Performance of Inverted Indices in Shared-Nothing Distributed Text Document Information Retrieval Systems

Published in PDIS '93

Anthony Tomasic

Department of Computer Science
Princeton University
Princeton, NJ 08540
tomasic@cs.stanford.edu

Hector Garcia-Molina

Department of Computer Science
Stanford University
Stanford, CA 94305
hector@cs.stanford.edu

Abstract

The performance of distributed text document retrieval systems is strongly influenced by the organization of the inverted index. This paper compares the performance impact on query processing of various physical organizations for inverted lists. We present a new probabilistic model of the database and queries. Simulation experiments determine which variables most strongly influence response time and throughput. This leads to a set of design trade-offs over a range of hardware configurations and new parallel query processing strategies.

1 Introduction

Full text databases of newspaper articles, journals, legal documents etc. are readily available. These databases are rapidly increasing in size as the cost of digital storage drops, as more source documents are available 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 and more important to have effective search capabilities, as provided by information retrieval systems.

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

To motivate the issues that will be addressed, let us start with a simple example. The left hand side of Figure 1 shows four sample documents, D0, D1, D2, D3, that could be stored in an information retrieval system.

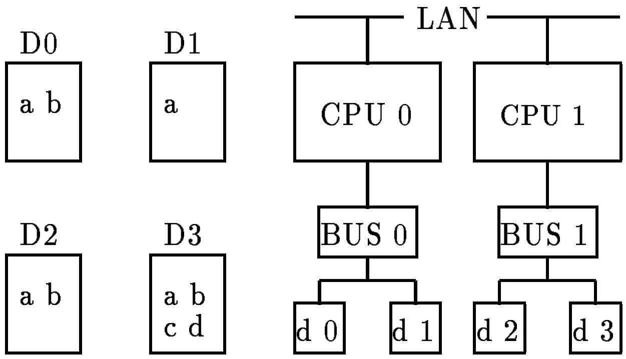

Figure 1: A example set of four documents and an example hardware configuration.

Each document contains a set of words (the text), and each of these words (maybe with a few exceptions) will be 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*. (Of course, in practice documents will be significantly larger and will contain many more words.)

To find documents quickly, full text retrieval systems traditionally build *inverted lists* [9] on disk. For example, the inverted list for word *b* would be *b*: (D0,1), (D2,1), (D3,1). Each pair in the list 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 only needs to retrieve 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 is used to answer queries involving distances, e.g., find documents where *a* and *b* occur within so many positions of each other.

Next suppose that we wish to store the inverted lists on a multiprocessor like the one shown in Figure 1 (on right). This system has two processors (CPUs), each with a disk controller and I/O bus. (Each CPU has its

Index	Disk	Inverted Lists
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. Inverted lists are *word: (Document, Offset)*.

own local memory.) Each bus has two disks on it. The CPUs are connected by a local area network. Table 1 shows three options for storing the lists.

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 earlier 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. Notice that to now answer the query "Find all documents with word b" we have to 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 disk attached to the CPU. For example, for CPU 1, the list for a is on $d2$, the list for b is on $d3$, and so on.

Query processing under each index organization is 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 then 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 sent its resulting document list to CPU 0, which would then merge the results. With the disk index organization, CPU 0 would retrieve the a and c lists off 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), merge them, and would then merge them 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: 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 in (this may involve the transfer of multiple blocks). In the disk index organization, however, 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 take place. However, each of the transfers is roughly a fourth of the size, and they may 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 be read into memory faster.

The system index organization may save disk resources, but it consumes more resources at the network level. In our example, the entire c list is transferred from CPU 1 to CPU 0, and we can expect these inverted lists to be 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. In our "Find documents with a and c" example, we can move the shorter list of a and c to the other host.

Thus, the performance of each strategy will depend 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 paper we will study these issues, discussing 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?

In Section 2 we describe our hardware scenario and query processing algorithms 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 a lot of work done on information retrieval systems, to our knowledge such models, appropriate for studying parallel query execution, have not been developed. In Section 3 we define simple models for these and other critical components. In Section 4 we describe the simulation, while in Section 5 we present our results and comparisons.

2 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 paper, we concentrate on "boolean and" queries of the form $a \wedge b \wedge c \ldots$. Such queries find the documents 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* of the document identifiers of all the documents which match the query. A *match* is a document which 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 property 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 additional CPU processing that compares the positions of the occurrences.

2.1 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 5). That is, our experiments start from a representative configuration, and from there, we explore the impact of changing the values. The base case does not represent any particular real system; it is simply a convenient starting place.

2.2 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 reside in a uniformly distributed manner across all disks d in the system ($d = Hosts \cdot I/OBusesPerHost \cdot DisksPerI/OBus$). Let the disks be numbered from 0 to $d-1$ as in Figure 1.

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

Table 2: Hardware configuration parameter values and definitions.

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 (if a given word does not appear in any documents on a disk, then that list is empty). 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 shown earlier 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. Note that regardless of the index organizations the same amount of data is stored in the system 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.

2.3 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 into memory all the lists involved, intersecting them, 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 to the query.

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 Section 1, the algorithm we have described for the system index organization can be improved. Here we describe three optimizations, called *Prefetch I, II* and *III*. Note that 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. This significantly reduces the size of all 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 out the subquery with the largest number of keywords. We expect that as the number of keywords in a subquery increases, its partial answer set will decrease in size. Thus, the amount of data returned by the one host that processes the phase 1 subquery should be small. If there is a tie (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 5) to understand what each optimization contributes. Keep in mind that Prefetch I and III require statistical information on inverted list sizes. Our results will tell us if it is worthwhile keeping 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.

2.4 Related Work

For full text retrieval systems, inverted lists are typically used. Compression of inverted lists is actively studied [19, ?]. However, much work has been done on other alternatives, such as signature schemes [7].

In [2], Burkowski 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 simulates a collection of servers on a local area network, as we do. Our work is complementary to this paper in that we concentrate on physical index organization. In [11], and independently from our work, the issue of partitioning by document vs. paritioning by keyword is studied for share-everything multiprocessors. The paper confirms the results presented here.

The work on document retrieval in multiprocessor systems (e.g. [1, 6, 12]) is also related to this paper in that physical index organization issues need to be addressed for those architectures. While some issues for these systems are not considered here, we believe that the issue of physical organization is an important one and that the prefetch algorithms presented in this paper probably perform well on multiprocessor architectures. Finally, in the debate on the relative advantages of parallel computers [15, 16, 17] and in other articles [10, 14, 18] various benchmark figures are given. However no systematic comparison has been done.

3 Models

There are two choices for representing documents and queries in a simulation study. 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 gives one more flexibility for studying a wide range of scenarios. Here we follow the synthetic data approach, as we feel it is more appropriate for a first study that explores options and tradeoffs, rather than predicts the performance of a particular document application.

3.1 Document Model

For the model of a document we first define several parameters in Table 3. The database consists of 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 uniquely represented by an integer w in the range $1 \leq w \leq T$ where

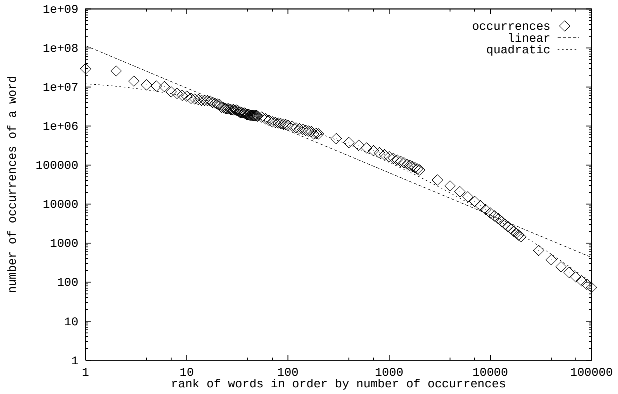

Figure 2: Curve fit to vocabulary occurrence data.

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

Table 3: Parameters of the document model.

$T = \mid V \mid$. 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 again represents our base case scenario. In this case, the values are from a legal document base described in [3].

To construct a specific probability distribution Z of $\mathcal{Z}$, a curve is fit to the rank/occurrence distribution of the vocabulary of the legal documents database [3] and then normalized to a probability distribution. Figure 2 shows the log/log graph of two curves which 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.

Given the quadratic fit curve, the form of the probability distribution Z is derived in [?] as

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

where the denominator is a normalization constant.

Parameter	Value	Description
K	5	number of words per 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.

(Although our distribution is similar to Zipf's [20], ours matches the actual distribution better.)

3.2 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 $Q(j)$. Thus, the occurrence of the words are mutually independent. Table 4 is 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 (however, see [5] for a different model). It does not follow the same distribution as the vocabulary (Figure 2), as relatively infrequent words are often used in queries. In light of this, the uniform distribution was chosen for Q, i.e. every word appears in a query with equal probability. The distribution allows easy comprehension 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. For this we introduce a parameter u to determine the range of the uniform distribution, giving Q the equation

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

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 will probabilistically have answer sets which are a large fraction of the database. On the other hand, if u is too large, answer sets will be unrealistically small. To estimate a good value for u, in [?] we compute the expected number of documents which match a query of length K for various values of u. In Section 5 the response time sensitivity to uT of the various index organizations is discussed.

3.3 Answer Set Model

At various points in the simulation we will 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 *Docs*. Note that under the system index organization, $Docs = D$ (D is the total number of documents). However, for the other organizations, *Docs* is the number of documents covered by the index (or indexes) used by the particular subquery. Given this, the expected number of documents which match the query is

$$Docs \cdot [1 - e^{-WZ(w_1)}] \cdots [1 - e^{-WZ(w_K)}]. \qquad (2)$$

(The term $[1 - e^{-WZ(w_1)}]$ is the probability that a document contains keyword w_1.)

3.4 Inverted List Model

To access an inverted list for a given vocabulary word, we assume that a memory resident data structure is searched. The size of the data structure will vary with each index organization, but in all cases is small compared to the size of the inverted lists.

An 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 (except for the white space and punctuation which is ignored when the document is added to the inverted index).

The expected number of occurrences of a word in a document is $\mathcal{Z}(w) \cdot W$. The expected number of entries of the corresponding inverted list is $Z(w) \cdot W \cdot Docs$ where $Z(w)$ is the value of Equation 1 for the word w, W is the number of words per document, and *Docs* is the number of documents spanned by the index.

4 Simulation

To study the index organizations and query algorithms, we implemented a detailed event-driven simulation using the DeNet [13] simulation environment. For details of the simulation, see [?].

The parameters controlling the hardware organization are listed in Table 5. The values for the disk and I/O bus portions of this table are from [4].

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

Parameter	Value	Description
$DiskBandwidth$	10.4	Mbits/sec Bandwidth
$DiskBuff$	32768	Size of a disk buffer
$BlockSize$	512	Bytes per disk block
$SeekTime$	6.0	ms of each seek
$BufferOverhead$	4.0	ms to seek one track
$I/OBusOverhead$	0.0	ms for I/O transfer
$I/OBusBandwidth$	24.0	Mbits/sec Bandwidth
$LANOverhead$	0.1	ms for LAN transfer
$LANBandwidth$	10.0	Mbits/sec Bandwidth

Table 5: Hardware parameter values and definitions.

expected length of the list. The length of an inverted list is a function of the expect number of occurrences of the word, the bits need for an entry, the compression factor, and the block size (see Table 6). This model assumes that the blocks of the inverted list are contiguous [8].

The length of a the answer set, in bytes, is determined by multiplying Equation 2 by the length of an element of an inverted lists, *AnswerEntry*.

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 various CPU requests. Each query consists of query start up, subquery start up, disk fetches, uncompression and merge of inverted lists, and the union of the subquery answer sets.

A disk services fetch requests from a CPU and sends the results to an I/O bus. Since one disk fetch corresponds to the read of one invert list, each fetch request has a length determined by $InvertedList(w)$. 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 I/O bus contention time. The overlap of the disk loading its track buffer and the transfer of data to the I/O bus is also simulated.

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). Subquery requests have variable length and consume a significant fraction of the local area network bandwidth when partial answer sets are transmitted. A request with identical source and destination 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.

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 accounting for query start up, subquery transmission, waiting for subqueries to finish, and merging the results of subqueries.

Parameter	Value	Description
CPUSpeed	1	Relative speed of CPU
Multiprogram	4	Number simultaneous queries *per Host*
QueryInstr	50000	Query start CPU cost
SubqueryInstr	10000	Subquery CPU cost
SubqueryLength	1024	Base size of subquery message
FetchInstr	5000	Disk fetch CPU cost
MergeInstr	10	Merge CPU cost/byte (decompressed list)
UnionInstr	1	Concatenation CPU cost/byte answer set
Decompress	10	Decompression CPU cost per byte of list
AnswerEntry	4	Bytes per entry in an answer set
EntrySize	10	Bits per entry of list on disk
Compress	0.5	Compression Ratio

Table 6: Base case 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 accounting for the subquery startup, the fetch requests to disks, waiting for the disks to finish, the intersection of the fetched inverted lists, and transmission of the answer set of the subquery back to the query. The answer is transmitted to the host cpu by inserting it in the *LAN* queue.

5 Simulation Results

Table 7 presents the data collected from a simulation run on the base case of values (Tables 2 - 6). The simulation runs are at a confidence level of 90% at 5%. 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 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, note that the host organization has the most balanced use of resources, and as we will see, this leads to better performance under more stressful scenarios.

The system index organization, as well as the prefetch optimizations, perform poorly in the base case scenario. 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 (see Section 2.3), and indeed we observe that the LAN utilization is much lower in these cases (see Table 7). Thus, the prefetch strategies perform substantially better than the system index organization.

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,

	Index						
	Disk	I/O	Host	Sys	I	II	III
(a)	2.17	1.75	2.12	8.42	4.99	5.78	5.83
(b)	.054	.043	.081	.311	.412	.462	.477
(c)	7.31	9.16	7.48	1.88	3.20	2.63	2.74
(d)	85.8	73.5	43.0	10.2	22.2	21.4	21.7
(e)	21.0	27.4	32.3	23.2	27.6	28.6	27.3
(f)	44.2	60.5	49.5	17.8	33.6	37.4	38.0
(g)	23.4	29.9	24.5	94.7	29.0	13.8	9.33

Table 7: Results of all metrics for the base case simulation experiment (I is Prefetch I, II is Prefetch II and III is Prefetch III). Labels are (a) query response time (sec), (b) response time error (sec), (c) query throughput (query/sec), (d) disk utilization (%), (e) I/O bus utilization (%), (f) CPU utilization (%), (g) LAN utilization (%).

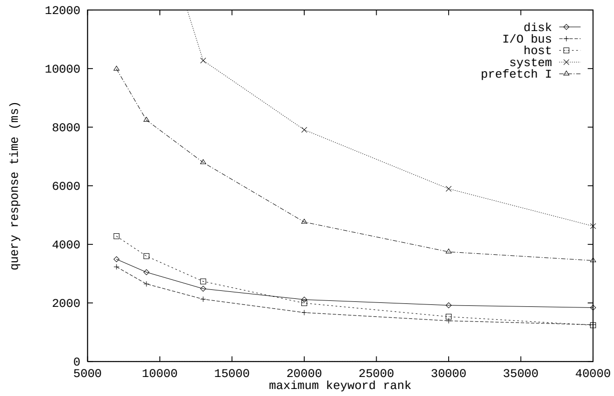

Figure 3: The sensitivity of response time to the maximum query keyword rank.

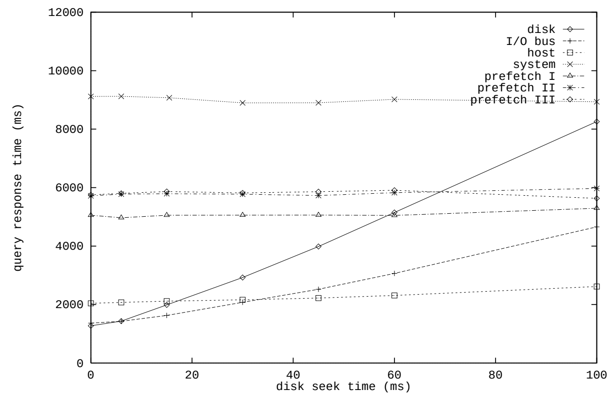

Figure 4: The sensitivity of response time to seek time.

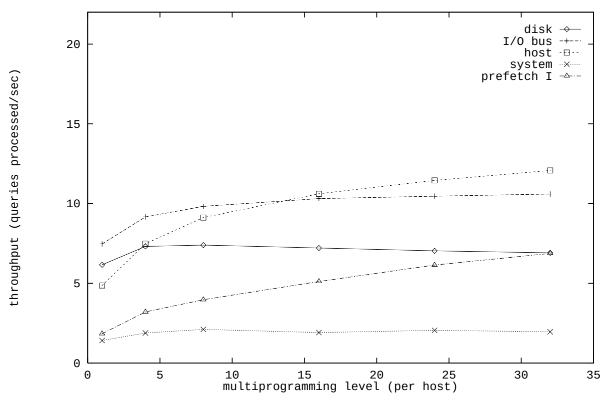

Figure 5: The effect of the load level.

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 *worse* than prefetch I (see Table 7). In Section 2.3 we argued that prefetch II and III would reduce the amount of data sent over the LAN. This is true as evidenced by the LAN utilization. However, with hindsight, we now see that 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. In the rest of our graphs this is not the case, so to avoid clutter we will only show the prefetch I results.

We now study how some of the key parameters affect the relative preformance of the index organizations. (We only report on the more interesting results; many more experiments were performed than what can be reported here.) We start by showing in Figure 3 the sensitivity of response time to the value of uT. Recall that T is the size of the vocabulary and u is the fraction of the vocabulary which 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 stripe 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 inverted lists are read whole. 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 a disk in Figure 4 shows that the disk and I/O bus index organizations are most sensitive to the seek time of the storage device. This is because 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 very insensitive to seek time since only a few inverted lists must be retrieved per query.

Figure 4 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 not only the seek time but also 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 practice, and hence multiple seeks. Thus, the higher seeks times shown in Figure 4 may occur in practice even without optical devices. In these cases, the disk and I/O organizations may not be advisable.

Figure 5 shows the effect of the load level on throughput for the various index organizations. As the load 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 which 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% respectively) and thus has more spare resources to consume as the multiprogramming level rises. At a multiprogramming level of 32 (128 total simultaneous queries since there are 4 hosts) the disk utilization has risen to over 74.3% and CPU utilization to over 78.2% for this index organi-

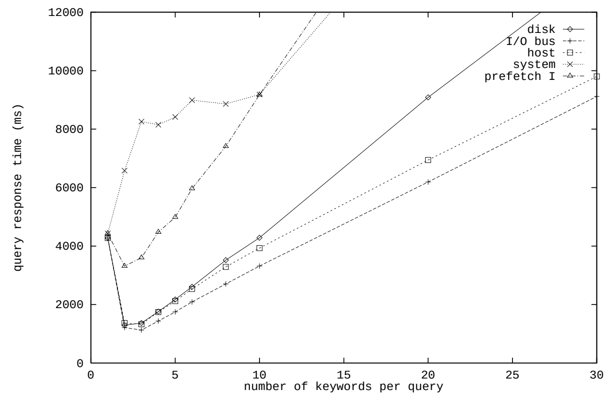

Figure 6: The sensitivity of response time to the number of keywords in a query.

zation. (Note that sufficient memory must be available to prevent excessive page faulting.)

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 load 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 sec., 12.0 sec., 10.6 sec., 62.6 sec., and 18.2 sec. respectively

The effect of large partial answer sets is shown clearly in Figure 6 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 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 needed dominates the response time.

In the system index organization, the size of the partial answer sets transmitted depends on which hosts 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 comparatively 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.

Note that after 15 keywords per query, prefetch I performs worse than the simple system organization. This is because in the system organization the probability of a single word answer set being transmitted is very small anyway. 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 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 generally not performed well. 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. We study the rise in query throughput as the seek time increases in Figure 7. Again, the disk organization is sensitive to the increase in seek time for the same reasons as Figure 4. 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 which 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 in Figure 7, the disk, I/O bus, host, system, and prefetch I response times are 27.1 sec, 15.0 sec, 15.0 sec, 10.8 sec, and 10.2 sec, respectively.

6 Conclusion

In this paper we have described various options for physical design of a text document retrieval system. We have studied the performance of several parallel query processing strategies, and the impact of the underlying technology. In particular, the choice of an index organization depends heavily on the access time of the storage device and the bandwidth of interprocessor communication. We also discovered some unexpected results, e.g., as the size of a query increases, its response time may drop; the fancier prefetch optimizations were usually counterproductive.

In general, our results indicate that the host index organization is a good choice. It uses system resources

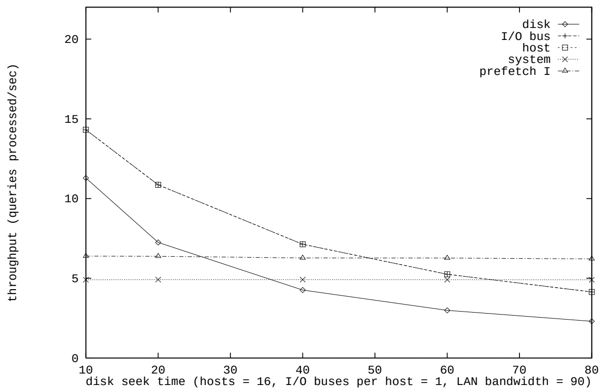

Figure 7: A good hardware configuration for the prefetch algorithm.

effectively and can lead to high query throughputs in many cases. When it does not perform the best, it is not very far off from 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. We should, however, point out four factors that may be unfair to this approach: (1) 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. This would make the system organization more attractive since it reduces the I/O load. (2) We are not modeling pipelining of I/O and CPU processing within a query. This can reduce query response time, and would be more beneficial to the system organization since it deals with longer inverted lists. (3) Another reduction in response time is *early termination* of the intersection algorithm. That is, if the inverted lists are in sorted order, the intersection algorithm can (in some cases) terminate having read only a fraction of the inverted lists. (4) We are using a closed simulation model where larger response times penalize throughput.

Acknowledgements: to Sam DeFazio, Ben Kao, Miron Livny, Sergio Plotkin, Mendel Rosenblum, and the referees for help on aspects of this paper.

References

[1] I. J. Aalbersberg and F. 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.

[2] F. 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.

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

[4] A. L. Chervenak. Performance measurements of the first raid prototype. Technical Report UCB/UCD 90/574, University of California, Berkley, May 1990.

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

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

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

[8] C. 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.

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

[10] D. Harman and G. 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.

[11] B.-S. Jeong and E. Omiecinski. Inverted file partitioning schemes for a shared-everything multiprocessor. Technical Report GIT-CC-92/39, Georgia Institute of Technology, College of Computing, 1992.

[12] Z. 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.

[13] M. Livny. DeNet user's guide. Technical report, University of Wisconsin-Madison, 1990.

[14] F. Rabitti and J. Zizka. Evaluation of access methods to text documents in office systems. In *Research and Development in Information Retrieval*, pages 21–40, King's college, Cambridge, 1984.

[15] G. Salton and C. Buckley. Parallel text search methods. *Communications of the ACM*, 21(2):202–214, February 1988.

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

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

[18] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. Technical Report STAN-CS-92-1434, Stanford University Department of Computer Science, 1992.

[19] E. M. Voorhees. The efficiency of inverted index and cluster searches. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval*, pages 164–174, Pisa, Italy, 1986.

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

[21] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley Press, 1949.

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