Query Processing and Inverted Indices in Shared-Nothing Text Document Information Retrieval Systems

Appears in The VLDB Journal (2) 3, pp. 243-271.

Anthony Tomasic and Hector Garcia-Molina

January 17, 1994

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 those variables that most strongly influence response time and throughput. This leads to a set of design trade-offs over a wide range of hardware configurations and new parallel query processing strategies.

Key Words. Performance, file organization, query processing, inverted file, inverted index, striping, shared-nothing, full text information retrieval.

1 Introduction

Full text document 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. 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.)

Anthony Tomasic, M.A., is Ph.D. Candidate, Department of Computer Science, Princeton University, Princeton, NJ 08540. Hector Garcia-Molina, Ph.D., is Professor, Department of Computer Science, Stanford University, Stanford, CA 94305. (Reprint requests to Anthony Tomasic, Dept. of Computer Science, Margaret Jacks Hall, Stanford University, Stanford, CA 94305, USA.) e-mail: `tomasic@cs.stanford.edu`, `hector@cs.stanford.edu`. This research was partially supported by the Defense Advanced Research Projects Agency of the Department of Defense under Contract No. DABT63-91-C-0025.

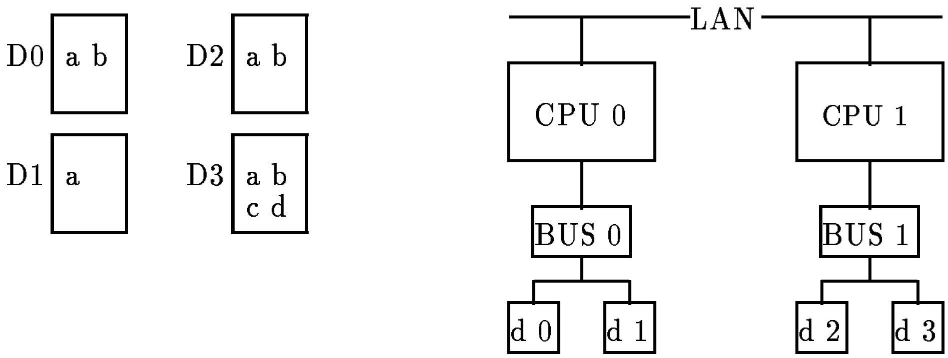

Figure 1: A example set of four documents and an example hardware configuration.

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. The host and I/O bus organizations are identical in this example because each CPU has only one I/O bus.

To find documents quickly, full text document 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 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.

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 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.

1.1 System index organization

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 *d*1.

1.2 Disk index organization

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.)

1.3 Host index organization

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.

1.4 I/O bus index organization

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 build and then uniformly dispered among the disks attached to the I/O bus. In our example, this results in the same organization 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.

1.5 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 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), would 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. 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 in (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 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. Notice that 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. For example, in our "Find documents with a and c", we can move the shorter list of a and c to the other computer.

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, query processing algorithms, physical index organization, and related work 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 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 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.

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.

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.

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. In particular, 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 Striping

Striping [17] is a method to decrease the response time and increase the throughput to read an inverted list by (a) allocating the blocks of an inverted list *horizontally* across several disks (by using modular arithmetic) and (b) reading the blocks in parallel. For example, suppose we have four blocks $b0$, $b1$, $b2$, $b3$ which store an inverted list for a word z which is 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 results in block $b0$ residing on disk $d1$ (the first block does not change its location), block $b1$ on disk $d2$ (since the blocks are allocated horizontally), block $b2$ on disk $d0$ (by using modular arithmetic) and block $b3$ resides on disk $d1$. Thus, disk $d0$ and $d2$ have one block of the inverted list for word z and disk $d1$ has two blocks.

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 disk $d0$, the entry (D1,0) would be on disks $d1$. Similarly for CPU 1, the (D2,0) entry would be on disks $d2$ and the (D3,0) entry would be on disk $d3$.

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 phyical allocation for any inverted list.

In the system index organization, the natural approach would be to stripe across all the disks in the system. However, this greatly complicates query processing, requiring for instance that the blocks of an inverted list 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, disk $d0$ would have the list (D0,0)(D2,0) and disk $d1$ would have the list (D1,0)(D3,0). This method avoids the complication of striping across the system, but still provides the advantage that the inverted list for a word is located in only one host in the system.

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. Then 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 (ignoring any queuing delays or contention) to $sk + l$ worst case when the reads are processed sequentially. Thus, under best case conditions, striping should improve response time when $s + l/k < s + l$. Note that the additional work required for a striped read is $s(k-1)$ and this quantity 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 5 reports the effect of striping the longer inverted lists for all the index organizations. This is studied by varying the fraction of the vocabulary that have striped inverted lists.

The attentive reader may wonder about the exact difference between the disk index organization for an inverted list and the striped host index organization for the same list. 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 very similar to controlling the number of inverted lists that have a disk index organization. We expect that 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.

2.5 Related Work

For an introduction to full text document retrieval, the reader is referred to [19]. In the design of full text document retrieval systems, there is a basic trade-off between the time taken to process the document database and the time taken to process queries [7]. In this paper we assume that queries can be answered without examining the text any documents. (The opposite approach, the direct scanning of documents (usually in combination with some indexing) is also possible [12].) For full text retrieval systems, inverted lists are typically used. Compression of inverted lists is actively studied [25] [28]. The probabilistic construction of inverted lists by assuming the independence of word occurrences also appears in [7] and the work presents an interesting variation on inverted lists. In addition, much work has been done on other data structures, such as signature schemes [8].

In [2], Burkowski examines the performance problem of the interaction between query process-

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.

ing 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. This article extends previous work [22] in describing the simulation fully, describing the mathematical basis of the work, and modeling striping. We include some performance comparisons for striping in Section 5. In [13], and independently from our work, the issue of partitioning by document vs. partitioning 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] [10] [14] [16]) 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. Inverted files are also used in some parallel computers [20] and this paper also assigns keywords to processors. Finally, in some articles on information retrieval [11] [18] [24] various benchmark figures are given.

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 predict 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 $T = \mid V \mid$. 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

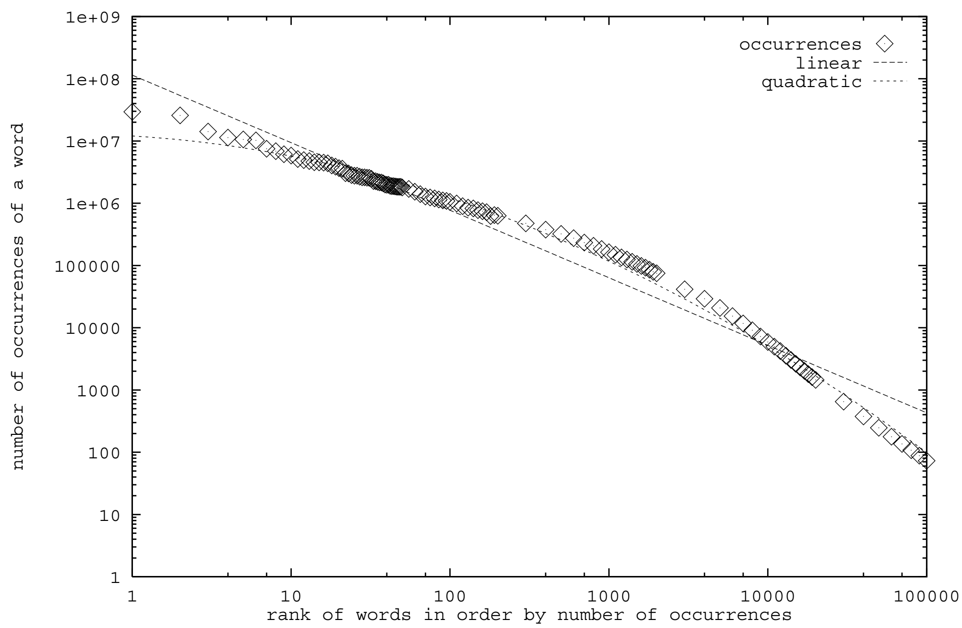

Figure 2: Curve fit to vocabulary occurrence data.

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 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. (We used [26] for curve fitting.)

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 [27], ours matches the actual distribution better. See Appendix A.)

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. 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 (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

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.

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 that 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 Appendix B we compute the expected number of documents that match a query of length K for various values of u. Note that the Q distribution has two other advantages. Since the distribution is simple, the impact of the distribution and consequently the impact of the work load on the system can be readily understood. (In Section 5, for example, we vary u and show the impact on performance.) Secondly, this distribution may favor very long inverted lists since very common words (such as "for") are part of the distribution. Thus, we consider this simulation to be a worst-case scenerio.

Using the parameter values in Table 3 and Equation 1, we graph the function Z for the various values of K and u in Figure 3. 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 300 documents on the average are found per query. Note that in this case the fraction uT of the vocabulary includes 96.3% of the cumulative keyword occurrences in [3] thus covering all but 3.7% of the words in the database. 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 *Documents*. Note that under the system index organization, *Documents* $= D$ (D is the total number of documents). However, for the other organizations, *Documents* is the number of documents covered by the index (or indexes) used by the particular subquery. Given this, the expected number of documents that match the query is

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

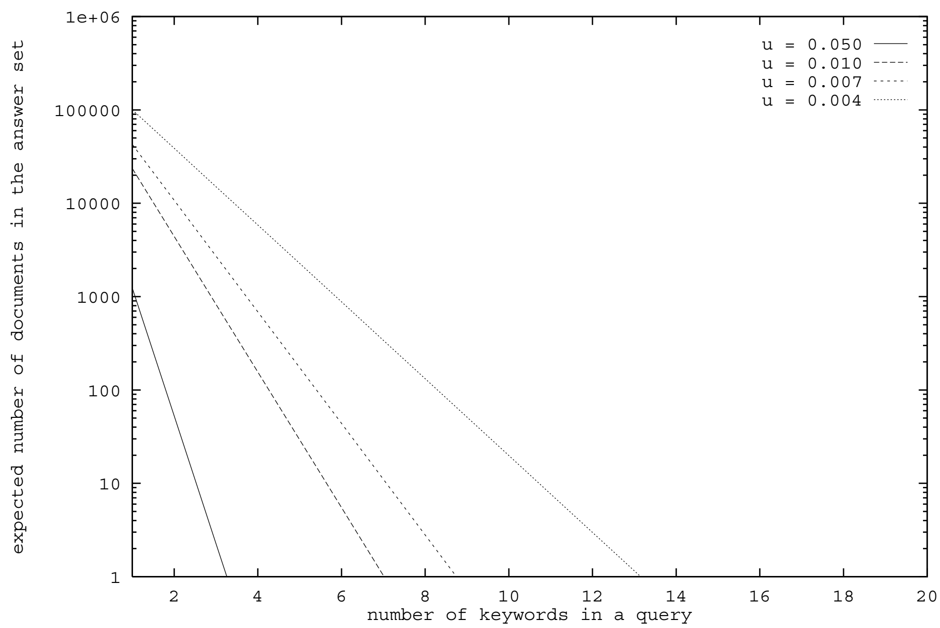

Figure 3: The Expected Number of Documents in an Answer Set for any Query.

(The term $[1 - e^{-WZ(w_1)}]$ is the probability that a document contains keyword w_1.) Equation 2 is similar to Equation 5 in Appendix B, except that here we are looking at a specific query, as opposed to averaging over all possible queries.

3.4 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 (except for the white space, punctuation, and a small number of common words that are ignored when the document is added to the inverted index).

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

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

where $Z(w)$ is the value of Equation 1 for the word w, W is the number of words per document, and *Documents* is, as before, 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 [15] simulation environment. In this section we describe important aspects of the simulation. Tables 5 and 6 describe the base parameters used.

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.

4.1 Hardware

The system model consists of several hosts 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 [4]. The hosts have parameter values that correspond to a typical workstation. See Figure 1 for an example hardware configuration.

4.2 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 modeled by the equation

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

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 approximently linear reduction is to represent the inverted lists in sorted order and then store (packed) the *difference* between an two consecutive entries (known as the *delta encoding*). More sophisticated compression schemes [28] result in better, nonlinear, compression ratios. The $BlockSize$ parameter permits studying the effect of internal fragmentation.

To determine if the inverted list for a word is striped, the predicate

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

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

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.

of the striped inverted 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).

4.3 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 (determined by parameter $QueryInstr$),
2. subquery start up (determined by parameter $SubqueryInstr$),
3. disk fetch (determined by parameter $FetchInstr$),
4. uncompression and merge of inverted lists determined by the equation

$$MergeInstr \cdot \sum_w InvertedList(w)$$

5. the union of subquery answer sets, determined by the equation

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

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

4.4 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 which 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 implicitly 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 is modeled by

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

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 ms) required to transmit y at bandwidth x. Let

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

then the BUS REQUEST messages is sent after $\max(0.0, a - b)$ units of time. This simulates the overlap of the disk loading its track buffer and the transfer of data to the I/O bus. The disk then waits until a BUS GRANTED message is received. 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 the I/O bus. In the case of a striped inverted list, the blocks of the inverted list which 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 the I/O buses are independent of each other.

4.5 LAN simulation

The system contains a single LAN that is simulated by a single *FCFS* infinite length queue which services subquery requests and answers that are transmitted between hosts. 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 3.3. The service time for a request is determined by the equation

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

where $LanBandwidth$ is a parameter. Note that subquery start up requests contend with answer set transmission, whereas disk fetch requests do not contend with fetch answers in I/O bus. This is because 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 local area network bandwidth when partial answer sets are transmitted. A request

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.

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 transmission.

4.6 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 representing query parsing and start-up,
2. subquery transmission to some or all hosts in the system,
3. block and wait for the subqueries to finish,
4. a CPU burst to merge the results of the subqueries.

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 representing subquery parsing and start-up,
2. a fetch request for an inverted list to one or more disks for each word appearing in the subquery,
3. a block and wait for the fetches to finish,
4. a CPU burst representing the computation of the intersection of the fetched inverted lists, and
5. the 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.

Metric	Index Organization 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
query 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).

4.7 Simulation

As mentioned earlier, the simulation is written in DENET [15]. 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 early. The values of these variables are shown in Table 7. These features are provided by the simulation programming language.

5 Simulation Results

Table 8 presents the data collected from a simulation run on the base case of values (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 have chosen not to add error bars to 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, 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 8). Thus, the prefetch strategies perform substantially better than the simple system index organization. (Note that the saturation of the LAN depends heavily on the ratio of the bandwidth of the LAN to the average length of an inverted list. In other work [21] we describe scenerios 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

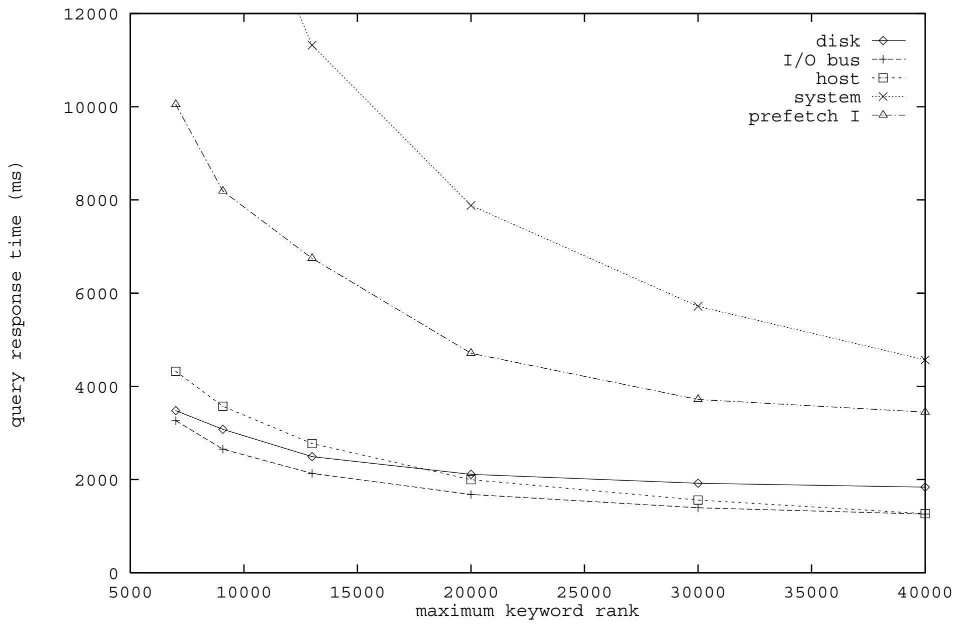

Figure 4: The sensitivity of response time to the maximum query keyword rank.

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). 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. 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 4 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 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 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 5 shows that the disk and I/O bus index organizations are most sensitive to the seek time of the

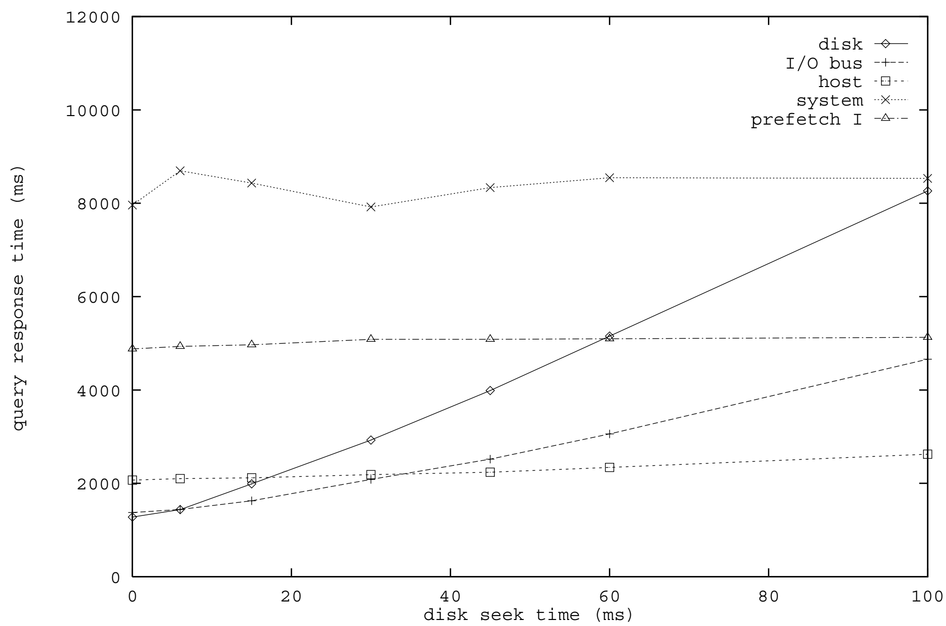

Figure 5: The sensitivity of response time to seek time.

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 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 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 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 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 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% 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 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 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

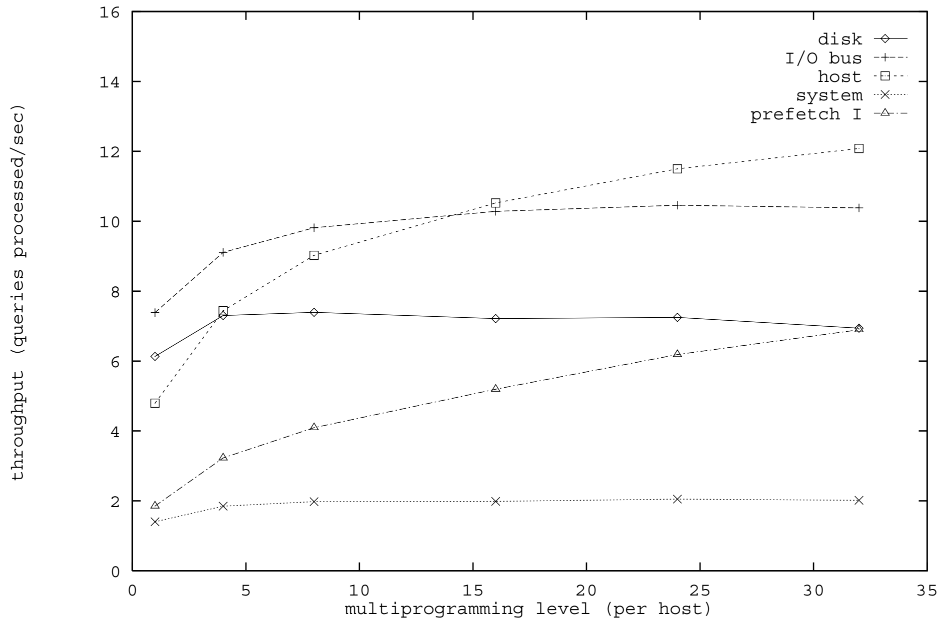

Figure 6: The effect of the load level.

18.2 sec. respectively

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 $Stripe \cdot u \cdot T$.) On the left-hand side of the graph, we see that 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 70% increase in throughput. Notice that 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 confirms the explanation of the effect of striping in Section 2.4.

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 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

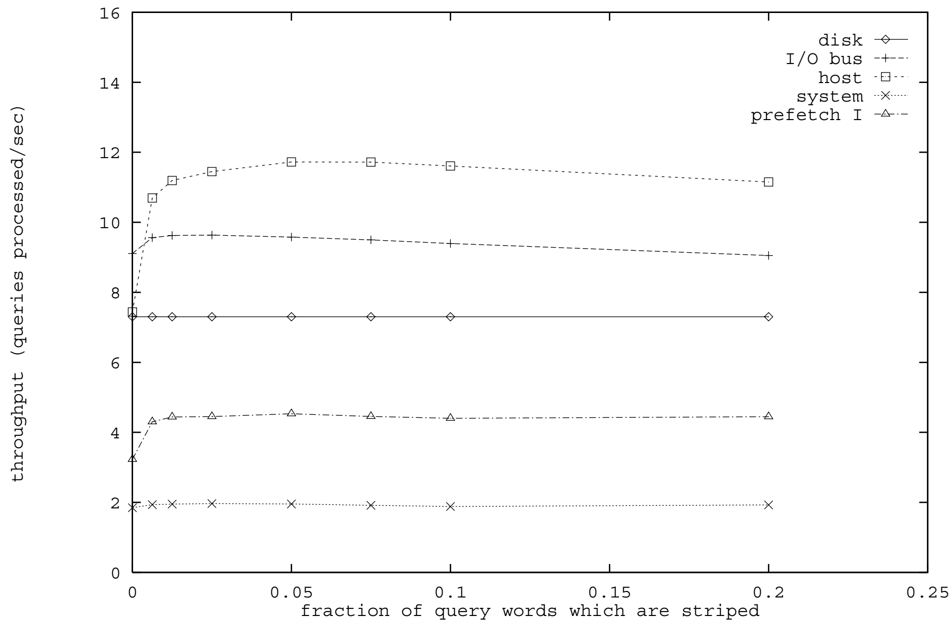

Figure 7: The effect of striping.

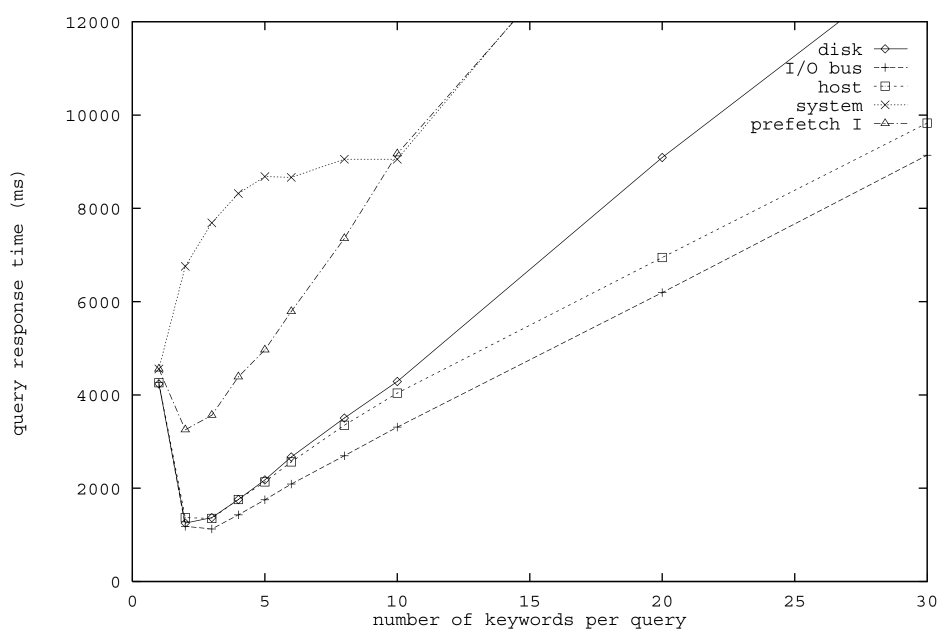

Figure 8: The sensitivity of response time to the number of keywords in a query.

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 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 that 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 9. 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 in Figure 9, 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, especially if very 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 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 prefetching, I/O and CPU processing within a query. This can reduce query response time, allow users to abort partially finished queries, and

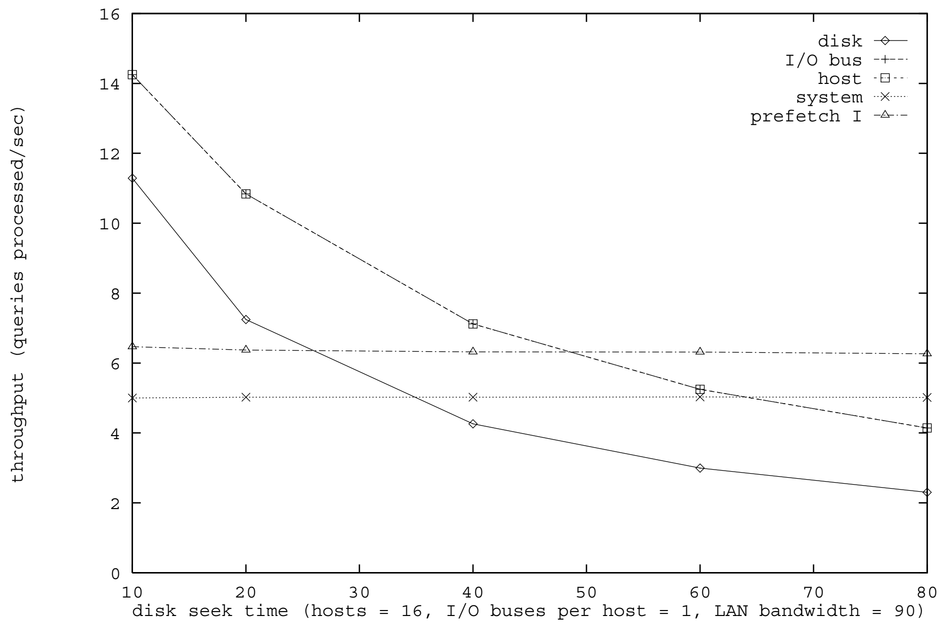

Figure 9: A good hardware configuration for the prefetch algorithm.

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.

In the future we plan to study the prefetch strategies more carefully, eliminating these potential biases. We also plan to build an actual experimental system, with a large collection of documents, in order to validate our models and results. We believe that the results in this paper will be very useful in guiding the construction of this system.

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

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

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

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

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

[10] Ophir Frieder and Hava Tova Siegelmann. On the allocation of documents in multiprocessor information retrieval systems. In *Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 220–229, Chicago, Illinois, 1991.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] Gabriel Matsliach and Oded Shmueli. An efficient method for distributing search structures. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 159–166, Miami Beach, Florida, 1991.

[17] 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, 1988.

[18] 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.

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

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

[21] 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.

[22] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the Second International Conference On Parallel and Distributed Information Systems*, San Diego, 1993.

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

[24] Ellen 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.

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

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

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

[28] 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.

A Derivation of the Probability Distribution Z

Given the curve fit equations, we wish to derive the form of the probability distribution Z. This is 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 form of the curve fit equation is

$$y' = mx' + b$$

and by replacement and raising exponents becomes

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

which then reduces by algebra to

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

Note that typically a Zipf Harmonic function [27] 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 [23] as follows. (Here, we model the distribution of all the words in the document. This 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. Then we rewrite x as

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

and from the above derivation we can write

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

for the equation of the distribution. By substitution and some algebra,

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

we derive the linear form

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

This 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 giving the derivation

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

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

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

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

to produce 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

$$Z(j) = \frac{j^{-0.0752528 \ln j - 0.150669} e^{16.3027}}{8.47291 \times 10^8} \tag{4}$$

where the denominator is a normalization constant. Thus, $\sum_{j=1}^{T} Z(j) = 1$ as required by probability distributions.

B Derivation of the effect of u on the expected size of a query answer set

A document matches a query when every word which 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, then

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

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:

$$D \cdot \sum_{Y=(v_1,\ldots,v_K)\in\mathcal{S}} \Pr(Y)[1-(1-Z(v_1))^W]\cdots[1-(1-Z(v_K))^W] \tag{5}$$

We can reduce this 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

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

is transformed to

$$\frac{D}{(uT)^K} \sum_{v_1\in V'} \cdots \sum_{v_K\in V'} [1-(1-Z(v_1))^W]\cdots[1-(1-Z(v_K))^W] \tag{6}$$

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

$$\frac{D}{(uT)^K} \sum_{v_1\in V'} [1-(1-Z(v_1))^W]\cdots \sum_{v_K\in V'} [1-(1-Z(v_K))^W] \tag{7}$$

and finally,

$$\frac{D}{(uT)^K} \left\{ \sum_{v=1}^{uT} [1-(1-Z(v))^W] \right\}^K . \tag{8}$$

Note that the in the above equation, 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 $p \leq 0.05$ [23]. 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 above equation can be rewritten as

$$\frac{D}{(uT)^K} \left\{ \sum_{v=1}^{uT} 1-e^{-WZ(j)} \right\}^K . \tag{9}$$

We use Mathematica [26] 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.