Issues in Parallel Information Retrieval

Anthony Tomasic
Stanford University*

Hector Garcia-Molina
Stanford University†

Abstract

The proliferation of the world's "information highways" has renewed interest in efficient document indexing techniques. In this article, we provide an overview of the issues in parallel information retrieval. Basic issues in information retrieval are described and various parallel processing approaches to the problem are discussed. To illustrate the issues involved, we discuss an example of physical index design issues for inverted indexes, a common form of document index. Advantages and disadvantages for query processing are discussed. Finally, to provide an overview of design issues for distributed architectures, we discuss the parameters involved in the design of a system and rank them in terms of their influence on query response time.

1 Introduction

As the data volume and query processing loads increase, companies that provide information retrieval services are turning to parallel storage and searching. The idea is to partition large document collections, as well as their index structures, across computers. This not only allows for larger storage capacities, but also permits searches to be executed in parallel.

In this article we sample research in the area of parallel information retrieval. We start by summarizing basic information retrieval concepts, and then describe how they have been applied in a parallel environment. We also give a short summary of our own research in this area, mainly as an example of the types of algorithms that need to be developed, and the system issues that need to be studied.

2 Information Retrieval Basics

For an introduction to full-text document retrieval and information retrieval systems, see reference [16]. An *information retrieval model* (IRM) defines the interaction between a user and an information retrieval system and consists of three parts: a *document representation*, a *user need* and a *matching function*.

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

*Department of Computer Science, Stanford, CA 94305-2140. e-mail: tomasic@cs.stanford.edu
†Department of Computer Science, Stanford, CA 94305-2140. e-mail: hector@cs.stanford.edu

The vector IRM is popular in academic prototypes for information retrieval systems and has recently gained commercial acceptance. Its document representation is the set of words that appear in each document and an associated *weight* with each word. The weight indicates the "relevance" of the corresponding word to the document. Thus, a document is represented as a vector. A vector IRM user need is represented by another vector (this vector can be extracted from a document or a set of words provided by a user). The matching function computes the similarity between the user need and the documents. Thus, all the documents can be ranked with respect to the similarity. Typically, the topmost similar documents are returned to the user as an answer. There is much research on the assignment of weights to words and on the effectiveness of various matching functions for information retrieval. However, both the boolean IRM and the vector IRM and associated variation of these models can be computed efficiently with inverted lists. (See Section 4 for a description of inverted lists.) Reference [28] surveys information retrieval models.

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

In the design of full-text document retrieval systems, there is a basic trade-off between the time to process the document database and the time to process queries. Broadly speaking, the more time spent processing the document database (i.e., building indexes) the less time is spent processing queries. In some scenarios (such as government monitoring of communication), a tremendous amount of information must be queried by only a few queries. In this case, time spent indexing is wasted and linear searching of documents is more efficient. Work in this area concentrates on hardware processors for speeding up the scanning of text [11]. More typically, indexing the documents is worthwhile because the cost can be amortized across many queries. We consider only these latter systems.

Emrath's thesis [6] explores this trade-off between query and update time by providing a data structure that can be tuned in the amount of information indexed. Essentially, the database is partitioned into equal sized "pages." A page is a fixed number of words located together in a document. Duplicate occurrences of words are dropped within a page. If the page is large, many duplicates are dropped from the index, speeding up indexing time. If the page is small, few duplicate words are dropped, slowing down indexing time. For certain applications this tuning of the data structure works well.

More recent work [18, 26, 27] uses physical index design to express the trade-off. The collection of documents is partitioned and each partition has an independent index at the physical index design level, but the entire collection has a single logical index. This provides fast update time but slow query time since each physical index must be searched. To provide fast query time, the physical indexes are merged according to a family of algorithms. More typically, indexing the documents in a single physical index is worthwhile because the cost can be amortized across many queries. We consider only these latter systems for the remainder of this article.

Much research has gone into designing data structures for indexing text. Faloutsos [7] is a survey of this issue. One approach is the use of *signature schemes* (also known as superimposed coding) [13]. Here, each word is assigned a random (hashed) k-bit code of an n-bit vector – for example the word "information" might correspond to bit positions 10 and 20 of a 2 kilobyte vector. Each document is represented by an n-bit vector containing the union of all the k-bit codes of all the words in the document. Queries are constructed by producing an n-bit vector of the k-bit codes of the words in the query. Matching is performed by comparing a query against the document vectors in the database. This scheme is used because the signatures of documents can be constructed in linear time. Unfortunately, the matching process produces "false drops" where different words or combinations of words are mapped into the same k-bit codes. One approach is to ignore false drops and inform the user that some additional documents may be returned. We do not consider this approach further.

Otherwise, each document in the result of the matching process must be checked for false drops. While the number of false drops can be statistically controlled for the average case, the worst-case behavior of this data structure implies checking *every* document in the database for some queries, which is prohibitively expensive for large document collections. Lin [14] describes a signature scheme where multiple independent signatures are used to control false drops and to improve parallel performance.

Another data structure is PATRICIA trees and PAT arrays [9, 10]. Here, the database is represented as one *database string* by placing documents end-to-end. A tree is constructed that indexes the semi-infinite strings of the database string. A semi-infinite string is a substring of the database string starting at some point and proceeding until it is a unique substring. The PAT system provides indexing and querying over semi-infinite strings. The New Oxford English Dictionary has be index using this data structure. The query time, indexing time, and storage efficiency are approximately the same as inverted lists. The techniques described here can be applied to this data structure.

For commercial full-text retrieval systems, inverted files or inverted indexes [8, 13] are typically used. Note that the information represented in each posting (each element of an inverted list) varies depending on the type of information retrieval system. For a boolean IRM full-text information retrieval system, the posting contains the document identifier and the position (as a byte offset or word offset from the beginning of the document) of the corresponding word. For a boolean IRM abstracts text information retrieval system, the posting contains the document identifier without a positional offset (since duplicate occurrences of a word in a document are not represented in these systems). For a vector IRM full-text or abstracts information retrieval systems the posting contains the document identifier and a weight. All of the above systems can be typed. In this case, the type system can be encoded by setting aside extra bits in each posting to indicate which fields the word appears in the document. Other methods of representing the type information are also used. As the information retrieval model becomes more complicated, more information is typically placed in each posting.

A related area of research is the compression of inverted indexes [29, 30]. The inverted index for a full-text information retrieval system is very large – typically on the same scale in size as the text. In fact, the original documents (minus punctuation) can be reconstructed from the inverted index. Thus, one interesting physical design issue is the impact of the compression ratio of the inverted index on response time. We return to this issue in Section 6.

3 Parallel Query Processing

Various distributed and parallel hardware architectures can be applied to the problem of information retrieval. A series of papers by Stanfill studies this problem for a Connection Machine. In reference [20], signature schemes are used. A companion paper by Stone [22] argues that inverted lists on a single processor are more efficient. In reference [21], inverted lists are used to support parallel query processing (in a fashion similar to that used by the system index organization that will discussed in Section 4). Finally, in reference [19], an improvement of the previous paper based on the physical organization of inverted lists is described. The technique essentially improves the alignment of processors to data.

An implementation of vector IRM full-text information retrieval is described in reference [1] for the POOMA machine. The POOMA machine is a 100-node, 2-d mesh communication network where each node has 16 MB of memory and a processor. One out of five nodes has an ethernet connection and one half of the nodes have a local disk. The implementation partitions the documents among the processors and builds a local inverted index of the partition. (This approach is similar to the host index organization of Section 4; however there are two processors per disk, as opposed to multiple disks per processor.) This paper cites a 2.098 second estimated query response time for a 191-term query on a database of 136,020 documents with a 20-node machine.

Some preliminary experimental results are reported in reference [3] for a 16 processor farm (Meiko Computing Surface). The vector IRM is used here and a signature scheme is used as the data structure. Unfortunately, the database has only 6,004 documents and the query workload only 35 queries.

The performance of some aspects of query and update processing of an implementation of a boolean IRM full-text information retrieval is discussed in reference [5] for a symmetric shared-memory multiprocessor (Sequent).

Reference [15] presents a discussion of the architecture issues in implementing the IBM STAIRS information retrieval system on a network of personal computers. This paper argues for the physical distribution of inverted lists across multiple machines when the size of a single database is larger than the storage capacity of a node on the network. This idea is essentially a special case of disk striping, where an object (in this case an inverted list) is partitioned across disks.

In the analysis of query processing, a query can be divided into three parts: parsing the query, matching the query against the database, and retrieving the documents in the answer. Parsing consumes few resources and is typically the same for all information retrieval systems. Retrieving of documents offers some interesting issues (such as placement of the documents) but again few resources are needed. Burkowski [2] examines the performance problem of the interaction between query processing and document retrieval and studies the issue of the physical organization of documents and indices. His paper models queries and documents analytically and simulates a collection of servers on a local-area network.

Schatz [17] describes the implementation of a distributed information retrieval system. Here, performance improvements come from changing the behavior of the interface to reduce network traffic between the client interface and the backend information retrieval system. These ideas are complementary our work. Three improvements are offered. First, summaries of documents (or the first page) are retrieved instead of entire documents. This scheme reduces the amount of network traffic to answer an initial query and shortens the time to present the first result of a query, but lengthens the time to present the entire answer. Second, "related" information such as document structure definitions are cached to speed up user navigation through a set of documents. Third, the contents of documents (as opposed to summaries) are prefetched while the user interface is idle.

Our own work [23, 24, 25] compares various options for partitioning an inverted list index across a shared-nothing multi-processor. (Reference [12] considers shared-everything multi-processors.) Simulated query loads are used in [24, 25], while [23] uses a trace-driven simulation.

4 Some Physical Design Choices

To illustrate more concretely the types of choices that are faced in partitioning index structures across machines, in this section we briefly describe the choices for an inverted-lists index, using the terminology of [25]. As stated earlier, this is the most popular type of index in commercial systems.

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

As discussed in Section 1, full-text document retrieval systems traditionally build inverted lists on disk to find documents quickly [8, 13]. For example, the inverted list for word *b* would be *b*: (D0,1), (D2,1), (D3,1). Each pair in the list is a posting that indicates an occurrence of the word (document id, position). To find documents containing word *b*, the system needs to retrieve only this list. To find documents containing both *a* and *b*, the system could retrieve the lists for *a* and *b* and intersect them. The position information in the list is used to answer queries involving distances, e.g., find documents

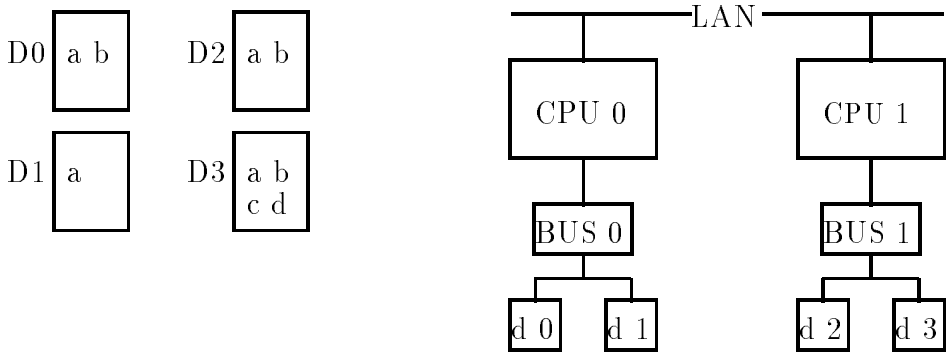

Figure 1: A example set of four documents and an example hardware configuration.

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

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

where a and b occur within so many positions of each other.

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

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

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

Clearly, many choices are available for physical index organization beyond those described here. We cannot consider all possible organizations. We do consider two other organizations where the inverted lists are divided among the disks and the I/O buses of the system. Our criteria for choosing these organizations focuses first on the optimization of queries as opposed to updates. Thus, we assume that the inverted lists on each machine are stored contiguously on disk. Second, we are interested in the interaction between the physical index organization and the allocation of resources (CPUs, disks,

I/O buses) of a shared-nothing distributed system. Our choices cover the entire range of interaction between indexes and hardware. In addition, we have studied issues such as striping and caching of the physical index organization with respect to a single host.

5 Query Processing

Given a physical index partition like the ones illustrated in the previous section, how does one process queries? To illustrate, let us focus on a particular type of query, a "boolean and" query. Such queries are of the form $a \wedge b \wedge c \ldots$, and find the documents containing all the listed words. The words appearing in a query are termed *keywords*. Given a query $a \wedge b \ldots$ the document retrieval system generates the *answer set* for the document identifiers of all the documents that match the query. A *match* is a document that contains the words appearing in the query.

Notice that boolean-and queries are the most primitive ones. For instance, a more complex search such as $(a \wedge b)$ OR $(c \wedge d)$ can be modeled as two simple and-queries whose answer sets are merged. A distance query "Find a and b occurring within x positions" can be modeled by the query $a \wedge b$ followed by comparing the positions of the occurrences. Thus, the query processing strategies for the more complex queries can be based on the strategies we will illustrate here for the simple boolean-and queries.

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

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

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

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

It is also important to notice that the query algorithms we have discussed can be optimized in a

Parameter	Base Value	Influence
Database scale	1.0	-359.6
Fraction of query words which are striped	0.0	278.4
Disk bandwidth (Mbit/sec)	10.4	112.7
Compression ratio	0.5	-67.4
Multiprogramming level (per host)	4	-48.1
CPU speed (MIPS)	20.0	47.7
Posting size (bits)	40.0	-44.5
Hosts	1	-27.9
Disks per I/O bus	4	25.4
I/O bus bandwidth (Mbit/sec)	24.0	11.2
Buffer overhead (ms)	4.0	-9.33
Disk buffer size (Kbyte)	32	9.12
LAN bandwidth (Mbit/s) (4 hosts)	100.0	2.33
I/O bus overhead (ms)	0.0	-1.96
Disk seek time (ms)	6.0	-1.93
Bytes per block	512	-0.81
Instructions per byte for a merge	40	0.0
Answer entry size (bytes)	4.0	0.0
Instructions per byte of decompression	40	0.0
Instruction count per query	500,000	0.0
Cache size (postings)	0	0.0
Instructions per byte of union operation	5	0.0
Subquery instruction count	100,000	0.0
Instructions per disk fetch	10,000	0.0
LAN overhead (ms)	0.1	0.0
LAN bandwidth (Mb/s)	100.0	0.0
Subquery length (bytes)	1024	0.0

Table 2: A ranking of the influence of simulation parameters on response time for the system index organization with Prefetch I query optimization.

variety of ways. To illustrate, let us describe one possible optimization for the system index organization. We call this optimization *Prefetch I*; it is a heuristic and in some cases it may not actually improve performance. (Other query optimization techniques have been studied in the literature.)

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

6 Experimental Parameters

In this section we summarize two studies we have performed to evaluate the index partition and query processing trade-offs. We believe they are representative of the types of analysis that needs to be performed to evaluate physical design alternatives for information retrieval. In particular, we focus on the experimental parameters used and their impact on response time. Our ranking of these parameters

gives an overview on the important areas to consider when designing an information retrieval system. In addition to the simulation work described here, a general interest in the performance of text document retrieval systems has led to a standardization effort for benchmarking of systems [4].

The first study [25] focused on full-text information retrieval. In full-text retrieval, the inverted index contains essentially the same information as the documents, since the position of each word in each document is recorded. Our inverted list model was based on experimental data, and our query model was based on a probabilistic equations. The second study [23] focused on abstracts text information retrieval where each electronic abstract is an abstract of a paper document. In this form of retrieval, the inverted index records only the occurrence of a word in an abstract, and not every occurrence. This dramatically reduces the size of the index with respect to full-text retrieval.

In general, our results indicate that the host index organization is a good choice, especially if long inverted lists are striped across disks. Long inverted lists are present in full-text information retrieval. Since the lists are long, the bottleneck is I/O performance. The host index organization uses system resources effectively and can lead to high query throughputs in many cases. When it does not perform the best, it is close to the best strategy.

For an application where only abstracts are indexed, the system organization (with the Prefetch I optimization) actually outperforms the host organization. The bottleneck for these systems is the network. This is because the inverted lists are much shorter, and can be easily moved across machines.

To study the impact of the experimental parameters on response time, we focus on the second study. Our inverted list model and query model were based on inverted lists of actual abstracts and traces of actual user queries from the Stanford University FOLIO information retrieval system. In both studies, query processing and hardware measurement where accomplished by using a sophisticated simulation containing over 28 parameters. Table 2 lists the parameters and the default values of each parameter. For each parameter in the table, a simulation experiment was run which linearly varied the values of the parameter. The simulation reflects the the architecture shown in Figure 1, as determined by the number of hosts, I/O buses and disks shown in the table. Full details of our experiments and our results are available in the references.

One way to succinctly show the parameters involved in the studies and their influence on performance is to "rank" the parameters by their (normalized) influence on performance. Here we only look at query response time as the performance metric. In particular, if a and b are the smallest and largest values measured for a parameter and x is the response time for a and y the response time for b, we compute $(y - x)/(a/b)$ as an estimate of the influence the parameter has on response time. Of course, this measure is only a rough indication of influence. The measure depends on the ranges of values over which a parameter is measured. It also assumes that response time is monotonic over the range of values chosen. We have inspected the data to insure that this last condition holds.

Table 2 shows the ranking of 28 parameters for the system index organization, as described in Section 4, with the Prefetch I query optimization, as described in Section 5. In previous work, the system index organization was shown to be the best overall choice for an index organization for abstracts text information retrieval. The positive or negative nature of the ranking is due to the positive or negative influence the parameter has on response time.

Database scale has the strongest influence – this parameter linearly scales the length of an inverted list and scales the lengths of all other objects in the system – such as the size of the answers to queries. With striping, a fraction of the inverted lists (in particular the longest ones) are striped across the disks within a computer system. This is a complementary technique to the list partitioning done by the basic index organization we have discussed, and can be very beneficial. Disk bandwidth is important due to the disk intensive nature of the computation. The compression ratio linearly scales the length of the inverted lists, but does scale any other parameter. The multiprogramming level is the number of simultaneous jobs which are run on each host. The relative CPU speed scales all computations which

compute the number of instructions needed to accomplish a task. The posting size is the number of bits needed to represent a posting. Hosts represents the number of processors in the system. When this parameter is increased, a copy of the processor is made. That is, if the parameter doubles, the number of I/O buses and disks in the entire system also doubles. In addition, the workload doubles, since the number of concurrent queries is allocated on a per host basis. Examining the parameters at the end of the table, we see that within the accuracy of the measurement, several parameters have no influence on response time. One surprising fact shows cache size as having no influence. In fact, caches have no influence on response time, but have a tremendous influence on throughput. Essentially, each query almost always has a cache miss. Thus, the response time of the query is dictated by the read from disk of the cache miss and thus the cache has little influence on response time. However, most queries have cache hits also, which dramatically improves throughput.

7 Conclusion

In this article, we have sampled issues in parallel information retrieval. As an introduction to the issues involved, we have discussed the literature in the area to introduce the various areas of research. We then focused on a specific example to illustrate the issued involved in distributed shared-nothing information retrieval. We discuss physical index organization and query optimization techniques. Then, to give the reader a sense of the important variables in the design of a system, we rank the various parameters in an experimental simulation study in terms of their influence on the response time of query processing.

References

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

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

[3] Janey K. Cringean, Roger England, Gordon A. Manson, and Peter Willett. Parallel text searching in serial files using a processor farm. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, pages 429–453, 1990.

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

[5] 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.

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

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

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

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

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

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

[12] 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.

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

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

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

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

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

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

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

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

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

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

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

[24] 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.

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

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

[27] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 1994.

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

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

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