Distributed Processing of Conjunctive Queries

Claudine Badue[1]
claudine@dcc.ufmg.br

Ramurti Barbosa[2]
ramurti@akwan.com.br

Paulo Golgher[3]
golgher@akwan.com.br

Berthier Ribeiro-Neto[4]
berthier@dcc.ufmg.br

Nivio Ziviani[5]
nivio@dcc.ufmg.br

[1,4,5]Federal University of Minas Gerais
Belo Horizonte, Brazil

[2,3]Akwan Information Technologies
Belo Horizonte, Brazil

ABSTRACT

We distinguish that Web query processing is composed of two phases: (a) retrieving information on documents related to the queries and ranking them, and (b) generating snippets, title, and URL information for the answer page. Using real data and a small cluster of index servers, we study four basic and key issues related to this first phase of query processing: load balance, broker behavior, performance by individual index servers, and overall throughput. Our study reveals interesting tradeoffs: (1) that load unbalance at low query arrival rates can be controlled with a simple measure of randomizing the distribution of documents among the index servers, (2) that the broker is not a bottleneck, (3) that disk and CPU utilization at individual servers depends on the relationship between memory size and the distribution of frequencies for the query terms, and (4) that load unbalance at high loads prevents higher throughput.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems, performance evaluation*; H.3.5 [**Online Information**]: [Web-based services]

General Terms

Performance, design

Keywords

Distributed query processing, search engines, load balance

1. INTRODUCTION

When a user query reaches a search engine, its processing is broken down into two phases: (a) the extraction of the data from the disks and the ranking of the answers, and (b) the production of the answer page to be sent to the user [4].

The first phase consists of taking the query terms, extracting from the disks information on documents that contain each query term, executing a conjunction of the sets of documents that contain each query term, and finally ranking the selected documents. It is a time consuming task that costs at least as much as the second phase (production of the answer page). Further, as the size of the search engine increases, the processing cost for the second phase remains basically constant while the processing cost of the first phase increases (because, for instance, larger lists of documents have to be read from the disks and manipulated). Thus, the first phase is critically important for the performance of modern search engines.

The second phase consists typically of taking the top 10 ranked answers and generating snippets, title, and URL information for each of them. For this, the search engine needs to examine the texts of the documents of the collection. Despite that, no matter how large is the collection, the processing has to be repeated for a fixed number of documents (typically 10). This indicates that this part of the query processing task is not affected (or is only slightly affected) by the size of the document collection. That is, the effort to produce the answer page for a search engine that handles 50 million documents and for a search engine that handles 5 billion documents is similar.

In this work, we concentrate our attention on the first phase of the query processing task. We study four basic issues that affect the internal operation and fine tuning of the cluster of index servers. To validate our conclusions we ran experiments on a real setup with real data obtained from a search engine.

The paper is organized as follows. Section 2 discusses the typical architecture of a cluster of index servers for modern search engines. Section 3 presents the cluster of servers we used in our experimentation and discusses the internal operation of the servers. Following, we analyse 4 basic issues that affect the performance of a cluster of index servers: load balance in Section 4, operation of the broker in Section 5, utilization of CPU and disk at the index servers in Section 6, and throughput estimates for our experimental

cluster in Section 7. Section 8 goes over research related to ours. Our conclusions follow.

2. ARCHITECTURE

Search engines use a network of workstations (or cluster of servers) as platform for query processing [4, 17]. Typically, in this network there are two types of machines: a single broker and p index servers. The whole collection of documents is partitioned among the index servers, such that each server stores its own local subcollection. That is, the p subcollections compose the document collection C. Figure 1 illustrates this architecture.

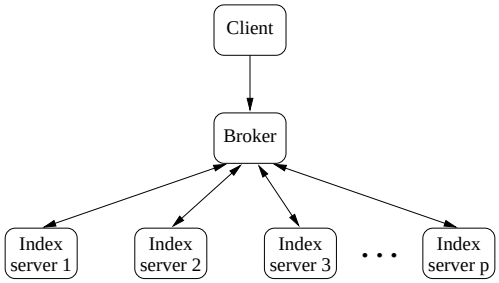

Figure 1: Architecture of a search engine: shared-nothing, local subcollections, local index organization, indexes are inverted files.

Assuming that the documents are evenly distributed among the servers, let b be the size (in bytes) of any local subcollection. Further, let n be the size (in bytes) of the whole collection C. Then,

$$b = \frac{n}{p} \tag{1}$$

An inverted file is adopted as the indexing structure for each subcollection. Inverted files are useful because they can be searched based mostly on the set of distinct words in all documents of the collection. They are simple data structures that perform well when the pattern to be searched for is formed by conjunctions and disjunctions of words.

The structure of our inverted files is as follows. It is composed of a *vocabulary* and a *set of inverted lists*. The vocabulary is the set of all unique terms (words) in the document collection. With each term is associated an inverted list. This list contains an entry for each document in which the term occurs. Each entry is composed of the document identifier (an integer) and the within-document frequency $f_{d,t}$ of occurrence of the term t in the document d. Additionally, the inverted lists are sorted by decreasing within-document frequencies.

Each local inverted file has a size that is $O(b)$. Thus, for p index servers, the size of the index for the whole collection is given by $p \times c_1 \times b$, where c_1 is a proportionality constant. This type of index organization, here referred to as a *local index organization* [16], is the standard "the facto" in all major search engines.

A user query reaches the broker through a client machine. The broker then broadcasts the query to all index servers. Each index server searches its own local index and produces a partial ranked answer. These partial ranked answers are collected by the broker and combined through an in-memory merging operation. The final list of ranked documents is then sent back to the client machine.

3. EXPERIMENTAL SETUP

For the experiments reported in this work we use a cluster of 8 index servers. We opted for a small configuration to facilitate experimentation and allow a more complete evaluation. Despite that, our conclusions hold for large clusters that satisfy our assumptions.

In our setup, each index server is a Pentium IV with a 2.4 gigahertz processor, 1 gigabytes of main memory and a ATA IDE disk of 120 gigabytes. The broker is an ATHLON XP with a 2.2 gigahertz processor and 1 gigabytes of main memory. The client machine, responsible for managing the stream of user queries, is an AMD-K6-2 with a 500 megahertz processor and 256 megabytes of main memory. All of them run the Debian Linux operating system version 2.6.

The test collection, referred to as WBR03, is composed of Web pages collected by the TodoBR [20] search engine from the Brazilian Web in 2003. Each index server holds a subcollection of 10 million pages. The inverted file for each subcollection occupies roughly 5.7 gigabytes. Thus, with 8 servers, the size of the index for the whole collection is 45.6 gigabytes.

The query set used in our tests is composed by 10,000 unique queries with two or more terms. In this query set, there are 8,052 unique terms. These queries were extracted from a query log submitted to the TodoBR search engine in January 2003. Notice that we use a set of *unique* queries to reduce the influence of disk cache.

In this work, we have used a ranking procedure that is based solely on the relative frequencies of occurrence of terms within documents. This means that our ranking computation is faster than the ranking computation in a modern large search engine (that takes into consideration other factors such as positions of terms within documents and anchor texts). Nevertheless, our results allow understanding basic tradeoffs in the operation of distributed index servers.

To rank the selected documents based on the relative frequencies of occurrence of terms within documents, we use the standard vector space model [3, 18, 24]. In this model, queries and documents are represented as weighted vectors in a t-dimensional space, where t is the number of terms in the vocabulary of the collection. With each pair term-document is associated a weight that is a function of the frequency $f(t,d)$ of the term t in the document d (the term frequency *tf*) and the inverse document frequency (*idf*) of the term t in the collection, computed as $log\frac{N}{n_t}$ (where N is the total number of documents in the collection and n_t is the number of documents in which the term t occurs). Because of that, this weighting scheme is usually referred to as a *tf-idf* weighting scheme. The rank of a document with regard to a user query is computed as the cosine of the angle between the query and document vectors.

In our experiments, the client machine submits the queries to the broker at a fixed query arrival rate (which is varied from one experiment to the other). For this, the client employs the software httperf [13]. A copy of each query is passed to each index server, which produces a partial ranked result relative to its local subcollection. The index server then sends its partial answer to the broker, which combines the 8 partial ranked answers into a final ranking.

The broker processes queries concurrently using multiple threads as follows. A thread continuously polls for incoming client query requests and broadcasts them to all index servers. A set of threads, one for each processing thread in the servers, continuously poll for partial results generated by the index servers. Whenever a partial result is retrieved it is inserted into a buffer. As soon as all partial results relative to a query have been received, the query is inserted into a merge queue. A distinct thread is responsible for extracting the queries from the merge queue, merging the partial results into a final ranking, and sending this final ranked results to the client machine.

To generate a partial ranked answer set, an index server receives query requests from the broker and inserts them in a queue. Queries in this queue are processed concurrently employing multiple threads. In our experiments we fine tuned the number of processing threads per index server to provide peak performance. As a result, all our experiments reported here use 8 threads per index server.

Once a processing thread takes a query out of the queue, it reads from disk the full inverted lists relative to the query terms, intersect the lists to produce the set of documents that contain all query terms (i.e., the conjunction of the query terms), compute a relevance score for each document, and sorts them by decreasing score. We read the full list because we assume that the query is a conjunction of the query terms, as done by modern search engines. As soon as the ranking has been computed, the top ranked documents are transferred to the broker machine.

Higher the variance in the processing times of the index servers, slower tends to be the throughput of our cluster of servers. Particularly, if a given index server is always the slowest machine independently of the query, that server becomes a bottleneck. Thus, it is critically important to reduce the load unbalance among the various index servers, as we discuss in the immediately following.

4. LOAD BALANCE

To reduce load unbalance among index servers we opt for balancing the distributions of the sizes of the inverted lists that compose the local inverted files (stored locally at the index servers). Fortunately, this can be accomplished by a simple procedure. Given a new document, we assign it to an index server through a *random* sorting.

A random distribution of documents among index servers works because it naturally spreads documents of various sizes across the cluster. As a result, the distributions of document sizes in the index servers become similar in shape, which reflects in inverted lists whose size distributions are also similar. Our motivation is to equalize the utilization of *space* at the various index servers, which should reflect in an equalization of processing *time* at the various servers, reducing load unbalance. Let us examine first the utilization of *space* at the disks of the index servers.

Figure 2 illustrates the distributions of sizes of the inverted lists that compose our 8 local inverted files. The terms in the horizontal axis are sorted by decreasing size of the corresponding inverted list. We observe that these distributions are very similar in shape indicating that the random assignment of documents to servers works.

Given that the utilization of disk space at the index servers is even, let us examine the impact on local processing times.

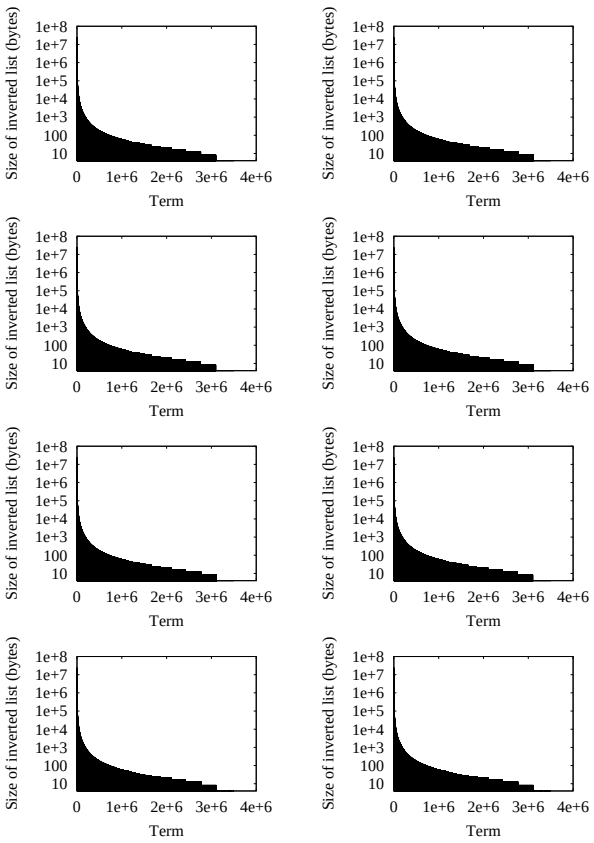

Figure 2: Distributions of sizes of the inverted lists that compose our 8 local inverted files.

For this, we reprogram the client to send queries to the broker one at a time. That is, the client sends query q_i and waits until the results have been produced. Only then does the client proceeds to query q_{i+1}. This procedure is executed for all queries in our log. The motivation is to evaluate local processing times independently of factors such as interference among the various processing threads at each index server.

Figure 3 illustrates the distributions of average, maximum, and minimum local processing times per query. Average time for a query q is computed as the average time among all 8 local processing times. The time interval surrounding average times have limits given by the slowest and the fastest index server for that query. To allow visual inspection, we display results for selected queries at intervals of 200 queries.

We notice that variance in processing times is small for the slower queries. The slowest query in our log, for instance, has average processing time of 210 milliseconds and maximum processing time of 225 milliseconds, a variance of 7%. For the faster queries, variance is much higher. For instance, the next to the last query shown in the figure has average processing time of 10 milliseconds and maximum processing time of 22 milliseconds, a variance of 120%. Not much can be done regarding this variance due to interference of the operating system and the inherent difficulties in controlling disk access times at this time granularity.

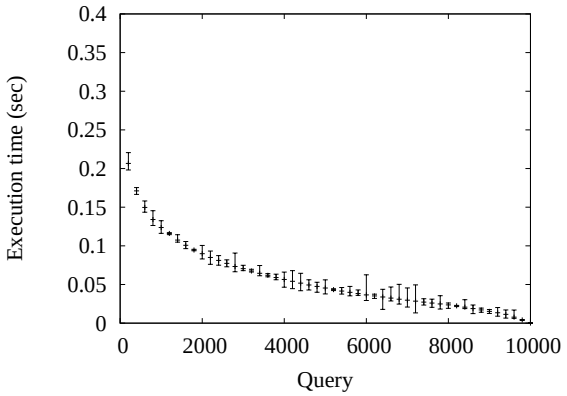

Figure 3: Distribution of average, maximum, and minimum local processing times (seconds) per query. Queries are executed one at a time. Processing times are sorted by decreasing order of average time. Only selected queries are shown explicitly, at intervals of 200 queries.

We should comment that early experimentation reported very high variance of local processing times. Our analysis of those results showed that the distributions of list sizes among index servers were very dissimilar. The reason was that we were assigning documents to the index servers according to a *sequential* assignment based on document identifiers. This suggested that a random assignment could reduce the problem, what really was the case, as reported by our experiments.

Besides reducing load unbalance among the index servers, another issue of concern is the fact that the broker is a potential bottleneck, which we now discuss.

5. THE BROKER IS NOT A BOTTLENECK

Observing the architecture in Figure 1, we notice that the broker constitutes a potential bottleneck. Every query that is submitted to our cluster of servers is processed by the broker, which has to merge the partial results produced by the various servers. However, this merging of partial results is done all in memory and can be done quickly. As a result, the broker works at relatively low loads at all times, as we now evaluate.

To stress the broker, we implemented a concurrent process monitor that simulates a cluster with p index servers. By varying p we can stress the broker and observe how it behaves. The broker takes queries at an arrival rate r (which we varied) and passes them to a single machine that runs our concurrency monitor. For each query, the broker sends p query requests to our monitor, one for each index server. For each query request it receives, the monitor simulates a local query processing task. For this, it sorts an execution time t_i from a real probability density function previously computed, associates this time to a task, and inserts that task in a time agenda, t_i units of time ahead of the present time.

Our monitor manages the time agenda by visiting its entries in circular fashion. Whenever it finds a task in an entry of the agenda, it sorts an answer from an array of precomputed answers stored in main memory, and sends this answer to the broker. Thus, from the viewpoint of the broker everything goes on as if there were indeed p index servers in the cluster.

Figure 4 reports the average time at the broker per query. We observe that even at high loads and with a large number of servers, time at the broker is not affected. In fact, with 256 servers and query arrival rates around 100 queries per second, average time at the broker per query is less than 10 milliseconds. That is, the broker is not a bottleneck in our architecture.

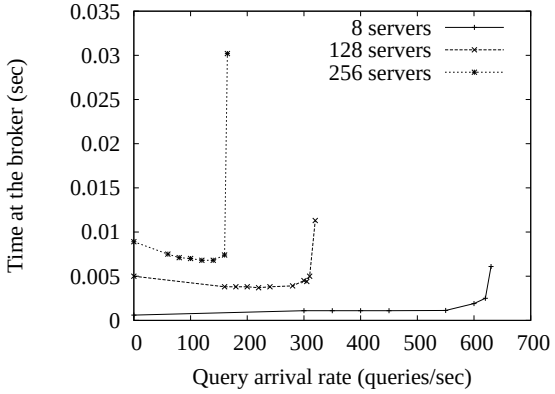

Figure 4: Average time (seconds) at the broker per query as a function of query arrival rate (queries/second).

This is because of two fundamental reasons. First, all the work the broker does is carried out fully in main memory. Second, all the tasks the broker executes are simple tasks that do not take much CPU time. The broker does not have to make ranking computations and does not have to execute algebraic operations (other than comparing document scores). Further, since it has to wait for the partial results for each query, it has plenty of free resources that can be shared with the various queries in the input stream.

Eventually the broker saturates as shown in Figure 4. In our case saturation is abrupt. The reason is that saturation is caused by contention at the network drive interface. With 256 servers, for instance, each user query requires 256 *write* operations to send the queries out and another 256 *read* operations to retrieve the partial answers from the network. At a rate above 150 queries per second, the network drive interface fails to handle the load. This is not critical though because, as we later show, the index servers saturate at a rate one order of magnitude smaller (i.e., index servers saturate at 18 queries per second). That is, a broker whose main task is to merge partial answer sets is not a bottleneck.

Given that load unbalance among index servers is controlled and that the broker is not a bottleneck, the individual performance of the index servers is a critical issue. Thus, let us examine in more detail how local processing time is affected by disk and CPU execution times.

6. CPU AND DISK TIMES AT SERVERS

Our interest is on examining how local processing time is split among disk and CPU at each index server. For this we again submitted the queries to the broker one at a time.

Figure 5 illustrates average local processing time, average disk access time, and average CPU execution time (in seconds). We observe that CPU times are remarkably dominant, particularly for the slowest queries. In fact, the shape of the distribution of local processing times is established by the distribution of CPU times. This is a result that we did not anticipate, so we proceeded with further experimentation.

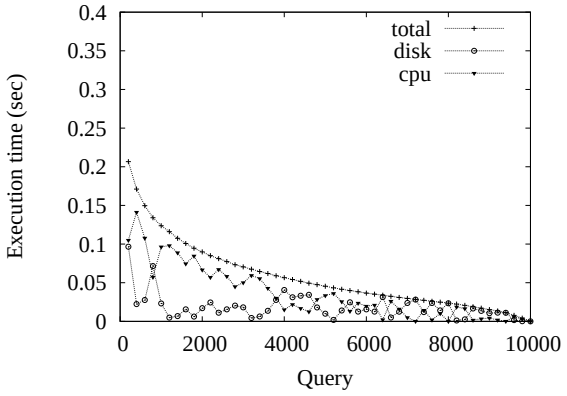

Figure 5: Average local processing time (total), average disk access time, and average CPU execution time (seconds) per query. Queries are sorted by decreasing order of local processing times (total).

We reasoned that maybe disk access time was relatively small in our experiments because of some particularity of our setup. For instance, a possibility could be that inverted lists were being held indefinitely in a disk cache accelerating I/O. To better understand this, we evaluated the distribution of sizes of inverted lists corresponding just to the terms that appear in our query log.

Figure 6 illustrates the distribution of sizes of inverted lists relative to the distinct query terms of our log for the index server 1. We notice that it presents a much "fatter" shape than the distribution of sizes for all terms (for the index server 1) illustrated in Figure 2. That is, the users prefer to write queries using terms that appear more frequently in the document collection. This suggests that terms that are more frequently used by the editors of the Web pages are also more likely to appear in the user queries.

The total space occupied by all lists in Figure 6 is 1.59 gigabytes (area under the curve). Index server 1 has 1 gigabytes of main memory of which roughly 800 megabytes are available for use by the operating system for caching I/O operations. This means that, in our setup, cache for disk is half the total size of all inverted lists that have to be actually retrieved from disk to answer all queries in our log. Notice that this is not evident from the examination of the distributions of list sizes for the whole index (depicted in Figure 2).

Given that disk cache (in our setup) is large compared to the total volume of data retrieved from disks, our disk times are smaller than CPU times. This is peculiar because it shows that, even if the size of the inverted file is large, the

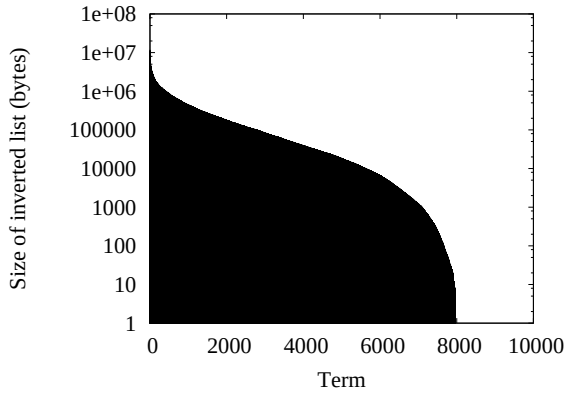

Figure 6: Distribution of sizes of inverted lists for the distinct (unique) terms in our query log at the index server 1.

total volume of data actually retrieved from disk might be much smaller, accelerating disk operations.

Figure 7 illustrates the distribution of average local processing time for the 50 queries executed first in our log. We notice that disk times now dominate the processing time and that they are much larger than previously reported. This clearly shows that our particular setup allowed the operating system to take large advantage of disk caching operations when the entire log of queries was executed.

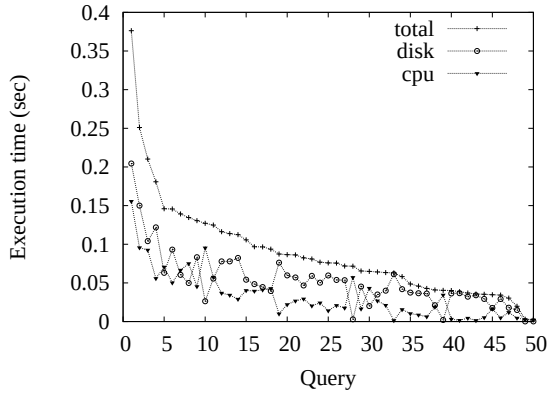

Figure 7: Average local processing times for the 50 queries executed first in our log.

In a large search engine, one might accommodate a larger collection of documents in each index server. To exemplify, by storing 50 million documents in each index server we could handle a collection of 5 billion documents using a cluster with 100 index servers (that our broker could manage without becoming a bottleneck). This means that inverted lists would be larger and that, consequently, disk access times would increase. Furthermore, if a positional index is used (storing the positions of each term in a document) one can expect the size of the inverted file to triple. In this context, CPU time could become the smaller component of the local processing time. This is one issue that deserves careful attention while tuning the cluster of servers.

The point that we make here is that, depending on the

setup of the search engine, CPU time at individual index servers might be significant, even dominant. In fact, in our particular scenario of a cluster of 8 servers storing a collection of 80 million documents, CPU time is determinant. In this context, buying faster SCSI disks is not indicated. Instead, if higher throughput is required with the same architecture of 8 machines, faster CPUs should be the focus of the buyer.

7. THROUGHPUT

We now examine throughput in our cluster of 8 servers. The client sends queries to the broker at an average rate of r queries per second. We vary r until the cluster saturates.

Figure 8 illustrates our results. Average throughput is computed for the 10 thousand queries in our log. Throughput increases linearly up to 18 queries processed per second when the cluster starts to saturate. Despite this linear increase in throughput, let us examine how load unbalance is affected as arrival rates increase.

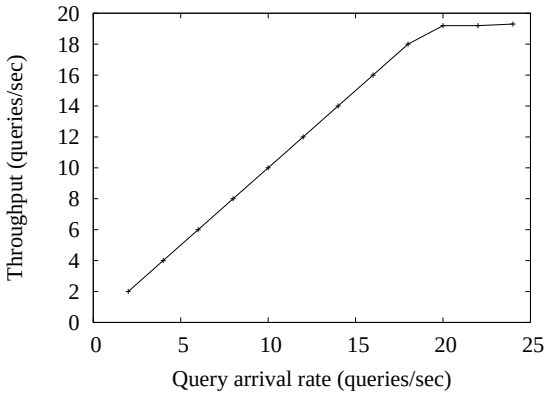

Figure 8: Throughput as a function of arrival rate for our cluster with 8 index servers. We vary arrival rate at intervals of 2 queries per second.

Figure 9 illustrates average query response time for the ten thousand queries in our log. Up to an arrival rate of 8 queries per second, our cluster provides answers to each query in less than 100 milliseconds on average. At 12 queries per second, average query response time is above 100 milliseconds, reaching 150 milliseconds for an arrival rate of 14 queries per second.

If we do not allow an arrival rate higher than 12 queries per second (deferring input traffic is a basic congestion control measure), and since our system is stable at this arrival rate, we have a throughput of 12 queries per second with low latency. This means that our cluster of 8 machines can process up to 43,200 queries per hour or roughly a million queries per day, with low latency. If we allow higher query latencies we can go up to the 16-18 queries processed per second, as illustrated in Figure 8.

Throughput of large search engines, while smaller because index sizes are larger, are expected to follow a similar pattern, since the broker is not a bottleneck.

Let us now examine the variance of query execution times. Figure 10 illustrates the distribution of query execution times at an arrival rate of 4 queries per second. We notice that

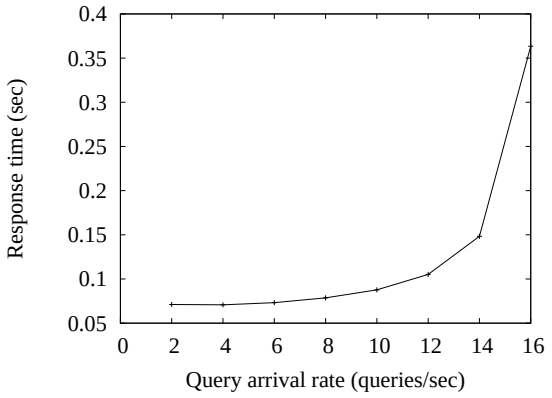

Figure 9: Average query response time as a function of arrival rate (cluster with 8 index servers).

load unbalance is controlled, similarly to what happens when queries are submitted sequentially (see Figure 3). That is, at low arrival rates the random distribution of documents among the servers does work as an effective measure of load balance.

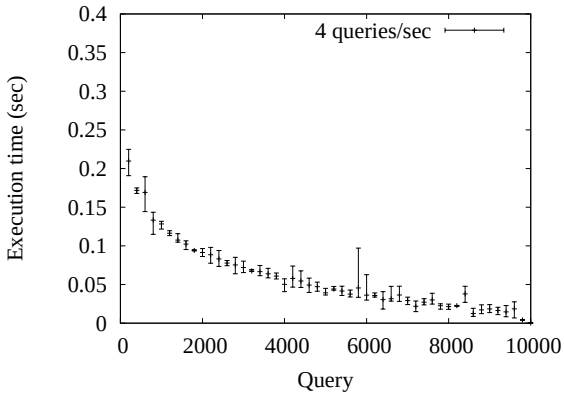

Figure 10: Distribution of query execution times at a rate of 4 queries per second (cluster with 8 index servers).

Figure 11 illustrates the distribution of query execution times at an arrival rate of 12 queries per second. We notice that, even though the cluster operation is still stable, load unbalance is severe. That is, at high arrival rates the random distribution of documents among the servers no longer works as an effective measure of load balance. This clearly suggests that other measures of load balance might be required, such as a load balancing module that takes decisions based on global information on the operation of the servers, if throughput is to be further increased with the same number of index servers. This issue is outside the scope of our work, but it is one that deserves attention, particularly by the community of systems performance and operating systems.

8. RELATED WORK

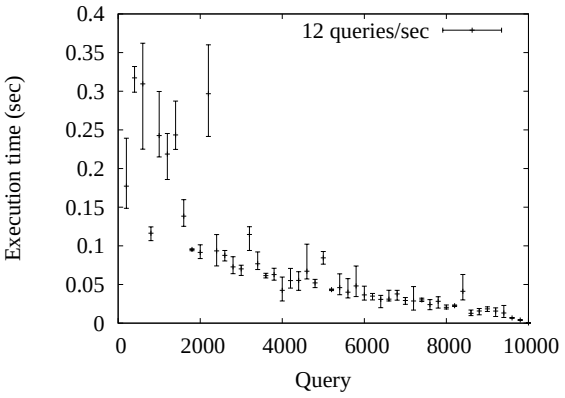

Figure 11: Distribution of query execution times at a rate of 12 queries per second (cluster with 8 index servers).

The Google and FAST search engine architectures are presented in [4, 5, 17]. In the first phase of query execution, index servers consult an inverted index and determine a set of relevant documents. In the second phase, document servers take the ordered list of document identifiers and compute their snippets. In both phases, documents are distributed into pieces, having multiple machines responsible for handling each piece. The work in [5] describes the features adopted by Google to produce search results, such as the link structure of the Web and anchor texts.

Our work is different from the works in [5, 4, 17] as follows. The work in [5] reports only four sample query times, and the work in [4] does not include any performance evaluation, while we analyse the strategy for partitioning a collection by document, broken down overall performance in costs of critical phases of query execution and identified a set of design trade-offs over a distributed architecture. The work in [17] investigates strategies for partitioning a collection by document, which take into consideration multiple properties of documents, and compares them against a reference system that distributes documents randomly. Nevertheless, they report only the ratio of query processing rate between proposed strategies and the reference system.

An overview of current Web search engine design is offered in [1]. After introducing a generic search engine architecture, they examine each engine component. Nevertheless, their work does not include any performance evaluation for distributed query processing.

The work in [14] presents a parallel and distributed architecture for a Web search engine. Both task-parallel and data-parallel approaches are exploited within their framework. Preliminary experiments highlighted the better performance of a hybrid task+data parallelization strategy. However, they considered a sequential query service, used a small collection as a benchmark, conducted experiments on a cluster with only three machines, and reported only average query time for performance evaluation.

The work in [7] evaluated the performance of a distributed information retrieval system using a variety of workloads. Their system consists of clients connected to information retrieval engines through a central administration broker. Engines hold independent collections. The connection server forwards requests to the appropriate engines, according to the command issued by the client.

Our work is different from the work in [7] as follows. First, while their work explores an architecture that distributes multiple autonomous text collections, we analyse distributed query processing performance of an individual Web collection. Second, while their work derives results from simulation, we implement and thoroughly evaluate distributed query processing on a real case distributed architecture. Third, we analyse the distinct phases of query processing, and identify performance tradeoffs not present in their work.

Early studies on distributed indexes for information retrieval systems appeared in [2, 10, 16, 8, 21, 11, 19]. In [2, 10, 16, 8, 21, 11] various data partitioning schemes among servers are discussed. This was an issue of early concern. With modern search engines, however, there is a clear preference for document-based partitioning because it simplifies management of the system and computation of conjunctions of query terms. In [19] a parallel SIMD algorithm for document retrieval is presented. In modern times, there is a clear preference for networks of low cost workstations (like our cluster of index servers).

Methods for efficient query evaluation have been studied extensively in the information retrieval literature. The work in [22] classify evaluation strategies in term-ordered query evaluation and document-ordered query evaluation and the work in [9] compares them. The works in [12, 15, 23] presents algorithms that process query terms in some order that lets the system identify the top n scoring documents without processing all query terms. The work in [6] presents a method based on a two level approach: at the first level, iterates in parallel over query term postings and identifies candidate documents using an approximate evaluation; at the second level, promising candidates are fully evaluated and their exact scores are computed. Strategies for efficient query evaluation using pruning techniques are beyond the scope of our paper.

9. CONCLUSIONS AND FUTURE WORK

We have studied 4 basic and key issues related to the internal operation and fine tuning of a cluster of index servers for search engines: load balance, broker behavior, CPU and disk times at individual index servers, and throughput for a particular setup. Some of our findings were not anticipated, as follows.

Load unbalance among index servers, which has to be limited to avoid penalties in throughput, can be controlled by randomizing the distribution of the documents of the collection among the index servers. This works well at low arrival rates, but is not effective at high arrival rates. This suggests that adding a module of dynamic load balance to the system might lead to improved throughput for a given setup. While load balance has been long discussed in distributed systems, to the best of our knowledge, it has not been addressed in the context of modern search engines. Thus, it might constitute an interesting topic of research.

While we have always thought of the broker as a serious candidate for bottleneck, this is not the case. In fact, a single broker can handle hundreds of index servers at loads 10 times higher than the saturation load of the individual index servers. The broker is not a bottleneck in a shared-nothing architecture of independent index servers, as the one used by modern search engines.

CPU and disk times at an index server are heavily affected

by the query load. Since users tend to use common (or high frequency) terms in their queries, the actual number of inverted lists that have to be retrieved from disk might be a fraction of the inverted file. And, since even low end servers have considerable availability in main memory, going up to 1 gigabytes and beyond, memory space for disk caching might considerably accelerate I/O operations making them comparable to CPU times. In our particular setup with 8 index servers and a collection of 80 million documents, CPU times at the index servers dominated local processing times. These results are not definitive, but indicate that care might be exercised during dimensioning of a cluster of servers for handling a particular document collection.

Throughput figures at low arrival rates (such as 4 queries per second) indicate that the cluster of servers operates stably with low variance in local processing times among index servers. At high loads, however, variance of local processing times quickly increases limiting scalability of cluster throughput. That is, randomizing the distribution of documents among servers does not help at high query arrival rates. This suggests that adding a module of dynamic load balance to a cluster of servers (of a search engine) might be an effective measure to improve throughput at high arrival rates.

Our views and findings on these basic issues regarding the internal operation of search engines indicate that much research needs to be done to fully address all the relevant issues.

10. ACKNOWLEGMENTS

This work was supported by the GERINDO project–grant MCT/CNPq/CT-INFO 552.087/02-5, by CNPq scholarship 140262/2001-6 (Claudine Badue), by CNPq grant 520.916/94-8 (Nivio Ziviani) and by CNPq grant 30.0188/95-1 (Berthier Ribeiro-Neto).

11. REFERENCES

[1] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technology*, 1(1):2–43, August 2001.

[2] C. S. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proceedings of the Eighth String Processing and Information Retrieval Symposium*, pages 10–20, Laguna de San Rafael, Chile, 2001. IEEE Computer Society.

[3] R. Baeza-Yates and B. Ribeiro-Neto, editors. *Modern Information Retrieval*. ACM Press New York, Addison Wesley, 1999.

[4] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. In *IEEE Micro*, pages 22–28, 2003.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[6] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 426–434, New Orleans, LA, USA, 2003.

[7] B. Cahoon, K. S. McKinley, and Z. Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions of Information Systems*, 18(1):1–43, 2000.

[8] B. S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[9] M. Kaszkiel and J. Zobel. Term-ordered query evaluation versus document-ordered query evaluation for large document databases. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 343–344, Melbourne, Australia, 1998.

[10] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In *Proceedings of the Seventh International Symposium on String Processing and Information Retrieval*, pages 209–220, La Coruña, Spain, 2000. IEEE Computer Society.

[11] I. Macleod, T. Martin, B. Nordin, and J. Phillips. Strategies for building distributed information retrieval systems. *Information Processing & Management*, 23(6):511–528, 1987.

[12] A. Moffat and J. Zobel. Fast ranking in limited space. In *Proceedings of the Tenth International Conference on Data Engineering (ICDE)*, pages 428–437, Houston, Texas, USA, 1994. IEEE Computer Society.

[13] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *Proceedings of the Internet Server Performance Workshop*, pages 59–67, 1998.

[14] S. Orlando, R. Perego, and F. Silvestri. Design of a parallel and distributed web search engine. In *Proceedings of the 2001 Parallel Computing Conference (ParCo 2001)*, pages 197–204, Naples, Italy, 2001. Imperial College Press.

[15] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[16] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM Conference on Digital Libraries*, pages 182–190, 1998.

[17] K. M. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In *Proceedings of the First Latin American Web Congress*, pages 132–143, Santiago, Chile, 2003.

[18] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[19] C. Stanfill, R. Thau, and D. Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of the 12th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 88–97, Cambridge, Massachusetts, USA, 1989.

[20] TodoBR. Main page: http://www.todobr.com.br.

[21] A. Tomasic and H. 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*, pages 8–17, San Diego, California, USA, 1993.

[22] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management:*, 31(6):831–850, 1995.

[23] A. N. Vo and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 290–297, Melbourne, Australia, 1998. ACM Press New York, NY, USA.

[24] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes - Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Inc., second edition, 1999.