0	Disk seek-time in ms
BufferOverhead	4.0	Cost to seek one track in ms
I/OBusOverhead	0.0	Cost of each I/O bus transfer in ms
I/OBusBandwidth	24.0	Mbits/sec Bandwidth of the I/O bus
LANOverhead	0.1	Cost of each *LAN* transfer in ms
LANBandwidth	10.0	Mbits/sec Bandwidth of the *LAN*

Table 5: Hardware parameter values and definitions.

3.3 Simulation

To study the index organizations and query algorithms, we implemented a detailed event-driven simulation using the DeNet [Liv90] simulation environment. In this section we describe important aspects of the simulation. Tables 5 and 6 describe the base parameters used.

3.3.1 Hardware

The system model consists of several hosts each with a CPU and memory, several I/O buses per host and several disks per I/O bus. The hosts are connected by a local-area network. See Table 5 for the parameters and base values that describe the the hardware configuration. The values for the disk and I/O bus portions of this table are from reference [Che90]. The hosts have parameter values that correspond to a typical workstation. Figure 1 shows an example hardware configuration.

Parameter	Value	Description
CPUSpeed	1	Relative speed of each CPU
Multiprogram	4	Multiprogramming level *per Host*
QueryInstr	50000	Query start up CPU cost
SubqueryInstr	10000	Subquery start up CPU cost
SubqueryLength	1024	Base size of subquery message
FetchInstr	5000	Disk fetch start up CPU cost
MergeInstr	10	Merge CPU cost per byte of a decompressed inverted list
UnionInstr	1	Concatenation CPU cost per byte of an answer set
Decompress	10	Decompression CPU cost per byte of inverted list on disk
AnswerEntry	4	Bytes to represent an entry in an answer set
EntrySize	10	Bits to represent an inverted list entry on disk
Compress	0.5	Compression Ratio
Stripe	0.0	Fraction of query words that have a striped inverted list

Table 6: Base case parameter values and definitions.

3.3.2 Inverted Lists and Answer Sets

In our simulation, we do not generate a synthetic document base *a priori*. Instead, when we require the length of the inverted list for a word w, we use the expected length of the list. Thus, the length in disk blocks of an inverted list is

$$InvertedList(w) = \lceil \frac{(Z(w) \cdot W \cdot Documents \cdot EntrySize \cdot Compress/8.0)}{BlockSize} \rceil$$

where $Z(w) \cdot W \cdot Documents$ is from Equation 3, *EntrySize* is the average number of bits used to represent an entry in the inverted list, 8.0 converts from bits to bytes, *BlockSize* is a parameter representing the size of a block on disk and *Compress* models the efficiency of the inverted list compression scheme. This compression scheme model assumes a linear reduction in the size of the inverted list. One simple way to accomplish an approximately linear reduction is use *delta encoding.* A list is sorted the difference between an two consecutive entries stored in a packed format. More sophisticated compression schemes [ZMSD92] result in better, nonlinear, compression ratios. The *BlockSize* parameter permits studying the effect of internal fragmentation.

If the inverted list for a word is striped, the predicate

$$w < Stripe \cdot u \cdot T$$

is true. Thus, if $Stripe = 0.0$ then no words have striped lists and if $Stripe = 1.0$ all words (which can appear in a query) have striped lists.

To fetch the inverted list for a word w in the unstriped case, one disk fetch corresponds to the read of one invert list and each fetch request has a length determined by *InvertedList(w)*. In the striped case, the total length is the same, but one fetch is issued for each disk that contains part of the striped list. In both cases, processing for the query waits until all the fetches have completed for all the words appearing in the subquery on a host.

The length of an answer set, in bytes, is determined by multiplying Equation 2 by the length of an element of an inverted lists, *AnswerEntry* (see Table 6).

3.3.3 CPU simulation

The relative weight of all CPU parameters is controlled by the single parameter *CPUSpeed*. Thus, the rate of the CPU can be varied independently of individual factors contributing to the length of various CPU requests. The CPU is simulated by a *FCFS* infinite length queue server. The number of CPU instructions needed by each request is determined by the type of request.

1. Query start up is determined by *QueryInstr*).
2. Subquery start up is determined by *SubqueryInstr*).
3. Disk fetch is determined by *FetchInstr*).
4. Uncompression and merge of inverted lists is determined by

$$MergeInstr \cdot \sum_{w} InvertedList(w).$$

5. The union of subquery answer sets is determined by

$$UnionInstr \cdot AnswerList(w_1, \ldots, w_k).$$

The amount of CPU time required by each request is scaled by *CPUSpeed*.

3.3.4 Disk and I/O bus Simulation

A disk services fetch requests from a CPU and sends the results to an I/O bus. The disk is a *FCFS* infinite length queue. An I/O bus is simulated by a *FCFS* infinite length queue that services request from disks. The disk service time for a request is determined by four factors: the constant seek-time overhead, the track-to-track seek-time and overhead to load the disk buffer, the transfer time off of the disk, and the time needed to gain access to the I/O bus. The seek-time overhead for the read is determined by the parameters *SeekTime* and includes the average rotational delay. Every read has a fixed overhead determined by the the initial seek and the track-to-track seeks and overheads. This overhead is modeled by

$$SeekTime + (InvertedList(w)/DiskBuff) \cdot BufferOverhead$$

After the simulation of the seek and the seeks between buffer loads, the disk negotiates access to the bus by sending a BUS REQUEST message to the I/O bus node. The function transmit(x, y) gives the time in milliseconds required to transmit y at bandwidth x. If

$$\begin{aligned} a &= \text{transmit}(DiskBandwidth, InvertedList(w)) \\ b &= \text{transmit}(I/OBusBandwidth, InvertedList(w))) + I/OBusOverhead) \end{aligned}$$

then the BUS REQUEST messages is sent after $\max(0.0, a - b)$ units of time. This model simulates the overlap of the disk loading its track buffer and the transfer of data to the I/O bus. The disk then waits for a BUS GRANTED message. Then both the disk and the I/O bus are busy for b units of time. The disk and I/O bus are then both freed to service the next request in each respective queue.

Since an I/O bus services requests one at a time in the order of their arrival, all the disks attached to an I/O bus compete for access to it. In the case of a striped inverted list, the blocks of the inverted list that reside on disks of an I/O bus are read in parallel but must be transmitted through the I/O bus sequentially. However, if the inverted list spans more than one I/O bus, some of the blocks are transmitted to the host entirely in parallel, since the operations of difference I/O buses are independent.

3.3.5 LAN simulation

The system contains a single LAN that is simulated by a single *FCFS* infinite length queue that services subquery requests and answers Subquery requests have a length determined by parameter *SubQueryLength* and any additional answer set appended to the query (as is the case with the prefetch algorithms). Answer sets lengths are described in Section 3.2.3. The service time for a request is determined by

$$\text{transmit}(LanBandwidth, RequestLength) + LANOverhead$$

where *LanBandwidth* is a parameter. Subquery start up requests contend with answer set transmission, whereas disk fetch requests do not contend with fetch answers in I/O bus. Disk fetch requests are of a short, constant length and consume an insignificant fraction of the I/O bus bandwidth. However, subquery requests have variable length and consume a significant fraction of the network bandwidth when partial answer sets are transmitted. A request with identical source and destination hosts is not transmitted through the local-area network. Note that for simplicity, broadcast messages are not modeled and thus the query algorithms do not use this feature. In an implementation, broadcast messages could be used to reduce the cost of transmission of subqueries by a factor of the number of hosts because the transmission of the prefetch subquery to each individual host would be replaced by a single broadcast.

3.3.6 Query Simulation

A query, consisting of a set of words, is issued to a host. The parameter *Multiprogram* determines the number of simultaneous queries *per host* in the simulation. The host processes the query with the following steps.

1. A CPU burst computes query parsing and start-up.
2. Subqueries are sent to some or all hosts.
3. The process blocks and waits for the subqueries to finish.
4. A CPU burst merges the results of the subqueries.

Parameter	Value	Description
SimulateTime	100000	Maximum time of an experiment
ConfidenceInter	5%	The size of the confidence interval
ConfidenceLevel	90%	The confidence level used with the t statistic
BatchSize	100	The batch size of response time values

Table 7: Simulation parameter values and definitions.

Subqueries are transmitted to hosts by inserting the subquery in the *LAN* queue. When a subquery arrives at a host, it is processed by the following steps.

1. A CPU burst starts-up the parses the subquery.
2. A fetch request is issues for an inverted list to one or more disks for each word in the subquery.
3. The subquery blocks and waits for the fetches to finish.
4. A CPU burst computes the intersection of the fetched inverted lists.
5. The answer set is sent back to the query.

The answer is transmitted to the host cpu by inserting it in the *LAN* queue.

3.3.7 Simulation

The simulation tracks the system response time and when the confidence interval is less than *ConfidenceInterval* for a confidence level of *ConfidenceLevel* of this value over batches of size *BatchSize*, the simulation terminates. Otherwise, the simulation terminates after *SimulateTime*. The values of these variables are shown in Table 7. This method of terminating the simulation usually shortens the time taken for any simulation run. These features are provided by the DENET simulation programming language.

	Index Organization						
Metric	Disk	I/O bus	Host	System	P I	P II	P III
query response time (sec)	2.17	1.75	2.14	8.68	4.96	4.98	4.88
response time error (sec)	0.049	0.044	0.081	0.324	0.417	0.366	0.385
throughput (query/sec)	7.30	9.11	7.44	1.85	3.23	3.22	3.25
disk utilization (%)	86.1	76.7	44.3	13.1	24.9	24.3	26.1
I/O bus utilization (%)	18.5	28.0	37.7	21.9	30.5	28.7	31.1
CPU utilization (%)	43.9	60.9	48.8	21.9	35.7	34.7	35.4
LAN utilization (%)	23.3	29.7	24.3	94.7	29.9	16.1	10.9

Table 8: Results of all metrics for the base case simulation experiment (P I is Prefetch I, P II is Prefetch II and P III is Prefetch III).

3.4 Simulation Results

Table 8 presents the data collected from a simulation run on the base case (Tables 2 – 7). The metrics of query processing response time, the error in response time (90% confidence interval), query throughput, disk, I/O bus, CPU and *LAN* utilization were monitored for every simulation experiment. The amount of error in the response time was controlled to prevent misinterpretation of results. To avoid clutter, we omitted error bars on the graphs.

The table reveals that the disk, I/O bus, and host index organizations have comparable performance. Of the three, the disk organization performs somewhat worse because it has the highest disk utilization, leading to longer I/O delays. The I/O bus index organization has the best response time and throughput in this case. However, the host organization has the most balanced use of resources which leads to better performance under more stressful scenarios.

The system index organization, as well as the prefetch optimizations, perform poorly. The main reason why this index organization (without prefetch) does so poorly is that it saturates the LAN by transmitting many long inverted lists. The prefetch organizations reduce the amount of data sent over the LAN, and indeed the LAN utilization is much lower in these cases (see Table 8). Thus, the prefetch strategies perform substantially better than the simple system index organization. The saturation of the LAN depends heavily on the ratio of the bandwidth of the LAN to the average length of an inverted list. Other work [TGM93a] describes scenarios where the prefetch index organizations perform better than the disk, I/O bus, or host index organizations.

However, the prefetch strategies still perform substantially worse than the disk, I/O bus, and host organizations. The main reason is that there is less parallelism in the prefetch strategies than in the others. The first phase of the prefetch requires waiting for one part of the query to be completed. Furthermore, since lists are not split across disks, it takes longer to read them. These delays lead to lower throughputs in our closed system model. That is, in our model, each computer runs a fixed number of queries. If they take longer to complete, less work is done overall. The main advantage of the prefetch strategies is that less work is done per query (i.e., fewer disk seeks, I/O starts). However, in this scenario, these resources are not at a premium, so the advantages of prefetch do not show.

To our surprise, prefetch II and III actually preform essentially the same as prefetch I (see Table 8). Section 3.1 predicted that prefetch II and III would reduce the amount of data sent over the LAN, which is true as shown by the LAN utilization. However, the additional work done in phase one of prefetch II and III is preformed sequentially with respect to the rest of the processing of the query, leading to longer response times. Thus, only in cases where the LAN is a bottleneck would prefetch II and III pay off. So we show only the prefetch I results.

We now study how some of the key parameters affect the relative preformance of the index organizations. (We report only the more interesting results; many more experiments were performed than what can be reported here.) Figure 4 shows the sensitivity of response time to the value of uT. In this figure and in the rest of the

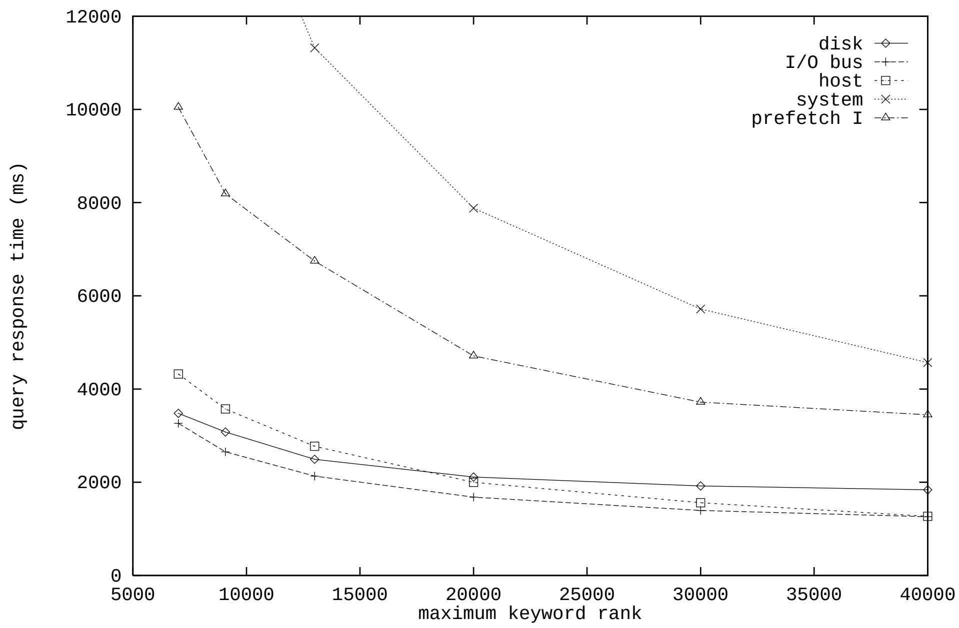

Figure 4: The sensitivity of response time to the maximum query keyword rank.

thesis, we connect our data points by lines as a visual aid to the reader. Recall that T is the size of the vocabulary and u is the fraction of the vocabulary that can appear in a query. Each line graphs the behavior of a different index organization. The line labeled prefetch is the prefetch I processing algorithm with a system index organization. The response times for each index organization decrease as uT increases because the number of word occurrences in the database for an average query word decreases. That is, as uT decreases, the inverted lists that have to be read increase in size. The disk and I/O bus organizations are relatively insensitive to uT because they distribute lists across many disks, i.e., the list fragments that need to be read grow at a slower rate. The system and prefetch curves are more sensitive to uT because entire inverted lists are read. The curve for the host organization is an intermediate case. Although not shown here, the effect of uT on throughput is similar.

A graph of the response time of the various configurations vs. the seek-time of

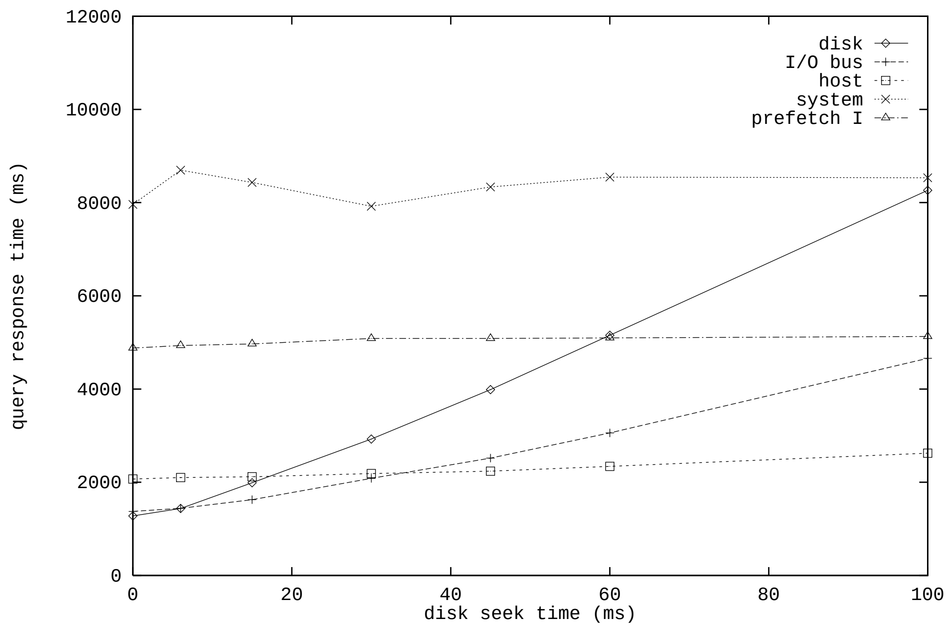

Figure 5: The sensitivity of response time to seek-time.

a disk in Figure 5 shows that the disk and I/O bus index organizations are most sensitive to the seek-time of the storage device. The disk index organization must retrieve for each query more inverted lists than any other organization. This same overhead is incurred by the I/O bus index organization to a lesser extent. The host index organization is insensitive to seek-time since only a few inverted lists must be retrieved per query.

Figure 5 indicates some potential for the host and prefetch index organizations if the storage devices are relatively slow (e.g. optical disks or a jukebox). It is important to note that our disk seek-time parameter captures the seek-time and other fixed I/O costs. For example, to get to the head of the inverted list, the system may have to go through a B-tree or other data structure. These additional I/O costs are modeled in our case by the "seek time." Furthermore, we are assuming that inverted lists (or fragments) are read with a single I/O. For longer lists there may be several I/Os in

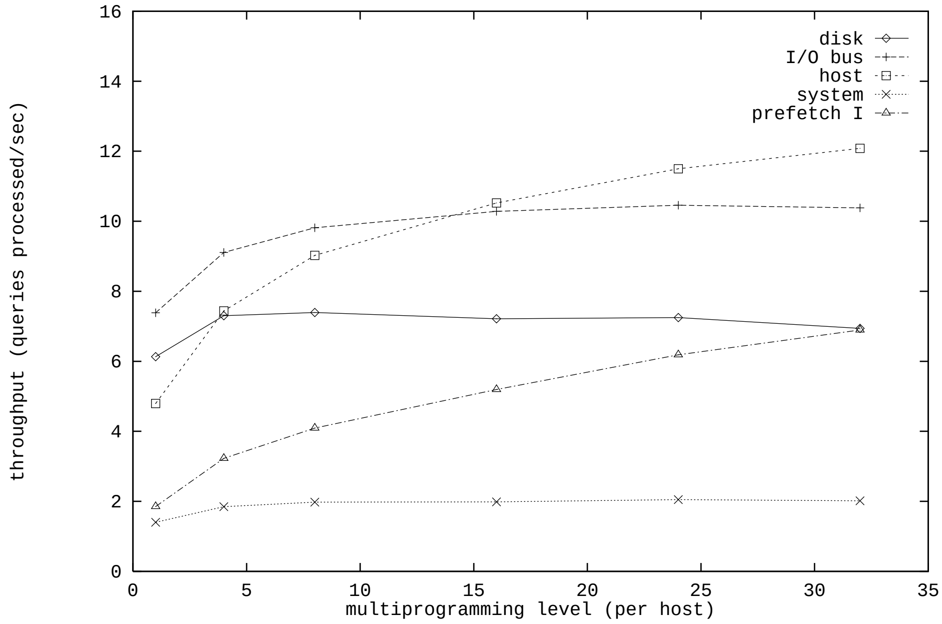

Figure 6: The effect of the multiprogramming level.

practice and hence multiple seeks. Thus, the higher seeks times shown in Figure 5 may occur in practice even without optical devices. In these cases, the disk and I/O organizations may not be advisable.

Figure 6 shows the effect of the multiprogramming level on throughput for the various index organizations. As the multiprogramming level rises, various bottlenecks in each index organization occur. Other collected data shows that the disk index organization has a disk utilization rate of 80.5% for a multiprogramming level of 1. The I/O bus index organization has a disk utilization of 58.7% for a multiprogramming level of 1 that rises to 77.5% at a multiprogramming level of 8. The host index organization has low disk and CPU utilization at a multiprogramming level of 1 (about 23.0% and 33.0%) and thus has more spare resources to consume as the multiprogramming level rises. At a multiprogramming level of 32 (128 total simultaneous queries using 4 hosts) the disk utilization has risen to over 74.3% and CPU utilization to over 78.2%

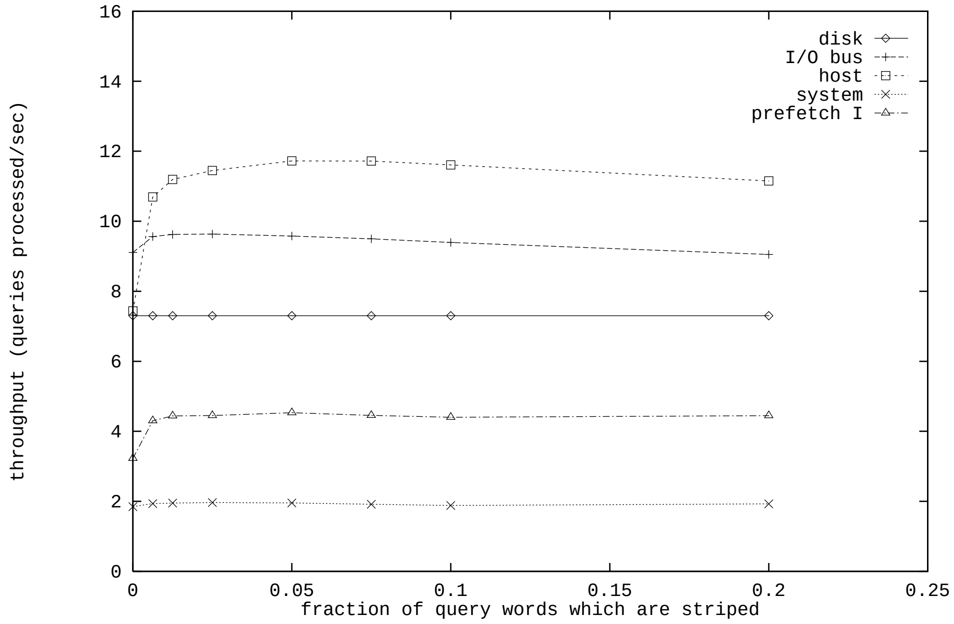

Figure 7: The effect of striping.

for this index organization.

The system organization has a *LAN* bottleneck even a low multiprogramming loads (94.7% at a multiprogramming level of 4) and thus does not improve as the multiprogramming level increases. With a multiprogramming load of 32, additional data shows that the response times for the disk, I/O bus, host, system and prefetch I index organizations are 17.9, 12.0, 10.6, 62.6, and 18.2 seconds.

Figure 7 shows the effect of striping on throughput. The horizontal axis, the variable *Stripe*, is the fraction of words that have striped inverted lists. The number of words that have striped inverted lists is $Stripe \cdot u \cdot T$. Striping 1% of the query words has a dramatic effect on the host index organization, giving a roughly 60% increase in throughput (with a similar decrease in response time). The system index organization shows no improvement due to the *LAN* bottleneck, however other collected data shows that with a 100 Mb/sec *LAN* the system index organizations shows an approximately

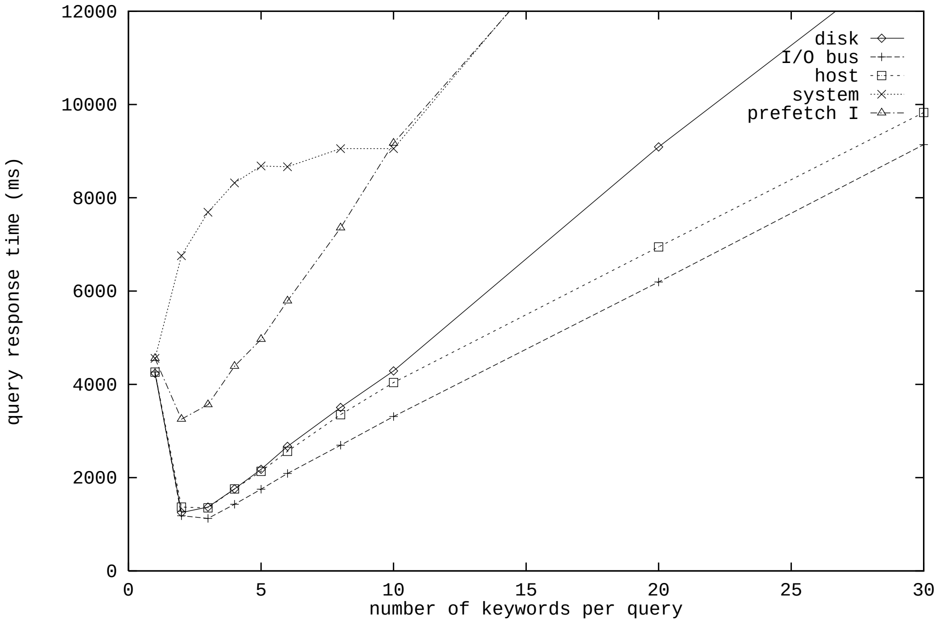

Figure 8: The sensitivity of response time to the number of keywords in a query.

70% increase in throughput. The disk index organization curve is flat indicating that this organization is independent of striping. Other collected data shows that if the horizontal axis is extended, the host and I/O bus index organizations approach the throughput of the disk index organization as the fraction of striped query words approaches 1. This graph confirms the explanation of the effect of striping.

The effect of large partial answer sets is shown clearly in Figure 8, which graphs response time as a function of the number of keywords. This graph shows a counter-intuitive result: in some situations, the response time of a query *decreases* as the number of keywords in a query *increases*. The sharp drop of the disk, I/O bus, and host lines from one keyword per query to two keywords per query is due to the reduced size of partial answer sets. That is, since the base case parameter set has four hosts, a query containing one keyword under the disk, I/O bus and host index organizations will transmit 3/4 of the answer set across the local area network for these three index organizations. In the case of a two-word query, again 3/4 of the answer set is transmitted. However, the total answer set size is much smaller since each partial answer set is the intersection of two inverted lists. This effect explains the sharp drop in the response time for these organizations from 1 to 2 keywords. As the number of keywords increases beyond 2, the additional work per keyword dominates the response time.

In the system index organization, the size of the partial answer sets transmitted depends on the hosts in which the particular words in the query reside. A subquery containing a single word has a large partial answer set. For 2 keywords, the probability of a single-word subquery at some host is high, thus leading to a large response time due to the transmission of these partial answer sets. At 5 keywords per query, the probability of a large partial answer sets is reduced and thus response time is correspondingly improved. With more than 15 keywords per query the probability of a large partial answer set is small and the response time for these queries is large due to the work required for query processing.

After 15 keywords per query, prefetch I performs worse that the simple system organization because the probability of a single word answer set being transmitted is tiny. Thus, the additional cost of the prefetch I algorithm is counterproductive. (This discrepancy can be eliminated by switching from the prefetch I algorithm to the system organization algorithm when the answer set of a subquery is expected to be small.) However, for small numbers of keywords, the prefetch I algorithm performs as expected and avoids transmitting large partial answers sets characteristic of the system level organization.

So far, the system organization, with or without prefetch, has performed poorly. To determine under what circumstances a prefetch algorithm performs well, we remove the *LAN* bandwidth bottleneck and increase the number of hosts to 16 while keeping the number of disks and I/O buses constant. Figure 9 shows the rise in

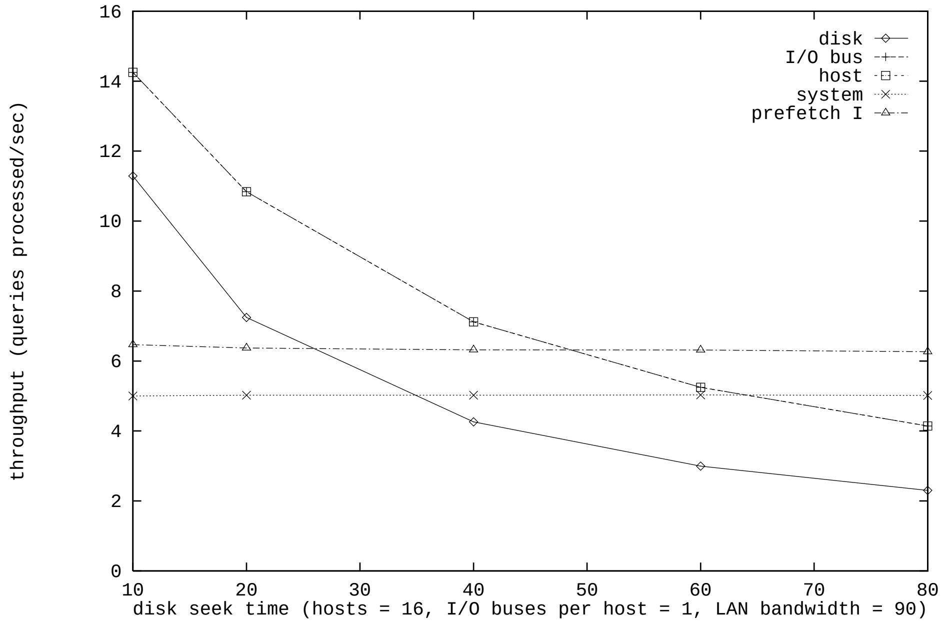

Figure 9: A good hardware configuration for the prefetch algorithm.

query throughput as the seek-time increases for this configuration. Again, the disk organization is sensitive to the increase in seek-time for the same reasons as Figure 5. The host and I/O bus index organizations are identical since each host has one I/O bus. The figure shows that the large number of hosts makes the these two index organizations sensitive to seek-time. The prefetch I algorithm performs well (with a disk seek-time above 50 ms) because an individual query (with 5 keywords) involves at most 6 hosts, which frees the other hosts to process other subqueries. Given the arguments for considering disk seek-time as a model of all fixed computation that consumes disk resources, 50 ms is not an unreasonable amount of time for a disk to be busy per inverted list fetch. For a disk seek-time of 80 ms, the disk, I/O bus, host, system, and prefetch I response times are 27.1, 15.0, 15.0, 10.8, and 10.2 seconds.

3.5 Conclusion

The choice of an index organization depends heavily on the access time of the storage device and the bandwidth of interprocessor communication. Somewhat unexpectedly, as the size of a query increases, its response time may drop, and the fancier prefetch optimizations are usually counterproductive.

In general, our results indicate that the host index organization is a good choice, especially if long inverted lists are striped. It uses system resources effectively and can lead to high query throughputs in many cases. When it does not perform the best, it is close to the best strategy.

Our results also indicate that the system organization, even with the prefetch organization, is not good unless disk seeks are high and network bandwidth is high. However, four factors that may be unfair to this approach. We are not modeling document fetches from disks. If the documents were stored on the same disks as the indexes, then disk utilizations would be higher, which would make the system organization more attractive since it reduces the I/O load. Second, we are not modeling pipelining of prefetching, I/O and CPU processing within a query. This can reduce query response time, allow users to abort partially finished queries, and would be more beneficial to the system organization since it deals with longer inverted lists. Third, *early termination* of the intersection algorithm can reduce response time if the inverted lists are sorted. The intersection algorithm can terminate after reading only a fraction of the inverted lists. Finally, we use a closed simulation model where larger response times penalize throughput.

Chapter 4

Distributed Queries – Trace-based Workload

This chapter studies distributed queries with a simulation system similar to the previous chapter. The workload in this chapter is based on actual user queries. We extend the simulation system to handle traces and to study several new issues. The chapter discusses the trace data, query processing, the simulation, and various results.

4.1 Trace Data

Stanford University provides on-campus access to its information retrieval system FOLIO from terminals in libraries and from workstations via *telnet.* FOLIO gives access to several databases; one of these is INSPEC, an abstracts database for technical documents in disciplines such as physics, electrical engineering, and computer science. A trace of all user commands for the INSPEC database was collected from 4/12/93 to 4/25/93. In addition, the number of postings of every word in the INSPEC database inverted index was also collected.

Each INSPEC abstract is divided into fields, such as *title* and *author.* One of these

```
68 04/25/93 10:45:02 CMD: fin a citrin and s exafs
68 04/25/93 10:45:04 SEA: CPU:262, Res:6, Find AUTHOR citrin and
                     SUBJECT exafs
68 04/25/93 10:45:04 CMD: !DISPLAY 1
```

Figure 10: Some example data from the raw trace. The first column is a unique integer representing the user login.

fields is *abstract*. To avoid confusion between the field and the complete record name, we refer to the complete abstract as the *document*. In a query, a user specifies the field where each word should appear. This specification is called the *field designation*.

An example of the raw trace is shown in Figure 10. The first column is the user identification[1], the second and third columns are the date and time of the command, and the fourth column specifies the type of data. The first line of the figure shows that user 68 issued a query for *author* `citrin` and the *subject* `exafs`. The user types *fin* as shorthand for *find*, *a* for *author*, and *s* for *subject*. The second and third lines show that six documents are the result for the query in the first line. (The trace repeats the query here.) The fourth line reports that the result of the query (short descriptions of each document in this case) where shown to the user.

To drive the simulation, only a subset of the raw trace is considered. Queries that have terms with no associated inverted list (12.0%) (e.g. misspellings) are ignored since the FOLIO query parser rejects these queries. Thus, they have no impact on performance beyond a small CPU overhead for parsing. Queries consisting of boolean AND operations on terms or wild-cards are simulated. We also simulate queries that are continuations of previous queries or are subject queries (discussed below). We do not consider 5.1% of the queries that are errors in the log, phrase queries, boolean OR queries, boolean NOT queries, or queries on chemical compounds. These queries have little impact on the performance results. The raw trace contained 8390 query

[1]To insure privacy, user login identifications have been replaced by unique integers.

Total Raw Trace Queries	8390	100.0 %
Discarded Queries	429	5.1 %
Nonexistent Terms	1001	11.9 %
Simulation Experiment Queries	6960	83.0 %

Table 9: Breakdown of raw trace and simulation trace.

commands. The remaining queries used to drive the simulation constitute 83% of the original queries (or 94.2% of the queries not caught as an error by the parser).

One important feature of FOLIO is the designation of the *subject* field in query matching. This field designation is a syntactic shorthand for matching the union of the field designations *abstract*, *conference*, *freeterm*, *document organization*, *thesaurus*, and *title*. Thus, the query *find subject theory* is conceptually a shorthand for the query *find abstract theory or conference theory or freeterm theory or document organization theory or thesaurus theory or title theory.* The subject field designation constitutes 37.2% of all field designations. Subject queries are handled by our simulation (see Section 4.2).

Two other features of FOLIO handled by our simulation are wild-card matching and continuation queries. A wild-card match is a keyword containing a "#" that matches zero or more characters. The addition of wild-cards can introduce performance problems and we discuss a specific data structure to handle them in Section 4.2. A continuation query adds extra conditions to the current query by using the command *and* instead of the command *find*. For example, the query *find subject exafs* produces 1595 answers. This query can be refined with the continuation query *and author* `citrin`, which produces the same 6 results as the query *find author* `citrin` *and subject* `exafs`. The simulation of this feature is discussed in Section 4.2.

Some statistics of the traces help interpret the results of the simulation. Table 10 summarizes some properties of the query traces. The mean number of keywords per

Description	Value	
Total Keywords	12444	
Number of *Subject* Field Keywords	4630	
Percent *Subject* Field of All Fields	37.2	
Number of Wild-Card Keywords	230	
Unique Keywords	4263	
Maximum Cache Hit Percent	65.7	
Description	Mean	Median
Keywords per Query	1.79	2
Result Size per Query	736.1	22
Matches per Keyword	25213.1	2344
Matches per Wild-Card Keyword	33894.9	6387
Postings per Keyword	43248.5	6139
Postings per Wild-Card Keyword	77897.5	35176
Words Matched per Wild-Card Keyword	73.3	23

Table 10: Statistical properties of the simulation trace.

query is less than two and the median is two. However, each use of the subject field designation is a shorthand for a query with multiple keywords. The mean size of the result of a query is large (over seven hundred), but the median is small at 22. We suspect that the queries with large results are immediately refined to produce smaller results.

Issuing multiple queries to refine an answer set is common in information retrieval systems. This query refinement behavior is an opportunity to cache inverted lists. Of all the keywords appearing in the traces, 65.7% of them are duplicate appearances. Thus, if caching every read of an inverted list we would achieve a cache hit ratio of 65.7% over the entire trace. While this figure is not a high as those reported in the file-system literature, Section 4.4 shows that caching does have a significant impact on mean throughput.

The INSPEC database holds 1,357,034 documents. The number of bytes per document is roughly 1,800. The total database size can be (roughly) estimated at 2.3

```
Count:          5  Key: CITRENBAUM
Count:         43  Key: CITRIN
Count:          5  Key: CITRINI
Count:          1  Key: CITRINOVITCH
Count:          4  Key: CITROEN
```

Figure 11: The number of postings for a sample set of words which appear in the *author* portion of the documents.

Description	Words	Postings	Mean	Median
Abstract	487247	74477422	152.9	1
Author	311632	4110892	13.2	2
Classification	2962	4211136	1421.7	634
Conference	11934	7246145	607.2	41
Free Term	319831	28110201	87.8	1
ISBN et. al.	12637	2445828	193.5	41
Author Org.	59546	8228475	138.2	2
Document Org.	2505	1145724	457.4	61
Report	7508	7833	1.0	1
Thesaurus	3695	11382655	3080.6	708
Publication	18411	6794557	369.0	4
Title	171537	10292321	60.0	1
Total	1409445	158453189	112.4	n/a

Table 11: The inverted indexes and associated statistics. The mean and median columns apply to the number of postings per word.

Gigabytes.

For the matching of queries, the total of lengths of the inverted lists (i.e. total number of postings) read determines the amount of work done in the matching process. We scanned the actual INSPEC inverted lists recording the number of postings and their field designations. For example, Figure 11 shows a sample of the file containing the number of postings for words with the *abstract* field designation. Table 11 lists some statistics on the postings for each field designation.

To drive our simulation, we combine the information from the trace and postings

```
8 68 0 ( CITRIN author 239427 43 47 ) ( BROMINE subject 236928 1137
       1138 )
9 68 6 ( CITRIN author 239427 43 47 ) ( EXAFS subject 248828 4225
       4226 )
10 68 21 ( EXAFS subject 248828 4225 4226 ) ( CHLORINE subject 241651
         3188 3194 )
```

Figure 12: Queries 8 through 10 of the trace input to the simulation.

files into a single trace file that is easy to use. Figure 12 shows a sample of this file. For example, the first line of the example shows query number 8, where user 68 issued a query that had 0 results. The query referred to the *author* `citrin` and the *subject* `bromine`. The value 239427 is a unique value of the keyword `citrin` and is used to determine the on which disks the inverted lists reside for the various index organizations. The next number, 43, is the number of posting for `citrin` that have the author field designation. The final number, 47, is the total number of posting entries for `citrin`, i.e., the total number of documents in which `citrin` appears regardless of the field designation. The number of postings for a subject field designation is the total of the postings for the constituent parts. This number is typically much higher than the number of documents that match because of the duplication of matches in the various field designations. For this sample trace file, two cache hits would occur (one for *author* `citrin` and one for *subject* `exafs`) assuming that the cache is initially empty.

The distribution of the lengths of the inverted lists that appear in the trace characterizes the work required to process the queries. Figure 13 shows the cumulative distribution of list lengths. The steep slope on the left-hand side of the figure shows that almost all of the inverted lists have less than 100,000 postings, and the flat slope shows that there are few long lists. These observations are confirmed by the following statistics: There are 12,503 inverted lists. The mean length of an inverted list is a

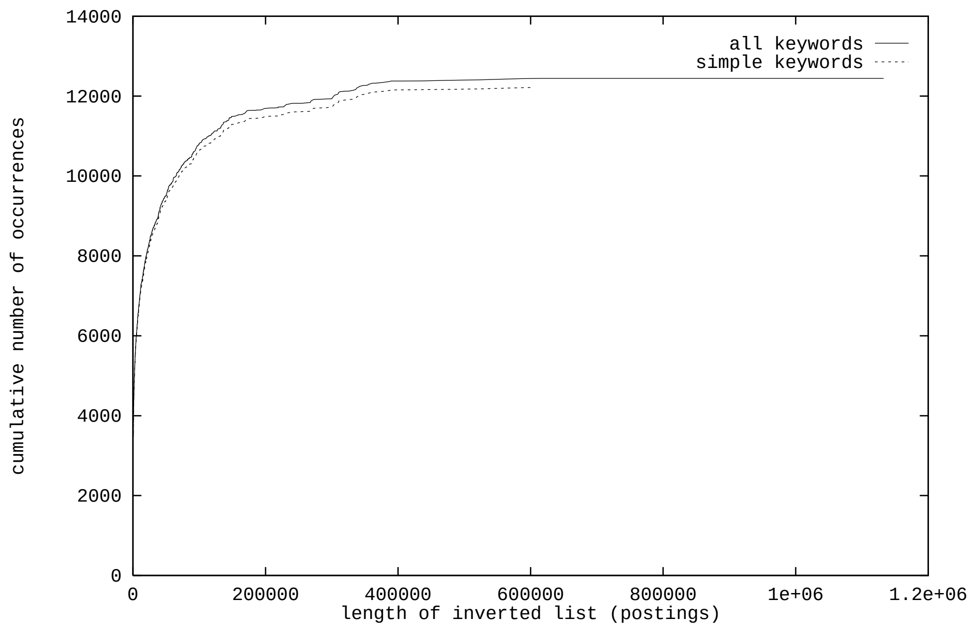

Figure 13: The cumulative distribution of occurrences of inverted indexes of a given length which appear in queries. Simple keywords do not use wild-cards.

little more than 43,000 postings. The median inverted list has 6139 postings. From the figure we see that the mean length is much larger because of the few long lists.

4.2 Query Processing

Information retrieval systems may partition their indexes by field designator or may build a combined index. In the partitioned case, all occurrences of a word in a *title* field are listed in one index, all *author* occurrences in another, and so on. In the combined case, a single index is built, and for each entry in an inverted list, a type annotation indicates the field where the word was used. The main advantage of the combined index is that a subject query such as *find subject art* can be answered by fetching a single inverted list. On the other hand, a query *find title art* can be processed faster with partitioned indexes, since only the relevant postings need to be processed. Since 37.2% of our queries involve *subject* searches, we assume a single combined index. Section 4.5 returns to this issue.

The use of a combined index allows a simple model of wild-card queries. We assume that the posting lists are allocated sequentially and alphabetically. Thus, the matching inverted lists can be read with a single disk access and the number of postings to read is the sum of the postings of the individual matching words. For the keyword *yak#*, the inverted lists for keywords *yak, yakube*, etc. cost only a single disk access. (FOLIO does not allow wild-cards of the form #yak.) The assumption of this data structure reflects a realizable data structure and is the "best case" for wild-card processing with inverted lists. Furthermore, we assume the system organization can be tuned so that words with the same alphabetical prefix reside on the same host. Our simulation results demonstrate that (with a proper data structure) wild-card processing has negligible impact on performance.

We model continuation queries through a dummy index called *user*. To illustrate, suppose user 5 issues the query *find subject A* and its returns 100 results. The continuation query *and author B* would be simulated as the query *find user 5 and author B* where *user 5* is an inverted list with 100 postings.

As discussed in the introduction, four physical index organizations are considered. We found in Chapter 3 that the LAN may be the bottleneck for the system index organization. To ameliorate this problem we adopt the "prefetch I" query processing optimization.

As in Chapter 3, to simulate the processing of a query, we consider five stages. The first stage covers the initial CPU processing for parsing the query and generating subqueries. Second, the subqueries are queued at the LAN for transmission to other hosts. Third, the process blocks, waiting for the subqueries to complete. When all the answers are returned the process wakes and simulates another CPU processing stage for intersecting the inverted lists. Finally, the process terminates, indicating that the query is complete. If the prefetch algorithm is used, several additional stages are added to account for the two phases.

A subquery goes through five stages also. First, initial start-up CPU processing is simulated. Second, the cache is checked for the words that appear in the query. For cache misses, reads are issued to the disks for the inverted lists. The process blocks, waiting for the disk reads to be returned through the I/O bus subsystem. When all the reads have returned, the subquery process wakes and simulates intersecting the inverted lists by a CPU processing stage. The answer is then queued at the LAN and the subquery terminates.

From our trace data, we can determine how many inverted lists have to be fetched to answer a given query, and how large the lists are. However, our simulation also requires the sizes of the intermediate results, and we estimate them by calculating the expect number of answers as follows. In the case of the disk, I/O bus and host index organizations, we make the assumption that the answers are distributed in equal proportion across all hosts. Thus, to compute the size of the subquery answer we simply divide the result size reported in the trace by the number of hosts. For the system organization, however, each subquery generally contains a subset of the keywords in the query. The following example illustrates how the expected answer size is calculated. Say the subquery is *find title A author B*. The full lists for A and B are fetched from disk; however, only the postings of the appropriate type (title for A, author for B) are used. The number of A postings with *title* designation is given in the trace, call it $n(A)$; the number of B *author* postings is $n(B)$. The expected size of the intersection of these lists is $n(A)n(B)/D$, where D is the total number of documents in the database. This estimate assumes that each word is equally likely to appear in any of the D documents and that the words occur independently. If an additional C word were in the query, the expected size would be $n(A)n(B)n(C)/D^2$.

For the disk, I/O bus and host index organizations, the model is accurate. For the system index organizations, this model is reasonably accurate for a small number of query terms (it is exact for single keyword subqueries). Emrath [Emr83] reports some measurements that support this model. However, as the number of keywords in the query increases, the expected number of answers approaches zero. To compensate for this effect, we use the result size (for the overall query) in the trace as a lower bound to the expected number of documents in a subquery. The size of the final answer to the match is bounded from above by the minimum of the sizes of the subquery answers in the system index organization.

Since the date and time of each query issued is reported in the trace, it is possible to simulate the exact sequence of query arrivals in the system in an open system type of queuing model. We have chosen a closed queuing system model that permits studying the effects of varying the multiprogramming level (the number of simultaneous queries in the system). At the start of a simulation run, a number of queries at the beginning of the trace equal to the multiprogramming level are started and then the next query is started whenever a query finishes in the system. This method maintains a constant number of queries in the system.

This approach concurrently simulates queries that are *from the same user* and thus could not have been requested simultaneously. This situation introduces a race condition for two queries that access the same inverted list. If the queries are executed sequentially, the second query will always be a cache hit. But if the queries are executed concurrently, both queries may simultaneously check the cache and both may miss. The negative impact of this loss in practice is minor and is outweighed by the advantages of studying the effect of the multiprogramming level on response time and throughput (cf. Section 4.4). Our cache hit results are slightly pessimistic due to the race condition.

To study the effects of scaling the database, the parameter *DatabaseScale* was added to the simulation. This variable linearly scales the number of postings for each inverted list, the number of answers to a query and the number of documents in the database. While scaling the number of postings and documents in the database is probably reasonable, scaling the number of answers is not. If a query matches 10 documents, a user will simply read all 10 documents to find those of interest. Faced with 1000 documents in a result, a user would probably issue a continuation query to prune the answer. As the database grows in size (say, linearly) the mean result size grows more slowly as users continue to construct queries with manageable result sizes. However, we did not incorporate this feature into the database scaling model.

4.3 Simulation

In this section, we use the same hardware organization as in Chapter 3 expect that we have as a default a single host. As before, every hardware organization consists of a LAN connecting several hosts together. Each host has a CPU and memory, a number of I/O buses, and a number of disks. Every host has the same number of I/O buses and every I/O bus has the same number of disks. In this chapter, each host also has a cache. Table 12 lists the variables that determine the hardware organization. The "Value" column in the table shows the base case value of each variable used in the experiments described in Section 4.4. Typically an experiment systematically varies one or more of the values to determine the effect of the variables. A *configuration* is the total collection of variable-value pairs used in an experiment. The base configuration is the collection of variable-value pairs given in the tables in this section.

Table 13 shows the base configuration variables for the hardware. The values for this table were taken from reference [Che90]. These values are better than, say, the theoretical maximum performance values available from the manufacturer's literature. The disks and I/O buses are simulated in the same way as Chapter 3. Requests for a disk read arrive from the CPU (after determining that they are cache misses). Each

Parameter	Value	Description
Hosts	1	Hosts
I/OBusesPerHost	4	Controllers and I/O Buses per Host
DisksPerI/OBus	4	Disks for each I/O bus

Table 12: Hardware configuration parameter variables, values and definitions.

read has a specified length in bytes. The reads are queued at the disk first-come-first-served (FCFS). Each request is first serviced by the disk by waiting an initial *SeekTime* milliseconds. The disk loads its track buffer at *DiskBandwidth* speed. When it finishes, the disk requests access to its I/O bus; only one disk at a time may occupy the I/O bus. When the I/O bus grants access both the I/O bus and disk are occupied for the transfer at *I/OBusBandwidth* speed. If multiple tracks must be loaded then the initial seek-time is extended by the *TrackToTrack* seek time.

The LAN handles the transmission of subquery and answer messages. Messages are serviced FCFS (except for messages that have the same source and destination – these are immediately returned to the host, simulating software loop-back). Each subquery has a length determined by *SubqueryLength* and each answer has a length determined by *AnswerEntry* times the number of postings in the answer. The service time for each message is *LANOverhead* plus the time taken to transmit the message at the given *LANBandwidth*.

Table 14 shows the parameters that affect the CPU and the time taken to process a query. The overall speed of a CPU is determined by *CPUSpeed*. Varying this value varies the rate at which instructions are executed. The number of instructions needed to execute various stages of the matching process are listed in the table. The multiprogramming level of the system is on a *per host* basis.

Finally, Table 15 lists the variables used to determine the size of the inverted lists.

Parameter	Value	Description
DiskBandwidth	10.4	Mbits/sec bandwidth per disk
DiskBuff	32768	Size of a disk buffer in bytes
BlockSize	512	Bytes per disk block
SeekTime	15.0	Disk seek-time in ms
TrackToTrack	4.0	Cost to seek one track in ms
I/OBusOverhead	0.0	I/O bus transfer in ms
I/OBusBandwidth	24.0	Mbits/sec bandwidth I/O bus
LANOverhead	0.1	*LAN* transfer in ms
LANBandwidth	100.0	Mbits/sec bandwidth *LAN*

Table 13: Hardware parameter values and definitions.

Parameter	Value	Description
CPUSpeed	20	Relative speed in MIPS
Multiprogram	4	Multiprogramming *per Host*
QueryInstr	500000	Query start up CPU cost
SubqueryInstr	100000	Subquery start up CPU cost
SubqueryLength	1024	Base size of subquery message
FetchInstr	10000	Disk fetch start up CPU cost
InterInstr	40	Intersection CPU cost per byte of a decompressed inverted list
Decompress	40	Decompression CPU cost per byte of inverted list on disk

Table 14: Base case parameter values and definitions.

Parameter	Value	Description
EntrySize	40	Bits to represent an inverted list entry on disk (uncompressed)
Compress	0.5	Compression Ratio
CacheSize	0.0	Inverted list cache (in postings)
ConcatInstr	5	Concatenation CPU cost per byte of an answer set
AnswerEntry	4	Bytes to represent an entry in an answer set
Documents	1357034	Number of documents
DatabaseScale	1.0	Database scale factor

Table 15: Base case parameter values and definitions.

EntrySize determines the number of bits needed to record a posting in an inverted list. *Compress* determines the reduction in bytes in the inverted list due to compression.

To illustrate the use of the variables in Tables 14 and 15, consider a subquery that intersects two inverted lists with 5 and 10 postings. The initial subquery CPU processing would be 120,000 instructions ($SubqueryInstr + 2 \cdot FetchInstr$) since each inverted list read is charged a start-up cost of *FetchInstr* instructions. The length of one list is 100 bits ($postings \cdot EntrySize \cdot Compress$). The length of the other list is 200 bits. The read length for both lists is 512 bytes (rounded up due to *BlockSize*). After the disk data is fetched, only the bits in the actual lists are used for subsequent computations. The number of instructions to process the intersection combines the costs of decompression and intersecting the lists. The size of the uncompressed inverted lists is 600 bits or 75 bytes ($postings \cdot EntrySize$). Then the number of instructions for this part of the subquery processing is 4,500 ($37.5 \cdot Decompress + 75 \cdot InterInstr$).

The size of the inverted list cache in postings is determined by *CacheSize*, which is measured in number of postings. The policy for the cache is least-recently-used. When an inverted list read is a cache miss, it is read from disk and the number of postings in the list is checked to determine if it will fit in the cache. If the inverted list is smaller or equal in size to the cache, the cache removes (in a least recently used fashion) enough inverted lists to make room for the new list and adds the new list. If the list is larger than the cache, the list is not placed in cache and no other lists are flushed. Both of these cases are cache misses. When an inverted list is a cache hit, it is moved to the end of the list of the least recently used inverted lists. (A possible improvement would be to also cache the intermediate and final results from the intersection computations.) For wild-card keywords, a cache hit occurs only if exactly the same wild-card keyword is used.

The number of bytes needed to represent a document in an answer is given in *AnswerEntry*. The instructions needed to concatenate the answers from the subqueries is given by *ConcatInstr*. The number of documents in the database is given by *Documents* and is equal to the number of abstracts in the INSPEC database. Finally, *DatabaseScale* permits scaling of the database as described in Section 4.2.

4.4 Results

In this section we present selected results of a set of experiments performed by the simulation. In conducting these experiments sensitivity analysis of all the variables in Section 4.3 were performed. An *experiment* is the execution of the simulation for the entire trace with a given configuration.

Figure 14 shows the mean response time of queries under the various index organizations as disk seek-time increases (the other simulation parameters for this configuration are given in Tables 2–15). The seek values on the left of the graph (around 10 ms) represent a typical magnetic disk. Values on the right could compare to optical disks. The graph shows that the disk index organization is most sensitive (i.e., has the largest slope) to the change in seek-time, followed by the I/O bus, host, system and prefetch index organizations. The host and system index organizations are identical because the base configuration has 1 host. This ordering of the index organizations is in decreasing number of inverted lists reads done by each organization. For a given query, the disk index organization does the largest number of reads, followed by the I/O bus index organization, etc. The increase in the number of reads leads to a higher disk utilization and increased queuing delays at every disk. The seek-time of the disk must thus dominate the cost of accessing an inverted list as opposed to the bandwidth limitations of the I/O subsystem. The host and system index organizations perform identically in the base configuration because there is only 1 host in the base configuration. The prefetch I index organization performs slightly worse than the host and system index organization because the prefetch of an inverted list is performed sequentially with respect to the processing of the remainder of the query. This slightly decreases the amount of parallelism in the processing of the query. (Recall that prefetch I index organizations was designed to reduce LAN traffic, which is not an issue in a one host configuration.)

Figure 15 shows the effect of the rise in the multiprogramming level on the mean throughput of queries processed. The graphs shows that the disk and I/O bus index organizations are relatively insensitive to the change in the multiprogramming level. Other collected data shows that I/O is the bottleneck in these two organizations. As the multiprogramming level rises, the same number of queries can be processed per second, but each query takes longer and longer. The host, system and prefetch I index organizations continue to improve across the range of the multiprogramming level in the graph because the resources are more evenly balanced. For a multiprogramming level of 32, the response times for the disk, I/O bus, host, system and prefetch I index organizations are 11.01, 3.17, 1.73, 1.73 and 1.74 seconds. Thus good response times are still available on a heavily loaded system.

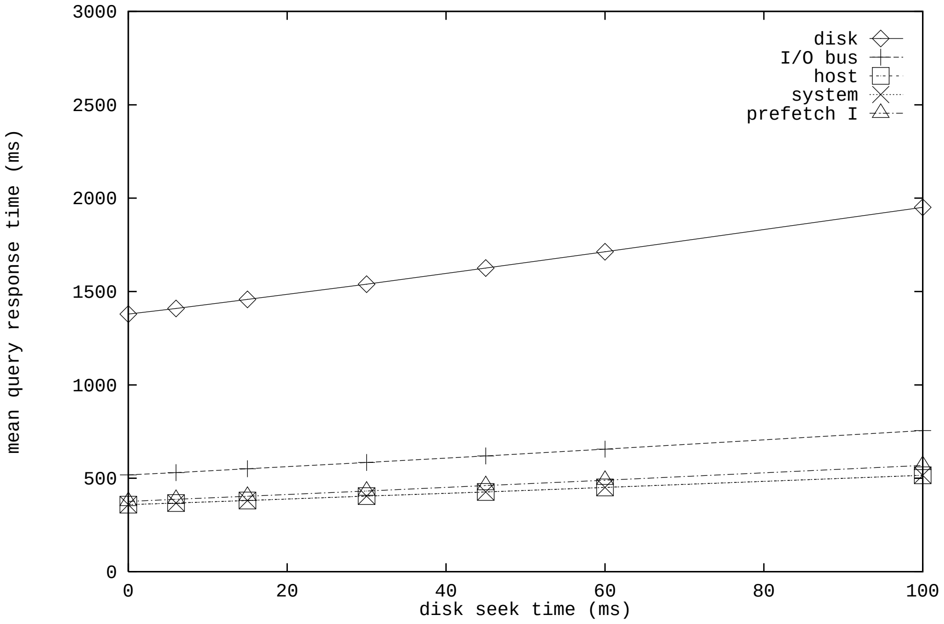

Figure 14: The sensitivity of response time to disk seek-time.

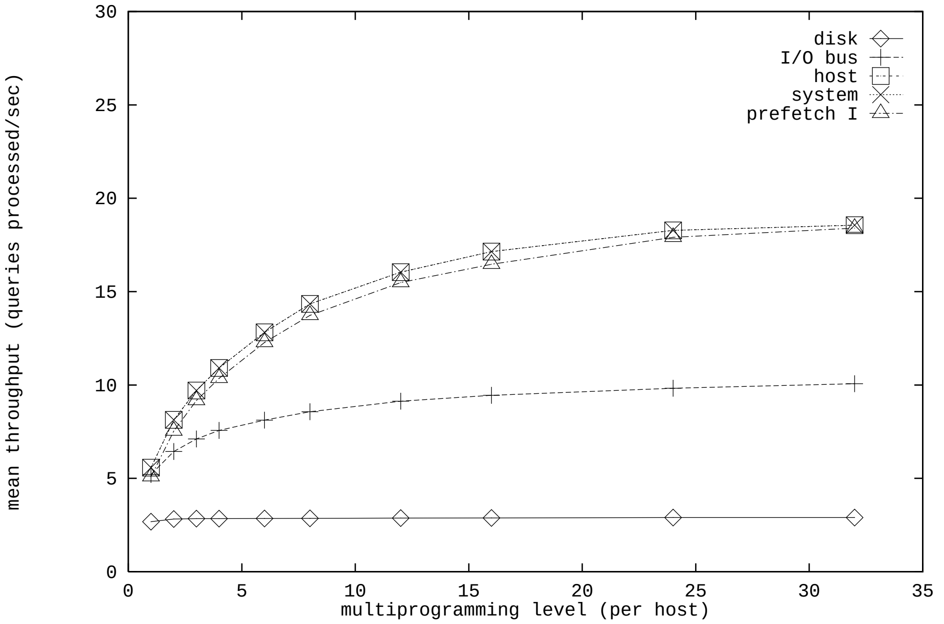

Figure 15: The effect of the multiprogramming level on throughput.

Experiment	BlockSize	DisksPerI/OBus	I/OBusesPerHost
0	512	2	2
1	512	2	4
2	512	4	2
3	512	4	4
4	16K	2	2
5	16K	2	4
6	16K	4	2
7	16K	4	4

Table 16: Enumeration of variable values.

Intuitively, experiments that vary the value of one variable in a configuration examine the change in a function along a single dimension. In some cases it is necessary to change the value of multiple variables in a systematic fashion in a 2^k *factor experiment* [Jai91]. For three variables, this experiment can intuitively be viewed as examining the values of a function at the corners of a three-dimensional cube; each axis of the cube corresponds to a variable.

To determine a reasonable base configuration, some of the values of the variables are provided by existing hardware, but other variables, such as *DisksPerI/OBus*, are less easily determined. We conducted a 2^k factor experiment on *BlockSize*, *I/OBusesPerHost* and *DisksPerI/OBus* to determine the configuration with the best response time. Table 16 lists the enumeration of the values of the variables.

The result of this experiment is shown in Figure 16. The data points for each index organization have been connected by lines to aid the reader in understanding the graph. The line connecting two points may represent the changing of the values of several variables. We see that the left-hand half of the graph (values 0-3) is the same shape as the right-hand half (values 5-7). This means that *BlockSize* has little effect on the response time, since it is the only variable to change value when comparing the halves of the graph. Next, examining each *sequential pair* of values (0,1), (2,3) etc.

shows no change in the response time for the disk and I/O bus index organizations. For each pair, only *I/OBusesPerHost* changes from 2 to 4. Thus, adding I/O buses (and implicitly, disks) does not improve the performance of these two index organizations. However, the response time for host, system, and prefetch organizations improve when more I/O buses are added because the total resources of the system are increased. Finally, consider the transition from the configuration in Experiment 1 to the configuration in Experiment 2. Here, the total number of disks is fixed at 8 but the arrangement of the disks changes because the number of I/O buses goes from 4 to 2. We see that disk index organization response time increases due to contention for the I/O bus (disk transfers on the same I/O bus are processed serially by the I/O bus). I/O bus index organization response time decreases because there are fewer inverted lists to read per keyword (since there are fewer I/O buses).

This graph shows the best combination of these variables for response time: *BlockSize* of 512, *DisksPerI/OBus* of 4 and *I/OBusesPerHost* of 4. This the worst configuration for the disk index organization.

To study database scaling, we first maximize the size of the database that can be processed with the base configuration. A 4-second mean response time is chosen as the limit for an effective information retrieval system. We scale the database on the base configuration (as described in Section 4.2) until the best response time increases to the threshold of 4 seconds. This graph is shown in Figure 17. From this graph the value of 10.0 is chosen for the maximum scaling of the database for a single host.

We now wish to observe the efficiency of the system as the number of hosts is increased. Increasing the number of hosts also increases the total number of I/O buses, disks, and queries (since the number of queries in the entire system is determined by $Multiprogram \cdot Hosts$). In Figure 18 the increase in response time is shown as the number of hosts is expanded. The increase in response time is due to two factors. First, the total load of the system is increasing in proportion to the number of hosts.

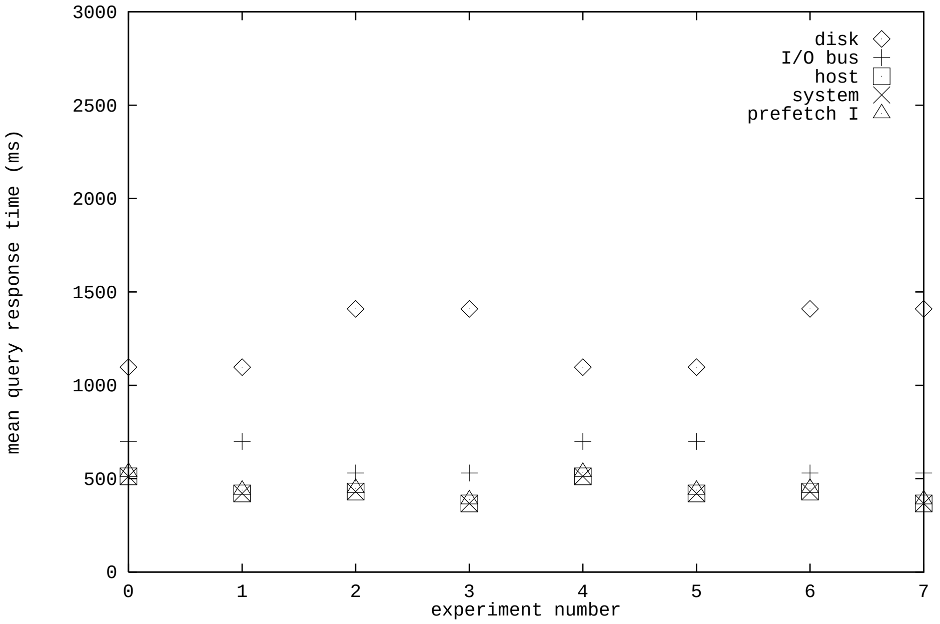

Figure 16: A 2^k factor experiment of three variables.

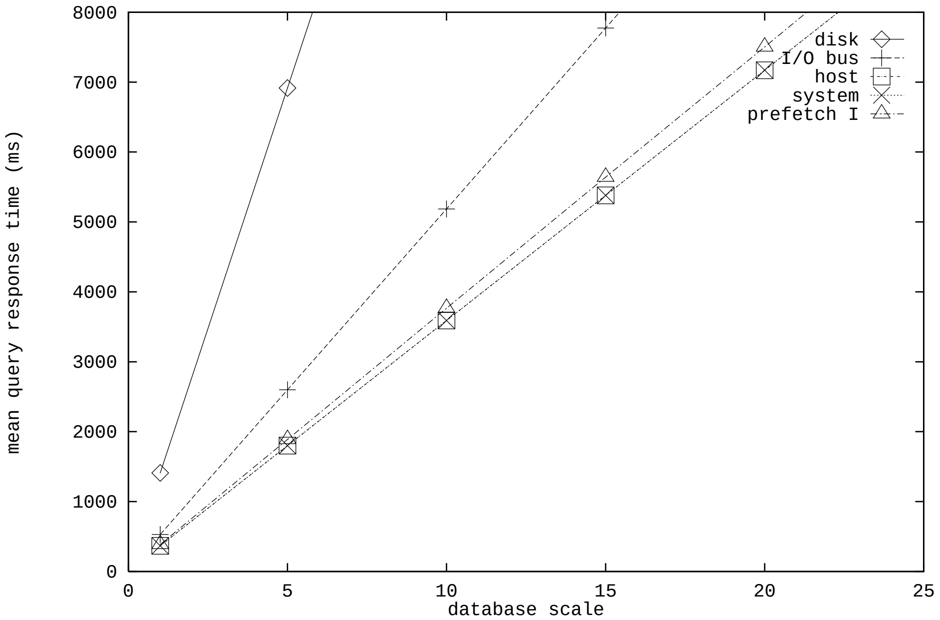

Figure 17: Scaling the database up to a 4-second response time for the best index organization.

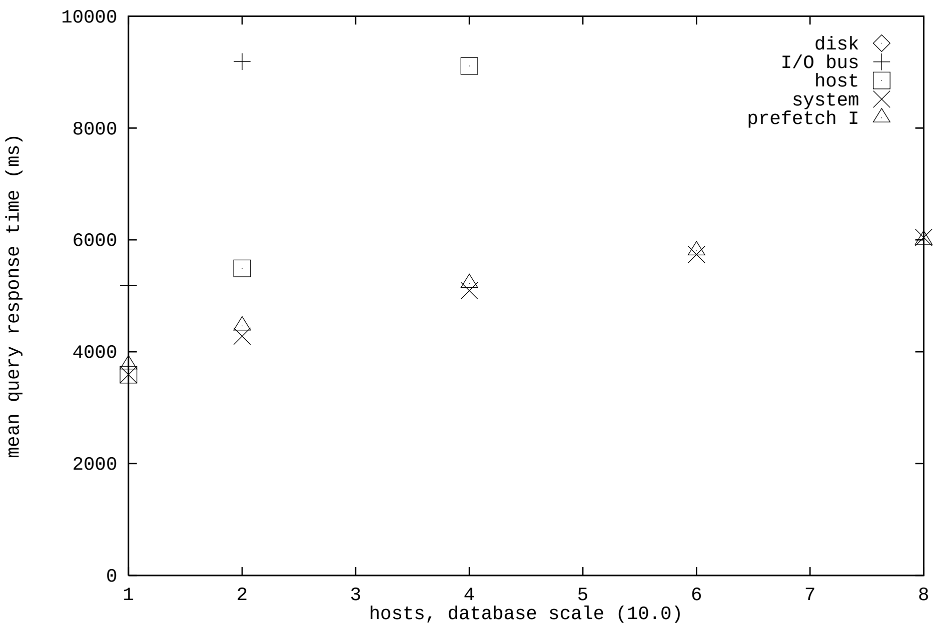

Figure 18: Increasing the number of hosts with a scaled database.

Second, as the number of hosts increases the traffic across the LAN increases. We see this effect appear at 8 hosts where the prefetch I index organization slightly outperforms the system index organization. Thus, the prefetch I organization scales well as the number of hosts increases.

However, the performance of the system organization depends strongly on the speed of the LAN. Figure 19 shows the impact of this variable on mean query response time. This figure does not show the data for the disk index organization since its mean query response time is approximately 55 seconds. The left hand side of the graph represents a Ethernet-type network at maximum bandwidth and the right hand side of the graph represents an FDDI network. The graph shows that the system response time is sensitive to the LAN bandwidth and that a sufficiently fast network eliminates the disadvantages of the system organization. The prefetch I organization performs more poorly at a high bandwidth with four hosts due to the sequential nature of the

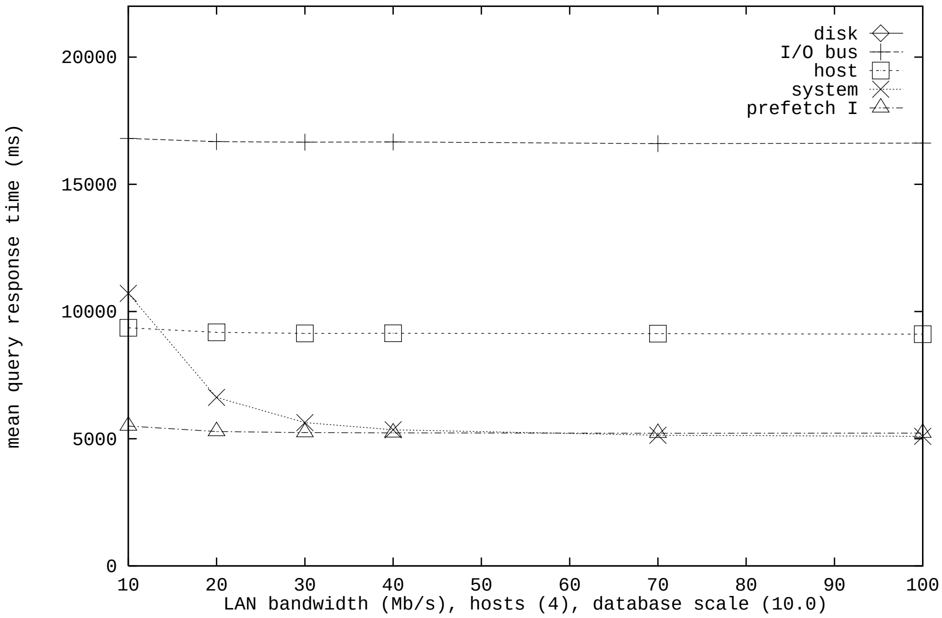

Figure 19: The sensitivity of mean query response time to LAN bandwidth.

two-phased approach. Note that the prefetch I organization is relatively flat in this graph. Thus, even with a fast network, this organization would be preferable if the network cannot be used to its maximum capacity. In addition, any of a number of other variables can make prefetching attractive e.g., the number of hosts, or a larger database.

Given that an index organization does well as the number of hosts increases, we can compare the base configuration of a single host to configurations with more hosts but the same total resources in terms of CPU speed, number of disks and I/O buses. This variation in parameters is essentially the trade-off between buying a single large mainframe processor or several slower workstations. Table 17 shows a enumeration of configurations that explore this trade-off under a fixed total system load. The main difference between a single host and multiple hosts is that the fast CPU has been replaced by several slower CPUs interconnected by a LAN. Figure 20 shows

Hosts	I/OBusesPerHost	Multiprogramming	CPU Speed	Database Scale
1	4	4	20	10.0
2	2	2	10	10.0
4	1	1	5	10.0

Table 17: Enumeration of variable values for fixed resources.

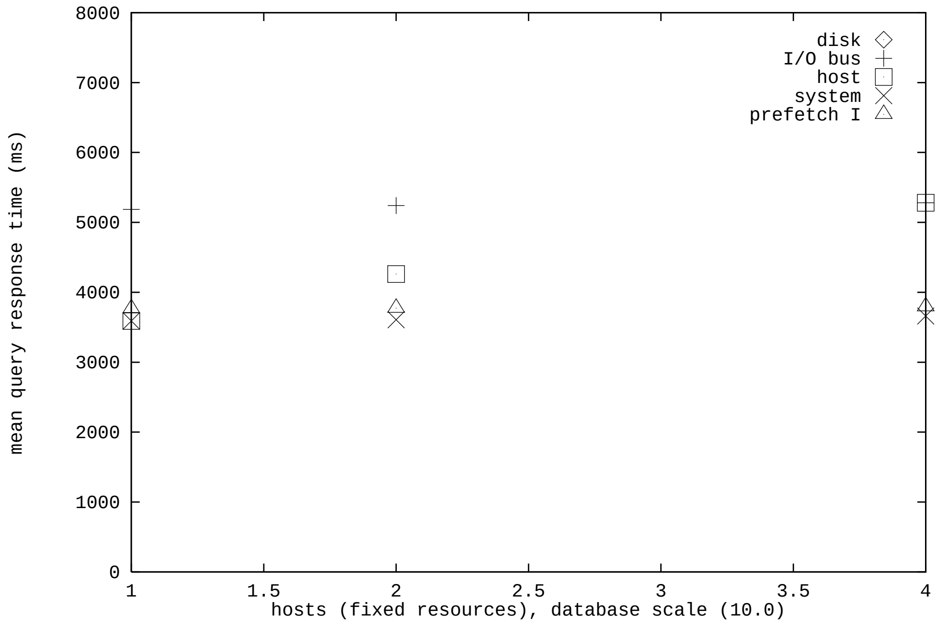

Figure 20: The mainframe vs. workstation trade-off.

that the best index organization (system) has a response time of 3.59, 3.61, and 3.66 seconds for 1, 2 and 4 hosts, indicating about a tenth of a second loss in response time when split among multiple hosts. Throughput for the system index organization is 1.12 queries/sec., 1.11 queries/sec., and 1.10 queries/sec. for 1, 2 and 4 hosts. Thus a minimal performance loss is incurred by using the multiple host organization. The results indicates that a "mainframe" is slightly more effective, but the small improvement has to be evaluated in light of the potentially higher mainframe cost.

Figure 21 shows the increase in the cache hit ratio as the size of the cache increases for a four-host system (database scale 1.0). Since the total cache size is the same regardless of the index organization, it is surprising that the cache hit rates vary depending on the organization. However, the behavior of the caches under the various organizations is quite different. For the system and prefetch I index organizations an inverted list is cached in only one place in the system. Thus, there are effectively *Hosts* number of independent caches. Also, suppose a list slightly larger in size than the cache is read from disk. In the system and prefetch I organization, the list does not fit in the cache and thus the caches would remain unchanged. In the disk, I/O bus, and host organizations, however, all four caches would hold a list of quarter the size, requiring some other lists to be removed from the cache. The figure shows that for the base configuration even a small cache has a good hit rate – achieving almost 40% where the maximum possible cache hit rate is about 65.7% (see Table 10). The cache hit rates for the disk, I/O bus, and host index organizations are the same since they access exactly the same lists on each host and so the cache contents are changed in the same way over the course of the simulation.

Figure 22 shows that the effect of caching is to free the I/O subsystem resources to speed up query processing for the system and prefetch I index organizations. The cache does not have a dramatic effect on throughput for the disk organization because it remains bottlenecked on the disks.

4.5 Conclusion

Our main result is that inverted lists referenced by queries in such systems tend to be relatively short and it does *not* pay to split them across hosts, much less across I/O subsystems or disks. Either system index organization, or the system index organization with the prefetch I optimization, performs best over wide ranges of parameter values. Prefetch I is especially good as the database size scales up. However, the

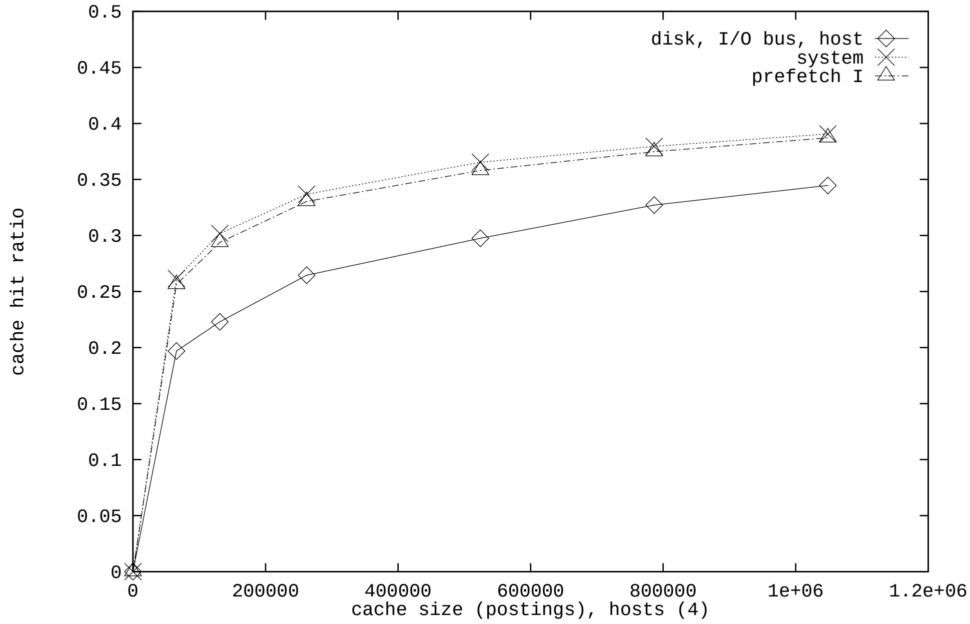

Figure 21: The improvement in the cache hit rate as the cache grows in size.

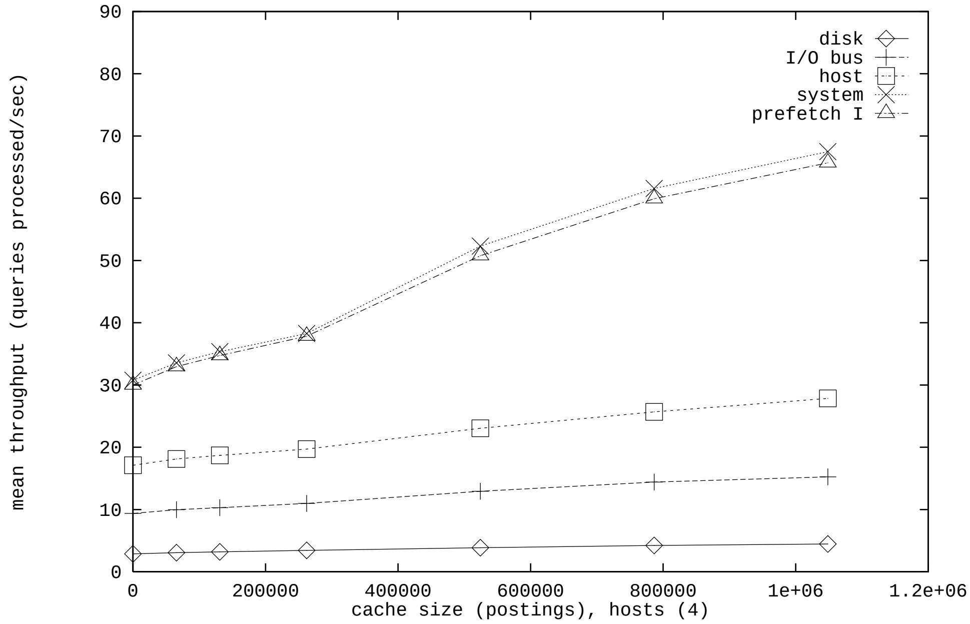

Figure 22: The impact of the cache size on throughput.

system organization does use the LAN or processor interconnect more heavily, so it is inappropriate for systems with slow networks.

Our conclusion is different from that of Chapter 3. In that case, inverted lists are much longer, and breaking them up (e.g., striping them) does pay off. In particular, the host organization was superior.

We also explored the impact of wild-card queries and found only a slight performance improvement and therefore have not included results on these experiments. Two factors contributed to this result. First, wild-card keywords are a small fraction of all the keywords that appear. Second, the data structure used to model wild-card queries is efficient, since the performance impact is restricted to simply reading longer inverted lists (about twice as long on average).

Our caching results indicate that a relatively small cache can improve performance significantly. For our INSPEC database that has an index size of 377 MB (129 million postings compressed) a cache of about 3.8 MB (800,000 postings uncompressed) – about 1.0% of the index – can improve throughput by about 171% for the prefetch strategy. For the other strategies, improvements are smaller. Although not reported here, we also experimented with various cache policies. For example, in one case, lists above a given threshold were not cached, even if they fit in the cache, on the presumption that they would flush out too many useful lists; we observed no significant improvement with this variation.

Chapter 5

Incremental Updates – Actual Workload

In this chapter, we assume that the postings in a new document are inserted into an in-memory inverted index. At some point, this index must be written to disk. Our objective is to update the disk incrementally as efficiently as possible. Collecting many documents into an in-memory index amortizes the cost of storing a posting.

5.1 Dual-Structure Index

The lengths of the inverted lists for a database of text documents have a roughly exponential distribution (see Figure 2). This distribution presents a dilemma for the in-place update of inverted lists since some inverted lists (corresponding to frequently appearing words) expand rapidly with the arrival of new documents while others (corresponding to infrequently appearing words) expand slowly or not at all. In addition, new documents contain previously unseen words. Table 18 shows some statistical properties of a database of NEWS articles (see Section 5.3 for a complete description of the database). Abstracts databases index general information about a

Text Document Database	NEWS
Total Raw Text	686 MB
Total Words	788,256
Total Postings	48,526,577
Documents	138,578
Average Postings per Word	61
Frequent Words (top 5%)	39,413
Infrequent Words	748,843
Postings for Frequent Words	93.6%
Postings for Infrequent Words	6.4%

Table 18: Statistics for a NEWS abstracts text database.

document such as author names, title, the set of words in the abstract, etc. A frequent word for this table ranks in the top 5% of all words (in order of frequency). Postings for frequent words are given as the percentage of all postings in the database. Infrequent words are all words that are not frequent. For example, if the frequently appearing words are those that rank in the top 5.0% of all words (in order of frequency) the postings for these words account for 93.6% of the postings.

There are two data structures for lists. We place short inverted lists (of infrequently appearing words) in a fixed-size region of disk (where the region contains postings for multiple words). These lists are *short lists* and the fixed-size regions are *buckets*. The idea is that every inverted list starts off as a short list; when a bucket fills up with inverted lists, the longest inverted list becomes a *long list*. We place the long lists (of frequently appearing words) in variable length contiguous sequences of blocks on disk. Each block of a long list contains postings for only one word. Given a word w, we examine a *directory* that determines if the word has a long inverted list. If the word does not have a long list, it has a short list or no list. In this case, a function $h(w)$ (e.g., a hash function or a tree search) returns the bucket that holds the short list for the word.

At some point, an in-memory list L for word w must be moved to disk. First, if w

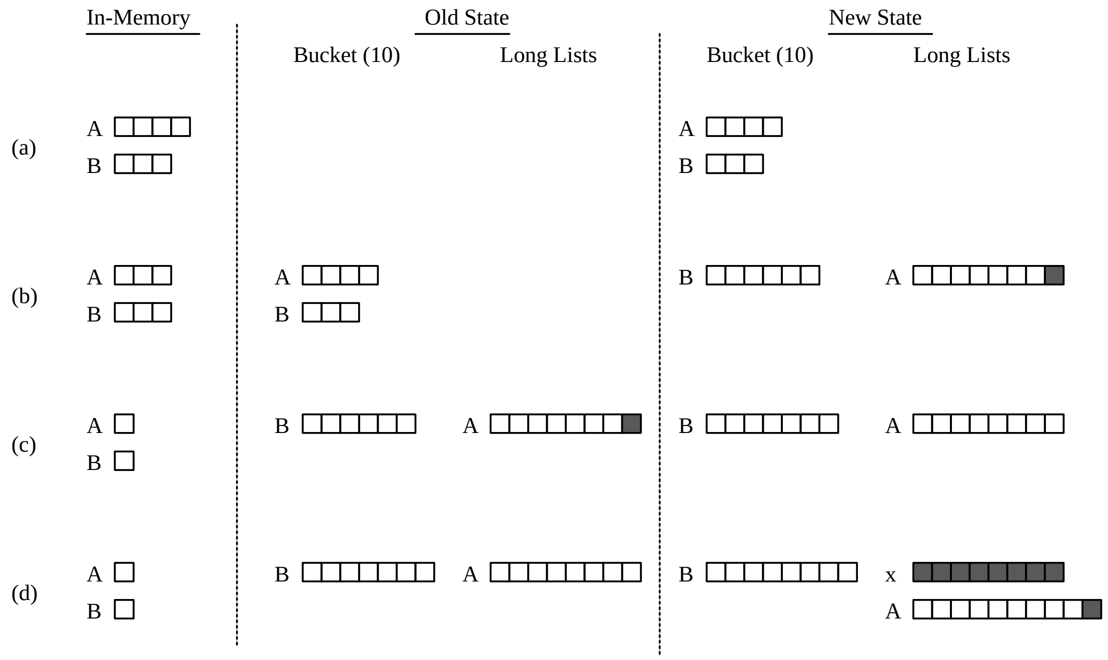

Figure 23: A running example of the behavior of the update algorithm.

already has a long list, L is appended to the long list as discussed in the next section. Otherwise, we assume L is a short list and insert it into bucket $h(w)$. If the bucket is not already in memory, it is read in, and L is inserted. If a list for w already existed in the bucket, L is added to it; else a new short list is created in the bucket. If the bucket overflows, we then pick the longest short list[1] in block, say M, remove it, and make M a long list. Once M is removed, the bucket will be partially empty. The updated bucket $h(w)$ is written to disk (eventually), and list M is written to disk as discussed in the next section. A word w never has both a short list and a long list. The buckets dynamically determine which words have inverted lists containing only a few postings, since these words are unlikely to grow enough to overflow i.e assuming that the bucket data structure is large enough to hold all the infrequent words.

Figure 23 illustrates the four different situations for our dual structured index.

[1]If there are multiple longest short lists, we choose one arbitrarily.

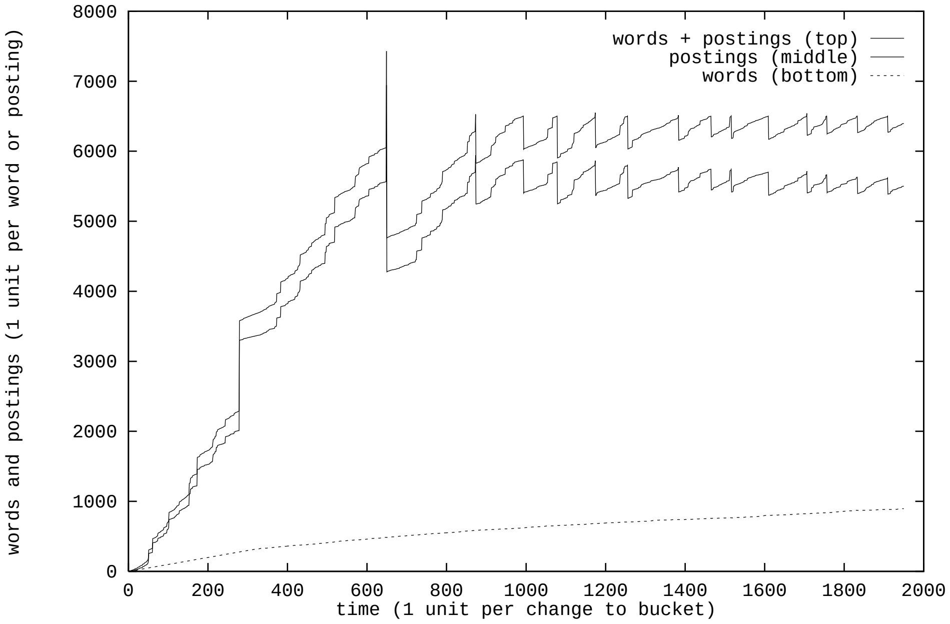

Figure 24: An animation of the behavior of bucket 0 for the first 6 updates for a system with 250 buckets.

For this example (and for this example only), we use two simplifications: a single bucket holds all words and one posting fits exactly in one disk block. The bucket has a capacity of 10 postings. Each row, labeled (a) through (d) is an example of the dual structure index. The in-memory column represents the new inverted lists generated by a batch of documents. The old-state column represents the state of the disk before the new lists are added; the new-state column is the result after insertion.

In Figure 23 (a), two words with seven postings are inserted into the empty bucket. Postings are represented by small empty squares. In Figure 23 (b), six postings for two words are inserted into the bucket. The bucket now contains 11 postings, so a word must overflow since the bucket capacity is 10 postings. The longest short list is chosen, thus the word "A" with seven postings is chosen over the word "B" with six postings. The short list overflows into the long list and it is written to disk with a 10% reserved space, which is one posting (the free posting is represented by a shaded square). In Figure 23 (c) two words with a single posting each are added. The posting for the word "B" is added to the short inverted list for "B" in the bucket. The posting for the word "A" is added as an in-place update to the long inverted list for the "A" word. Finally, in Figure 23 (d) two words with a single posting each are added. The posting for the word "B" is again added to the short inverted list for "B" in the bucket. The posting for "A" cannot fit in the reserved space, so the long inverted list is moved to a new location with both the extra posting appended to the end and with new reserved space. The old long list for "A" is freed for subsequent reuse by any word, as indicated by the "x" label.

Figure 24 shows an animation of the behavior of buckets. We choose bucket 0 as an example bucket and run the bucket algorithm for a short time on a small system. Data is from a NEWS, as explained in Section 5.3. The bucket has a size of 6500 units, where each posting is charged 1 unit and each word is charged one unit. For each inverted list in the bucket, we need to store the word it represents plus all of its postings. Each time step on the x-axis corresponds to a change in the bucket: inserting a new word with its postings, appending the postings to an existing word, or the removing a word and its postings from the bucket. The y-axis measures the combined number of words and postings. The top line in the figure is the total number of words and postings, the middle line is the total number of postings and the bottom line is the total number of words in the bucket. For the total number of words in the bucket, we see a slow rise in the number of words in the bucket as new words are continuously inserted into the bucket. For the number of postings in the bucket (the middle line), we see a steep climb as the bucket fills up and two leaps where a long in-memory list is inserted into the bucket at approximately time 300 and time 700. The second insertion of the long in-memory list causes an overflow and the list is removed from the bucket, shown as a downward spike in the graph. After the spike, the bucket continues to fill to about time 900 where it overflows and again the longest short list is removed.

In summary, the dual-structure index allows us to apply different storage structures to the huge number of infrequent words and to the relatively few frequent words. Through the use of fixed-size buckets, this approach dynamically discovers the frequent words that require their own long list. Updates to the large number of infrequent words are amortized into a relatively small number of disk operations, since the buckets are small enough to fit in memory. In addition, coalescing infrequent words reduces wasted disk space due to allocation of complete disk blocks to short lists.

5.2 Policies for Allocation of Long Lists

Information retrieval systems merge inverted lists to compute the answer to a boolean query. This merging is possible because the document identifiers appear in sorted order in inverted lists. We assume that new documents are numbered with identifiers in increasing order and that all long lists are updated by appending new postings to them. With these assumptions, the merge operation can be used to compute answers to boolean queries with our long list data structure.

Long lists are created initially by bucket overflow. Once a word has a long list on disk, subsequent in-memory lists for that word are appended to that long list. In this section, we consider only long lists and refer to them as *lists*.

In allocating lists to disk, there are two extreme policies. One extreme policy optimizes the time to update incrementally. L is the list for a word w and M is the in-memory list to be appended to L. If M is written to disk sequentially on disk, irrespective of L, then update performance is optimized because the disk head never seeks during a sequence of updates. The other extreme policy optimizes the reading of a list during query processing. To update a word w, we read L from disk, append M to it, and write the new combined list to a new location on disk. This policy optimizes query performance because exactly one seek is required to read any list. However, query performance for the update optimized policy is poor because the list for a word will be spread over the disk and the update performance for the query optimized policy is poor because reads are intermixed with writes.

Between these two extremes, there are intermediate policies that move lists and allocate new space for them in a variety of ways. This section presents a framework for describing some of these intermediate policies.

The first issue is to compose lists from disk blocks. We use the term *chunk* for variable sized contiguous regions of disk and reserve the term *extent* for fixed-sized contiguous regions of disk. Multiple chunks for an inverted list may be allocated. The pointers to all chunks are recorded in the directory. The directory entries for a word may point to chunks on multiple disks. The directory resides in memory at all times. Periodically, the directory is written to disk.

The second issue is to assign a disk unit to a new word or chunk. When the list for a new word w is added to the directory or a new chunk of a list for a word w is allocated, a disk is chosen. Let there be n disks, numbered from 0 to $n-1$, and let i be the disk chosen when the last new word or chunk was allocated (i is initially 0). The strategy considered here choses disk $i+1 \bmod n$. Other strategies, not considered here, could be to look for the most empty disk or a disk where the list has the fewest chunks.

The third issue is to combine in-memory lists with long lists. We have an in-memory list M that we wish to append to a long list L. Both lists are for a word w. Let x be the size (in postings) of the long list, let y be the size (in postings) of the in-memory list, and let z be the size (in postings) of the space remaining in the chunk that can accommodate new postings. As described below, when a chunk is allocated to disk, it may have *reserved space* at the end of the chunk where future postings may be append. Variable z may be zero or positive and x and y are always positive. Our strategies for appending use the following basic operations, which operate on long list L. The RELEASE list is used to delay the deallocation of long lists while they are copied.

UPDATE(a) reads the last block containing postings for word w of in-memory list a, appends a to it, and then writes the result back as an in-place update.

b := **READ**(a) reads all the postings for long list a, places a on the RELEASE list, and returns the postings read as in-memory list b.

WRITE(a,b) writes up to e blocks worth of postings from in-memory list a and returns the remaining postings as in-memory list b. The global parameter e is the *extent size.* The fill style, below, breaks up in-memory lists into extent size chunks. If a contains less than e blocks worth of postings, e blocks are still allocated on disk.

WRITE RESERVED(a) writes the a in-memory list to disk with reserved space at the end of the list.

A strategy for appending an in-memory to a long lists is specified by two variables, *Limit* and *Style*. *Limit* is either 0 or z. *Style* is *fill*, *new*, or *whole*. Table 19 summaries the variables and values governing policies.

Figure 25 shows the algorithm for updating long lists. The first three lines check if the existing chunk can and should be extended with the in-memory postings. If extension isn't possible or desirable ($Limit = 0$), lines 4-6 (*whole*) copy the old postings to a new location with the in-memory postings appended. Lines 7-9 (*fill*) write out multiple extents. Lines 10-11 (*new*) write a new chunk with reserved space.

Variable	Value	Description
Limit	0	Never update in-place
	z	Update in-place if enough space
Style	*fill* $(e = 3)$	Fill in fixed-size extents
	new	Write a new chunk when appropriate
	whole	Long lists are single whole chunks
Alloc	*constant* $(k = 10)$	Constant extra postings reserved
	block $(k = 3)$	Multiple of a fixed-sized block reserved
	proportional $(k = 1.1)$	Proportional extra postings reserved

Table 19: The variables and values that determine a policy for the allocation of long inverted lists to disks. The values in parenthesis are for each allocation strategy or style.

```
if y ≤ Limit then
   UPDATE(M) update long list in-place with the in-memory list
else
   if Style = whole then
      b := READ(L) read long list
      WRITE RESERVED(M and b) write with reserved space
   if Style = fill then
      while (M not empty) in-memory postings remain
            WRITE(M, M) write in-memory postings
   if Style = new then
      WRITE RESERVED(M) write in-memory postings with reserved space
```

Figure 25: The algorithm for updating long lists.

One consequence from lines 1-2 is that an in-memory inverted list is never split into two different chunks for an in-place update.

Periodically, the buckets and the directory are written to disk. At this time, the disk blocks for the previous buckets and directory are returned to free space for the disks. In addition, in the case of the whole strategy, the old long lists on the RELEASE list are returned to free space for the disks.

The fourth and final issue is to allocate space on a disk for a chunk. Given a request for a chunk of size f and a disk, we need a contiguous region of free space on the disk to satisfy the request. We use a first-fit strategy by scanning the free list for the disk from the beginning of the disk. Upon finding a contiguous sequence of f or more blocks, the chunk is placed at the beginning of the free blocks and the remaining free blocks are returned to free space.

In addition, the WRITE RESERVED call reserves space at the end of every list for future growth. That is, additional space is allocated to a chunk to hold postings which will appear in subsequent updates. Let x be the size (in postings) of the inverted list being written to disk and let $f(x)$ be the allocated space (list plus reserved space). The resulting size (in blocks) of a chunk is the number of blocks needed to hold $f(x)$. For the new style x is typically the size of an in-memory list. For the whole style x is typically the size of the entire long list for a word.

We consider three choices for the definition of $f(x)$. The *constant* strategy adds a constant number k of postings to the end of the inverted list, i.e., $f(x) = x + k$. The *block* strategy insures the chunk is of constant multiple of size k, i.e., $f(x) = k \cdot \lceil \frac{x}{k} \rceil$. (In practice, we specify k for this strategy in terms of blocks instead of postings.) Finally, the *proportional* strategy allocates a chunk in proportion to the number of postings being written to disk, i.e., $f(x) = kx$. The variable *Alloc* equals *constant*, *block*, or *proportional*, for the corresponding choice of strategy.

5.2.1 Policies

A policy is determined by the values of the variables *Style*, *Limit* and sometimes *Alloc*. If $Limit = 0$, then any reserved space for a chunk is never used, so we automatically set $Alloc = constant$ with $k = 0$. If $Style = fill$ then the allocation strategy is irrelevant since it is never considered.

The update optimized policy described above can be achieved by setting $Limit = 0$ and $Style = new$. This policy minimizes update time by simply writing out the update list blocks as fast as possible. No reading is done because no in-place updates occur. We expect that this policy will have the best update time and that the query time for the resulting index will be poor.

The policy for fast queries sets $Limit = 0$ and $Style = whole$. Setting *Style* to *whole* insures that the inverted list for any word will always be a single contiguous chunk and thus minimizing query time. We expect that the update time for this organization will be high, because lists must be moved. To ameliorate this situation, we can let $Limit = z$ and $Alloc = proportional$ with a constant of, say, 1.1 (i.e., reserved space that is 10% of the size of the long list). With each move of the long list, the reserved space grows by 10%, permitting more in-place updates of in-memory lists.

Finally, we consider a policy that attempts a trade-off: to minimize query time and keep the cost of updates low by organizing inverted lists into chunks that never move once they are full. Let $Limit = z$ and $Style = fill$ with an extent e size, say, 3 blocks. With this policy, each inverted list grows until it reaches the limit of its chunk and then a new chunk is started on a new disk. We expect comparatively good query and update times for this policy. Our model of extents uses only one size for an extent. We do not model multiple fixed extent sizes since this policy is approximated by the new style with a block allocation strategy.

So far our discussion has focused on the addition of documents to an index since typically databases only grow in size, or deletion is infrequent enough that the entire index is rebuilt. The addition of incremental deletion of documents poses some problems to the design of an index. One method maintains an index of document identifiers and all the words in the document (or the words are extracted from the original document). Given this index, each inverted list for a word in the document would be fetched, the reference to the document deleted, and the new inverted list rewritten to disk. However, the size of this index is the same as the size of the inverted index. To avoid this cost, existing implementations typically maintain a list of deleted document identifiers and filter any answer to a query through this list. This approach deletes the document from the point of view of the user. To reclaim the space taken by the deleted document identifiers in the index, a background process sweeps the lists in the index one list at a time, removing any deleted documents. After a sweep of the index, the list of delete document identifiers is discarded. Since this issue is orthogonal to the the issues in this thesis, we do not consider deletion further.

In summary, the parameters described in this section span range of approaches to storing long lists. By varying these parameters, we can model schemes that keep the lists sequential and those that break the lists into contiguous chunks. We can control the size of the chunks allocated, either as fixed-length chunks or as chunks whose size is controlled by the frequency of a word. Finally, we can control whether or not unused space at the end of a list is used. The choice of parameters permits trade-offs between index build performance, query performance, and index disk space consumption.

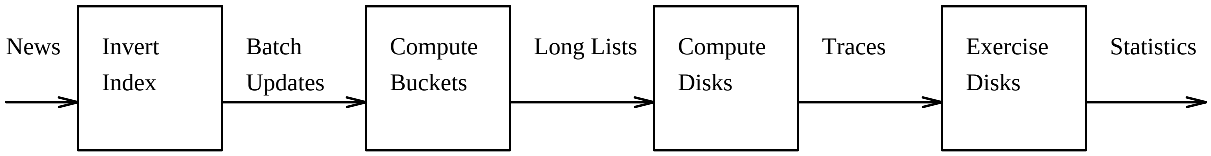

Figure 26: The flow of data for the experiment design. Arrows represent data. Boxes represent the transformation of data by a process.

5.3 Experiment Design

Figure 26 shows the flow of data for our experiments in building inverted indexes. Each arrow represents a data set and each box represents a process that transforms data sets. The diagram also serves as an outline for this section.

5.3.1 NEWS

The source text document database is 64 days of NEWS articles gathered from November 18th, 1992 to January 21st, 1993. (December 10th is missing.) See Table 18 for statistics on the database. Once per day the the local server was scanned for new documents. NEWS documents less than 2560 characters in length were eliminated to increase the average document size to a more typical range of about 5K characters. Also, non-English language documents (e.g., encoded binaries and pictures) were filtered out [TGMS93].

Each day of documents is a *batch* and is processed separately from other days. While the dual-structure index does not require periodic updates, this arrangement is good for measuring activity at periodic intervals.

5.3.2 Invert Index

The *invert index* process accepts a sequence of document batches as input, processes them, and generates a *batch update* for each batch. A batch update contains a

```
for years. And it was a total flop, in all the years it was available
very few people ever took advantage of it so it was dropped.
```

(a)

```
a advantage all and available dropped ever few flop for in it of
people so the took total very was years
```

(b)

Figure 27: (a) A fragment of a document from November 18th, 1992; (b) the tokens in sorted order.

abandons 1	abashed 2	abate 1
abated 1	abatement 1	abb 2

Table 20: A part of the batch update for November 18th, 1992, shown as pairs of words and the number of documents the word occurs in.

list of words that appear in the documents of the batch and the number of times each word occurs in the batch. A word and its frequency of occurrence is a *word-occurrence pair*.

To generate a batch update, each document in the batch is analyzed lexically to produce a token stream. Sequences of letters and sequences of numbers are tokens — all other characters are ignored. Certain lines of a document (such as "Date: " lines) are also ignored. Finally, duplicate tokens for a document are dropped. After all documents for a batch are reduced to sets of tokens, an inverted file is constructed for the batch. Tokens are converted to words by converting upper case letters to lower case. The batch update containing all the words and the lengths of the inverted lists for each word is then constructed. Figure 27a shows a fragment of a document and Figure 27b shows the resulting set of tokens. Table 20 shows a part of a batch update. Note that the misspellings of words are part of the batch update as well. At this point, all words in batch updates are converted to unique integers to simplify the

Variable	Value	Description
$Buckets$	1,500	Number of buckets
$BucketSize$	6,500	Size of bucket
$BucketTotal$	9.75 M	$Buckets \cdot BucketSize$
$BlockPosting$	683	Postings per Block
$Disks$	3	Number of Disks
$BlockSize$	4,096	Bytes per Block
$BufferBlock$	400	I/O buffer memory

Table 21: The experimental parameters and base-case values.

```
0 0
125144 4746
133663 4123
180761 4084
124376 3637
313269 4567
```

Figure 28: A part of the output of the compute buckets process. Each line is a word-occurrence pair.

remaining computations.

An implementation of an information retrieval system proceeds in the same way we have described here, except that it would keep, for each word, its complete inverted list, as opposed to the simple word-occurrence pair we keep here. For our performance evaluation, we do not need to know the contents of each inverted list, only its size, which is what the word-occurrence pair gives us. Thus, our batch update is our representation of the in-memory index of Section 5.1.

5.3.3 Compute Buckets

The *compute buckets* process takes the sequence of batch updates as inputs, runs the bucket algorithm described in Section 5.1 on the sequence and generates a single trace file of updates to long lists. Each update in the file indicates the word involved, and the number of postings to be add to the corresponding long list on disk. Postings for an update can come from the new postings in a batch or from previous postings in a bucket. In addition, a marker for the end of each batch update is added to the trace. Figure 28 shows a part of the output of the compute buckets process. The left column contains integers representing words. (Words are numbered alphabetically.) The right column contains the lengths of the corresponding in-memory lists. The line "0 0" indicates the end of a batch update. For instance, in the second line of this figure, 125,144 is the unique identifier for a particular word and 4,746 is the number of postings to be appended to the long list for that word.

Table 21 lists the variables that control the bucket computation and the base values used for those variables in the experiments reported in the next section. *Buckets* records the number of buckets, *BucketSize* records the size of each bucket (we count 1 for each word and posting placed in a bucket). *BucketSize* implicitly models the efficiency of the compression algorithm applied to in-memory inverted lists since computations are in terms of postings instead of bytes. The remaining variables in this table are described in the next section.

An information retrieval system would perform a similar computation using inverted lists as the compute bucket process does using word-occurrence pairs. An implementation would produce the same set of long lists. We assume that during the update process the buckets are kept in memory since they are referenced much more frequently than the long lists. At the end of each batch update, all buckets are flushed to disk. The cost of maintaining all the buckets in memory during the update process can be avoided by sorting the in-memory lists into bucket order and then merging the in-memory list with the buckets, requiring only one bucket to be in memory at any single point in time.

```
update bucket disk 0 id 0 size 1587
update bucket disk 1 id 0 size 1587
update bucket disk 2 id 0 size 1587
update chunk disk 0 id 0 size 0
write word 125144 posting 4746 disk 1 id 1587 size 7
write word 133663 posting 4123 disk 2 id 1587 size 7
write word 180761 posting 4084 disk 0 id 1587 size 6
write word 124376 posting 3637 disk 1 id 1594 size 6
write word 313269 posting 4567 disk 2 id 1594 size 7
```

Figure 29: An I/O trace corresponding to the previous figure.

5.3.4 Compute Disks

The *compute disks* process takes as input the trace file of long list updates and computes the sequence of I/O system calls required to implement the policies described in Section 4.2. In addition, the write operations for saving the buckets and the directory are added at the end of each batch update. Figure 29 shows a sample of a I/O trace file. The first three lines indicate that the write of the bucket data structure occurs on three disks starting at location 0 and continuing for 1,587 blocks. The next line writes an empty directory. (The directory is empty because this sample is the beginning of the trace, i.e., no long lists have been written to disk — no actual I/O is performed for this line). The following lines write inverted lists for each word. For instance, the first "write word" line indicates that word 125,144 writes 4,746 postings on disk 1 starting at block 1,587 for a size of 7 blocks.

5.3.5 Exercise Disks

The *exercise disks* process takes a trace of I/O operations as input and executes it on an IBM RS 6000 Model 350 computer (64 MB memory, UNIX AIX 3.2 operating system) with 3 disks (Seagate ST41200NM, 1 GB capacity, 5.25 inch, SCSI-1 standard) and an I/O bus (SCSI-1 standard). Each line of the trace generates a read or write system call request and after the update of the buckets and the directory all system buffers are flushed to disk.

Requests to each disk are issued by independent processes to achieve maximum parallelism. Request are directed to "raw" partitions of the disk, bypassing the operating system's file-system and disk buffer pool. Our algorithms do not revisit the same blocks within a single batch, thus eliminating the advantage of a buffer pool. In addition, we assume that relevant data will not remain in buffers from one batch update to the next. Furthermore, bypassing the file system saves CPU overhead and results in slightly superior data rates. Finally, bypassing the operating system isolates our experiment design from effects introduced by the file system and thus experiments are independent of any particular file system implementation.

One drawback to using raw disk partitions is that the operating system obeys the disk requests exactly and does not coalesce adjacent write requests into single disk I/O operation. For this reason, the disk exerciser program does its own coalescing of I/O operations where possible without reordering the execution trace. To be faithful to real systems with a finite amount of buffering, the disk exerciser will only coalesce up to *BufferBlock* blocks (each of size *BlockSize*) in a single request.

One advantage of the experimental design is the decoupling of each process from the subsequent process, which permits varying parameters of a process to study the effects on the corresponding data transformation. However, the design rests on the assumption that the CPU costs of each process do not dominate the total computation time. To test this assumption, we tested an actual running information retrieval system. We selected the RUFUS system [SLS+93] and built an inverted index for 307 megabytes of documents from a collection of IBM internal bulletin board articles. These experiments confirm that I/O time dominates.

5.4 Results

An *experiment* is the execution of the sequence of processes described in Section 5.3 over the 64 days of collected data[2] using the defaults values given in Tables 19 and 21, except as otherwise noted. As in previous chapters, the values of variables are sometimes systematically varied to show the sensitivity of a variable to some measurement. An *update* refers to the incremental batch update of the index. Some measurements apply only to the update. Other measurements apply to the index that results from the sequence of updates, which we refer to as the *index after update*. For this section, the *final index* is the index produced after all the updates have been processed.

5.4.1 Compute Buckets

Tuning the size of the buckets and the number of buckets is a complex issue in itself. For the issues presented in this thesis, tuning of the bucket essentially affects the results presented here *uniformly*. Thus, while there are quantitative changes in the behavior of the system, the qualitative differences remain the same.

To show the behavior of the buckets, we measure the number of long lists in an update. For each word-occurrence pair in an update, we can categorize the word of the pair as one of three types: a *new* word, a *bucket* word (a word that is already in a bucket), or a *long* word (a word that has a long list).

Figure 30 shows, for each update, the fraction of words belonging to each category. Initially, all word-occurrence pairs contain new words since the buckets are empty and there are no long lists. This behavior drops off rapidly as the buckets fill up. Eventually, after about 40 updates, the fraction of new words per update stabilizes around 10%. The fraction of words in an update that are in buckets rises rapidly until about the 15th update. Since the majority of words are the same in every update, this

[2] We explore larger data sets in reference [TGMS93]

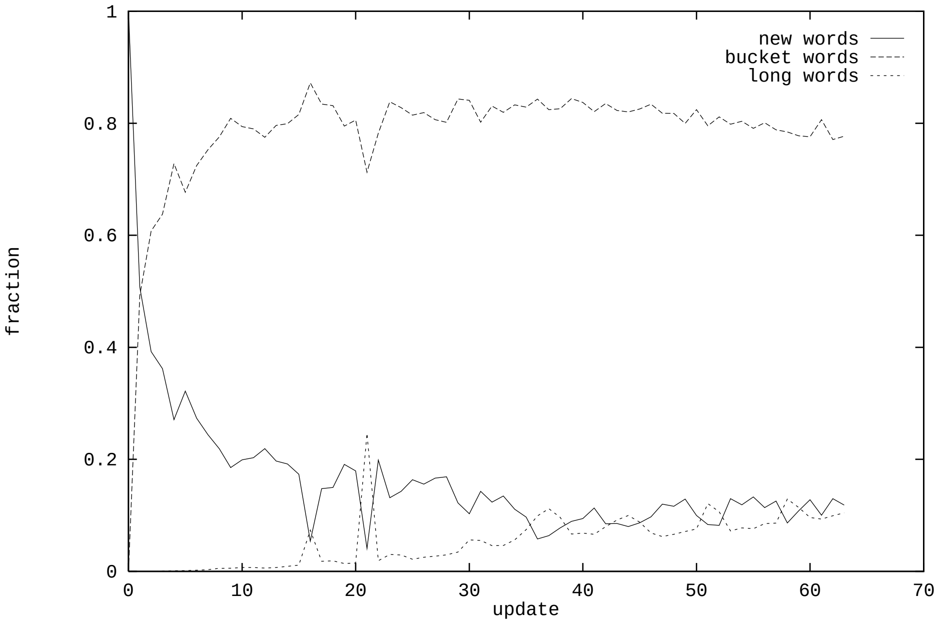

Figure 30: The fraction of words per update in each category.

rise indicates that the buckets are filling up with words. The curve declines (roughly linearly) after update 40 as new words (containing typically short in-memory lists) fill the buckets and cause words to overflow into long lists. Initially, no word-occurrence pairs contain long words because a few initial updates fit into the bucket. The fraction of long lists rises (roughly linearly) after the buckets fill up in the initial stage. The spike at update 21 is due to the small size of the update for that day introduced by an interruption in the gathering of data. Finally, the periodic peaks every seven days on the "long words" curve occur because each peak corresponds to a Saturday when the update is smallest. Small updates have higher fractions of frequently appearing words.

5.4.2 Compute Disks

I/O operations are our unit of measurement for this section only. Each I/O operation corresponds to a call to the operating system that results in a disk seek and transfer of information. Counting I/O operations only estimates the time taken for a sequence of I/O operations. We consider the actual time taken for I/O operations in Section 5.4.3, which presents the exercise disk. We study I/O operations in addition to actual times because they provide insights into the behavior of the long list policies, because a wider range of parameters can be studied, and because I/O operations closely estimate actual times. To compare allocation strategies, first we compare the three styles with zero reserved space to study the effect of in-place updates with respect to index build time, disk space utilization, and the query performance of the resulting long lists. Then allocation strategies are added to study the effects of reserved space for the same issues.

Styles

The number of I/O operations needed for each of the three policies is shown in Figure 31. When $Limit = z$, we use $Alloc = constant$ with a constant of 0 which removes the effect of the allocation policies. However, in-place updates are still possible by filling the empty space in the block(s) at the end of the list. The x-axis is the index after the given update. The y-axis is the *cumulative* number of I/O operations needed to build the index incrementally. Each curve is label with the values of *Style* and *Limit*.

All the curves in the graph have increasing slope, which means that the time to run each update takes longer as the index grows in size. This behavior is due to the increasing number of long lists. Second, the bottom two lines have $Limit = 0$ and the next two lines have $Limit = z$ for the new and fill styles. This behavior means that

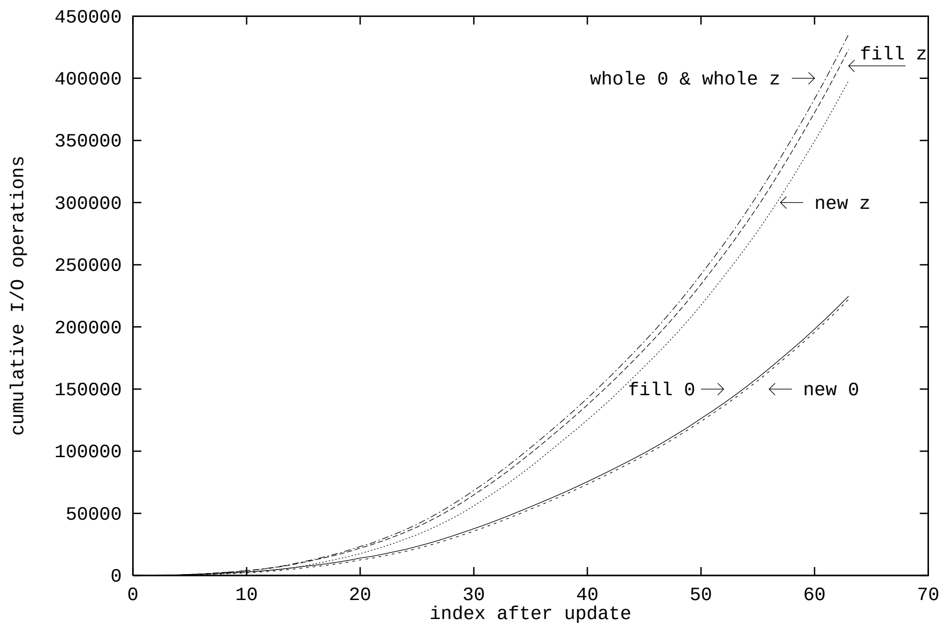

Figure 31: The *cumulative* number of I/O operations needed to build the final index.

in-place updates *double* the number of I/O operations required because each in-place update reads and writes. The graph shows that the whole style requires more I/O operations than either the fill or new style, regardless of the use of in-place updates. Since the whole style costs one read and one write for each append of an in-memory list to a long list, whether an in-place update occurs or not, the whole style is the upper bound in number of I/O operations for any style. The values for the final index for the whole style and for the fill and new styles with in-place updates are within 10% of each other. Thus, these policies use approximately the same time to build the final index.

Another measure of performance of a style is the long list utilization rate, namely the fraction of space allocated in long lists disk blocks that have postings. Thus we measure the *internal* utilization of the long lists. Figure 32 shows the long list utilization rate for the index, measured at the end of each update, for the same set

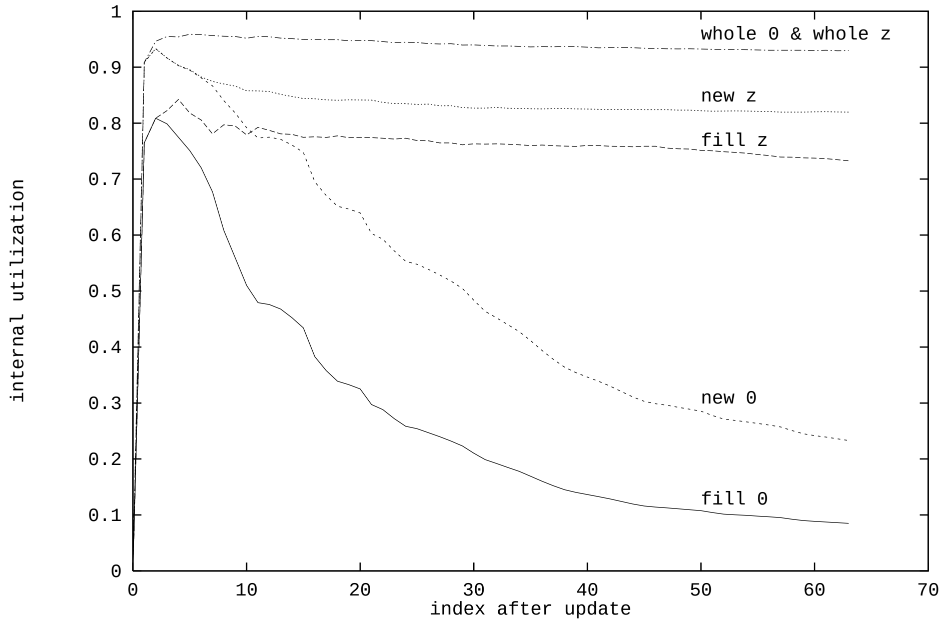

Figure 32: The long list disk utilization of the various policies.

of policies as the previous figure. The x-axis is the database after the update. The y-axis is the fraction of space in the long lists that contain postings. Each curve is label with the values of *Style* and *Limit*. The spike for all curves between update 0 and 1 is due to the utilization rate of 0 when there are no long lists. Utilization without in-place updates for the new and fill styles falls dramatically because there are large amounts of wasted space for small in-memory lists for these styles. Adding in-place updates to the new and fill style permits blocks to be used more efficiently. The whole style has good utilization regardless of in-place updates since each list is stored contiguously.

Comparing the I/O performance of policies to their corresponding utilization rates, the two best performing policies have poor utilization rates. Thus, the doubling of the I/O operations for update cannot realistically be avoided. In choosing among the remaining alternatives, if update performance is crucial then the new style with

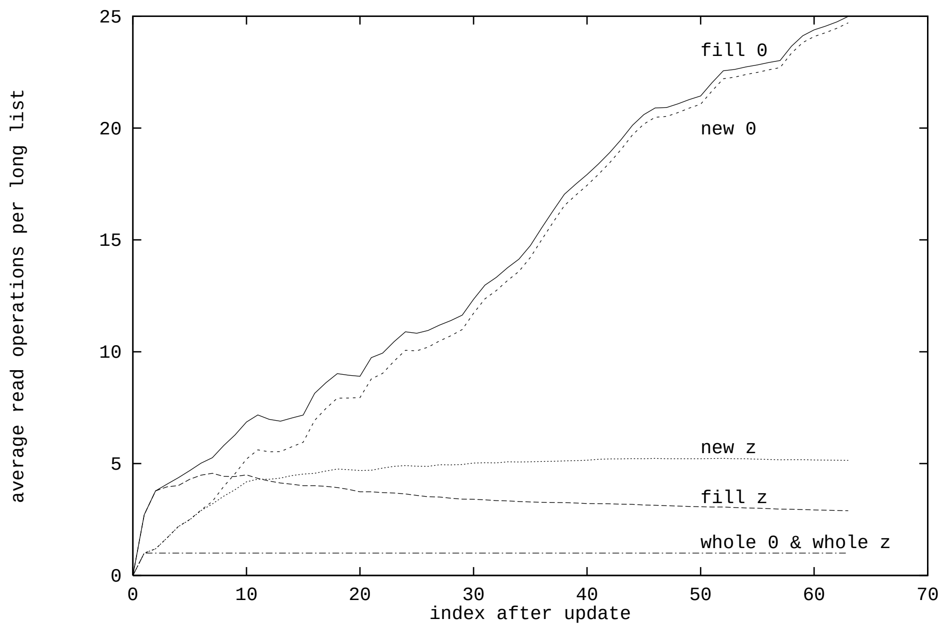

Figure 33: The average number of read operations to read a word with a long list.

in-place updates is best, and if utilization is crucial then either of the whole policies is best.

Measuring query performance for a policy is difficult since the typical workload depends on the information retrieval model (IRM). For a typical boolean IRM, a query contains a few words (less than 50) and the words tend to be the less frequently appearing words since frequently appearing words do not discriminate strongly between documents. Thus we would expect many query words to reside in buckets for this model. For a typical vector space IRM, a query may be derived from a document, consequently the query often contains many words (more than 100) and the words tend to be frequently appearing words. We concentrate on the vector space IRM for this chapter (see reference [TGMS93] for results on the boolean IRM).

For a vector space IRM, we assume the distribution of words in a query approximates the frequency of words in documents. To measure query performance, we measure at the end of each update the average number of read operations needed to read a long word, which is computed by counting the total number of chunks in the index and dividing by the number of words with long lists. Figure 33 graphs the results for the various policies and shows that in-place updates are needed for competitive query performance for the new and fill styles. The x-axis is the database after the given update. The y-axis is the average number of read operations to read a long list. In the final index, the whole style performs about 2.8 times better than the fill style with in-place updates and about 5 times better than the new style with in-place updates.

Allocation Strategies

There are several issues to consider with the allocation strategies. How is the constant value for an allocation strategy selected? Given an allocation constant, is there some rule to select its value independent of a policy? Given any style, is one of the allocation strategies best? Let us start by focusing on a particular style, say the new style. We assume in-place updates since allocation strategies are not otherwise used.

As the amount of reserved space for each list rises (by increasing k), the number of in-place updates rises and behavior converges towards a style where most updates long lists are in-place. In addition, as the amount of reserved space rises the disk utilization falls and the average number of reads for a long list approaches 1. This behavior presents a classical trade-off between disk utilization and query performance. Experiments described in reference [TGMS93] describe this trade-off in more detail. In-place updates also increase the update time for the new style, but the range of update times for in-place updates is only 10% in terms of I/O operations (see Figure 31). Thus, allocation strategies can only have a small impact on update time.

Table 22 compares various allocation strategies and constants for the new style. The "Read" column is the average number of read operations required to read a long

Allocation	k	Read	Utilization	In-place	Fraction
constant	500	3.60	0.78	189865	0.89
constant	1000	2.66	0.73	198292	0.93
block	2	3.36	0.78	192059	0.90
block	3	2.61	0.73	198746	0.93
proportional	2.0	2.89	0.80	196271	0.92
proportional	3.0	1.88	0.73	205316	0.96

Table 22: A comparison of allocation strategies with respect to the final index for the new styles.

Allocation	k	Utilization	In-place	Fraction
constant	0	0.93	179603	0.84
constant	500	0.90	188836	0.89
constant	1000	0.82	198309	0.93
block	2	0.87	194003	0.91
block	3	0.80	200156	0.94
block	5	0.68	205735	0.96
proportional	1.1	0.90	193695	0.91
proportional	1.25	0.85	202087	0.95
proportional	2.0	0.67	209994	0.98

Table 23: A comparison of allocation strategies with respect to the final index for the whole style.

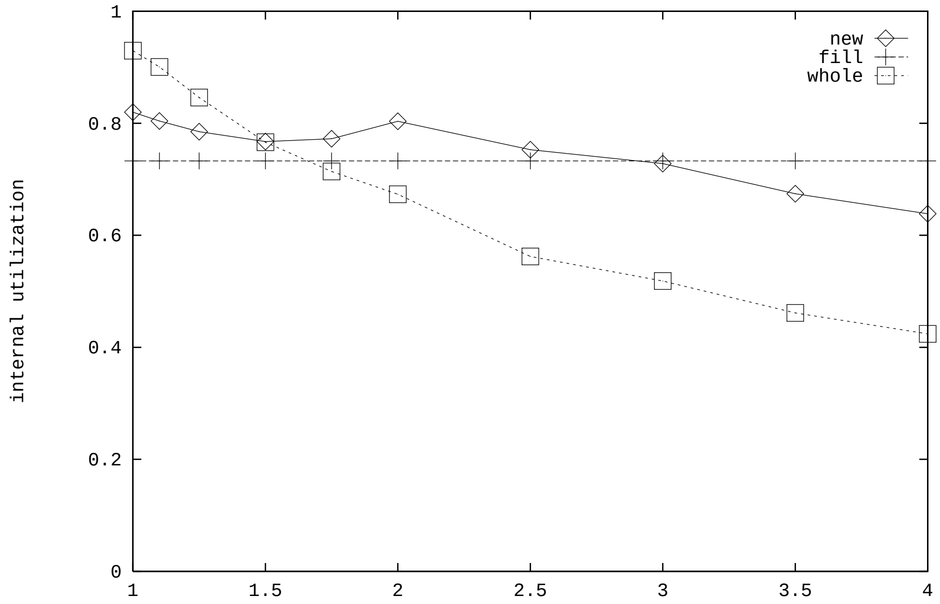

Figure 34: The impact of the constant k for the proportional allocation strategy on the utilization of long lists in the final index. The fill style (with the extent allocation strategy with extent size 3) is include for comparison.

list. The "Utilization" column is the internal utilization of the long lists. The "In-place" column is the total number of in-place updates needed to build the final index incrementally. The "Fraction" column is the fraction of in-place updates of the total possible number of in-place updates. (The total possible number of in-place updates is 213,256.) The "In-place" and "Fraction" columns are included only for comparisons with Table 23. The constant value for each strategy was chosen by increasing it until long list utilization was at 73%. This utilization rate was chosen because it offered good read performance, which was not available at higher long list utilization rates. Some additional values of interest are also included in the table. The table suggests (and other results not shown here confirm) that the new style with a proportional allocation strategy offers the best trade-off by having the best read performance at this level of utilization.

There is also a space-time trade-off for the allocation strategies for the whole style. The space trade-off is the utilization of long lists (as for the new style) but the time trade-off is only update time, not query performance, since all allocation strategies offer the same query performance. To compare update time, we cannot count I/O operations since this measure not distinguish between reading the tail of a list to append an in-memory list and reading the entire list, so we compare the number of in-place updates for each allocation strategy directly.

Table 23 shows statistics for various allocation strategies. The number of read operations for a long list is always 1 with the whole style. The "Util" column is the internal long list utilization. The "In-place" column is the total number of in-place updates needed to build the final index incrementally. The "Frac" column is the fraction of in-place updates of the total possible number of in-place updates. The table shows that the proportional allocation strategy is the best overall strategy since it is the only strategy to offer at least 90% for both utilization and the fraction of in-place updates.

Recall that the fill style has its own extent allocation strategy. The same space-time trade-off as with the new strategy exists. So, as the number of extents e is increased, disk utilization falls and query performance improves. We conducted the same analysis for this style as for the others, increasing e until utilization falls to 73%. Our experiments show that a value of 3 for e gives an average number of read operations for a long list of 2.90, and 198,746 in-place updates. Both of these performance measures are worse than the best new style policy. However, the fill style as an advantage of limiting the maximum required contiguous region of disk (in this case to 3 blocks, since e has a value of 3).

We have seen that the proportional allocation strategy is a good choice for the new and whole styles. We now consider the selection a good constant k for this allocation strategy. Figure 34 shows the impact of varying k on the utilization of long lists. The

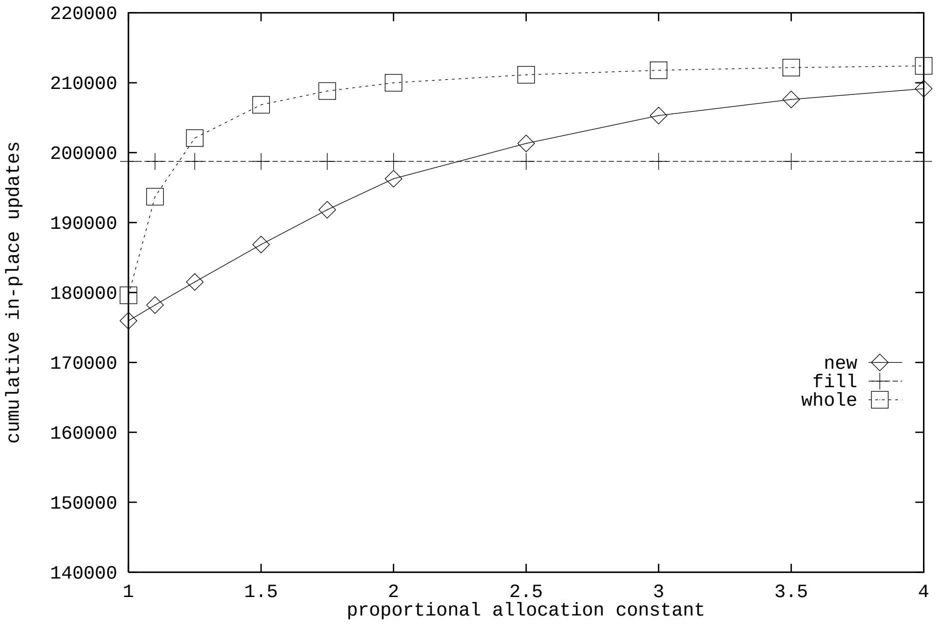

Figure 35: The impact of the constant k for the proportional allocation strategy on the cumulative number of in-place updates that occur in building the final index. The fill style (with the extent allocation strategy with extent size 3) is include for comparison.

figure shows that, generally, as k rises, the utilization falls for both the new and whole styles (the fill style does not interact with the proportional allocation strategy and it is included for comparison only). However, there is a cusp in the new style at 2. This cusp is because multiple updates to the same word have approximately the same length. A constant value of 2 reserves space for one additional in-place update. The simultaneous increase in in-place updates is shown in Figure 35. The y-axis starts at 140,000. There is only a marginal improvement from 84% of the in-place updates at a constant value of 1.0 to 100% of the in-place updates at a constant value of 4.0 is possible. Considering both figures, we see the the majority of gains are from constant values less or equal to 3.0. Based on the trade-offs presented, we recommend the proportional allocation scheme with a constant of 1.1 for the whole style and 3.0 for the new style.

5.4.3 Exercise Disks

Figure 36 shows the cumulative time taken to build the final index incrementally. The x-axis is the index after the given update. The y-axis is the cumulative time needed to build the index incrementally. Each curve is labeled with the values of *Style* and *Limit*. The fill style without in-place updates (fill 0) is not shown since our disks were not large enough to store the long lists for this policy due to gross underutilization of disk space (see Figure 32). The range of cumulative times for the final index varies by a factor of 4 as opposed to a factor of 2 determined by comparing total I/O operations (see Figure 31). The significant difference between the policies implies that a policy must be chosen with care.

Comparing Figure 36 with Figure 31, we see that measuring cumulative I/O operations produces the same *qualitative* comparison of policies as measuring real execution time. That is, the ordering of policies from best to worst is the same (accounting for the addition of whole with in-place updates and the removal of fill without in-place

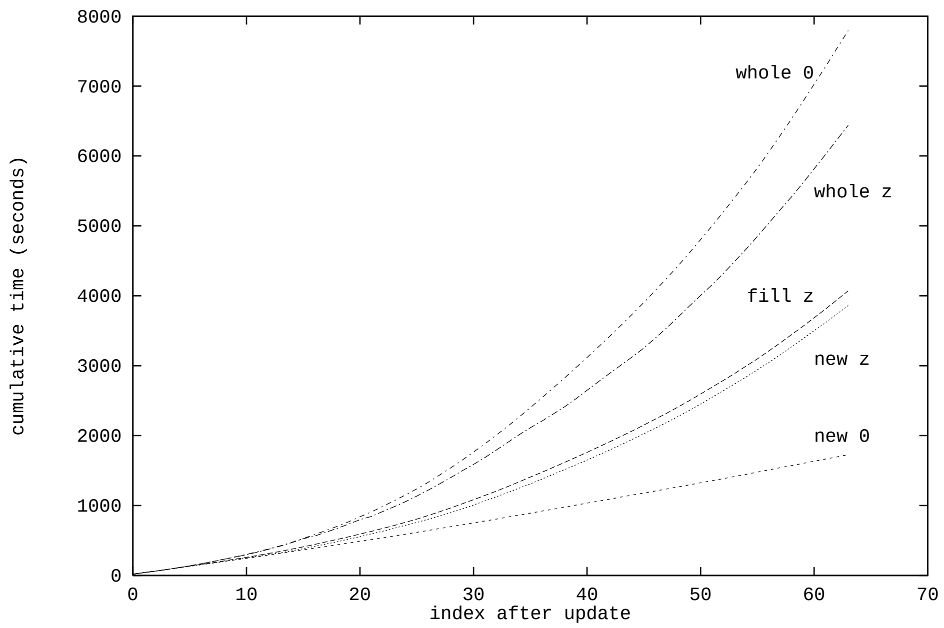

Figure 36: The *cumulative* time needed to build the final index.

updates). This result justifies using I/O operations to compare policies.

Also the new style with limit of 0 has an almost linear growth in the cumulative time taken as opposed to a more steep increase in the cumulative number of I/O operations. This behavior occurs because the exercise disk process coalesces I/O operations. That is, since for long list updates this policy only writes sequentially to the disk, all the write operations in an update can be coalesced (up to the buffer size imposed by the exercise disk process). Figure 36 also shows that the whole style without in-place updates takes the longest cumulative time to build the final index. This behavior is due to the additional movement of long lists compared to in-place updates.

Figure 37 shows the time taken to perform each update; this figure is the non-cumulative version of the previous figure. The x-axis is the update number. The y-axis is the time to execute the update by the exercise disk process. Each curve

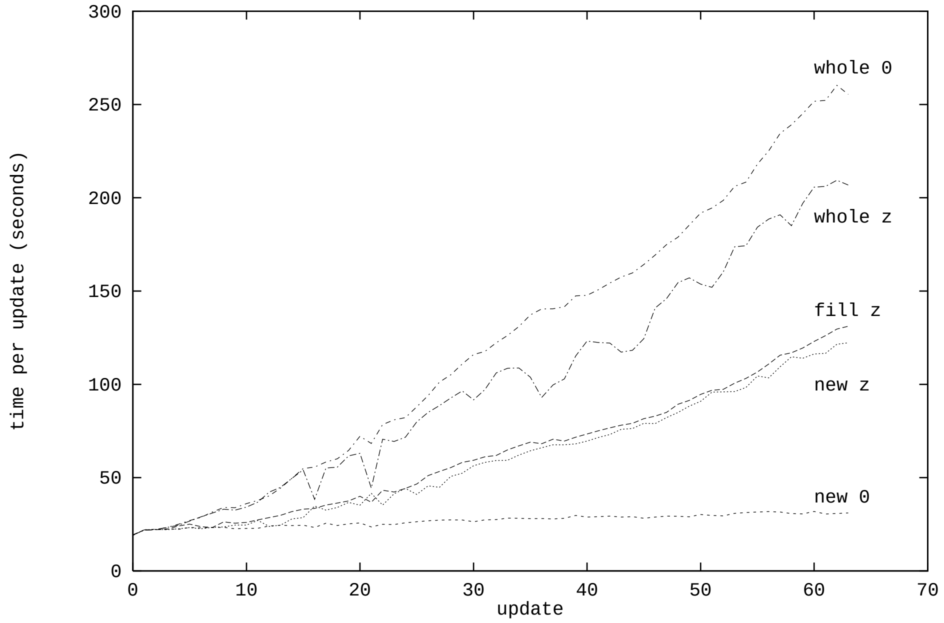

Figure 37: The time per update.

is labeled with values of *Style* and *Limit*. The update times grow over time as the number of long lists in the index grows. However, the increase for new style without in-place updates is slight because updates to different long lists are coalesced into single I/O operations. A second effect shown in the figure is that the whole style with in-place updates ($Limit = z$) is the only policy whose per update time is sensitive to the variations in the size of the update. The average number of in-place updates for this policy is sensitive to the average number of postings in a long list update.

5.5 Conclusion

In dynamic, time-critical text-document databases, it is important to modify index structures in place as documents arrive. Our dual-structure index strategy to addresses this problem. Comparing the results presented here with the literature, we have argued that the dual-structure index has better performance than existing implementations and provides incremental updates. The principle sources of the improvements are the dynamic dividing the postings into short and long inverted lists and using appropriate data structures for each type of list.

In studying our index, we found a classical trade-off between update time and query time. That is, the time spent incrementally updating the index is repaid with better query performance. Performance varies by a factor of 4 in the time to build an index and a factor of 25 in query performance. Another classical trade-off was found between space and time. As the amount of space wasted in storing long inverted lists rises, the query performance to read those inverted lists falls. We described three different methods for allocating additional space on disk to improve query performance and quantitatively describe the trade-off for these methods. In addition, we quantitatively compared overall performance.

Generally, the schemes that rewrite the unused space at the end of a long list before allocating more space take considerably longer than the schemes that don't. However, the extra space consumed by *not* rewriting the tail of long lists makes that option impractical for most applications.

While we have analyzed various situations, a designer of an information retrieval system is faced with the issue of chosing a policy. Is there a best policy? In general, no policy is best. Some policies favor update time and others favor query time.

New style

The new style provides the best update performance, since only the last block of each long list need ever be read during the updates. The new style without in-place updates is the best if update time is critical. The policy offers a factor of 2 in update time over the next best policy and a factor of 4 over the slowest policy. However, the policy exhibits poor query time and disk utilization. The new style with proportional allocation with constant 3 compensates for the poor query time and disk utilization. This policy is best if update time is important with reasonable query time. The policy is faster by a factor of 2 over the slowest policy and offers query performance within a factor of 2 of the best query performance. Use the new style with a proportional allocation strategy with a constant of 3.0 only if query performance is not critical.

Fill style

The fill style offers no advantages over the new style except that the maximum contiguous section of disk required is limited (it is unbounded in the new style). This requirement has an advantage in that long lists are automatically divided into sections of disks that can be written to disk and read in parallel (e.g., with a disk array). The cost of satisfying this requirement is small in the case of the fill style with extent allocation with constant 3 since this policy has slightly worse performance on all three metrics than the new style with proportional allocation with constant 3.

Whole style

The major advantage of the whole style is its guarantee of 1 read operation for any long list. Providing this guarantee has a cost in index build time. The penalty arises from the costs of moving long lists to keep them sequential. However, this penalty is not as large as might be expected, due the relative efficiency of performing sequential disk reads and writes. The whole style without in-place updates has an about 12% slower update time compared to the whole style with a proportional allocation with constant 1.1. The latter policy has a long list utilization rate of 90%. Thus the latter policy is the best if query time is critical. Use the whole style with a proportional allocation strategy with a constant value of 1.1 if query performance is critical.

An important issue is selecting the right amount of space for buckets and partitioning this space into the right number of buckets. In reference [TGMS93], we illustrate the trade-offs involved, but a more detailed study is required. We also need to study how to grow the bucket space dynamically since, unfortunately, as the size of the index grows from the addition of more documents the performance of the index degrades. This behavior implies that we need a strategy to rebalance the division between short and long lists. Some possible strategies include periodically rebalancing or rebalancing as the buckets are read and written for each update.

Our results are also limited because we considered only a relatively small database of 686 MB. In reference [TGMS93], we generate synthetic databases with the same characteristics as our real database. Those results indicate that, given the correct parameters, our algorithms scale well to larger databases.

Chapter 6

An Alternative Storage Technology

In this chapter, the implementation of an alternative storage technology (AST) for an information retrieval system is described. AST implements the algorithms described in Chapter 5. We describe the implementation of WAIS, the implementation of AST, and reports some performance comparisons between the two systems.

The AST implementation is based on replacing the underlying inverted index storage structure of the WAIS information retrieval system. The remaining infrastructure, such as the document management, the user interface, and the information retrieval model offered by WAIS are retained in both implementations. From an end user's point of view, both implementations behave identically since they return identical answers for any given query of a database. From an administrator's point of view, AST offers faster and more flexible updates.

One feature of WAIS was removed from both the WAIS and AST implementations. When new files are appended to an existing inverted index, the WAIS implementation checks to see if the new files have already been indexed. The checking algorithm is $O(n^2)$ where n is the number of file names. The checking algorithm searches the file of filenames, so the constant associated with the algorithm is large. With our experiments, the number of files is large and the checking algorithm dominates the total computation time. Every file is unique in our experiments, so the checking algorithm was removed from the WAIS and AST implementations. The checking algorithm probably should be replaced with more efficient algorithm. Removing the check caused the execution time of one experiment to drop from 36 hours to 12 hours.

Finally, the current AST implementation is limited in certain ways. With respect to the WAIS implementation, AST does not implement boolean or exact match searching. With respect to the simulation, AST works with a single disk (as does the WAIS implementation) whereas the simulation handles multiple disks. AST can be easily modified to handle these features.

6.1 Design of WAIS

To build a database of documents, WAIS parses each document and generates the postings for each word. WAIS maintains five files for this infrastructure. The *catalog* file contains a description of the documents in the index. This file is returned to the user if a search produces no answer. The *document* file maps document to files (there may be multiple documents per file). The *filename* file records the name and type of each file. The *headline* file records a one line description of each file. Finally, the *source* file records the server description of the database. AST uses this infrastructure also.

In WAIS, two files contain the inverted index for the database. The *inverted index* file contains the inverted lists for the words. The *dictionary* file is a dictionary (or index) that maps words to their inverted lists in the inverted list file.

To build inverted indexes of documents, WAIS proceeds as shown in the pseudocode of Figure 38. WAIS repeatedly (line 1) reads documents, parses them (line 3)

```
1.  while documents remain do
2.      while in-memory inverted index not full do
3.          parse next document
4.          insert postings into in-memory inverted index
5.      end
6.      sort in-memory inverted index alphabetically
7.      flush in-memory inverted index to new temporary file
8.      free in-memory inverted index
9.      merge temporary files
10. end
11. merge temporary files into final inverted index
12. build catalog
```

Figure 38: The WAIS pseudocode for building inverted indexes.

Name	Bytes	Description
INDEX_BLOCK_FLAG	1	Flag indicating type of block
NEXT_INDEX_BLOCK	4	Offset to the next index block
INDEX_BLOCK_SIZE	2	Size of the index block
NUMBER_OF_OCCURRENCES	4	Number of postings for the word
Word	≤ 21	String representation of the word

Table 24: The fields of the dictionary (or index) block of WAIS temporary files.

and inserts the resulting postings into an in-memory index (line 4). The in-memory index is implemented as a hash table with open addressing. The elements of the hash table are records containing a word and its associated inverted list. This processing continues (line 2) until the in-memory index is "full" (that is, it holds certain number of postings). At this point, the in-memory index is sorted (line 6) and flushed to a new temporary file (line 7), and the in-memory index is freed (line 8). The new temporary files are then merged (line 9) as described below. This processing continues until all documents have been processed.

The format of the temporary files consists of a pair of records for each word. The

Name	Bytes	Description
DOCUMENT_ID	4	Document identifier for this posting
CHARACTER_POSITION	3	Position of posting in document
WEIGHT	1	Weight or relevance of the posting

Table 25: The fields of the posting block of WAIS temporary files.

first record of the pair is a variable length dictionary (or index) record that contains the word itself and data that describes the format of the second record of the pair. The fields for the dictionary record are described in Table 24. The second record of the pair is a variable length list for the word. Table 25 describes the fields for each posting. Each pair of records for a word contains all the information for that word. Thus, little state information is maintained in memory between each flush of the in-memory index.

A temporary file is always written sequentially. After several temporary files are written, they are merged into larger temporary files of the same format. (The merging operation saves some disk space). Merging two temporary files of length n_1 and n_2 takes time $O(n_1 + n_2)$. The exact sequence of merges of temporary files into larger temporary files varies across the implementations of WAIS. The exact sequence is unimportant here. Finally, when all documents have been processed, the last in-memory index is written to a temporary file. The existing temporary files are merge into one final inverted index (line 11).

At this point (line 11) a temporary copies and the final index itself exist, so the total disk requirement for this implementation is about twice the size of the final index. During the generation of the final index, a dictionary file is also created. The dictionary file contains the words and offsets to each word's list in the final inverted index.

```
1.  while documents remain do
2.      while in-memory inverted index not full do
3.          parse document
4.          insert postings into in-memory inverted index
5.      end
6.      sort in-memory inverted index by bucket then alphabetically
7.      for each bucket
8.          read bucket
9.          merge in-memory inverted index for bucket and bucket
10.         queue overflow and long inverted lists
11.         write new bucket
12.     end
13.     process queue as long inverted lists
14. end
15. build catalog
```

Figure 39: The AST pseudocode for building inverted indexes.

6.2 Design of the Alternative Storage Technology

The implementation is a modification of the freeWAIS version of WAIS produced by CNIDR [FRE92]. The document parsing and construction of the in-memory inverted index infrastructure is retained. The build of the catalog is also retained. The following are dropped from the implementation: sorting the in-memory index, flushing it to a temporary file, merging temporary files, and building the dictionary. Thus, the catalog file, the document file, the filename file, the headline file and the source file are identical in both implementations since these files are related to the parsing of documents and files and the organization of the server information.

Figure 39 shows the pseudocode for the AST implementation. Lines 1-4 are the same as in Figure 38. Thus documents are processed in an identical way up to the construction of the in-memory inverted index. For the AST implementation, when the in-memory inverted index is full, the dual structure index is updated as described in Chapter 5. This update is done in three stages. First, the inverted lists in the in-memory inverted index are sorted into bucket order and within each bucket are sorted (line 6). This sort partitions the in-memory index into the inverted lists for each bucket. Next, each bucket is read (line 8) and the corresponding inverted lists for the bucket are merged (line 9) with the inverted lists read from the bucket. Since the lists for an in-memory bucket within the index and the lists in the bucket are in alphabetical order, the merge is linear time in the sum of the sizes of these lists.

During the course of the merge of a partition of the in-memory inverted index and a bucket, each word and its inverted list is classified into one of five types: a word that has never appeared before, a word that has a bucket inverted list, an word that has a long inverted list (as defined in Chapter 5), a word that another inverted list already queued, and a word (with the longest inverted list in the bucket) that overflows from the bucket. The first two types are handled by the merge operation with the bucket. The last three types are handled by adding the word and its inverted list to a queue (line 10). The checking of each inverted list to determine its type is a hash-table search.

After the buckets are processed, the queued lists are processed (line 13). Every list in the queue is a long inverted list. They are processed according to the algorithms in Chapter 5. The system administrator chooses among the new, fill, and whole policies and the constant associated with the proportional allocation scheme. Because of the results of Chapter 5, only the proportional allocation scheme was implemented since it offers superior performance.

During the course of the merge the lists of the in-memory inverted index are copied to another data structure of essentially the same form. This copy is technically unnecessary. It is simply a by-product of the integration of the AST implementation with the WAIS implementation. This copy doubles the space needed to represent the in-memory inverted index in the AST implementation and introduces a small number

Name	Bytes	Description
word	≤ 21	String representation of the word
offset	2	Index offsets to the posting list
occur	2	Number of occurrences of a word
posting	8	A posting

Table 26: The fields of the bucket format on disk.

Bytes	Value	Description
4	"rip"	The string for a word (with a null byte terminator)
8	"roaring"	The string for a word (with a null byte terminator)
1	0	The null string (a null byte terminator)
1	0	An alignment byte for the offset table
2	3	The start offset
2	4	The end offset for the word "rip"
2	1	The number of occurrences for the word "rip"
2	5	The end offset for the word "roaring"
2	1	The number of occurrences for the word "roaring"
4	0	Alignment bytes for the posting table
8		The posting for the word "rip"
8		The posting for the word "roaring"

Table 27: The bucket disk data structure for two words.

of page faults.

The fields of the bucket format on disk are listed in Table 26. It consists of a sequence of words, a sequence of offsets with one offset per word, and the sequence of inverted lists for each word. A word consists of the word as a null terminated string, and a null string terminates the sequence of words. Next is a sequence of offset indexes into the sequence of postings. Offsets have an offset index that points to the array of postings and a two-byte integer that gives the number of occurrences of the word. The first offset is the start of the posting lists. (It occurrence field is set to 0). Each subsequent offset points to the slot beyond the *end* of the corresponding

Name	Bytes	Description
padded_word	20	String representation of the word
extent	4	Pointer to the extent list
next	4	Next element in extent table list

Table 28: The fields of the extent table.

Name	Bytes	Description
block	4	Pointer to a block description
used	4	Number of postings used in this block
next	4	Next element in extent descriptions list

Table 29: The fields of an extent description.

word's posting list. An index is an offset represented by a 2-byte integer indicating the last posting of the inverted list for the word. (Only 2 bytes are needed because of the way we index an array of postings in the bucket.) The inverted lists are sequences of postings in the format described in Table 25.

Table 27 lists the bytes and contents for each field of a bucket that contains two words "rip" and "roaring" each with a single posting. The word sequence is the string "rip", a null character, the string "roaring", a null character, and a null string. Then there is a single null byte to align the subsequent offset table. The first offset in the offset table, for the word a two-byte integer index to the posting of the word rip (contains 3), a two byte integer index to the posting for the word roaring (contains 4), the posting for the word rip (8 bytes) and finally the posting for the word roaring (another 8 bytes).

An extent table records which words have long lists. The disk data structure and the in-memory data structure for the extent table are identical. It is a singly linked list of records, one record for each word with a long list; the fields are described in Table 28. To record where each extent for a word resides, the extent-table record

Name	Bytes	Description
id	4	The block identifier
size	4	Number of blocks in this block
next	4	Next element in the free block list

Table 30: The fields of a record of a region of the inverted file.

points to a linked list of extent descriptions. The fields of the records for the extent descriptions are given in Table 29. Each extent consists of one variable length region of the inverted file. A free-block manager deals with freed regions of the inverted file. The fields for the records describing each free region are described in Table 30. Each region is of a variable number of blocks. Every block is of the same fixed-sized and each record describes the starting block (numbered from the first block in the file) and the number of blocks for a region. For free block management, a list of records is maintained in increasing block order. This implementation requires linear time to insert a freed block but permits constant time coalescing of adjacent freed regions. We use a first fit algorithm for allocating of regions. There is always exactly one region for each extent description. The use of separate records for recording regions simplifies the implementation (at the expensive of some memory) since region records can be passed directly to the free-block manager when necessary.

Figure 40 shows the data structure layout for two words "cat" and "mouse." The word cat has two extent regions. The first extent contains 200 postings and is stored starting in block 14 in a region consisting of 1 block. The second extent contains 300 postings and is stored starting in block 17 in a region consisting of 2 blocks. The word mouse has one extent containing 1000 postings starting in block 28 in a region consisting of 2 blocks. Given the size of each posting and the size of a region, the number of free posting slots in any region can be computed.

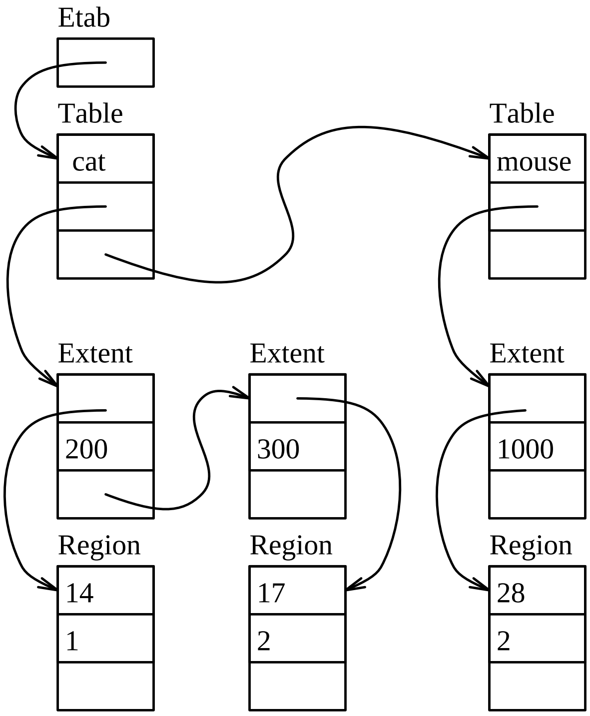

Figure 40: The data structure layout for two long list words "cat" and "mouse."

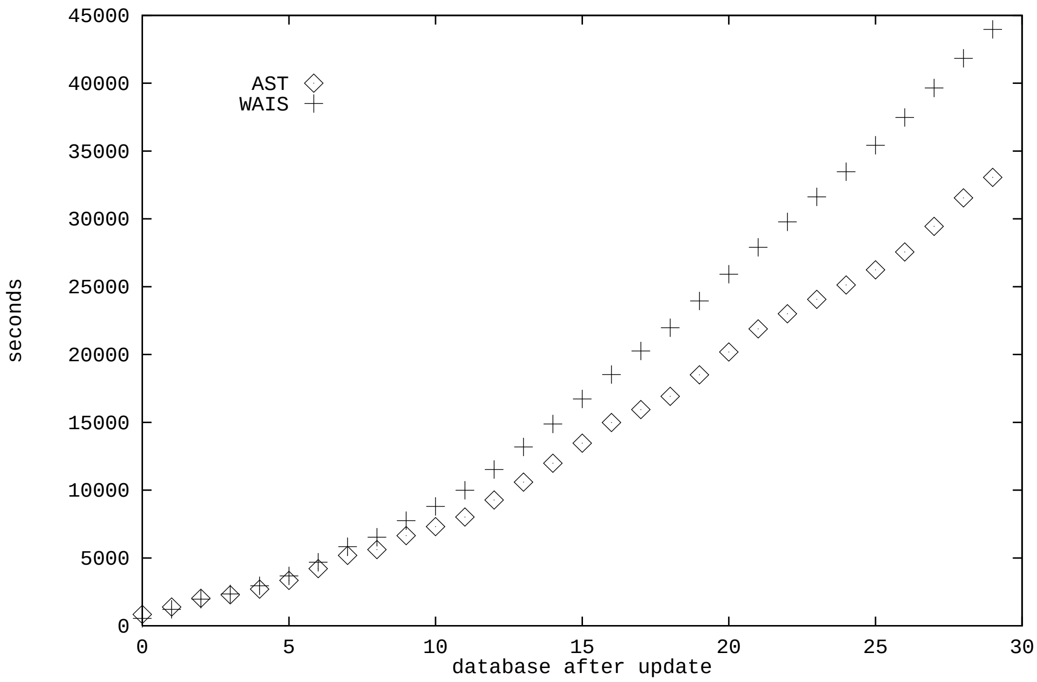

Figure 41: The *cumulative* time to build inverted index for the month of December for WAIS and AST with the new policy.

6.3 Results

To test the performance of our implementation, we built an index for the month of December from our experiment database described in Chapter 5. Figure 41 shows the cumulative build time for this data. The x-axis has only 30 data points since one day of data for the month of December is missing. The y-axis is the cumulative building time in seconds. Comparing this graph to Figure 36 shows that the implementation is lower than expected.

6.3.1 Simulation vs. Implementation

The difference in the scale of the graphs is due to the several differences between the simulations of that chapter and the AST implementation. We first consider the differences are based on variations in the modeling of the workload and implementation.

For modeling, in the simulation the vocabulary of the index is the set of words that appear in documents. The WAIS information retrieval model, however, in addition to the set of words in documents, also represents pairs of capitalized words in sequence as separate "words." Thus, the text fragment "the Thin Man" has four words, the three in the text fragment and an addition word "thinman." Thus, in comparing the simulation to the implementation, we must remember that the implementation's vocabulary is larger for a document database.

The implementation differs from the simulation in several ways. The implementation is executed over a single disk and the simulation uses three disks (a factor of 3 improvement). The simulation keeps the entire bucket data structure in memory. At the end of each batch update, the entire bucket data structure is written to disk. In the implementation, only one bucket is kept in memory at any time. This saves memory at a factor of 2 cost in I/O operations that deal with buckets. The simulation times assume that document processing and index building are overlapped disk and CPU operations (a factor of 2 improvement). Finally, the simulation writes to raw disk and the implementation writes to the file system (a factor of 2 improvement). These differences total a factor of 24 difference in performance.

The implementation of the bucket data structure requires about 11 bytes on average to represent a word and 8 bytes to represent a posting. The simulation represented a word as one unit and a posting as one unit, i.e. a word and posting have equal size. I/O operations are not explicitly sorted in the implementation. They are sorted by the file system. The simulation sorts the I/O operations explicitly. These two differences have a small performance impact.

6.3.2 WAIS vs. AST

In the WAIS implementation, there is no overlap between the building of the in-memory inverted index and the disk-based inverted index. Performance measurements were taken to measure the time spent in various parts of the implementation. Of the total 43,971 seconds, our measurements show that 10.9% (4,820 seconds) of the time taken to construct an index is spent in all of the processing needed to build an in-memory index. An additional 24.6% (10,822 seconds) of the time is spent maintaining the headline files, catalog files, etc. The largest fraction 61.8% (27,185 seconds) is spent merging inverted lists. Only 1.6% (723 seconds) of the time is spent writing in-memory indexes to temporary files. Finally, 0.9% (421 seconds) is spent sorting the in-memory index.

In the AST implementation, there is also no overlap between the building of the in-memory inverted index and the disk-based inverted index, since both implementations share this code. The total time to build the index is 33060 seconds. 15.2% (5,056 seconds) is spent building the in-memory index. The difference in the raw number of seconds in building in-memory indexes is due to the additional time spent opening the extent table catalog. Maintaining the headline, catalog etc. files takes 30.1% (9,970 seconds). Writing the postings to long lists takes 2.2% (734 seconds), and reading, writing and updating the bucket data structure takes 49.4% (16,338 seconds) of the total time. Clearly, there is room for improvement in the implementation of the bucket data structure. Finally, sorting the bucket data structure takes 2.7% (912 seconds). The larger sort time for sorting the bucket data structure in the AST implementation compared to the WAIS implementation is due to the increased complexity of the comparison function. In the AST implementation, each comparison requires a two hash computations and perhaps a string comparison; the WAIS implementation does only a string comparison is required.

Finally, the major difference is that the WAIS implementation discards words with long inverted lists. Thus, the curve for the WAIS implementation in Figure 41 does not represent the entire inverted index; the curve for the AST implementation does. This can be seen by examining the curve. The curve flattens out after update 11. Since the WAIS implementation is an $O(n^2)$ implementation, this curve should continue to climb in a nonlinear fashion. This feature of WAIS explains why the performance improvement of AST is relatively small.

The current implementation of AST does work correctly. The same set of documents for a given query are produced by both the AST and the WAIS implementation. However, performance improvements still remain to be done. We suggest the following improvements: expanding of the implementation to use multiple disks; using raw I/O instead of the file system; and overlapping of I/O operations and CPU operations.

Chapter 7

Conclusion

The falling costs of processors, main memories, and disks have encouraged storaging of increasing numbers of documents on-line. The advent of the National Information Infrastructure will provide access for millions of users to thousands of document repositories. As a result, there is renewed interest in efficient document indexing techniques.

In this thesis we demonstrate that distributed architectures can efficiently solve this problem. Since our architecture is manufactured on a commodity basis, this architecture is also cost effective. This thesis has quantified the engineering trade-offs of information retrieval systems. Essentially, we provide a guide to the engineer constructing such a system.

In Chapter 3 we explore four different physical organizations for inverted lists. The disk index organization constructs an inverted file for all the documents logically assigned to a disk. The I/O bus index organization constructs inverted lists for all the documents logically assigned to the disks of an I/O bus and distributes them among the disks attached to the I/O bus. The host index organization constructs an inverted lists for all the documents logically assigned to a processor and distributes them. The system index organization constructs inverted list for all documents and distributes them among all disks.

Each physical organization has an associated query processing algorithm. The disk, I/O bus and host index organizations are similar in that they issue identical subqueries to all processors in the system, process the subqueries independently, and then concatenate the subresults to produce the final result. However, each of these index organizations processes a subquery in a different way, depending on the number of disk and I/O buses attached to each host. The system index organization divides the query into different subqueries depending on the distribution among the processors of the keywords of the query. Each subquery is processed independently and the subresults are merged to produce the final result. As the number of processors increase the disk, I/O bus and host index organization issue more and more subqueries. The system index organization, however, is independent of the number of processors at least for boolean information retrieval models.

In addition to the query processing algorithm described above, we studied three optimizations of the system index organization. These optimizations are based on prefetching inverted lists in an attempt to reduce the amount of traffic on a local area network.

That chapter used simulation to analyze the performance of the four index organizations and seven query processing algorithms. Our work load is based on an analytic model. Generally, the host index organization works best, particularly if the inverted lists are striped across multiple disks. Because the inverted lists are so long in full-text systems (particularly those examined under our analytic model) the host index organization avoids the transmission of inverted lists across the processor interconnect — the bandwidth bottleneck of these systems.

In Chapter 4, we extended the work of the previous chapter by considering a different work load based on actual user traces gathered from the Stanford University FOLIO information retrieval system. These traces permit the study of a specific implementation and use of the physical index organizations and query processing algorithms. We also studied the impact of caching and the behavior of systems as the inverted lists were scaled and as the number of processors were scaled. In this situation, the system index organization worked the best since the inverted lists that are queried in the trace are generally short. We found that caching worked well in increasing query throughput since the architecture does less total I/O work per query. Caching had little effect on query response time, principally due to the I/O operations needed from cache misses (usually at least one per query). We studied database scaling of the inverted list and processor scaling. We found that some query optimization methods worked well as the processors scaled, others did not. One disadvantage of prefetching is that while it reduces the total amount of traffic on the network, it also decreases the amount of parallel processing done per query.

In Chapter 5 we return to an initial assumption of the previous two chapters. In typical commercial information retrieval systems each inverted list is stored contiguously on disk to maximize query performance. We assume contiguous inverted lists in the previous two chapters. However, this choice for the physical organization of inverted lists comes with a high cost in update time. Each time the inverted file is updated, the inverted file must be reorganized to keep each inverted list contiguous on disk. We invent new data structures and algorithms to address this problem. Essentially, we store infrequent words with short inverted list in a fixed-sized structure and we store frequent words with long inverted lists in a variable sized structure. The division between the two types of words is determined dynamically.

In this thesis, our methodology is based on applying the principles of distributed databases to this problem. We draw on work from distributed query optimization, indexing methods, system optimization issues, data structures, and caching techniques, among other issues, to construct efficient systems. We construct algorithms to optimize the incremental update problem and describe the engineering trade-offs involved. We quantitatively explore three strategies for laying out inverted lists. One strategy, for each incremental update, starts a new contiguous region of disk for each inverted lists. This strategy optimizes update time at the expense of query time. A second strategy organizes inverted lists into contiguous regions of disk. This strategy optimizes query time at the expense of update time. The third strategy organizes inverted lists into extent sized contiguous regions of disk. The strategies can be modified by permitting in-place updates. We quantitatively explore three strategies for determining the amount of space to reserve for in-place updates at the end of each long inverted list. One strategy adds space for a constant number of postings. A second strategy adds space in fixed blocks of postings (essentially rounding up to the next block size). A third strategy adds a reserved space that is proportional to the size of the inverted lists. We show that the update optimized strategy with in-place updates using a proportional reserved space strategy for inverted lists provides fast update times and reasonably good query performance. The query-optimized strategy performs well on space utilization.

Our performance evaluation is based on multiple methods. We use simulation in several forms and model user work load in three ways — analytically and trace-based for queries, and an actual work load for updates. While each type of performance evaluation has advantages and disadvantages, we found that calibrated simulation models offer flexibility, speed of construction, and realistic results. The quantitative basis for the study of database systems will become increasingly important.

For future work, we envision several avenues. Several specific issues that are direct extension of this thesis need to be addressed. There are hundreds of query optimization techniques in the database literature. Our study only scratches the surface of optimization of inverted lists. Two avenues that may be fruitful are pipelining of query processing and using indexes on inverted lists. In our simulations, an entire inverted list is read at once. In pipelining, only part of an inverted list is read at any one time for query processing. The advantage of pipelining is that any system resource is busy for a shorter period of time, which usually increases system throughput. Secondly, query response time for the *initial* answer will be reduced. However, the total work for the query will be increased. For indexes, we use no secondary indexing method to access some part of an inverted list randomly, i.e., entire inverted lists are always read. In the case of merging a long inverted list and a short one, it may be faster to read the short inverted list and then probe, via a secondary index, the long inverted list.

The update algorithms of Chapter 5 essentially apply to the host index organization described in the introduction. However, the system index organization offers performance advantages in some situations and the incremental update problem for this organization needs to be studied.

Constructing a prototype would permit the exploration of issues such as the performance impact of a file system, the impact of various nonlinear compression functions, and the impact of memory contention.

More generally, a merging of information retrieval and databases would be benefit both areas. For instance, some information retrieval systems automatically route new documents to users based on user profiles of interest. Selective dissemination of information can be implemented in an active rule-based system that processes information retrieval queries. Another example is distributed processing of queries across a wide area network. Initial studies [GGMT93] indicate that information retrieval across the INTERNET is possible in some situations. In the other direction, the information retrieval community has done extensive modeling of the effectiveness of queries. These models assume that the answer to a query is imprecise. Database theory has typically assumed that the model is a precise expression of the world. However, combining information retrieval and database systems will require that database models handle imprecise answers.

Appendix A

Derivation of the Probability Distribution Z

Given the curve fit equations, we wish to derive the form of the probability distribution Z, which accomplished by transforming the continuous curve fit equation from a logarithmic domain to a linear domain and then using this equation to approximate an integer probability distribution. The distribution that results from a linear curve fit is derived by introducing two auxiliary equations

$$x' = \ln x,\ y' = \ln y$$

that describe the relationship between the domains. The curve fit equation is

$$y' = mx' + b$$

and by replacement and exponentiating becomes

$$e^{\ln y} = e^{m \ln x + b}$$

which reduces to

$$y = e^{m \ln x} e^b = e^{\ln(x^m)} e^b = x^m e^b.$$

Typically a Zipf Harmonic function [Zip49] is used to approximate the distribution of the occurrences of high frequency words in a document. Such a function corresponds to a linear fit in log space. The definitions of the Zipf Harmonic function appear in reference [Tri82] as follows. Here, we model the distribution of all the words in the document, which simplifies the analysis and has little impact since we simulate only the high frequency words. To show this relationship, suppose for the moment that Z is this function. We arrange the probabilities of $Z(j)$ in nonincreasing order $Z(1) \geq \cdots \geq Z(T)$. Zipf's law states that

$$Z(i) = \frac{c}{i},\ 1 \leq i \leq T,$$

where the constant c is determined from the probability distribution normalization requirement, $\sum_{i=1}^{T} Z(i) = 1$. Thus $c = \frac{1}{H_T}$ where H_T is the T^{th} Harmonic number. Given this definition, we derive the linear form of the Zipf Harmonic function in log/log graphs as follows. Let

$$x' = \ln x,\ y' = \ln y$$

again describe the relationship between the the logarithmic and linear domains, and rewrite x as

$$e^{x'} = x$$

From the derivation above we can write

$$y = \frac{1}{H_T x}$$

for the equation of the distribution. By substitution,

$$y' = \ln \frac{1}{H_T x} = \ln 1 - \ln H_T - \ln x$$

we derive the linear form

$$y' = -x' - \ln H_T.$$

This derivation demonstrates that the Zipf Harmonic function is at best some linear fit on the data shown. However, Figure 2 shows that the quadratic fit is better an any linear fit.

Returning to the problem of determining equation Z from the quadratic fit, we can use a derivation similar to the one above:

$$x' = \ln x, \;\; y' = \ln y$$

$$y' = ax'^2 + bx' + c$$

$$e^{\ln y} = e^{a(\ln x)^2 + b \ln x + c}$$

$$y = e^{a(\ln x)^2} e^{b \ln x} e^c \;\; = e^{a \ln x \ln x} e^{\ln(x^b)} e^c \;\; = \left(e^{\ln(x^a)}\right)^{\ln x} e^{\ln(x^b)} e^c$$

which produces the general form

$$y = x^{a \ln x + b} e^c.$$

Thus, by using this continuous approximation to the integer probability distribution and extracting the values of a, b and c from the curve fit, we can express Z as

$$Z(j) = \frac{j^{-0.0752528 \ln j - 0.150669} e^{16.3027}}{8.47291 \times 10^8}$$

where the denominator is a normalization constant.

Appendix B

Derivation of the effect of u

In this appendix we derive the effect of u on the expected size of a query answer set. A document matches a query when every word that appears in the query also appears in the document. For expected number of documents to match a query of length K, we write

$$D \cdot \Pr(\text{query } Y \text{ of length } K \text{ matches document } A)$$

by the independence of documents;

$$D \cdot \sum_{Y \in \mathcal{S}} \Pr(Y) \Pr(Y \text{ matches } A \mid Y)$$

by the theorem of total probability. The conditional probability

$$\Pr(Y \text{ matches } A \mid Y)$$

$$\Pr((v_1, \ldots, v_K) \text{ matches } A \mid Y = (v_1, \ldots, v_K))$$

$$\Pr(v_1 \text{ matches } A) \cdots \Pr(v_K \text{ matches } A) \mid Y = (v_1, \ldots, v_K)$$

reduces to a multiplication by the independence of each match. The probability of a match of a word v and a document A

$$\Pr(v \text{ matches } A)$$

$$\Pr(v \text{ occurs at least once in } A)$$

$$1 - \Pr(v \text{ does not occur in } A)$$

$$1 - \Pr(v \text{ does not occur as } word_1, \ldots, word_W \text{ in } A)$$

$$1 - (1 - Z(v))^W$$

reduces to a simple function of Z and W by the independence of each word trial. Thus, by replacement, we arrive at the expected number of documents to match a query of size K:

$$D \cdot \sum_{Y=(v_1,\ldots,v_K)\in\mathcal{S}} \Pr(Y)[1-(1-Z(v_1))^W]\cdots[1-(1-Z(v_K))^W] \tag{4}$$

We can reduce this equation further by using the independence assumption about the set of queries $\mathcal{S}$. Let the words of a query be chosen independently according to a uniform distribution $Q(j)$, then $\Pr(Y) = (\frac{1}{uT})^K$ and

$$\frac{D}{(uT)^K} \sum_{(v_1,\ldots,v_K)\in\mathcal{S}} [1-(1-Z(v_1))^W]\cdots[1-(1-Z(v_K))^W]$$

becomes

$$\frac{D}{(uT)^K} \sum_{v_1\in V'} \cdots \sum_{v_K\in V'} [1-(1-Z(v_1))^W]\cdots[1-(1-Z(v_K))^W]$$

by independence of the words that appear in the query. (This assumption is tentative; some features of user interfaces such as thesauri and wild-cards will invalidate this assumption.) We rewrite this as

$$\frac{D}{(uT)^K} \sum_{v_1\in V'} [1-(1-Z(v_1))^W]\cdots \sum_{v_K\in V'} [1-(1-Z(v_K))^W]$$

and finally,

$$\frac{D}{(uT)^K} \left\{ \sum_{v=1}^{uT} [1-(1-Z(v))^W] \right\}^K .$$

In the equation above, the expression $1-(1-Z(v))^W$ can be viewed as the probability of at least one success in W trials where a success is determined by the distribution $Z(j)$. Since the summation in the above equation is difficult to compute, we approximate this expression by the use of a Poisson approximation of the Binomial theorem as follows. The probability of x successes of probability p in Y trials is the binomial distribution $b(x;Y,p)$. The Poisson distribution is $p(x;\lambda) = \frac{\lambda^x e^{-\lambda}}{x!}$. The approximation of the Binomial distribution by a Poisson distribution is by writing $\lambda = pY$, which is valid when $Y \geq 20$ and $p \leq 0.05$ [Tri82]. Let $Y = W$, $p = Z(j)$, $\lambda = WZ(j)$. The probability of 0 successes in the Poisson distribution is $p(0;\lambda) = e^{-\lambda}$. The probability of at least one success is $1 - e^{-\lambda}$. Thus, $1-(1-Z(v))^W = 1 - e^{-WZ(j)}$. The equation above can be rewritten as

$$\frac{D}{(uT)^K} \left\{ \sum_{v=1}^{uT} 1 - e^{-WZ(j)} \right\}^K .$$

We use Mathematica [Wol91] to perform the summation and using the parameter values in Table 3 and Equation 1 for Z, we graph this function for the various values of K and u in Figure 3.

Bibliography

[AS91] Ijsbrand Jan Aalbersberg and Frans Sijstermans. High-quality and high-performance full-text document retrieval: the parallel infoguide system. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 151–158, Miami Beach, Florida, 1991.

[BCCM93] Eric W. Brown, James P. Callan, Bruce Croft, and J. Eliot B. Moss. Supporting full-text information retrieval with a persistent object store. Technical Report 93-67, University of Massachusetts, Amherst, Department of Computer Science, August 1993.

[Bur90] Forbes J. Burkowski. Retrieval performance of a distributed text database utilizing a parallel processor document server. In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, pages 71–79, Dublin, Ireland, 1990.

[CD90] D. Chapman and S. DeFazio. Statistical characteristics of legal document databases. Technical report, Mead Data Central, Miamisburg, Ohio, 1990.

[CEMW90] Janey K. Cringean, Roger England, Gordon A. Manson, and Peter Willett. Parallel text searching in serial files using a processor farm. In

Proceedings of Special Interest Group on Information Retrieval (SIGIR), pages 429–453, 1990.

[Che90] Ann L. Chervenak. Performance measurements of the first RAID prototype. Technical Report UCB/UCD 90/574, University of California, Berkeley, May 1990.

[CP90] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, pages 405–411, 1990.

[DeF92] Samuel DeFazio. Document retrieval benchmark. Working Draft Version 1.2, Sequent Computer Systems, 1992.

[DeF93] Samuel DeFazio. Full-text document retrieval benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, chapter 8. Morgan Kaufmann, second edition, 1993.

[DH91] Samuel DeFazio and Joe Hull. Toward servicing textual database transactions on symmetric shared memory multiprocessors. In *Proceedings of the International Workshop on High Performance Transaction Systems*, Asilomar, 1991.

[Emr83] Perry Alan Emrath. *Page Indexing for Textual Information Retrieval Systems.* PhD thesis, University of Illinois at Urbana-Champaign, October 1983.

[Fal85] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17:50–74, 1985.

[FBY92] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[Fed87] J. Fedorowicz. Database performance evaluation in an indexed file environment. *ACM Transactions on Database Systems*, 12(1):85–110, 1987.

[FJ92a] Christos Faloutsos and H. V. Jagadish. Hybrid index organizations for text databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings 3rd International Conference on Extending Database Technology – EDBT '92*, Vienna, 1992. Springer–Verlag.

[FJ92b] Christos Faloutsos and H. V. Jagadish. On B-tree indices for skewed distributions. In *Proceedings of 18th International Conference on Very Large Databases*, pages 363–374, Vancouver, British Columbia, Canada, 1992.

[FRE92] freewais release 0.202 beta. Source code available from the Clearinghouse for Networked Information Discovery and Retrieval (CNIDR) via anonymous ftp from host ftp.cnidr.org., 1992.

[GBYS91] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. Lexicographical indices for text: Inverted files vs. PAT trees. Technical Report OED-91-01, University of Waterloo Centre for the New Oxford English Dictionary and Text Research, Canada, 1991.

[GGMT93] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The efficacy of gloss for the text database discovery problem. Technical Note STAN-CS-TN-93-2, Stanford University, 1993. Anonymous FTP: db.stanford.edu: /pub/gravano/1993/stan.cs.tn.93.002.ps.

[GGMT94] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The effectiveness of gloss for the text database discovery problem. In *Proceedings of 1994 ACM SIGMOD International Conference on Management* *of Data*, Minneapolis, MN, 1994. Anonymous FTP: db.stanford.edu: /pub/gravano/1993/stan.cs.tn.93.002.short1.ps.

[HC90] Donna Harman and Gerald Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41(8):581–589, 1990.

[Hol92] Lee A. Hollaar. Implementations and evaluation of a parallel text searcher for very large text databases. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 300–307. IEEE Computer society Press, 1992.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, New York, 1991.

[JO92] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes for a shared-everything multiprocessor. Technical Report GIT-CC-92/39, Georgia Institute of Technology, College of Computing, 1992.

[KMD+92] B. Kahle, H. Morris, F. Davis, K. Tiene, C. Hart, and R. Palmer. Wide area information servers: an executive information system for unstructured files. *Electronic Networking: Research, Applications and Policy*, 2(1):59–68, 1992.

[Knu73] Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, 1973.

[Lin91] Zheng Lin. Cat: An execution model for concurrent full text search. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 151–158, Miami Beach, Florida, 1991.

[Liv90] Miron Livny. DENET user's guide. Technical report, University of Wisconsin-Madison, 1990.

[MMN86] Patrick Martin, Ian A. Macleod, and Brent Nordin. A design of a distributed full text retrieval system. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, pages 131–137, Pisa, Italy, September 1986.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *International Conference on Management of Data (SIGMOD 88)*, pages 109–116, Chicago, Illinois, 1988.

[Sal89] Gerard Salton. *Automatic Text Processing.* Addison-Wesley, New York, 1989.

[Sch90] Bruce Raymond Schatz. Interactive retrieval in information spaces distributed across a wide-area network. Technical Report 90-35, University of Arizona, December 1990.

[SK86] Craig Stanfill and Brewster Kahle. Parallel free-text search on the connection machine system. *Communications of the ACM*, 29:1229–1239, 1986.

[SLS+93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The Rufus system: Information organization for semi-structured data. In *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993.

[Sta90] Craig Stanfill. Partitioned posting files: A parallel inverted file structure for information retrieval. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, 1990.

[STGM94] Kurt Shoens, Anthony Tomasic, and Hector Garcia-Molina. Synthetic workload performance analysis of incremental updates. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, Dublin, Ireland, 1994.

[Sto87] Harold S. Stone. Parallel querying of large databases: A case study. *IEEE Computer*, pages 11–21, October 1987.

[STW89] Craig Stanfill, Robert Thau, and David Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of the Twelfth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 88–97, Cambridge, Massachusetts, 1989.

[TC92] Howard R. Turtle and W. Bruce Croft. Uncertainty in information retrieval systems. In Amihai Motro and Philippe Smets, editors, *Proceedings of the Workshop on Uncertainty Management in Information Systems*, pages 111–137, Mallorca, Spain, September 1992.

[TGM93a] Anthony Tomasic and Hector Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proceedings of the Special Interest Group on Management of Data (SIGMOD)*, Washington, D.C., May 1993.

[TGM93b] Anthony Tomasic and Hector Garcia-Molina. Query processing and inverted indices in shared-nothing document information retrieval systems. *The VLDB Journal*, 2(3):243–271, July 1993.

[TGMS93] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. Technical Note STAN-CS-TN-93-1, Stanford University, 1993. Available via FTP db.stanford.edu:/pub/tomasic/stan.cs.tn.93.1.ps.

[TGMS94] Anthony Tomasic, Hector Garcia-Molina, and Kurt 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.

[Tri82] Kishor Shridharbhai Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[Wei90] Peter Weiss. *Size Reduction of Inverted Files Using Data Compression and Data Structure Reorganization.* PhD thesis, George Washington University, 1990.

[Wol91] Stephen Wolfram. *Mathematica.* Addison-Wesley, Redwood City, California, 2nd edition, 1991.

[Zip49] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.

[ZMSD92] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proceedings of 18th International Conference on Very Large Databases*, Vancouver, 1992.