NORTH-HOLLAND

Informatics and Computer Science

Large-Scale Information Retrieval with Latent Semantic Indexing

TODD A. LETSCHE

and

MICHAEL W. BERRY

Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996-1301, USA

ABSTRACT

As the amount of electronic information increases, traditional lexical (or Boolean) information retrieval techniques will become less useful. Large, heterogeneous collections will be difficult to search since the sheer volume of unranked documents returned in response to a query will overwhelm the user. Vector-space approaches to information retrieval, on the other hand, allow the user to search for concepts rather than specific words, and rank the results of the search according to their relative similarity to the query. One vector-space approach, Latent Semantic Indexing (LSI), has achieved up to 30% better retrieval performance than lexical searching techniques by employing a reduced-rank model of the term-document space. However, the original implementation of LSI lacked the execution efficiency required to make LSI useful for large data sets. A new implementation of LSI, LSI++, seeks to make LSI efficient, extensible, portable, and maintainable. The LSI++ Application Programming Interface (API) allows applications to immediately use LSI without knowing the implementation details of the underlying system. LSI++ supports both serial and distributed searching of large data sets, providing the same programming interface regardless of the implementation actually executing. In addition, a World Wide Web interface was created to allow simple, intuitive searching of document collections using LSI++. Timing results indicate that the serial implementation of LSI++ searches up to six times faster than the original implementation of LSI, while the parallel implementation searches nearly 180 times faster on large document collections.

1. INTRODUCTION

As storage becomes more plentiful and less expensive, the amount of information retained by businesses and organizations is likely to increase.

INFORMATION SCIENCES **100**, 105–137 (1997)

655 Avenue of the Americas, New York, NY 10010

0020-0255/97/$17.00
PII S0020-0255(97)00044-3

Searching that information and deriving useful facts, however, will become more cumbersome unless new techniques are developed to automatically partition the data into sets small enough for a human to understand.

Richard Wurman, in his book *Information Anxiety*, reports three startling facts:

- "Approximately 9600 different periodicals are published in the United States each year,"
- "The amount of available information doubles every five years; it will soon double every four...," and
- "The number of books in top libraries doubles every fourteen years..." [26].

Unfortunately, although the amount of information available is growing at an exponential rate, our ability to search that information and derive useful facts is diminishing. Traditional lexical (or Boolean) retrieval techniques, while sometimes valuable to experts trained to search collections from a specific discipline, often return too much information to the user. Other times, because the terms used in the query differ from the terms used in the document, valuable information is never found in the document collection.

Latent Semantic Indexing (LSI) [7], a vector-space approach to conceptual information retrieval, is useful in situations where traditional lexical information retrieval approaches fail. LSI estimates the semantic content of the documents in a collection, and uses that estimate to rank the documents in order of decreasing relevance to a user's query. Since the search is based on the concepts contained in the documents rather than the document's constituent terms, LSI can retrieve documents related to a user's query even when the query and the documents do not share any common terms. Also, since LSI ranks the documents according to their relevance to the user's query, the system helps the user decide which information may be more specific to the user's interests.

Although LSI is capable of achieving significant retrieval performance gains over standard lexical retrieval techniques (see [8]), the complexity of the LSI model often causes its execution efficiency to lag far behind the execution efficiency of the simpler, Boolean models, especially on large data sets. By carefully examining the LSI model and noting the various optimizations that can be applied to its underlying implementation, though, both the retrieval benefits of the LSI model and an execution efficiency near that of the Boolean retrieval techniques can be attained. Here, an efficient, extensible, maintainable, and portable implementation of the LSI model is presented, and a simple user interface, created with the new implementation of the LSI model, is explored. Using both the new imple-

mentation of the LSI model and its corresponding user interface, users can quickly search large data sets without understanding any details of the LSI model or implementation.

The following sections outline the development and use of an efficient, extensible, portable, and maintainable implementation of the Latent Semantic Indexing model. Section 2 explores the general ideas behind vector-space models for information retrieval and describes LSI in particular. Section 3 introduces LSI++, a C++ class library for searching with the LSI model, and examines how LSI++ (with a World Wide Web interface) can be used to build both serial and distributed search engines. Finally, Section 4 examines the execution efficiency of the search engines written with LSI++.

2. VECTOR-SPACE MODELS FOR INFORMATION RETRIEVAL

Research in information retrieval has followed several parallel, yet similar, developmental paths. Latent Semantic Indexing (LSI), because of the way it represents terms and documents in a term-document space, is considered a vector-space information retrieval model. In the following sections, general vector-space models as well as LSI are introduced.

2.1. INTRODUCTION TO VECTOR-SPACE MODELS

The vector-space models for information retrieval are just one subclass of retrieval techniques that have been studied in recent years. The taxonomy provided in [4] labels the class of techniques that resemble vector-space models "formal, feature-based, individual, partial match" retrieval techniques since they typically rely on an underlying, formal mathematical model for retrieval, model the documents as sets of terms that can be individually weighted and manipulated, perform queries by comparing the representation of the query to the representation of each document in the space, and can retrieve documents that do not necessarily contain one of the search terms. Although the vector-space techniques share common characteristics with other techniques in the information retrieval hierarchy, they all share a core set of similarities that justify their own class.

Vector-space models rely on the premise that the meaning of a document can be derived from the document's constituent terms. They represent documents as vectors of terms $d=(t_1,t_2,\ldots,t_n)$ where t_i $(1 \leqslant i \leqslant n)$ is a nonnegative value denoting the single or multiple occurrences of term i in document d. Thus, each unique term in the document collection

corresponds to a dimension in the space. Similarly, a query is represented as a vector $q=(\hat{t}_1,\hat{t}_2,\ldots,\hat{t}_m)$ where term $\hat{t}_i$ $(1\leqslant i\leqslant m)$ is a nonnegative value denoting the number of occurrences of $\hat{t}_i$ (or merely a 1 to signify the occurrence of term $\hat{t}_i$) in the query [4]. Both the document vectors and the query vector provide the locations of the objects in the term-document space. By computing the distance between the query and other objects in the space, objects with similar semantic content to the query presumably will be retrieved.

Vector-space models that do not attempt to collapse the dimensions of the space treat each term independently, essentially mimicking an inverted index [13]. By applying different similarity measures to compare queries to terms and documents, properties of the document collection can be emphasized or deemphasized. For example, the dot product (or inner product) similarity measure finds the Euclidean distance between the query and a term or document in the space. The cosine similarity measure, on the other hand, by computing the angle between the query and a term or document rather than the distance, deemphasizes the lengths of the vectors. In some cases, the directions of the vectors are a more reliable indication of the semantic similarities of the objects than the distance between the objects in the term-document space [13].

Vector-space models were developed to avoid many of the problems associated with lexical matching. In particular, since words often have multiple meanings (polysemy), it is difficult with lexical matching to differentiate between two documents that share a given word, but use it differently, without understanding the context in which the word was used. Also, since there are many ways to describe a given concept (synonomy), related documents may not use the same terminology to describe their shared concepts. A query using the terminology of one document will not retrieve the other related documents. In the worst case, a query using terminology different from that used by related documents in the collection may not retrieve any documents using lexical matching, even though the collection contains related documents [5].

Vector-space models, by placing terms, documents, and queries in a term-document space and computing similarities between the queries and the terms or documents, allow the results of a query to be ranked according to the similarity measure used. By basing their rankings on the Euclidean distance or the angle measure between the query and terms or documents in the space, vector-space models are able to automatically guide the user to documents that might be more conceptually similar and of greater use than other documents. Also, by representing terms and documents in the same space, such models often provide an elegant method of implementing relevance feedback [23]. Relevance feedback, by

allowing documents as well as terms to form the query, and using the terms in those documents to supplement the query, increases the length and precision of the query, helping the user to more accurately specify what he or she desires from the search.

Information retrieval models typically express the retrieval performance of the system in terms of two quantities: precision and recall. Precision is the ratio of the number of relevant documents retrieved by the system to the total number of documents retrieved. Recall is the ratio of the number of relevant documents retrieved for a query to the number of documents relevant to that query in the entire document collection. Both precision and recall are expressed as values between 0 and 1. An optimal retrieval system would provide precision and recall values of 1, although precision tends to decrease with greater recall in real-world systems [13].

2.2. LATENT SEMANTIC INDEXING

The Latent Semantic Indexing information retrieval model builds upon the prior research in information retrieval and, using the singular value decomposition (SVD) [16] to reduce the dimensions of the term-document space, attempts to solve the synonomy and polysemy (Section 2.1) problems that plague automatic information retrieval systems. LSI explicitly represents terms and documents in a rich, high-dimensional space, allowing the underlying ("latent"), semantic relationships between terms and documents to be exploited during searching.

LSI relies on the constituent terms of a document to suggest the document's semantic content. However, the LSI model views the terms in a document as somewhat unreliable indicators of the concepts contained in the document. It assumes that the variability of word choice partially obscures the semantic structure of the document. By reducing the dimensionality of the term-document space, the underlying, semantic relationships between documents are revealed, and much of the "noise" (differences in word usage, terms that do not help distinguish documents, etc.) is eliminated. LSI statistically analyzes the patterns of word usage across the entire document collection, placing documents with similar word usage patterns near each other in the term-document space, and allowing semantically related documents to be near each other even though they may not share terms [8].

LSI differs from previous attempts at using reduced-space models for information retrieval in several ways. Most notably, LSI represents documents in a high-dimensional space. Koll [18], for instance, used only seven dimensions to represent his semantic space. Secondly, both terms and

documents are explicitly represented in the same space. Thirdly, unlike Borko and Bernick [6], no attempt is made to interpret the meaning of each dimension. Each dimension is merely assumed to represent one or more semantic relationships in the term-document space. Finally, because of limits imposed mostly by the computational demands of vector-space approaches to information retrieval, previous attempts focused on relatively small document collections. LSI is able to represent and manipulate large data sets, making it viable for real-world applications [7].

Compared to other information retrieval techniques, LSI performs surprisingly well. In one test, Dumais [8] found that LSI provided 30% more related documents than standard word-based retrieval techniques when searching the standard MED collection. Over five standard document collections, the same study indicated that LSI performed an average of 20% better than lexical retrieval techniques. In addition, LSI is fully automatic, easy to use, requires no complex expressions or syntax to represent queries, and can be used to index extremely large text corpora [9, 10]. Because terms and documents are explicitly represented in the space, relevance feedback [23] can be seamlessly integrated with the LSI model, providing even better overall retrieval performance.

The following sections briefly describe the LSI model. A computational example of the model can be found in [5].

2.2.1. Term-Document Representation

In the LSI model, terms and documents are represented by an $m \times n$ incidence matrix A. Each of the m unique terms in the document collection is assigned a row in the matrix, while each of the n documents in the collection is assigned a column in the matrix. A nonzero element a_{ij}, where

$$A = [a_{ij}],$$

indicates not only that term i occurs in document j, but also the number of times the term appears in that document. Since the number of terms in a given document is typically far less than the number of terms in the entire document collection, A is usually very sparse [5].

2.2.2. Weighting

LSI typically uses both a local and global weighting scheme to increase or decrease the relative importance of terms within documents and across

the entire document collection, respectively. The product of the local and global weighting functions is applied to each nonzero element of A,

$$a_{ij} = L(i,j) * G(i),$$

where $L(i,j)$ is the local weighting function for term i in document j and $G(i)$ is the global weighting function for term i. Among the popular local and global weighting functions tried with LSI (see [8, p. 233]), Dumais found that the log-entropy weighting scheme provided a 40% advantage over raw term frequency on several standard document test collections [8].

2.2.3. Computing the SVD

Once the $m \times n$ matrix A has been created and properly weighted, a rank-k approximation ($k \ll \min(m,n)$) to A, A_k, is computed using an orthogonal decomposition known as the singular value decomposition (SVD) [16]. The SVD of the matrix A is defined as the product of three matrices,

$$A = U\Sigma V^T,$$

where the columns of U and V are the left and right singular vectors, respectively, corresponding to the monotonically decreasing (in value) diagonal elements of Σ which are called the singular values of the matrix A. As illustrated in Figure 1, the first k columns of the U and V matrices and the first (largest) k singular values of A are used to construct a rank-k approximation to A via $A_k = U_k \Sigma_k V_k^T$. The columns of U and V are orthogonal, such that $U^T U = V^T V = I_r$, where r is the rank of the matrix A. A theorem due to Eckart and Young [17] suggests that A_k, constructed from the k-largest singular triplets[1] of A, is the closest rank-k approximation (in the least squares sense) to A [5].

With regard to LSI, A_k is the closest k-dimensional approximation to the original term-document space represented by the incidence matrix A. As stated previously, by reducing the dimensionality of A, much of the

[1] The triple $\{U_i, \sigma_i V_i\}$, where $\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_k)$, is called the ith singular triplet. U_i and V_i are the left and right singular vectors, respectively, corresponding to the ith largest singular value σ_i of the matrix A.

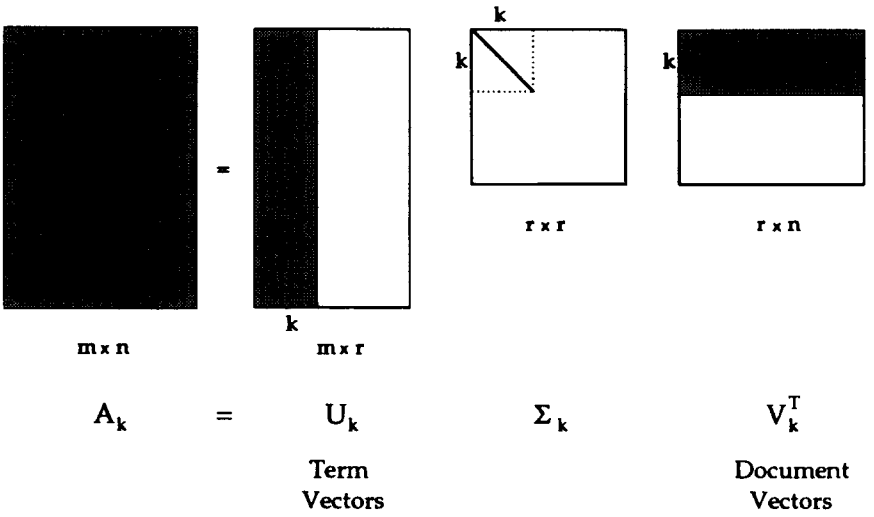

Fig. 1. A pictorial representation of the SVD. The shaded areas of U and V, as well as the diagonal line in Σ, represent A_k, the reduced representation of the original term-document matrix A.

"noise" that causes poor retrieval performance is thought to be eliminated. Thus, although a high-dimensional representation appears to be required for good retrieval performance, care must be taken to not reconstruct A. If A is nearly reconstructed, the noise caused by variability of word choice and terms that span or nearly span the document collection will not be eliminated, resulting in poor retrieval performance [5].

In LSI, the left and right singular vectors specify the locations of the terms and documents, respectively, in the reduced term-document space. The singular values are often used to scale the term and document vectors, allowing clusters of terms and documents to be more readily identified. Within the reduced space, semantically related terms and documents presumably lie near each other since the SVD attempts to derive the underlying, semantic structure of the term-document space [5].

2.2.4. Query Projection and Matching

In the LSI model, queries are formed into *pseudo-documents* that specify the location of the query in the reduced term-document space. Given q, a vector whose nonzero elements contain the weighted (using the

same local and global weighting schemes applied to the document collection being searched; see Section 2.2.2) term-frequency counts of the terms that appear in the query, the pseudo-document $\hat{q}$ can be represented by

$$\hat{q} = q^T U_k \Sigma_k^{-1}.$$

Thus, the pseudo-document consists of the sum of the term vectors ($q^T U_k$) corresponding to the terms specified in the query scaled by the inverse of the singular values (Σ_k^{-1}). The singular values are used to individually weight each dimension of the term-document space [5].

Once the query is projected into the term-document space, one of several similarity measures can be applied to compare the position of the pseudo-document to the positions of the terms or documents in the reduced term-document space. One popular similarity measure, the cosine similarity measure, is often used because, by only finding the angle between the pseudo-document and the terms or documents in the reduced space, the lengths of the documents, which can affect the distance between the pseudo-document and the documents in the space, are normalized. Once the similarities between the pseudo-document and all the terms and documents in the space have been computed, the terms or documents are ranked according to the results of the similarity measure, and the highest ranking terms or documents, or all the terms and documents exceeding some threshold value, are returned to the user [5]

2.2.5. Relevance Feedback

Relevance feedback is an effective method for iteratively improving a query without increasing the computational requirements to perform the query. Relevance feedback uses the terms contained in relevant documents to supplement and enrich the user's initial keyword query, allowing greater retrieval performance (in terms of precision and recall; see Section 2.1). Since LSI explicitly represents both terms and documents in the same space, a relevance feedback query is constructed in essentially the same way as a regular query.

Given two vectors:

q, from the previous section, specifying the terms in the query, and
d, a vector whose elements specify the documents in the query,

the pseudo-document representing the relevance feedback query is given by

$$\hat{q} = q^T U_k \Sigma_k^{-1} + d^T V_k.$$

Then, by matching the pseudo-document against each term or document vector (as described in the previous section) and sorting the results, the highest ranking terms or documents (or all those that exceed some threshold value) can be returned to the user.

3. THE DESIGN AND IMPLEMENTATION OF A LATENT SEMANTIC INDEXING SERVER

The LSI model described in the previous section does not specify a particular implementation strategy. The abstract model can be broken into two phases: a preprocessing phase where the term-document space is built, and the query phase where queries are projected into that space and a similarity measure is applied to find terms and documents near the query. Although both phases of the LSI model are equally important, the preprocessing phase is a one-time cost. It is independent of the number of queries posed to the database in the second phase. Since the user must interact with the query phase, the functionality and execution efficiency of that phase is very important. Here, adequate software to perform the preprocessing phase is assumed to exist. Instead, the query phase is considered.

3.1. FUNCTIONAL SPECIFICATION

Minimally, any implementation of Latent Semantic Indexing must take as input a six-tuple $q = \langle b, r, n, t, d, f \rangle$, where

b is the number of the document collection to search,
r indicates whether the query engine should return a ranked list of terms, documents, or both terms and documents,
n is the number of terms, documents, or both to return to the user,
t is a set of words (terms) used to describe the query,
d is a set of documents whose corresponding text is used to refine the query through relevance feedback (Section 2.2.5), and
f is the current number of factors used to represent the query, terms, and documents.

On any particular query, b, r, n, f, and at least one of t and d are required. However, t and d cannot both be empty.

Given q, the LSI server must perform the following tasks:

1. *Eliminate from t those terms considered common in the document collection.* Typically, when a document collection is assembled, the fre-

quency of each word in the collection is analyzed, and words that occur most often in the collection are not indexed, but instead are added to the stoplist of the collection. A stoplist for a given document collection is a list of words generally not worth indexing in the collection [13]. Words that occur frequently in the collection are not valuable in the index since they make it difficult to distinguish documents from one another, increase the amount of space required to store the index, and increase the time required to process a query. When a search is performed, terms that appear on the stoplist are removed from the query since they were not indexed in the document collection.

2. *Eliminate from t the terms that do not occur in the document collection.* Words that do not occur in the document collection cannot be used to project the query into the term-document space.

3. *Compose the pseudo-document that represents the query.* Each term in the document collection (except those terms considered too common to index) occupies a particular position in the term-document space. The position of a particular term in the space is given by a vector, called its term vector. Similarly, each document in the collection is represented by a document vector that specifies the document's position in the term-document space. In the LSI model, a document is placed at the centroid of the terms that comprise the document. The pseudo-document that represents a query consists of the weighted sums of the term vectors associated with the terms in t combined with the sums of the document vectors associated with the documents in d.

4. *Project the pseudo-document into the term-document space and find the n highest ranking terms, documents, or both.* Although several different similarity measures can be used to determine the highest ranking terms and documents for a given query, the cosine similarity measure is most often used to find terms and documents near the pseudo-document in the term-document space. The cosine similarity measure computes the angle between the vector representing the pseudo-document and each term or document vector. The higher the cosine between the pseudo-document and a term or document, the closer the term or document is to the pseudo-document, regardless of Euclidean distance. When computing the cosine, only the first f elements of the term and document vectors are used. This allows the user to adjust whether he or she desires a more conceptual match (which corresponds to smaller values of f) or a more literal match (corresponding to larger values of f). Then, the results of each cosine are sorted, and the top n ranking terms or documents are returned to the user.

Typically, the results returned to the user include the similarity of the pseudo-document to each of the top n ranked terms or documents and the corresponding terms or document titles. The flowchart in Figure 2 summarizes the process.

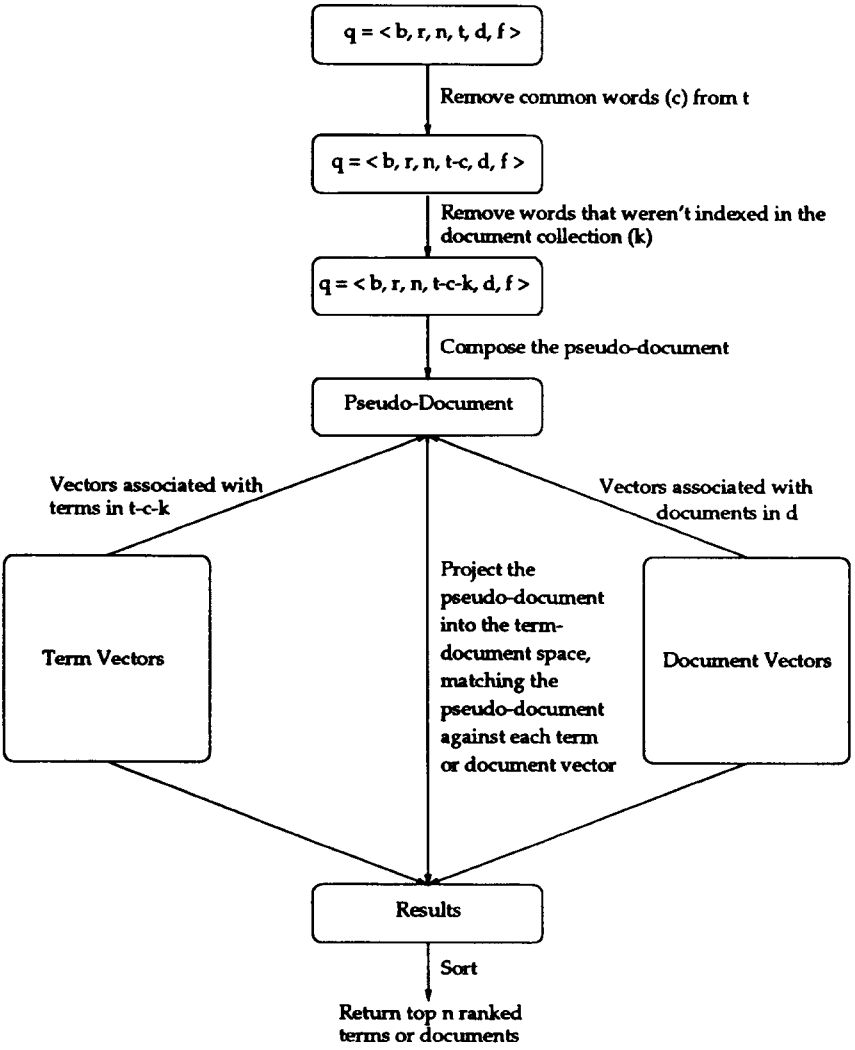

Fig. 2. The query phase of the LSI model.

3.2. *ORIGINAL IMPLEMENTATION*

The initial implementation of LSI [7] primarily was used by Bellcore as a research vehicle to test the LSI model and to discover any modifications that might have improved the model. The design of the initial implementation was weighted toward maximum flexibility rather than optimal execution efficiency.

While the initial implementation served as an excellent research tool, it has several shortcomings as a production-level system. Although most of the code that performs computation was written in the C programming language, UNIX[2] shell scripting languages (Bourne shell and C shell) as well as the UNIX utilities *awk* and *sed* were used for text manipulation. The computational portions of the code were linked together with UNIX pipes, often with an *awk* or *sed* script sandwiched between them to convert the output of one computation into the form required for the next computation.

Besides the maintenance issues that arise when dealing with a single system consisting of so many different languages, efficiency issues become apparent for large document collections. For example, the time required to spawn several processes and establish pipes between them is significantly larger than the time required to begin executing a single, compiled program. In addition, the scripting languages as well as *awk* and *sed* are interpreted, resulting in even slower execution. Converting data formats between phases of execution and writing temporary files adds to the inefficiency.

A more subtle issue than either maintenance or efficiency is portability. While most platforms currently offer a means to compile C code, operating systems other than UNIX may not offer equivalent scripting languages, and may not provide utilities like *awk* and *sed*. Most importantly, though, other operating systems may not offer pipes as a way to communicate information among separately executing processes.

3.3. *A NEW IMPLEMENTATION*

A new implementation of LSI, LSI++, seeks to solve the maintenance, portability, and efficiency problems of the Bellcore implementation while

[2] UNIX is a registered trademark of AT&T.

providing a great deal of flexibility to both the programmer and the user of an LSI system that incorporates LSI++. LSI++ provides an Application Programming Interface (API) that allows the programmer to create a variety of user interfaces that do not depend on the underlying implementation details of the LSI system. The programming interface allows LSI++ to be used in a variety of applications without modification, and also allows LSI++ to take advantage of special features of the machine architecture (such as parallelism) without affecting the application-level code that uses it.

3.3.1. Serial Implementation

A serial implementation of the query phase for large document collections requires careful thought and planning. Since a single, multiuser machine has finite memory and swap space, projecting a query into the term-document space must use as little memory as possible while executing as efficiently as possible. Any operation that depends on the size of the document collection must be performed incrementally, without loading large sections of the collection into memory. Objects that are no longer needed must be reused or discarded to keep memory usage as low as possible.

The time required to perform the query phase increases in proportion to the number of terms and documents in the database (see [20]). The query phase consists of a small amount of bookkeeping (adding terms and documents to the query, keeping track of the terms actually used in the query, etc.) and a large amount of computation as the query is matched against each term or document vector. In practice, the bookkeeping costs are fairly low since they depend on the size of the query instead of the number of terms and documents in the collection. Typically, matching costs far exceed the bookkeeping costs.

An efficient serial implementation of the query phase should load a term or document vector as it is needed, perform all the computation that involves that vector, and then discard the vector, freeing memory for the next vector. Since only one vector is kept in memory at a time, the amount of memory required does not increase with the number of terms and documents in the database. Also, this pipeline is fairly time-efficient since the algorithm accesses contiguous vectors, and some I/O buffering is performed at the operating system level.

3.3.2. Distributed Implementation

Since a parallel computer or network of workstations presumably has more total memory than a single processor, the amount of memory consumed during matching is less of a concern. In fact, if the machine can be used as a back-end processor (a continuously running server that receives queries from a client, processes them, and sends the results back to the client) and has a large enough collective memory, each processor can cache term and document vectors in main memory, saving the cost of loading the vectors from secondary storage for each query. Depending on the machine, the cost of starting a process on a parallel computer or network of workstations might be very high. Because of this, we consider only the distributed implementation executing as a backend processor.

Computing the similarity between the pseudo-document and each term or document is a highly parallel operation. When applying the similarity measure, the scaling and multiplication of one term or document vector are completely independent of the operations performed on another term or document vector. Even a simple blocked data distribution [20] provides a high degree of parallelism while balancing the computational load among the processors. If each processor loads its respective blocks of term and document vectors upon initialization, the costs associated with loading term and document vectors from secondary storage will only have to be paid once rather than each time a query is performed.

Processing a query in parallel is very similar to processing a query on a serial machine. One processor, the root processor, is responsible for receiving a query from the client, building the pseudo-document, and performing the bookkeeping tasks associated with the query. When the root processor has finished building the pseudo-document, it broadcasts it to the other processors. It also must broadcast the number of factors to use, whether to return terms, documents, or both, and the number of terms or documents to return. Each processor then computes the similarity between the pseudo-document and each local term or document vector, recording the results. At the end of computation, a global sort returns the ranked similarity measures to the root processor. The root processor then returns the results to the client.

For the price of a few broadcasts before computation begins and a single global merge sort after computation ends, a query can be processed very quickly without accessing secondary storage or causing network contention. In addition, since there is very little difference between the serial algorithm and the parallel algorithm, much of the code from the serial implementation can be used by the distributed implementation.

3.3.3. C++ Classes

LSI++ is written in C++, a language that provides the ability to encapsulate data and functions while still providing the efficiency and maintainability that are essential to a large software system.

LSI++ consists of objects that contain both data and functions to manipulate the data. Figure 3 shows the inheritance relationship between the LSI++ classes. Most of the classes are independent, but a few classes share similar characteristics. The *lsiSerial* and *lsiParallel* classes provide serial and distributed implementations of the LSI++ API. The common characteristics of each class are contained in their base class, the *lsi* class. The *termVector*, *docVector*, and *singValues* classes provide the term vectors, document vectors, and singular values that are used to project the pseudo-document into the term-document space. They are all derived from the *singTriplets* class, which provides caching facilities and the ability to retrieve specific vectors from secondary storage.

The *lsi* class and its derived classes represent the Application Programming Interface for the LSI++ search engine. It serves as a base class from which more specific, architecture-dependent classes are derived. It

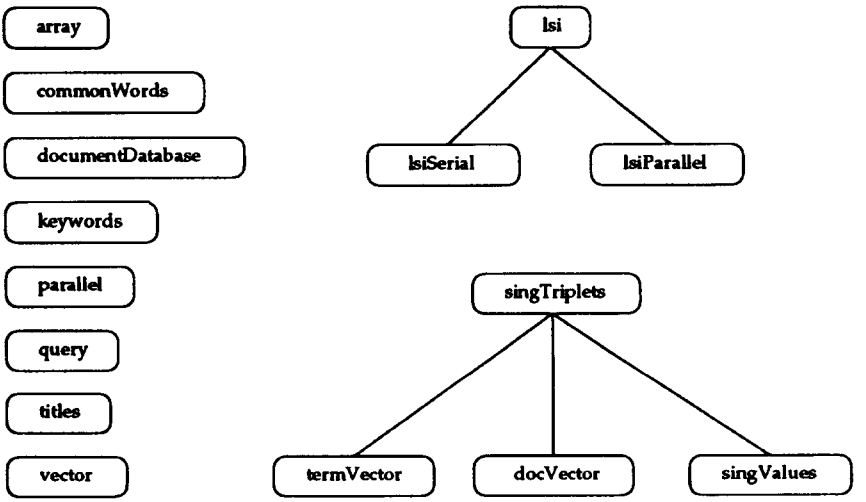

Fig. 3. The LSI++ inheritance hierarchy. The classes on the left side of the diagram are independent, while the *lsiSerial* and *lsiParallel* classes are derived from the *lsi class*. The *termVector*, *docVector*, and *singValues* classes are derived from the *singTriplets* class.

allows applications to compose a query, perform the search, and set various attributes that influence the search (for example, the number of terms or documents to return, the number of factors to use in the query, etc.). In addition, it stores the terms and documents that were used to compose the query, the terms in the query that remain after the common words and words not in the document collection are removed, and the total number of terms and documents in the collection. By accessing only the *lsi* class and its derived classes, the application programmer is able to pose a query to the search engine and receive the results of the query without knowing the underlying LSI algorithms or implementations.

Both the *lsiSerial* class and the *lsiParallel* class are derived from the *lsi* base class. Both classes provide the same basic services, and both support the API described in the *lsi* base class. The *lsiSerial* class is designed for applications that will execute on a single machine. To minimize memory requirements, the term vectors, document vectors, and singular values are incrementally loaded from secondary storage as they are needed. The *lsiParallel* class is designed for applications that require exceptionally fast query matching. It can use any number of processing elements, and it is written to use the Message Passing Interface (MPI) [24], making it highly portable. Since the time required to start a parallel application tends to be large [20], an application using the *lsiParallel* class will probably be started as a continuously running back-end server, with one or more clients passing queries to it and receiving the results.

The GNU version of C++, g++, includes an excellent library [14] of low-level classes to aid the programmer. The library, called libg++, contains classes that allow easy and intuitive string manipulation (with regular expression matching and extraction), input/output, and data storage (such as sets, bags, and associative arrays), among other data structures. LSI++ uses libg++ extensively for low-level data manipulation and storage, although it is not dependent on the features of libg++. Any library implementing basic data structures will satisfy the demands of LSI++. See [20] for details on the design and use of all the classes listed in Figure 3.

3.4. A CLIENT–SERVER INTERFACE

The LSI++ Application Programming Interface allows any number of applications and user interfaces to be constructed with a minimal knowledge of the internal details of the LSI system. Here, a World Wide Web (WWW) interface for LSI, based on LSI++, is described. A World Wide Web interface was selected as the first application of LSI++ because

interfaces created for the World Wide Web are portable and simple to create and maintain, eliminating much of the design, implementation, and maintenance time required for a stand-alone GUI.

3.4.1. The LSI WWW Interface

The original WWW interface for LSI was devised by Susan Dumais and Loren Shih at Bellcore. The prototype interface was designed to work with the original Bellcore implementation of LSI, making it obsolete when LSI++ was completed. The interface has been completely reworked to add new features, to make it easier to use, and to hide the details of LSI from the user as much as possible. The new LSI WWW interface uses the LSI++ Application Programming Interface to set the parameters for a query, process the query, and formulate the results.

The LSI WWW interface seeks to provide a useful, flexible interface to LSI without overwhelming the user with an abundance of esoteric options. It allows the user to:

- select a document collection ("book") to search,
- enter a series of keywords to form a query,
- choose the number of factors to use (controlling the precision of query matching) when performing the search,
- select whether to return terms, documents, or both terms and documents,
- and elect to return the 10, 20, 30, 40, or 50 highest ranked objects (according to the cosine values).

In addition, once an initial query has been submitted, any objects returned by the search can be selected to become part of the next query, allowing relevance feedback (Section 2.2.5) to be seamlessly integrated with the rest of the interface.

Before performing a search, a user first must choose the book to search. The *LSI:Book* form allows the user to select a document collection to search. Once the user selects a document collection, all queries are performed in the context of that document collection until the user explicitly changes the book being searched. When the *LSI:Book* form is submitted to the server, the second form, the *LSI:Query form*, is returned.

The *LSI:Query* form actually consists of two similar, yet separate forms. The initial *LSI:Query*, form (see [20]) allows the user to enter a query, adjust the number of factors, and select the number and type of objects to be returned. The user is also allowed to change to a different document

collection, taking him or her back to the *LSI:Book* form, and to create a new query, which allows the user to start a fresh query in the same document collection.

The subsequent *LSI:Query* form, which is returned after the user enters his or her initial query, includes the initial *LSI:Query* form, but also displays the results of the previous search (see [20]). To refine the query, the user can add terms and documents returned as the result of the prior search, add new keywords, and adjust the parameters of the search.

3.4.2. Implementation

Both the *LSI:Book* and the *LSI:Query* HTML documents are created by programs executing under the Common Gateway Interface (CGI) [21]. In this section, the creation of *LSI:Query* HTML documents is considered, and the interactions between their generating programs and the LSI++ Application Programming Interface is discussed. See [20] for details on the creation of *LSI:Book* document and C++ classes used to implement the interface.

The *LSI:Query* form can be created in several different ways, depending on the configuration. In the most straightforward configuration, *LSI:Query* is created by an LSI search engine running on the HTTP server itself. The HTTP server starts the search engine and passes the contents of the form to the search engine with CGI. The search engine processes the query and passes the resulting *LSI:Query* HTML document to the server, who then passes it to the browser (Figure 4).

If the HTTP server is not dedicated specifically to LSI (for example, the HTTP server handles all the WWW requests for an entire department), is expected to handle a large number of LSI queries, or is installed on a slow machine, the querying process can be offloaded to one or more continuously running back-end processors. The HTTP server, acting as a front end, receives an LSI search request from a browser and passes it, via CGI, to a small client program, *lsiRemote*, who is then responsible for connecting to the back end for that book, sending the query, and receiving the resulting *LSI:Query* document. As *lsiRemote* receives the resulting *LSI:Query* form, it passes it to the HTTP server, who then passes it to the browser (Figure 5).

4. RESULTS AND CONCLUSIONS

Both retrieval effectiveness (how reliably a system returns relevant documents) and execution efficiency (the average time required to search

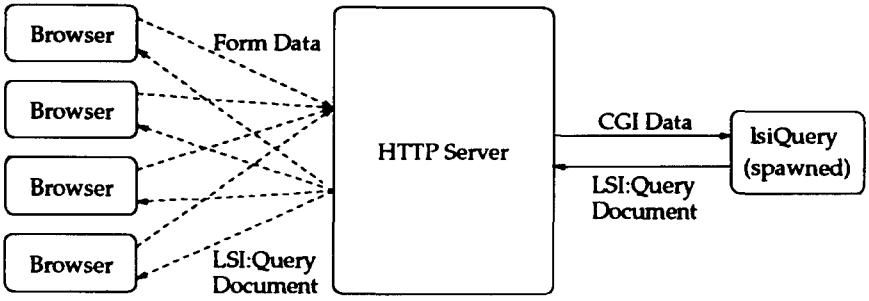

Fig. 4. *LsiQuery* executing on the HTTP server. The HTTP server spawns *lsiQuery* on the same machine as the server, using CCI to pass the HTML form input to *lsiQuery*. The resulting *LSI:Query* document, created by *lsiQuery*, is relayed to the browser by the HTTP server. The solid lines between the HTTP server and *lsiQuery* are used to show that *lsiQuery* executes locally. The dashed lines between the browsers and the HTTP server signify that the browsers typically make a remote connection to contact the HTTP server.

a document collection) of an interactive information retrieval system are very important when discussing the overall usefulness of the system [13]. If a system consistently returns large amounts of irrelevant information, or the user must wait a long time for a search to be performed, the system is less useful than one that quickly returns a higher percentage of relevant information. The retrieval effectiveness of LSI was discussed in Section 2.2. Since LSI++ merely implements the LSI model, its retrieval performance is governed by the effectiveness of the model. However, the execution efficiency is directly influenced by its implementation. LSI++

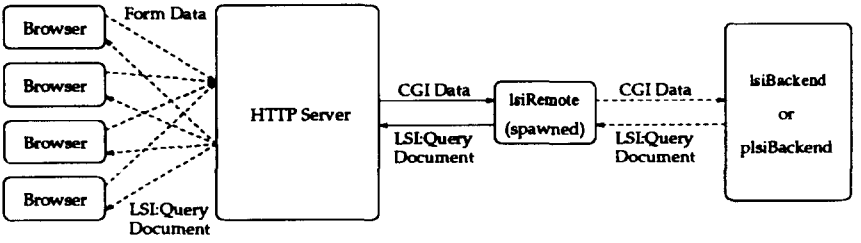

Fig. 5. *LsiBackend* or *plsiBackend* (a parallel back-end server) executing on a remote machine. The HTTP server spawns *lsiRemote*. *LsiRemote* gathers the CGI data and passes them to the back-end server. After the back-end server processes the query, the resulting *LSI:Query* document is sent to the browser through *lsiRemote* and the HTTP server. The dashed lines represent a connection to a remote machine.

has proven to be an efficient implementation of the LSI model described in Section 2.2. The following sections describe the performance of both the serial and distributed implementations of LSI++, and compare the performance of LSI++ to the performance of the original Bellcore implementation of LSI.

4.1. COMPUTING ENVIRONMENT

All performance timings were obtained on a homogeneous collection of 24 dedicated 70 MHz Sun SPARCstation 5 computers, each with 97,968K of main memory. Each machine was connected to both a 10 and a 100 Mb/s Ethernet network. The 100 Mb/s network, shared only by the 24 machines, was used by the distributed implementation of LSI++ to communicate between the nodes of the parallel machine. Since the disks containing the document collections were not local to the 24-machine cluster, all file access occurred over the 10 Mb/s network. In addition, communication between lsiRemote and both lsiBackend and plsiBackend took place over the 10 Mb/s Ethernet network.

4.2. METHODOLOGY

Four LSI search engines were tested to determine their relative execution efficiency while processing queries. Each search engine received its input from CGI and wrote an HTML document as output. The LSI search engines tested included:

lsiFinder—lsiFinder is written in the Perl programming language [25], and has approximately the same functionality as the original LSI WWW interface written by Loren Shih at Bellcore in 1994. Given a query, *lsiFinder* calls *mlsisearch* (the original Bellcore implementation of LSI) with the appropriate parameters. The output of *mlsisearch* is then processed by *lsiFinder* to produce an HTML page. Since *lsiFinder* uses the original Bellcore implementation of LSI to perform the search, *lsiFinder* is used as the base case to which the LSI++ code is compared.

lsiQuery—lsiQuery is a serial search engine based on LSI++. *lsiQuery* loads the terms and documents incrementally from disk as they are needed. It receives the query via CGI, performs the search using LSI++, and writes the results as an HTML document.

lsiBackend—lsiBackend is a continuously running, serial back-end server that receives a query from *lsiRemote*, performs the search using LSI++,

and passes the resulting HTML document back to *lsiRemote*. Like *lsiQuery*, *lsiBackend* incrementally loads the term and document vectors from disk.

plsiBackend—*plsiBackend* is a continuously running, distributed backend server. Like *lsiBackend*, *plsiBackend* receives its input from *lsiRemote*, performs the search using LSI++, and passes the resulting HTML document back to *lsiRemote*.

To measure the performance of *lsiFinder*, *lsiQuery*, *lsiBackend*, and *plsiBackend*, 25 random queries were selected for each of four document collections (see Table 1 for a description of each document collection). Each query consisted of one–five terms known to be in the document collection and zero–four terms not in the document collection. In addition, approximately half of the queries were relevance feedback queries, with one–five documents added to the query. Each set of 25 queries was posed

TABLE 1

The Document Collections Used during Performance Testing to Determine the Execution Efficiency of LSI++. The Usenet News Archive Document Collection Is a Compilation of USENET Newsgroup Postings Taken from a Variety of Newsgroups between February 2, 1996 and February 8, 1996

Document collection	Abbrev.	Number of terms	Number of documents	Number of factors
PVM: Parallel Virtual Machine [15]	PVM	2547	170	55
All of the following combined: PVM: Parallel Virtual Machine [15] LAPACK Users' Guide Release 2.0 [2] MPI: The Complete Reference [24] Templates for the Solution of Linear Systems [3] Parallel Computing Works! [12]	PLMTP	9842	1154	122
The Concise Columbia Encyclopedia [1]	CCE	31,110	15,486	104
Usenet News Archives	USENET	534,493	100,000	100

TABLE 2
A Summary of the 25 Random Queries Selected for Each Document Collection

Document Collection	Terms Not in Collection			Terms in Collection			Documents		
	Min.	Max.	Ave.	Min.	Max.	Ave.	Min.	Max.	Ave.
PVM	0	4	2.2	1	5	2.7	0	5	2.0
PLMTP	0	4	2.3	1	5	3.0	0	5	2.0
CCE	0	4	2.2	1	5	3.0	0	5	1.4
USENET	0	4	2.0	1	5	2.9	0	5	1.8

to the search engines twice, once to find related terms and again to find related documents. Table 2 summarizes the queries selected for each document collection.

The total wall-clock time for each search was recorded by the search engine. Each search engine used the *gettimeofday( )* system call to record the time the search began and the time the search finished. The total wall-clock time was found by subtracting the starting time from the ending time. Since Perl lacks adequate timers, *lsiFinder* determined the starting and ending times by calling an external program that reported the results of a call to *gettimeofday( )*.

Due to the configuration of the machines on which the performance timings were taken, an HTTP server was not available to pass queries to the search engines. In order to perform the tests, the HTTP server was simulated by setting the environment variables required by CGI and passing the queries to each search engine as a CGI-encoded string. Thus, the time required by the HTTP server to receive a query from a remote browser and spawn either *lsiQuery* or *lsiRemote* is not included in the timings given in the following sections.

4.3. SERIAL RESULTS

Both serial LSI++ search engines, *lsiQuery* and *lsiBackend*, fared well relative to *lsiFinder*, the search engine that uses the original Bellcore implementation of LSI. *LsiQuery*, which is designed to execute directly on the HTTP server, was between five and ten times faster than *lsiFinder* on the four document collections tested. *LsiBackend*, designed to execute on a remote machine and communicate with the HTTP server through *lsiRemote*, showed similar performance gains over *lsiFinder*.

4.3.1. Timing on the Network

Although the timings were taken on dedicated machines, the network was necessarily shared by many users when the timings were taken. In particular, since the timings were taken during the early morning hours, each search engine had to compete with both incremental and full backups of disks occurring over the network, as well as a daily software distribution to each machine. Even though the network shared by the dedicated machines was relatively isolated, the term and document vectors for each document collection were stored on disks outside the cluster of dedicated machines. *LsiQuery*, *lsiBackend*, and *lsiFinder* all load the term and document vectors from disk into memory as they are needed, making all three of them dependent on the speed of the network. Since little could be done to prevent other users from accessing the network while the timings were being taken, the network was monitored during testing to ensure that the network stayed at approximately the same load while each search engine was running. However, since accurately monitoring the network was difficult, the times reflect natural variances as the network became more and less congested.

4.3.2. LsiQuery versus LsiFinder

Figure 6 reports the average number of seconds required to search for both terms and documents in each of the four document collections using *lsiQuery* and *lsiFinder*. Because of memory constraints, *lsiFinder* was unable to search for terms in the USENET document collection.

Although the logarithmic scale of the graph conceals much of the true time differences between *lsiQuery* and *lsiFinder*, *lsiQuery* outperformed *lsiFinder* on all the document collections tested, especially the larger collections. Significant time differences (as much as an average of 167.7 s when searching for related documents in the USENET collection) were observed for the CCE and USENET collections. Greater time differences are expected for even larger document collections.

Empirical data suggest that as much as 84% of the total time required by *lsiFinder* to process a query was spent in one program called by *mlsisearch*, *syn. Syn*, which computes the cosine between the query vector and the term and document vectors, does not free or reuse memory that has already been allocated, but instead allocates memory for each vector as it is loaded. Not only does the continual allocation of memory slow *syn's* execution, it also prevents large document collections (such as the USENET document collection) from being searched.

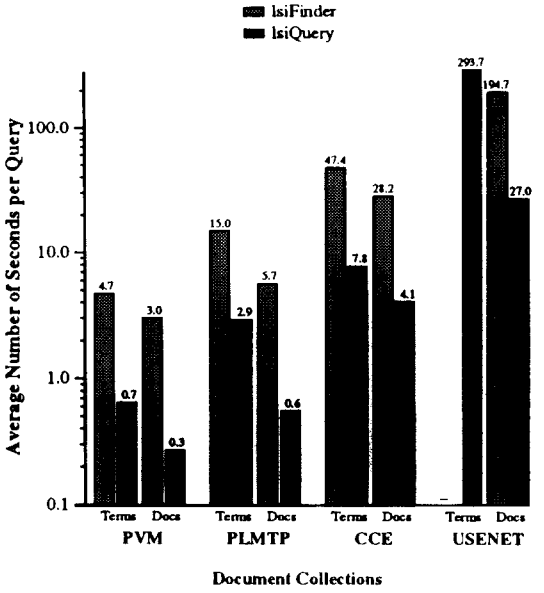

Fig. 6. *lsiQuery* versus *lsiFinder*. The average number of seconds required by *lsiFinder* and *lsiQuery* to search for both terms and documents in each of the four document collections.

4.3.3. LsiBackend versus LsiFinder

The execution efficiency of *lsiBackend* was very similar to the efficiency of *lsiQuery* (see Figure 7). Like *lsiQuery*, *lsiBackend* exceeded the performance of *lsiFinder* while searching each of the document collections. Although *lsiBackend* appears slightly faster than *lsiQuery* for the USENET document collection, the difference is probably due to network congestion, as discussed in Section 4.3.1.

4.4. DISTRIBUTED RESULTS

PlsiBackend, the continuously running, distributed LSI++ back-end server, also performed well during the tests. Although the time to search each document collection was improved by using several processors rather than a single processor, the larger document collections like CCE and USENET benefited the most from parallelism. Also, the timings demonstrated that adding more processors to the smaller collections generally did

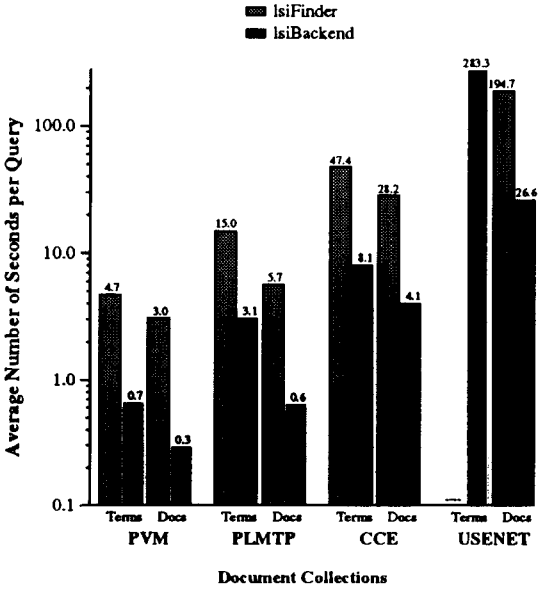

Fig. 7. *lsiBackend* versus *lsiFinder*. The average number of seconds required by *lsiFinder* and *lsiBackend* to search for related terms and documents in each of the four document collections.

not have a significant effect, but larger collections were able to take advantage of the greater parallelism and decrease the overall search time.

4.4.1. Execution Time

Figure 8 shows the average number of seconds required by *plsiBackend* to search for terms related to the test queries for 2, 4, 8, 12, 16, 20, and 24 processors. Figure 9 displays the average number of seconds used by *plsiBackend* to search for documents related to the queries using the same machine configurations. No data point is given for *plsiBackend* with two processors on the USENET collection since the collective memory of the two machines was not large enough to hold all the term and document vectors for the collection.

Both graphs show an overall downward trend as more processors are added to the parallel machine. However, the smaller document collections, with fewer term and document vectors, were less able to take advantage of the increased number of processors, and their curves became flatter as

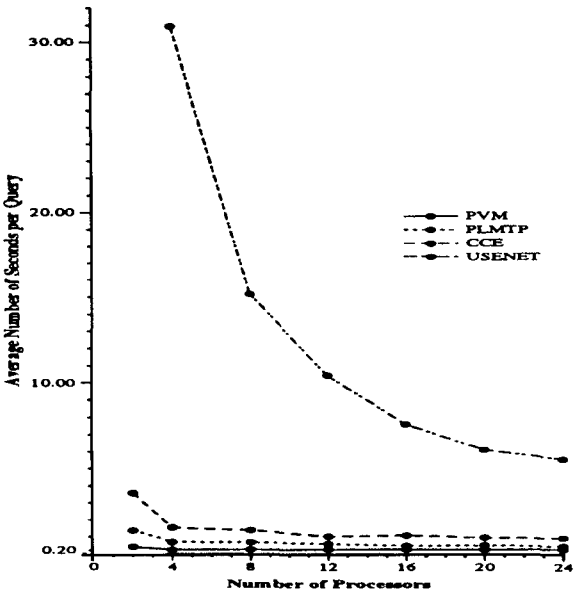

Fig. 8. The average search time of *plsiBackend* (terms). The average number of seconds required by *plsiBackend* to search for related terms in each document collection on 2, 4, 8, 12, 16, 20, and 24 processors.

more processors were added. The largest document collection, the USENET collection, was able to use all 24 processors to its advantage, decreasing the time required to process queries significantly from the serial results.

4.4.2. Speedup and Scalability

When computing the speedup of *plsiBackend*, the timings provided by *lsiBackend* were used as the base serial performance. Since *lsiBackend* must load each term and document vector from disk as it is needed while each processor executing *plsiBackend* loads its set of term and document vectors into memory upon initialization (before timings on individual queries begin), the times produced by *lsiBackend* and *plsiBackend* are not directly comparable. However, the algorithm used by *lsiBackend* may be considered to be the best available serial algorithm since any single processor does not have enough local storage to load all the term and document vectors for a given collection into memory at once.

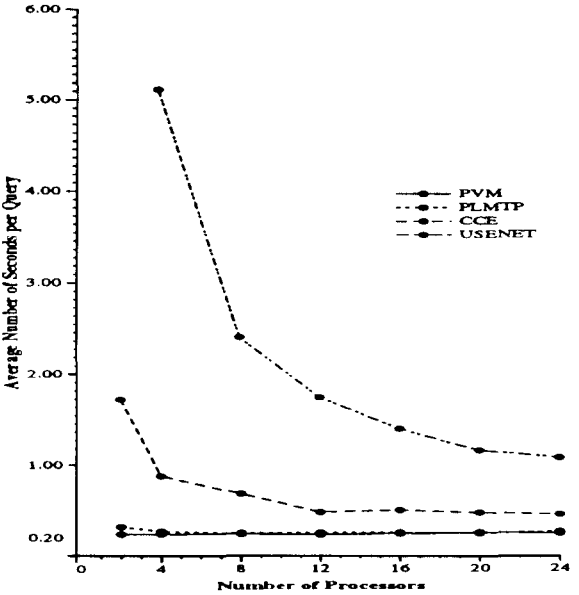

Fig. 9. The average search time of *plsiBackend* (documents). The average number of seconds required by *plsiBackend* to search for related documents in each document collection.

Speedup: Speedup is defined as the ratio between the time required by the best serial algorithm to solve a given problem and the time required by a parallel algorithm executing on p processors to solve the same problem [19]. That is, given T_1, the time required by the serial algorithm, and T_p, the time required by the parallel algorithm on p processors, the speedup of the parallel algorithm over the serial algorithm S_p is given by

$$S_p = \frac{T_1}{T_p}.$$

Figure 10 shows the average speedup achieved while searching for terms related to the test queries using 2, 4, 8, 12, 16, 20, and 24 processors. Similarly, Figure 11 shows the average speedup while searching for documents related to the same queries. Again, memory constraints prevented testing a two-processor machine serving the USENET collection.

Since the PVM and PLMTP collections are fairly small, they both produced relatively flat speedup curves. Spreading their term and docu-

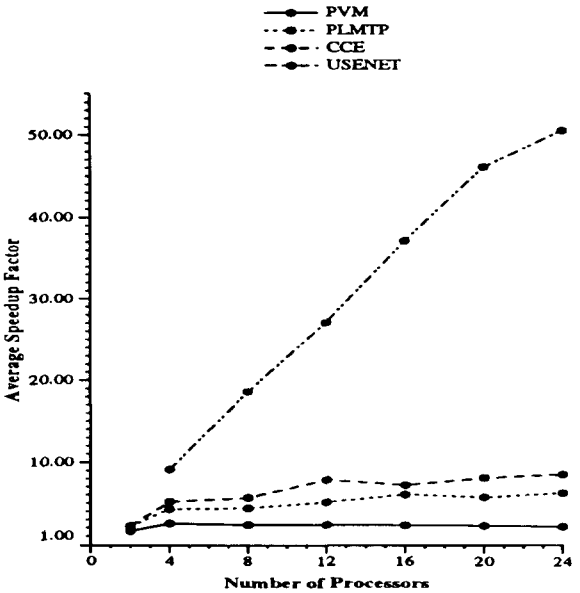

Fig. 10. The average speedup of *plsiBackend* (terms). The average speedup achieved by *plsiBackend* while searching for terms related to the test queries using 2, 4, 8, 12, 16, 20, and 24 processors.

ment vectors over increasing numbers of processors did not significantly decrease the time required to process queries in those collections. The CCE collection showed a slight speedup, although its curve appears to reach its asymptotic limit between 8 and 9 as the number of processors is increased. The USENET collection showed the most dramatic speedup of all the document collections. However, the super-linear speedup of *plsiBackend* on the USENET collection is due to the data being maintained in memory compared to the relatively inefficient way *lsiBackend* must load its data from disk incrementally.

Scalability: Since *plsiBackend* has very little communication (a few broadcasts before computation begins and a global sort after all the cosines have been computed) and the data are distributed evenly between processors, its scalability depends more on the size of the document collection being searched than the computation required to search the collection. As shown by the speedup graphs, smaller document collections limit the number of processors that can be applied to the problem while still decreasing the time to search the collection. For example, neither the PVM nor the PLMTP document collections benefit from more than

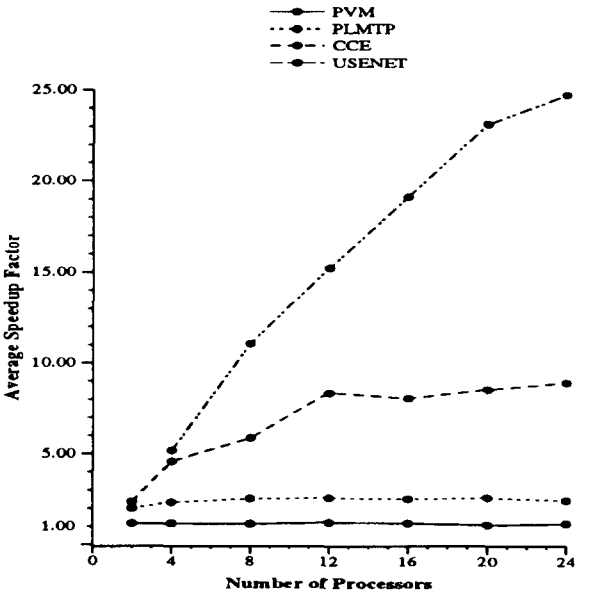

Fig. 11. The average speedup of *plsiBackend* (documents). The average speedup achieved by *plsiBackend* while searching for documents related to the test queries.

six–eight processors. However, the CCE collection appears to use up to 16 processors efficiently, while the USENET collection might be able to use more than 24 processors without difficulty.

Given enough term and document vectors, *plsiBackend* is scalable to a relatively large number of processors. Since *plsiBackend* requires very little communication, and the data are distributed evenly across the nodes, *plsiBackend* should scale to almost any number of processors.

4.5. FUTURE WORK

Although LSI++ and the LSI++ WWW interface provide the flexibility and efficiency needed to search a document collection using LSI, a great deal of work remains to make LSI a viable model for large-scale information retrieval. In particular, the existing tools for preprocessing the document collection are fairly slow and resource-intensive. Currently, each time a document collection changes, the entire preprocessing phase, including computing the SVD of the term-document matrix, must be repeated if the singular vectors are to remain orthogonal, keeping the

retrieval performance of LSI from degrading. Rather than processing the entire document collection each time it changes, updating (adding documents to the collection) and downdating (removing documents from the collection) would require less time and memory, especially for rapidly changing collections. A prototype of an LSI updating algorithm was described in [22], although it was never fully integrated into the LSI system. Downdating the SVD is an interesting problem that has not yet been attempted with LSI. With efficient updating and downdating facilities, LSI will become useful for large, rapidly changing document collections like the World Wide Web and financial and law databases.

4.6. CONCLUSIONS

Latent Semantic Indexing is a novel information retrieval model that will become more and more valuable as the amount of electronic information increases. It has been used effectively in a variety of applications, including information filtering [11] and cross-language information retrieval [27]. With the recent explosion in the amount of information available on the World Wide Web, the advent of electronic commerce, and the ability of LSI to find information similar to (rather than literally matching) other information, LSI has the potential to become the information retrieval method of choice by those who desire an alternative to keyword searching.

LSI++ is an efficient, flexible, and extensible implementation of LSI. Its object-oriented design makes it easy to maintain and modify. In addition, the LSI++ Application Programming Interface allows it to be used in a variety of applications without modification. The LSI++ WWW interface provides a highly portable, simple, and powerful interface to the LSI++ search engine. It completely hides the details of the LSI model, allowing even a naive user the ability to harness the full power of LSI. Coupled with the high performance of LSI++, the LSI++ WWW interface can provide access to a large amount of information quickly and intuitively.

This research was supported by the National Science Foundation under Grants NSF-ASC-92-03004, NSF-ASC-94-11394, and NSF-CDA-95-29459.

REFERENCES

1. *The Concise Columbia Encyclopedia*, 2nd ed., Columbia University Press, New York, 1989.

2. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1995.
3. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Interative Methods*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1994.
4. N. Belkin and W. Croft, Retrieval techniques, in: M. Williams (Ed.), *Annual Review of Information Science and Technology* (*ARIST*), Vol. 22, Elsevier Science Publishers B.V., 1987, Chap. 4, pp. 109–145.
5. M. Berry, S. Dumais, and G. O'Brien, Using linear algebra for intelligent information retrieval, *SIAM Rev.* 37(4):573–595 (1995).
6. H. Borko and M. Bernick, Automatic document classification, *ACM* 10:151–162 (1963).
7. S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman, Indexing by latent semantic analysis, *J. Amer. Soc. for Inform. Sci.* 41(6):391–407 (1990).
8. S. Dumais, Improving the retrieval of information from external sources, *Behavior Res. Methods, Instruments, & Comput.* 23(2):229–236 (1991).
9. S. T. Dumais, LSI meets TREC: A status report, in: D. Harman (Ed.), *The First Text REtrieval Conference* (*TREC1*), National Institute of Standards and Technology Special Publication 500-207, Mar. 1993, pp. 137–152.
10. S. T. Dumais, Latent Semantic Indexing (LSI) and TREC-2, in: D. Harman (Ed.), *The Second Text REtrieval Conference* (*TREC2*), National Institute of Standards and Technology Special Publication 500-215, Mar. 1994, pp. 105–116.
11. P. Foltz, Using latent semantic indexing for information filtering, in: *Proc. ACM Conf. Office Inform. Syst.* (*COIS*), 1990, pp. 40–47.
12. G. Fox, R. Williams, and P. Messina, *Parallel Computing Works!*, Morgan Kaufmann, San Francisco, CA, 1994.
13. W. Frakes and R. Baeza-Yates (Eds.), *Information Retrieval: Data Structures & Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
14. The Free Software Foundation, *The GNU G++ Class Library*, 1992.
15. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*, MIT Press, Cambridge, MA, 1994.
16. C. Golub and C. Van Loan, *Matriz Computations*, 2nd ed., Johns Hopkins, Baltimore, MD, 1989.
17. G. Golub and C. Reinsch, *Handbook for Automatic Computation II, Linear Algebra*, Springer-Verlag, New York, 1971.
18. M. Koll, WEIRD: An approach to concept-based information retrieval, *SIGIR Forum* 13:32–50 (1979).
19. V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings, Redwood City, CA, 1994.
20. T. A. Letsche, Toward large-scale information retrieval using latent semantic indexing, Master's Thesis, University of Tennessee, Knoxville, Aug. 1996.
21. C. Liu, J. Peek, R. Jones, B. Buns, and A. Nye, *Managing Internet Information Services*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

22. G. O'Brien, Information management tools for updating an SVD-encoded indexing scheme, Master's Thesis, University of Tennessee, Knoxville, Dec. 1994.
23. G. Salton and C. Buckley, Improving retrieval performance by relevance feedback, *J. Amer. Soc. for Inform. Sci.* 41(4):288–297 (1990).
24. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1996.
25. L. Wall and R. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., Sebastopol, CA, 1991.
26. R. Wurman, *Information Anxiety*, Doubleday, New York, 1989.
27. P. Young, Cross-language information retrieval using latent semantic indexing, Master's Thesis, University of Tennessee, Knoxville, Dec. 1994.

Received 1 March 1996; revised 1 November 1996