Caching and Database Scaling in Distributed Shared-Nothing Information Retrieval Systems

Stanford University Computer Science Technical Report STAN-CS-92-1456

Anthony Tomasic and Hector Garcia-Molina
Stanford University Department of Computer Science

December 22, 1992

Abstract

A common class of existing information retrieval system provides access to abstracts. For example Stanford University, through its FOLIO system, provides access to the INSPEC database of abstracts of the literature on physics, computer science, electrical engineering, etc. In this paper this database is studied by using a trace-driven simulation. We focus on physical index design, inverted index caching, and database scaling in a distributed shared-nothing system. All three issues are shown to have a strong effect on response time and throughput. Database scaling is explored in two ways. One way assumes an "optimal" configuration for a single host and then linearly scales the database by duplicating the host architecture as needed. The second way determines the optimal number of hosts given a fixed database size.

1 Introduction

Information retrieval systems, of the type found in libraries, provide indexed access to the abstracts of documents. Information vendors such as Dialog and BRS Search also provide access to such abstracts databases. The number of such databases is rapidly growing, as more and more information is stored digitally. At the same time, an increasing number of users have access to these databases through the networks. To handle the increased load, a distributed architecture can be used, dispersing the data and index structures across several computers and performing searches in parallel. This paper studies the performance trade-offs in such a shared-nothing distributed system. Our work complements an earlier paper [12] where a full-text information retrieval system was studied (entire document is indexed, as opposed to just its abstract). The study reported here also represents the first time (to our knowledge) that an actual user query trace drives the performance evaluation of a distributed architecture.

An abstracts database typically uses an *inverted index* to speed up query processing (see [6] for a survey of access methods for text). For each word, an *inverted list* is constructed that gives all the abstracts in which the word appears. In a multiprocessor environment, the inverted lists can be distributed in various ways:

1. Disk Organization. The abstracts are logically partitioned by physical disk, that is, each disk is assigned (a hopefully equal) number of abstracts. An inverted index is then constructed for the abstracts of each disk.

2. I/O Bus Organization. Each I/O bus controls a subset of disks. An inverted index is constructed for all the abstracts of the disks on each I/O bus. Each inverted list is stored in one of the disks in the I/O bus group.

author: A. System
title: Theory of System
id: 0
abstract: A practical system, good for one year or one million dollars, whichever comes first.

author: B. Theory
title: Theory of Theory
id: 1
abstract: A theory, might be useful, might not.

author: C. Hardware
title: Hard Ware
id: 2
abstract: A very hard piece of ware.

author: D. Student
title: Thesis
id: 3
abstract: A direct extension of my advisor's will.

Figure 1: A example set of four documents.

3. Host Organization. An inverted index is constructed for the abstracts assigned to the disks of each host. The inverted lists are allotted evenly across the disks of the host.

4. System Organization. In the previous organizations, for each word, there are multiple inverted lists, one at each disk, I/O bus, or host. In the system organization, a single inverted list is generated for each word. Each inverted lists is allotted to one of the disks of the system.

To illustrate these organizations, consider the four documents in Figure 1. Each document contains four fields: author, title, abstract, and id. An example hardware organization is shown Figure 2; it has two hosts, labeled CPU 0 and CPU 1, each with a cache, one I/O bus and two disks. Table 1 shows the various inverted index organizations for the figures. (The Host and I/O Bus organization are equivalent in this example since each host has a single I/O bus.) Note that in this table, each entry in an inverted list is typed with the field name where the word appears ("A" for author, "T" for title, "B" for abstract).

For instance, the word "a" appears five times in the example set of documents. The word appears in each abstract and it appears in the author field for document 0. For the system organization, all the appearances of a word are in the same inverted list. In the table this inverted list in located on disk d 0. For a given keyword and field designation, all the corresponding entries in the inverted list are the *postings* for that keyword and field designation. Thus, this inverted index organization combines all the postings of a word for all the field designations into a single inverted list. In the other organizations, the "a" list is split. For example, in the Host organization, there is one "a" list covering abstracts in disks D0 and D1 (stored in d0 in this example), and another "a" list for the d2, d3 abstracts (stored in d2).

To answer a query originating on a host (the *home* host) in the disk, I/O bus, and host index organizations, a copy of the query is made for each host. This subquery is sent to each host which then *matches* the subquery against its inverted lists. Since queries consist of keyword-field pairs connected by boolean ANDs, matching is accomplished by constructing the intersection of the inverted lists. The result of the intersection, the *answer* to the subquery, is then transmitted to the home host. The home host concatenates all the subquery answers to produce the final answer.

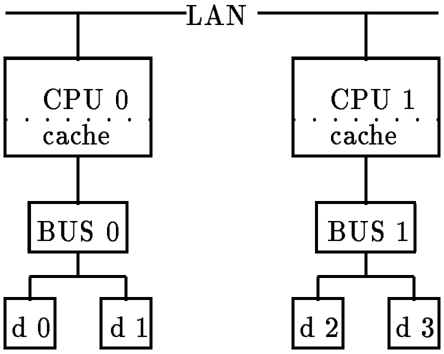

Figure 2: An example hardware configuration.

Index	Disk	Inverted Lists in *word: (Id, Field)* form
Disk	d 0	a: (0, A), (0, B); theory: (0, T); system: (0, A), (0, T), (0, B)
	d 1	a: (1, B); theory: (1, A), (1, T), (1, B)
	d 2	a: (2, B); hard: (3, T), (3, B)
	d 3	a: (3, B);
Host, I/O bus	d 0	a: (0, A), (0, B), (1, B); theory: (0, T), (1, A), (1, T), (1, B)
	d 1	system: (0, A), (0, T), (0, B)
	d 2	a: (2, B), (3, B)
	d 3	hard: (3, T), (3, B)
System	d 0	a: (0, A), (0, B), (1, B), (2, B), (3, B)
	d 1	system: (0, A), (0, T), (0, B)
	d 2	hard: (3, T), (3, B)
	d 3	theory: (0, T), (1, A), (1, T), (1, B)

Table 1: The various inverted index organizations for the words “a”, “hard”, “system”, and “theory” in Figures 1 and 2. The field “A” is for author, “T’ is for title, and “B” is for abstract. (Other lists are not shown.)

To answer a query in the system index organization, a subquery is sent to each host relevant to the query. (A host is relevant to a query if the inverted list for at least one keyword in the query resides on the hosts.) The subqueries are processed in the same manner as the other index organizations. When the answers to the subqueries are returned to the home host, another intersection is performed on the answers to produce the final answer.

To illustrate, consider the query *find abstract: system and title: theory* issued to the home host CPU 0. In the system index organization two subqueries are issued. One is the subquery *find abstract: system* and is sent to the host CPU 0, the other is the subquery *find title: theory* and is sent to host CPU 1. The subqueries are processed in parallel. CPU 0 generates the subquery answer (the list of matching abstracts) (0) and CPU 1 generates the answer (0, 1). Both answers are transmitted to the home host which constructs the intersection of the answers producing the answer (0) indicating that document 0 matched the query. In the host index organization two subqueries are also issued, each consisting of the query *find title: theory and abstract: system.* The subqueries are processed in parallel. Host CPU 0 transmits the answer (0) and host CPU 1 transmits the empty answer. The home host concatenates the two answers to produce final answer to the query.

Note that the abstracts themselves do not have to reside on the same disks as the inverted index. That is, for constructing the various index organizations, abstracts were *logically* assigned to disks or hosts, but they can be physically stored wherever we wish. Also note that the abstracts themselves are not accessed during query processing; to see an abstract the user must issue a separate display command. In this paper we assume that abstracts are stored on separate disks (a common technique in practice) and we only study query processing, i.e., we ignore display commands.

In this paper we address four basic types of questions:

1. What index organization yields best performance? In the system organization, typically a query only involves a subset of the hosts, leaving the irrelevant hosts free to process other queries. On the other hand, the other strategies allow more intra-query parallelism and may generate more uniform loads at the hosts. So which one leads to lower response times or higher throughputs? Also, there are various optimizations to the system organization, dealing with the order in which lists are fetched and intersected (see Section 3). How do these improve performance?

2. What are the critical hardware resources? What is the optimal arrangement for a given set of hardware resources? In particular, many current information retrieval systems run on large "mainframes." Can we really improve performance by having instead a collection of less expensive machines, implementing distributed indexes?

3. How well do the algorithms and hardware scale as the database size grows? As mentioned earlier, current data collections are growing rapidly, and it is not clear what index organization scales best, or whether it is more important to add disk or processor or communication resources as the database grows.

4. What is the impact of caching inverted lists in main memory? Is there enough locality of reference between queries to make caching worthwhile?

Our evaluation is based on query traces from the FOLIO library information retrieval system at Stanford University, run against a detailed event-driven simulation of the hardware and query processing. The trace data is described in Section 2, while our processing model is described in Section 3. The hardware model is presented in Section 4. Our results are given in Section 5 and conclusions in Section 6.

1.1 Previous Work

A substantial amount of work has been done in the general area of Information Retrieval. For a good explanation of the current state of the art in terms of improved precision and recall, see [13].

Not that much has been done on distributing inverted lists and searching them in parallel. Reference [11] discusses some of the basic issues. Burkowski simulates a shared-nothing information retrieval system [1] to study the performance impact of the placement of documents and inverted indexes. Jeong and Omiecinski [9] independently study for a shared-everything architecture similiar issues of the physical index design as in this paper. Work has also been done on searching with a variety of other architectures, e.g., processor farms [3], fine-grained parallelism [10], and shared-memory multiprocessors [4].

As mentioned earlier, our work is a continuation of an earlier paper [12]. The four index organization we have described are from that earlier paper. There are, however, two important differences between the earlier study and this one.

```
172 05/03/92 10:34:58 CMD: f au fan,g.y.
172 05/03/92 10:34:58 DIS: SEA:1, BRIEF:1
172 05/03/92 10:34:58 SEA: Res:5,  Find AUTHOR fan,g.y.
172 05/03/92 10:34:58 CMD: $DISPLAY 1
172 05/03/92 10:35:42 CMD: f au fan,g.
172 05/03/92 10:35:42 DIS: SEA:5, BRIEF:5
172 05/03/92 10:35:42 SEA: Res:26,  Find AUTHOR fan,g.
172 05/03/92 10:35:42 CMD: $DISPLAY 1
```

Figure 3: Some example lines from the raw trace.

- Here we study an abstracts database as opposed to a full-text system. In a full-text system, every single word occurrence is indexed. In an abstracts system, only the abstract is indexed. If we compare two systems with the same *number* of documents, the index in the full-text case will be much larger. Even if the volume of raw data is equal (example: abstracts are a tenth of the size of the full-text documents but there are 10 times as many abstracts), the inverted lists for the abstracts case will still be smaller. This is because repeated words are indexed in the full-text case only. For instance, if a word appears 10 times in a document, there will be 10 index entries (pointing to each occurrence) in the full-text case, and only one entry in the abstracts case. As we will see, the fact that inverted lists are shorter for abstracts dramatically changes the relative performance of the various organizations. Emrath [5] focuses on the performance trade-offs involved in partial or complete indexing.
- Here we drive our study with real traces from the Stanford library system. Furthermore, the traces also give the result sizes, so we can use that in our simulation. In [12] we modeled queries probabilistically, assuming query terms were picked at random from a vocabulary. Clearly, using traces (which incidentally are hard to get from commercial information vendors) yields more realistic results. It also lets us study caching issues.

2 Data

Stanford University provides on-campus access to its information retrieval system FOLIO from terminals in libraries and from workstations via telenet sessions. FOLIO gives access to several databases; one of these is INSPEC, an abstracts database for technical documents in disciplines such as physics, electrical engineering, and computer science. A trace of all user commands for the INSPEC database were collected from 5/3/92 to 5/9/92. In addition, the number of postings of every word in the INSPEC database inverted index was also collected.

Each INSPEC abstract is divided into fields such as *title* and *author.* One of these fields is called *abstract.* To avoid confusion between the field and the complete record name, we will henceforth refer to the complete abstract as the *document.* In a query, a user specifies the field where each word should appear. This is called the *field designation.*

An example of the raw trace is shown in Figure 3. The first line of the figure shows that user 172 issued a query for the author "G. Y. Fan."[1] The user types "f" as shorthand for "find." The "au" (author) string is the field designation of the string "G. Y. Fan." The second line reports how the results of the match will be displayed. The third line of the raw trace shows the result

[1]To insure the privacy of the data, the user login identification has been replaced by a unique integer.

Raw Trace Queries	2583
Simulation Experiment Queries	1894
Percentage Simulation Query of Raw Queries	73.3%
Total Keywords	3593
Mean Keywords per Query	1.90
Number of *Subject* Field Keywords	1611
Percent *Subject* Field of All Fields	44.8%
Mean Result Size per Query	833.8
Median Result Size	23
Unique Keywords	1551
Mean Cache Hit Percent	56.8%

Table 2: Statistical properties of the simulation trace input.

for the query in the first line. (The trace repeats the query here.) After examining the 5 results of the match, the user executed a query for the author "G. Fan" as shown in line 5 of the figure. The user then examined the 26 match results. Lines two, four, six and eight of the example raw trace describe the details of how the results are displayed.

Note that a certain amount of syntactic processing is done to detect command abbreviations and that the first name initials of an author follow the last name by a comma. In the FOLIO system all author names are indexed by surname only so the two queries of Figure 3 retrieve the *same* inverted list. After the list is fetched, FOLIO filters it based on the specification of the first and middle initials. If the inverted list in the first query was cached during the processing of the match, the second query in this example would score a cache hit.

To drive the simulation, only a subset of the raw trace is considered. Only the match results of queries consisting of boolean AND operations on simple terms is considered. We do not consider the other features of FOLIO such as wild-carding, thesaurus aided queries, phrase queries, etc. Queries that have terms with no associated inverted list (e.g. misspellings) are ignored since the query parser would catch these queries and reject them. Finally, the raw trace contained several queries with a reported result size of 322,737. When these queries were repeated by hand in an attempt to confirm this result, completely different result sizes were reported. We assume that this is an error in the trace log and have thrown out these queries. The raw trace contained 2583 query commands. The remaining queries used to drive the simulation constitute 73.3% of the original queries. This subset has the advantage that the impact of the traces on performance can be easily determined. We plan to include the remaining queries in a future study.

One important feature of FOLIO is the designation of the *subject* field in query matching. This field designation is a syntactic shorthand for matching the field designations *freeterm*, *abstract*, *organization*, *title*, and *conference* simultaneously. Thus, the query *find subject: theory* is a shorthand for the query *find freeterm: theory or abstract: theory or organization: theory or title: theory or conference: theory.* The subject field designation constitutes 44.8% of all field designations. Subject queries are handled by our simulation (see Section 3).

Some statistics of the traces will be helpful in interpreting the results of the simulation. Table 2 summarizes some properties of the query traces. To our surprise, the mean number of keywords per query is less than two. However, note that each use of the subject field designation is a shorthand for a query with multiple keywords. The mean size of the result of a query is large (over eight hundred) but the median result size is small at 23. We suspect that the queries with large results

```
Count: 7 Key: ABACO
Count: 39 Key: ABACUS
Count: 1 Key: ABACUSES
Count: 1 Key: ABAD
Count: 15 Key: ABADI
Count: 1 Key: ABADIA
```

Figure 4: The number of postings for a sample set of words which appear in the *abstract* portion of the documents.

Name	Description	Words	Postings	Postings/Words
abstract	Abstract of Document	456153	66090337	144.9
author	Author Surnames	282054	3399151	12.1
chemical	Chemicals	82397	102992	1.2
classcode	Classification Codes	2962	3728665	1258.8
conference	Conference Title	11148	6360111	570.5
freeterm	Extra Words	299732	24933897	83.2
number	ISBN (and other codes)	11825	2184484	184.7
organization	Author Organizations	55803	7292210	130.7
report	Technical Report Number	6849	7158	1.0
source	Publication Source	17518	6011824	343.2
title	Title of Document	161380	9149251	56.7
Total		1387821	129260080	93.1

Table 3: The field designators and the number of word occurrences for each.

are immediately refined to produce smaller results.

Issuing multiple queries to refine an answer set is common in information retrieval systems. This query refinement behavior by a user provides an opportunity for the caching of inverted lists. Of all the keywords appearing in the traces, 56.8% of them are duplicate appearances. Thus, if we cached every single read of an inverted list in the system we would achieve a mean cache hit ratio of 56.8% over the entire trace. (While this figure is not a high as those reported in the file system literature, Section 5 shows that caching does have a significant impact on mean throughput.)

For the database, total number of documents for the INSPEC database is reported at approximately 1,165,059 documents. The number of bytes per document is approximately 1,800. The total database size can be (very roughly) estimated at 2 Gigabytes.

For the matching of queries, the total of lengths of the inverted lists (i.e. total number of postings) read is important in determining the amount of work done in the matching process. As mentioned earlier, we scanned the actual INSPEC inverted lists recording the number of postings and their field designations. For example, Figure 4 shows a sample of the file containing the number of postings for words with the *abstract* field designation. Table 3 lists for each field designator the total number of words and the total number of postings.

To drive our simulation, we combine the information from the trace and postings files into a single trace file that is easy to use. Figure 5 shows a sample of this final trace file. For example, the first line of the trace shows that user 168 issued a query which had 181 results. The query referred to a single keyword "CARTER." The value 449 is a hash of the keyword "CARTER"

```
168 181 ( CARTER 449 1086 1222 )
168 110 ( CARTER 449 1086 1222 )
168 3039 ( MINNESOTA 686 3041 3758 )
168 0 ( CARTER 449 1086 1222 )
168 110 ( CARTER 449 1086 1222 )
168 12 ( OKEEFE 431 103 111 )
26 12 ( GAAS 284 60868 61215 ) ( MIS 233 3708 3751 ) ( ALGAAS 425 11862 11863 )
```

Figure 5: The trace input to the simulation.

and is used to determine the disk(s) which the inverted list(s) will reside for the various index organizations. The next number, 1086, is the number of posting for "CARTER" that have the field designation specified in the query. (We do not show what the field designation, say *author*, was. All that matters is that there are 1086 documents where "CARTER" is an author.) Note that the number of retrieved documents is less than 1086 because FOLIO filtered for the author initials. The final number, 1222, is the total number of posting for "CARTER," i.e., the total number of documents where "CARTER" appears, regardless of the field designation. For queries with multiple search terms (e.g., the one by user 26), each term is listed, together with the number of postings as described above.

The distribution of the lengths of the inverted lists which appear in the trace characterizes the work required to process the queries. Figure 6 shows the cumulative distribution of list lengths. The steep slope on the left-hand side of the figure shows that most of the inverted lists are less than 50,000 postings in length. The flat slope on the right-hand side of the figure shows that there are a few long inverted lists. These observations are confirmed by the following statistics: There are 3,593 inverted lists referenced in the trace. The mean length of an inverted list is 40,258.3 postings. The median inverted list has 10,353 postings. From the figure we see that the mean length is much larger because of a few very long inverted lists.

3 Query Processing

Information retrieval systems may partition their indexes by field designator or may build a combined index. In the partitioned case, all occurrences of a word in a *title* field are listed in one index, all *author* occurrences in another, and so on. In the combined case, a single index is built, and for each entry in an inverted list, a type annotation indicates the field where the word was used. The main advantage of the combined index is that a query such as *find subject: art* can be answered by fetching the single inverted list for "art," as opposed to fetching the lists for "art" in the *title*, *author*, etc. indexes. (Recall that a *subject* search is shorthand for a combined search, see Section 2.) On the other hand, a query *find title: art* can be processed faster with partitioned indexes, since only the relevant postings need to be processed. Since 44% of our queries involve *subject* searches, here we assume a single combined index. (In Section 6 we return to this issue.)

As discussed in the introduction, four physical index organizations are considered. We found in previous work [12] that the LAN may be the bottleneck for the system index organization. To ameliorate this problem we adopt one query processing optimization named "prefetch I" that operates as follows: we divide the processing for a query into two phases. In the first phase, the home site sends a subquery to the host holding the shortest inverted list for the query. This host broadcasts the shortest list on the LAN to all other hosts. In the second phase, the remaining

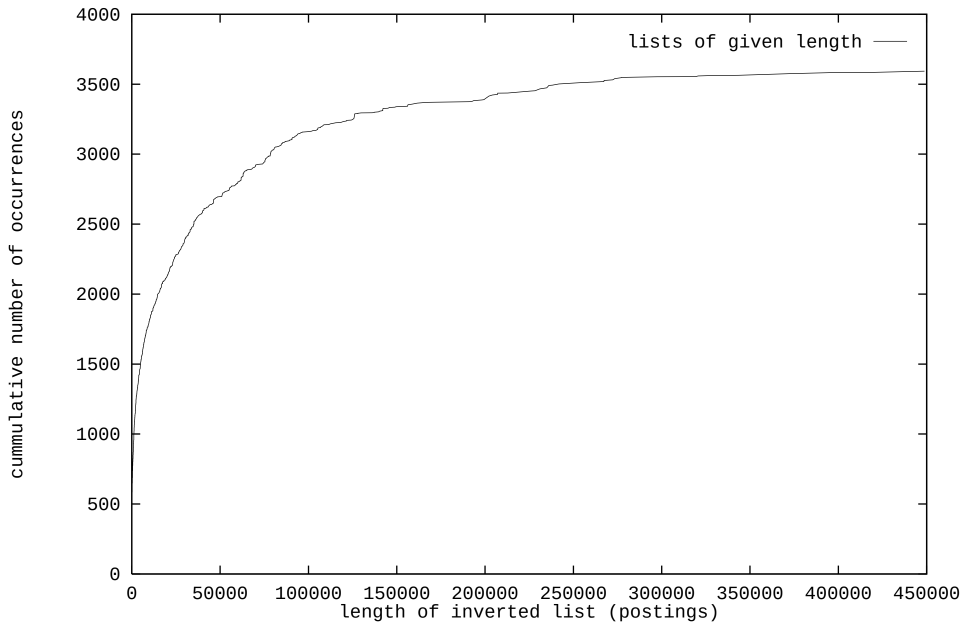

Figure 6: The cumulative distribution of occurrences of inverted indexes a given length which appear in queries.

inverted lists are retrieved (and intersected if more than one list resides at the same host), except that before results are sent to the home host, they are intersected with the first list broadcast. This significantly reduces the data volume on the LAN by reducing the mean subquery answer size. (In [12] two other prefetch variations are studied. For our current study, we evaluated all three variations; Prefetch I was the variation with the best performance (as in [12]), so to economize on space, we only describe the winning variation and its performance.)

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

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

From our trace data, we can determine how many inverted lists have to be fetched to answer a given query, and how large the lists are. However, our simulation also requires the sizes of the intermediate results, and we estimate them by calculating the expect number of answers as follows. In the case of the disk, I/O bus and host index organizations, we make the assumption that the answers are distributed in equal proportion across all hosts. Thus, to compute the size of the subquery answer we simply divide the result size reported in the trace by the number of hosts.

For the system organization, however, each subquery generally contains a subset of the keywords in the query. The following example illustrates how the expected answer size is calculated. Say the subquery is *find title: A author: B*. The full lists for A and B are fetched from disk; however, only the postings of the appropriate type (title for A, author for B) are used. The number of A postings with *title* designation is given in the trace, call it $n(A)$; the number of B *author* postings is $n(B)$. The expected size of the intersection of these lists is thus $n(A)n(B)/D$, where D is the total number of documents in the database. This assumes that each word is equally likely to appear in any of the D documents and that the words occur independently. (If an additional C word were in the query, the expected size would be $n(A)n(B)n(C)/D^2$.)

For the disk, I/O bus and host index organizations, we believe the model is very accurate. For the system index organizations, we believe that this model is reasonably accurate for a small number of query terms (it is exact for single keyword subqueries). (Emrath [5] reports some measurements which support this model.) However, as the number of keywords in the query increases, the expected number of answers approaches zero. To compensate for this effect, we use the result size (for the overall query) in the trace as a lower bound to the expected number of documents in a subquery. This is logically valid since the size of the final answer to the match is bounded from above by the minimum of the sizes of the subquery answers in the system organization.

Finally, we note one case in which we take license with the data in the previous section. Since the date and time of each query issued is reported in the trace, it is possible to simulate the exact sequence of query arrivals in the system in an "open system" type of queuing model. We have chosen a closed queuing system model that permits studying the effects of varying the multiprogramming level (the number of simultaneous queries in the system). This is accomplished by starting a number of queries at the beginning of the trace equal to the multiprogramming level and then starting the next query in the trace whenever a query finishes in the system. This method maintains a constant number of queries in the system.

The effect of this is to concurrently simulate queries that are *from the same user* and thus could not have been requested simultaneously. We believe that the negative impact of this loss in realism is minor and is outweighed by the advantages of studying the effect of the multiprogramming level on response time and throughput (cf. Section 5).

To study the effects of scaling the database, the parameter $DatabaseScale$ was added to the simulation. This variable linearly scales the number of postings for each inverted list, the number of answers to a query and the number of documents in the database. While scaling the number of postings and documents in the database is probably reasonable, a word of caution is in order with respect to the scaling the number of answers. If a query matches 10 documents, a user will simply read all 10 documents to determine which ones are of interest. Faced with a query with a result size of 1000, a user would probably issue a modified query to produce a smaller answer. We believe that as the database grows in size (say, linearly) the mean result size grows more slowly as users continue to construct queries with manageable result sizes. However, we do not incorporate this into the database scaling model.

4 Simulation

In this section the hardware simulation is described, together with the parameters that specify the resources consumed by each stage of query execution. An example hardware organization is shown in Figure 2. Every hardware organization consists of a local area network (LAN) connecting several hosts together. Each host has a CPU and memory, a number of I/O buses, and a number of disks. Every host has the same number of I/O buses and every I/O bus has the same number of disks. Each host also has a cache. Table 4 lists the variables that determine the hardware

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

Table 4: Hardware configuration parameter variables, values and definitions.

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
$SeekTime$	15.0	Disk seek time in ms
$TrackToTrack$	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
$LANBandwidth$	30.0	Mbits/sec Bandwidth of the LAN

Table 5: Hardware parameter values and definitions.

organization. The "Value" column in the table shows the "base case" value of each variable used in the experiments described in Section 5. Typically an experiment systematically varies one or more of the values to determine the effect of the variables. A *configuration* is the total collection of variable-value pairs used in an experiment. The *base configuration* is the collection of variable-value pairs given in the tables in this section.

Table 5 shows the base configuration variables for the hardware. The values for this table were taken from [2]. The disks and I/O buses are simulated as follows. Requests for a disk read arrive from the CPU (after determining that they are cache misses). Each read has a specified length in bytes. The reads are queued at the disk in a first-come-first-served (FCFS) manner. Each request is first serviced by the disk by waiting an initial $SeekTime$ milliseconds. The disk loads its track buffer at $DiskBandwidth$ speed. When it finishes, the disk requests access to its I/O bus (only one disk at a time may occupy the I/O bus). When the I/O bus grants access both the I/O bus and disk are occupied for the transfer at $I/OBusBandwidth$ speed. If multiple tracks must be loaded then the initial seek time is extended by the needed track-to-track seeks (variable $TrackToTrack$).

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

Table 6 shows the parameters which affect the CPU and the time taken to process a query. The overall speed of a CPU is determined by the parameter $CPUSpeed$. Varying this value proportionately varies the rate at which instructions are executed. The number of instructions needed to execute various stages of the matching process are listed in the table. Note that the multiprogramming level of the system is on a *per host* bases.

Parameter	Value	Description
$CPUSpeed$	1	Relative speed of each CPU in MIPS
$Multiprogram$	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
$InterInstr$	20	Intersection CPU cost per byte of a decompressed inverted list
$Decompress$	20	Decompression CPU cost per byte of inverted list on disk

Table 6: Base case parameter values and definitions.

Finally, Table 7 lists the variables used to determine the size of the inverted lists. The variable $EntrySize$ determines the number of bits needed to record a posting in an inverted list. The variable $Compress$ determines the reduction in bytes in the inverted list due to compression.

To illustrate the use of the variables in Tables 6 and 7, consider a subquery which intersects two inverted lists with 5 and 10 postings, respectively. The initial subquery CPU processing would be 60,000 instructions ($QueryInstr + 2 \cdot FetchInstr$) since each inverted list read is charged a start-up cost of $FetchInstr$ instructions. The length of one list is 100 bits ($postings \cdot EntrySize \cdot Compress$). The length of the other list is 200 bits. The disk read length for both lists is 512 bytes (rounded up due to $BlockSize$). After the disk data is fetched, only the bits in the actual lists are used for subsequent computations. The number of instructions to process the intersection combines the costs of decompression and intersecting the lists. The size of the uncompressed inverted lists is 600 bits or 75 bytes ($postings \cdot EntrySize$). Then the number of instructions for this part of the subquery processing is 2,250 ($37.5 \cdot Decompress + 75 \cdot InterInstr$).

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

The number of bytes needed to represent a document in an answer is given in $AnswerEntry$. The instructions needed to concatenate the answers from the subqueries is given by $ConcatInstr$. The number of documents in the database is given by the variable $Documents$ and is equal to the number of abstracts in the INSPEC database. Finally, the variable $DatabaseScale$ permits scaling of the database as described in Section 3.

Parameter	Value	Description
EntrySize	40	Bits to represent an inverted list entry on disk (uncompressed)
Compress	0.5	Compression Ratio
CacheSize	0.0	Size of inverted list cache in postings
ConcatInstr	5	Concatenation CPU cost per byte of an answer set
AnswerEntry	4	Bytes to represent an entry in an answer set
Documents	1165059	Number of documents
DatabaseScale	1.0	Database scale factor

Table 7: Base case parameter values and definitions.

5 Result

In this section we present selected results of a set of experiments performed by the simulation. Space limitations prevent us from showing all the results. In conducting these experiments sensitivity analysis of all the variables in Section 4 were performed. An *experiment* is the execution of the simulation for the entire trace with a given configuration. As an example of this, Figure 7 shows the mean response time of queries under the various index organizations as disk seek time increases (the other simulation parameters for this configuration are given in Tables 4–7). The graph shows that the disk index organization is most sensitive (i.e., has the largest slope) to the change in seek time, followed by the I/O bus, host, system and prefetch index organizations, respectively. (The host and system index organizations are identical because the base configuration has 1 host.) This ordering of the index organizations is in decreasing number of inverted lists reads done by each organization. For a given query, the disk index organization does the largest number of reads, followed by the I/O bus index organization, etc. The increase in the number of reads leads to a higher disk utilization and increased queuing delays at every disk. We conclude that the seek time of the disk dominates the cost of accessing an inverted list as opposed to the bandwidth limitations of the I/O subsystem. The host and system index organizations perform identically in the base configuration because there is only 1 host in the base configuration. The prefetch I index organization performs slightly worse than the host and system index organization because the prefetch of an inverted list is performed sequentially with respect to the processing of the remainder of the query. This slightly decreases the amount of parallelism in the processing of the query. (Recall that prefetch I index organizations was designed to reduce LAN traffic, which is not an issue in a one host configuration.)

In Figure 8 the effect of the rise in the multiprogramming level on the mean throughput of queries processed is shown. The graphs shows that the disk and I/O bus index organizations are relatively insensitive to the change in the multiprogramming level. Other collected data shows that these two organizations are bottlenecked in the I/O subsystem. As the multiprogramming level rises, the same number of queries can be processed per second, but each query takes longer and longer. The host, system and prefetch I index organizations continue to improve across the range of the multiprogramming level in the graph because the resources are more evenly balanced. For a multiprogramming level of 32, the response times for the disk, I/O bus, host, system and prefetch I index organizations are 10.98 sec., 2.91 sec., 1.09 sec., 1.09 sec. and 1.09 sec., respectively. Thus good response times are still available on a heavily loaded system.

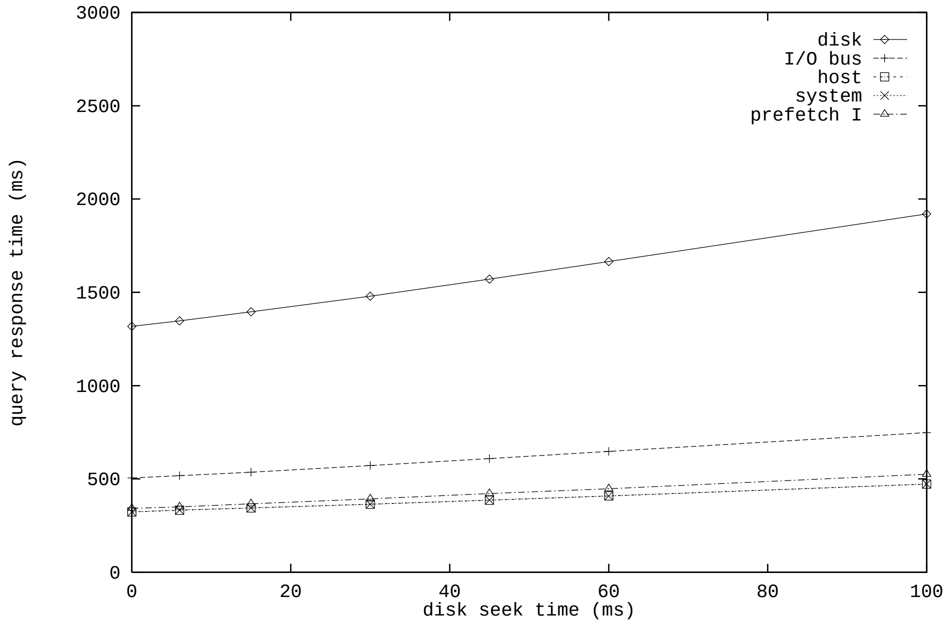

Figure 7: The sensitivity of response time to disk seek time.

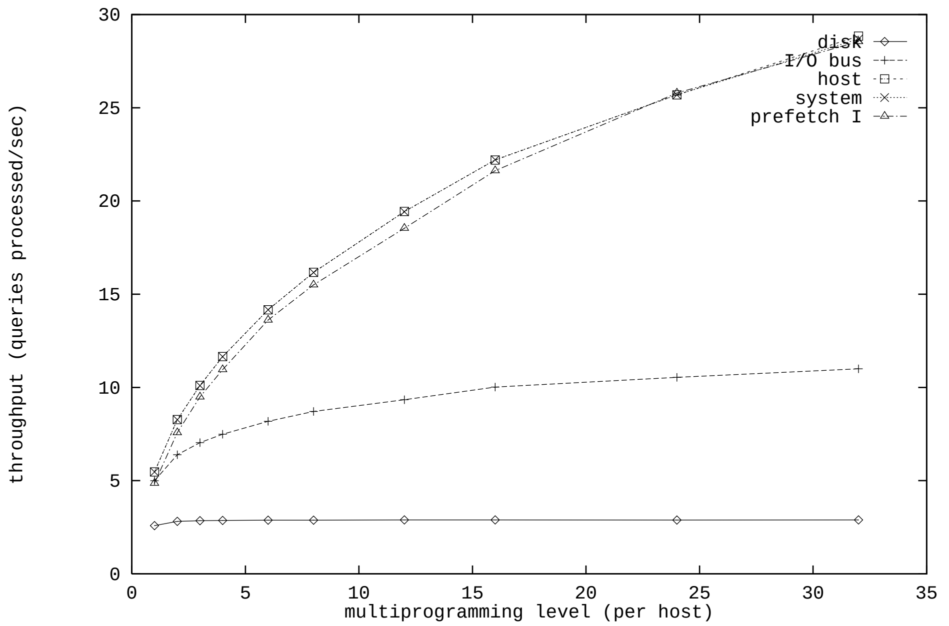

Figure 8: The effect of the multiprogramming level on throughput.

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

Table 8: Enumeration of variable values.

Intuitively, experiments which vary the value of one variable in a configuration examine the change in a function along a single dimension. In some cases it is necessary to change the value of multiple variables in a systematic fashion in a 2^k *factor experiment* [8]. For three variables, this can intuitively be viewed as examining the values of a function at the corners of a three-dimensional cube (each axis of the cube corresponds to a variable).

To determine a reasonable base configuration, some of the values of the variables are provided by existing hardware, but other variables such as $DisksPerI/OBus$ are less easily determined. We conducted a 2^k factor experiment for $k = 3$ on the variables $BlockSize$, $I/OBusesPerHost$ and $DisksPerI/OBus$ to determine the configuration with the best response time. Table 8 lists the enumeration of the values of the variables.

The result of this experiment is graphed in Figure 9. The data points for each index organization have been connected by lines to aid the reader in understanding the graph. (Note that the a line connecting two points may represent the changing of the values of several variables.) We see that the left-hand half of the graph (values 0-3) is the same shape as the right-hand half (values 5-7). This means that $BlockSize$ has little effect on the response time, since it is the only variable to change value when comparing the halves of the graph. Next, examining each *sequential pair* of values (0,1), (1,2) etc. shows an increase in the response time for the disk and I/O bus index organizations. For each pair the only variable to change is $I/OBusesPerHost$ which changes from 2 to 4. Thus, adding I/O buses (and implicitly, disks) decreases the performance of these two index organizations. However, the response time for host, system, and prefetch organizations improve when more I/O buses are added because the total resources of the system are increased. From this graph we pick the best combination of these variables for response time, namely $BlockSize$ of 512, $DisksPerI/OBus$ of 4 and $I/OBusesPerHost$ of 4 as the value for these variables in the base configuration. (Note that this increases the response time of the disk index organization.)

To study database scaling, we first maximize the size of the database which can be effectively processed with the base configuration. We choose a 4 second mean response time as the limit for an effective information retrieval system. We scale the database on the base configuration (as described in Section 3) until the best response time increases to the threshold of 4 seconds. This graph is shown in Figure 10. From this graph we choose the value of 10.0 for the maximum scaling of the database for a single host.

We now wish to observe the effectiveness of the system as the number of hosts is increased. Increasing the number of hosts also increases the total number of I/O buses, disks, and queries (since the number of queries in the entire system is determined by $Multiprogram \cdot Hosts$). In Figure 11 the increase in response time is shown as the number of hosts is expanded. The increase in response time is due to two factors. First, the total load of the system is increasing in proportion to the

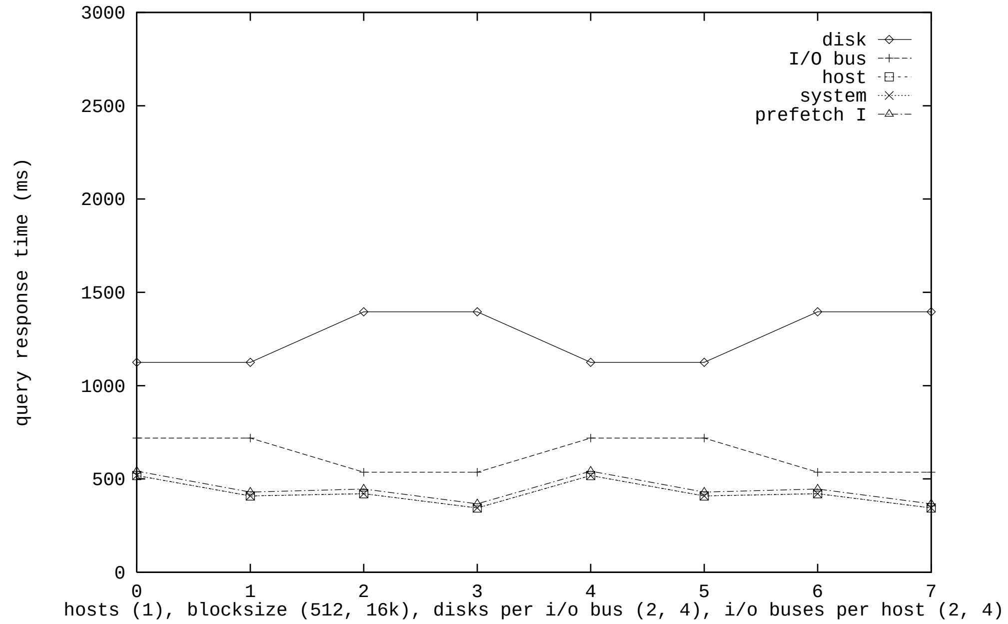

Figure 9: A 2^k factor experiment of three variables.

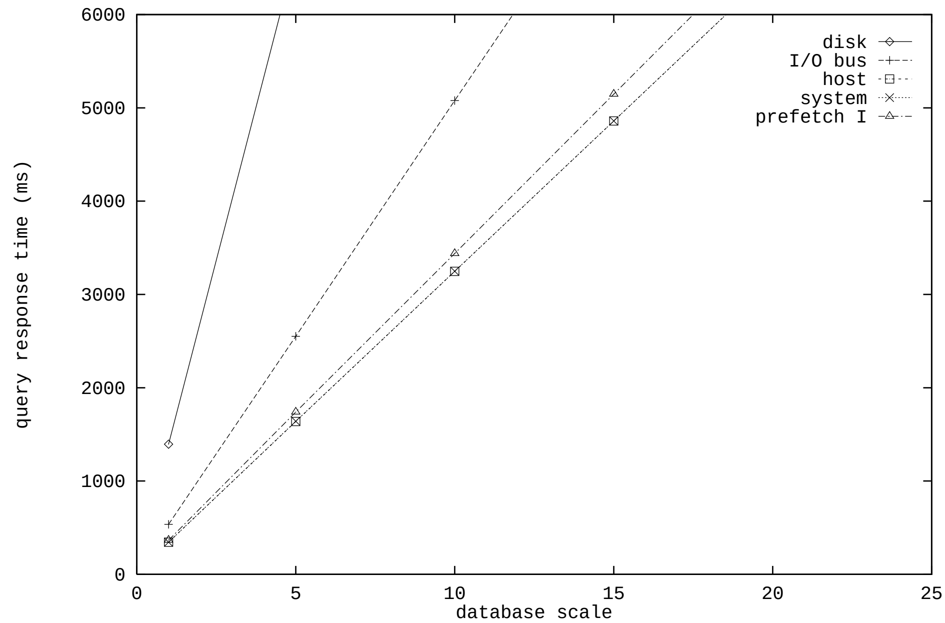

Figure 10: Scaling the database up to a 4 second response time for the best index organization.

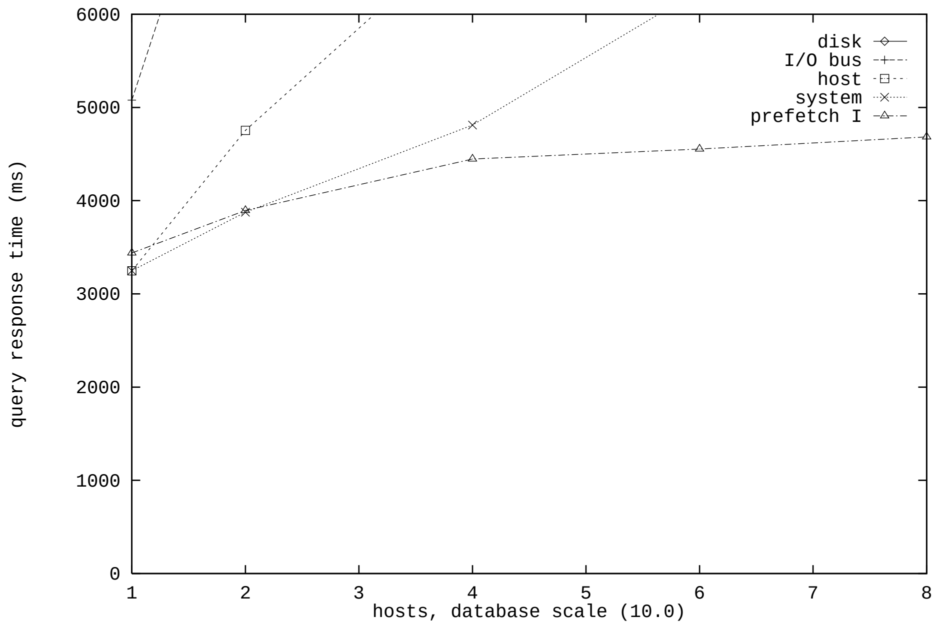

Figure 11: Increasing the number of hosts with a scaled database.

Hosts	I/OBusesPerHost	Multiprogramming	CPU Speed	Database Scale
1	4	4	1.0	10.0
2	2	2	0.5	10.0
4	1	1	0.25	10.0

Table 9: Enumeration of variable values for fixed resources.

number of hosts. Second, as the number of hosts increases the traffic across the LAN increases. We see this effect appear at 3 hosts where the prefetch I index organization outperforms the system index organization. Thus, the prefetch I organization scales well as the number of hosts increase.

Given that the prefetch I organization does well as the number of hosts increases, we can compare the base configuration of a single host to configurations with more hosts but the same total resources in terms of CPU speed, number of disks and I/O buses. This is essentially the trade-off between buying a single large mainframe processor or several slower workstation size processors. Table 9 shows a enumeration of configurations which explore this trade-off under a fixed total system load. The main difference between a single host and multiple hosts is that the fast CPU has been replaced by several slower CPUs interconnected by a LAN. Figure 12 shows that the best index organization (prefetch I) has a response time of 3.44 sec., 3.59 sec., and 3.83 sec. for 1, 2 and 4 hosts respectively, indicating about a half second loss in response time when split among multiple hosts. Throughput for the prefetch I index organization is 1.17 queries/sec., 1.12 queries/sec., and 1.05 queries/sec. for 1, 2 and 4 hosts respectively. Thus a modest performance loss is incurred by using the multiple host organization. The results indicates that a "mainframe" is more effective, but the small improvement has to be evaluated in light of the potentially higher mainframe cost (compared to workstations and a LAN).

Finally, we turn to the issue of caching and address two simple questions. How rapidly does

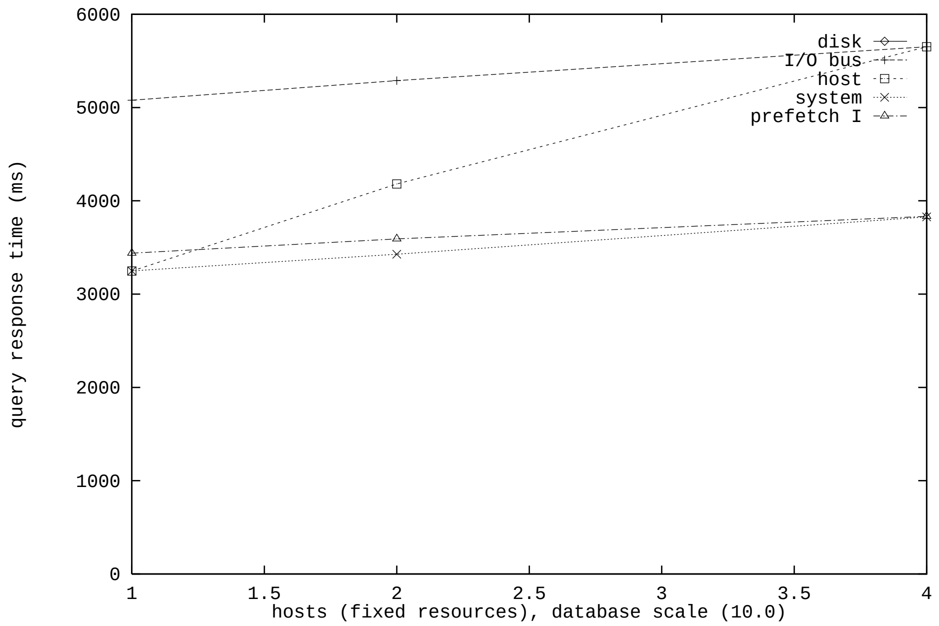

Figure 12: The mainframe vs. workstation trade-off.

the cache hit rate rise as the size of the cache increases? What is the effect of the rising cache hit rate on performance? Figure 13 shows the increase in the cache hit ratio as the size of the cache increases for a four host system (database scale 1.0). Since the total cache size is the same regardless of the index organization, it is suprising that the cache hit rates vary depending on the organization. However, the behavior of the caches under the various organizations is quite different. For the system and prefetch I index organizations an inverted list is cached in only one place in the system. Thus, there are effectively $Hosts$ number of independent caches. Also, suppose a list slightly larger in size than the cache is read from disk. In the system and prefetch I organization, the list does not fit in the cache and thus the caches would remain unchanged. In the disk, I/O bus, and host organizations, however, all four caches would hold a list of quarter the size, requiring some other lists to be removed from the cache. The figure shows that for the base configuration even a small cache has a good hit rate - achieving over 40% where the maximum possible cache hit rate is about 56% (see Table 2). The cache hit rates for the disk, I/O bus, and host index organizations are the same since they access exactly the same lists on each host and so the cache contents are change in the same way over the course of the simulation.

Figure 14 shows that the effect of caching is to free the I/O subsystem resources to speed up query processing for the system and prefetch I index organizations. The cache does not have a dramatic effect on throughput for the disk organization because it remains bottlenecked on the disks.

6 Conclusion

We have studied an information retrieval system for bibliographic entries or abstracts. Using queries from the INSPEC database on the FOLIO system at Stanford University, we analyzed strategies for distributing indexes across a set of processors and for performing queries in parallel.

Our main result is that inverted lists referenced by queries in such systems tend to be relatively

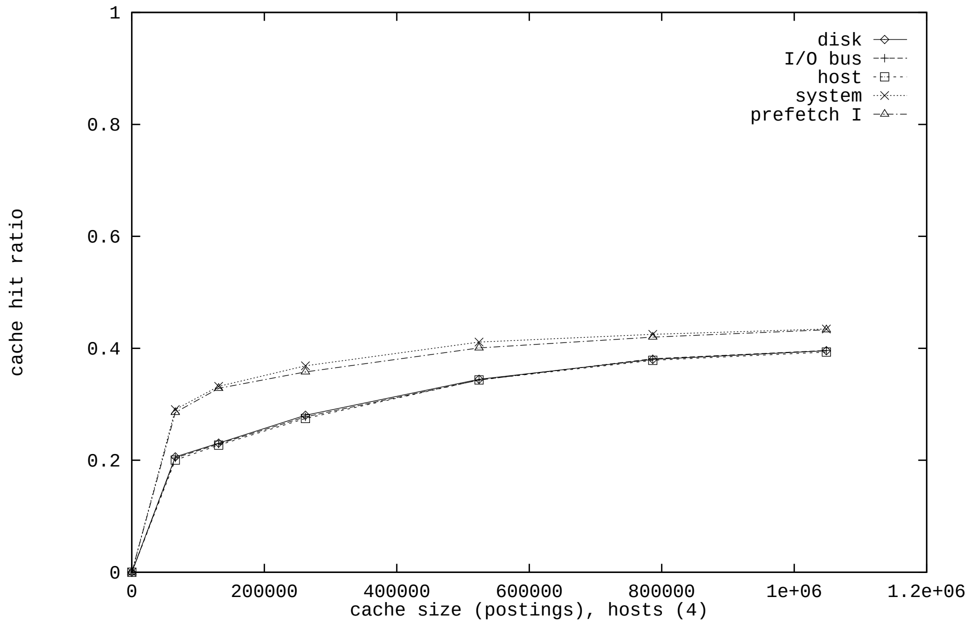

Figure 13: The improvement in the cache hit rate as the cache grows in size.

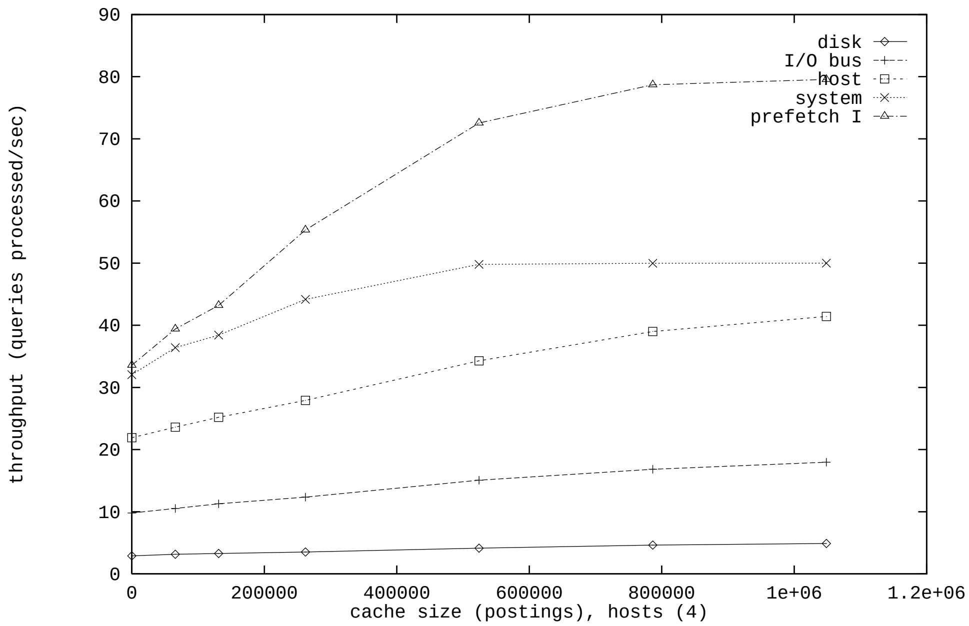

Figure 14: The impact of the cache size on throughput.

short and it does *not* pay off to split them across hosts, much less across I/O subsystems or disks. Either system index organization, or the system index organization with the prefetch I optimization, performs best over wide ranges of parameter values. Prefetch I is especially good as the database size scales up. However, the system organization does utilize the LAN or processor interconnect more heavily, so it would not be appropriate for systems with slow networks.

Our conclusion is different from that of our earlier work [12] where a full-text information retrieval system was analyzed. In that case, inverted lists are much longer, and breaking them up (e.g., striping them) does pay off. In particular, the host organization was superior in that scenario.

In our experiments, we explored the "mainframes vs. workstations" issue. That is, we took a specific index distributed over a fixed number of disks and I/O buses. Then we considered whether it would be best to connect all these resources to a single fast processor, or to connect them to n processors each of $(1/n)^{th}$ the speed. The mainframe does achieve moderately higher throughput, but the gains have to be evaluated in light of the higher mainframe cost. In other words, one has to take the throughput rates we report here, and divide them by the dollar cost of each configuration, to obtain a query/sec/dollar measure, as is done in transaction processing systems [7].

Our caching results indicate that a relatively small cache can improve performance significantly. For our INSPEC database that has an index size of 308MB (129 million postings compressed) a cache of about 3.8 MB (800,000 postings uncompressed), on the order of 1.2% of the index, can improve throughput by about 136% for the prefetch strategy. For the other strategies, improvements are smaller. Although not reported here, we also experimented with various cache policies. For example, in one case, lists above a given threshold were not cached, even if they fit in the cache, on the presumption that they would flush out too many useful lists. However, we observed no significant improvement with this caching variation.

In our study we assume that a single combined index was constructed for all fields (see Section 3). Although we have not yet experimented with partitioned indexes, our current results indicate that performance will *not* improve by partitioning. Partitioned indexes forces *subject* field searches to read more, but shorter lists, which would only exacerbate the disk seek problem. Some systems actually have both combined *and* partitioned indexes. This does not seem attractive either. For some queries, lists read will be shorter, but the same number of disk seeks need to be performed. As long as seeks remain the critical resource, the gains should not be significant. (This assumes that the system is dedicated to query processing. If the hosts are used for other tasks, then reducing the size of lists to intersect may still be desirable.) This conjecture needs to be verified.

Finally, as stated in Section 2, we excluded from our trace 26.7% of the queries. Many of these are wildcard searches (e.g., searching for the keyword "recession*" to cover words such as "recessionary" and "recessional".) For each wildcard term, a number of inverted lists have to be read. This effectively increases the system load, as if simply more queries were run. Thus we do not expect the relative performance of the index organizations to change. As a matter of fact, the disk and I/O bus organization may perform even worse due to the increased disk traffic. Furthermore, the system organization can be tuned so that words with the same prefix hash to the same host (e.g., all inverted lists for words that match "recess*" can be placed on the same host), so that the wildcard search does not involve sending additional lists over the LAN.

Acknowledgements: Thanks to Norman Roth who gathered the raw trace and posting counts from FOLIO. Ben Kao and Luis Gravano provided several useful suggestions.

References

[1] 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.

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

[3] Janey K. Cringean, Roger England, Gordon A. Manson, and Peter Willett. Parallel text searching in serial files using a processor farm. In *SIGIR 1990*, pages 429–453, 1990.

[4] 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.

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

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

[7] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

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

[9] 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.

[10] Craig Stanfill. Partitioned posting files: A parallel inverted file structure for information retrieval. In *ACM Special Interest Group on Information Retrieval (SIGIR)*, pages 413–428, 1990.

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

[12] 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. (To Appear).

[13] Howard R. Turtle and W. Bruce Croft. Uncertainty in information retrieval systems. Presented at an NSF Workshop in Majorca, September 1992.