Toward Large-Scale Information Retrieval Using Latent Semantic Indexing

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Todd A. Letsche

August 1996

Acknowledgments

I am deeply indebted to Dr. Michael Berry, my major advisor, for his kind guidance and support. I also thank Dr. Susan Dumais, director of the Information Sciences Research Group at Bellcore, for her technical advice. In addition, she graciously allowed us (me as well as all of us at the University of Tennessee who have worked on LSI) to modify and extend the LSI model. I thank my committee members, Dr. Brad Vander Zanden and Dr. David Straight, for their time and suggestions.

My grateful thanks go to Dr. Bill Harrod of Cray Research, Inc. and Michael J. Merchant, formerly of Cray, for providing the opportunity to intern at Cray. They are friends as well as wonderful mentors. Also, many thanks to Curt Eide of IBM, Inc. for helping me start my professional career. I will always remember my experiences at IBM as part of the Red Badge Crew.

This research was supported in part by the National Science Foundation under grant numbers NSF-ASC-92-03004 and NSF-ASC-94-11394.

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 the serial implementation of LSI++ searches up to 6 times faster than the original implementation of LSI, while the parallel implementation searches nearly 180 times faster on large document collections.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

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

1.1 Motivation and Goals

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. . . [Wur89]."

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) [DDF+90], 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 [Dum91]), 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 implementation 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.

1.2 The Future of Information Technology

With the amount of information available increasing each year and the fact that a great deal of the information exists in electronic form at some point (whether it is published electronically or merely created with a word processor), information retrieval techniques appear poised to become increasingly valuable to casual users, scientists, and businesses. Since information that is inaccessible is as much a problem as the lack of information, searching techniques that can automatically derive conceptual data from a document and allow a user to search that conceptual data will become more and more popular among those who require large amounts of information to be indexed and, perhaps, cross-indexed with seemingly unrelated information. The following sections describe three interesting applications of information technology: data mining, digital libraries, and information filtering.

1.2.1 Data Mining

"Data mining" is a fairly new phrase used to describe the process of automatically discovering patterns and deriving meaning from large data sets. The chemical/pharmaceutical, banking and financial, and retailing and manufacturing industries have built large databases of information related to their day-to-day business operations. They want to use the data they have collected to predict future business trends, develop new products, and determine the effects of external factors (for example, advertising) on their current products and services. In particular, the chemical/pharmaceutical industry often must determine the legal, social, and biological implications of new products as they are developed. Using data mining, the industry hopes to streamline the search for new chemicals and medicines, decreasing the time and cost of developing new products [DF95].

Although many attempts at data mining have focussed of neural networks and machine learning, conceptual and vector-space models for information retrieval may be useful for representing the vast quantities of information generated by these industries, and, by examining the clusters of information, deriving meaningful conclusions from the data. Clearly, conceptual information retrieval models, because of the way they represent information independently of the actual data itself (typically, they represent the data as a point in space rather than as a set of words), are superior to lexical matching techniques when one is attempting to derive meaning from a large, heterogeneous data set.

1.2.2 Digital Libraries

Whether the words "digital library" evoke images of new methods of accessing and viewing information or merely an electronic form of the traditional library, information retrieval technologies will play a significant role in the way the information is organized and searched. Already, several World-Wide Web sites have begun indexing computer science technical reports, allowing users to search abstracts of the technical reports or the full text of the technical reports themselves [Fox95]. Futurists hope digital libraries will be able to provide greater access to information, increase the royalties paid to authors, and allow information to be dynamic (and, therefore, also continually up-to-date) rather than static [Wie95].

1.2.3 Information Filtering

As information becomes more abundant and easier to disseminate, people will become less able to handle the continuous flow of information. To help people better deal with the barrage of information, automatic information filtering techniques can be used to discard information irrelevant to a person's interests while allowing the rest of the information to pass through the filter. Information filtering is very similar to information retrieval. Instead of retrieving information related to a user's query, though, the user typically creates a profile of what he or she wishes to view, and the incoming documents are compared to the profile. Documents that match the profile are allowed to pass through the filter, while the other documents are discarded. The process of creating profiles and matching documents with the profiles is similar to relevance feedback, a technique used by information retrieval systems to enrich a user's query. Relevance feedback will be discussed in Section 2.2.6.

Since information filtering must deal with some of the same issues as information retrieval (for example, differences in word usage), conceptual information retrieval models are often used for information filtering applications. So far, results have indicated that information filtering is useful for decreasing the amount of information a person must consider, while still providing the information he or she might consider useful [FD92].

1.3 The Development of an LSI Programming Library

The following chapters outline the development and use of an efficient, extensible, portable, and maintainable implementation of the Latent Semantic Indexing model. Chapter 2 explores the general ideas behind vector-space models for information retrieval and describes LSI in particular. Chapter 3 introduces LSI++, a C++ class library for searching with the LSI model, and Chapter 4 examines how LSI++ can be used to build both serial and distributed search engines. In addition, Chapter 4 presents a World-Wide Web interface for LSI that provides users the opportunity to interact with the LSI++ search engines. Finally, Chapter 5 examines the execution efficiency of the search engines written with LSI++.

Chapter 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 are introduced and various improvements to the models are discussed. In the final section, LSI itself is considered.

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 [BC87] 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 don't 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 $(1 \leq i \leq n)$ is a non-negative 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 $\hat{t}_i$ $(1 \leq i \leq m)$ is a non-negative 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 [BC87]. 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 don't attempt to collapse the dimensions of the space treat each term independently, essentially mimicking an inverted index [FBY92]. However, vector-space models are more flexible than inverted indices since each term can be individually weighted, allowing that term to become more or less important within a document or the entire document collection as a whole. Also, 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 [FBY92].

Vector-space models were developed to eliminate many of the problems associated with exact, lexical matching techniques. In particular, since words often have multiple meanings (polysemy), it is difficult for a lexical matching technique 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 than that used by related documents in the collection may not retrieve any documents using lexical matching, even though the collection contains related documents [BDO95].

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. Unlike lexical matching techniques that provide no ranking or a very crude ranking scheme (for example, ranking one document before another document because it contains more occurrences of the search terms), the vector-space models, by basing their rankings on the Euclidean distance or the angle measure between the query and terms or documents in the space, 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, vector-space models often provide an elegant method of implementing relevance feedback [SB90]. 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 [FBY92].

2.2 Early Work

The idea of using a vector-space model to represent information has existed since the advent of the computer era. Through the years, various modifications have been suggested to improve the retrieval performance of the models. In addition, as computing resources became more available and reliable, the models have become increasingly sophisticated and complex, often accompanied by significant increases in retrieval performance.

2.2.1 H.P. Luhn and the Origin of Vector-Space Models

In 1953, H.P. Luhn published an initial discussion of vector-space models for information retrieval [Luh53] that summarized many of the key issues and concepts still being considered today. Luhn was motivated by the concern that the controlled vocabularies and classification schemes used in manual indexing may change over time. Luhn was also concerned that by only classifying concepts in a document that seemed important at the time, aspects of the document that might become more important in the future would be lost.

The 1953 article, along with a subsequent article in 1957 [Luh57], proposed the use of a "notion-space" to alleviate the problems of classification and the lack of uniformity in word usage. Luhn suggested each document should be represented in the notion-space by an array whose elements correspond to the various notions (language-independent ideas) in the document collection. To search the collection, the notions contained in the query would be encoded like the notions in the document, and the array would be compared to the other arrays in the space. Since documents sharing similar notions would presumably lie near each other in the notion-space, a similarity measure could be used to find documents relevant to the query. The results of the search could then be used to better specify the query, allowing better retrieval performance.

To aid in defining the notions for a document collection, Luhn suggested two dictionaries were needed. The first dictionary would contain a listing of the notions used to define the documents' position in the space along with their corresponding index numbers. The second dictionary would contain an alphabetical listing of the words worth indexing in the collection and the notions to which the words belonged. By examining the two dictionaries, the arrays used to represent queries and new documents could be constructed automatically.

Although Luhn's proposed vector-space representation lacked many important implementation details, it provided an intuitive explanation of the purpose for the vector-space model and laid the foundation for later implementations and improvements. Modern, more complex retrieval systems are at least partially based on the ideas presented by Luhn in the 1950's. However, it is interesting to note that some of the issues he raised over 40 years ago (for example, how to overcome the difficulties of synonomy and polysemy) have not yet been totally resolved.

LSI was developed to solve many of the information retrieval problems Luhn anticipated in the 1950's. The LSI model will be discussed in Section 2.3.

2.2.2 Borko and Bernick on Reduced-Space Document Classification

A short time after Luhn's ideas were published, H. Borko and M. Bernick [BB63] presented a method by which documents could automatically be classified into predefined categories. Although document classification has different goals than information retrieval, Borko and Bernick's approach to document classification can be viewed as a special case of information retrieval. Like all vector-space approaches, Borko and Bernick assumed the terms in a document were a fairly reliable indicator of the semantic content of the document. They hoped documents belonging to the same classification category contained similar words. Two sets of documents, an experimental set and a validation set, were used in their experiments. The experimental set was used to derive both the classification categories and a function to predict the correct category for a given document. The validation set was used to test whether the prediction formula correctly categorized documents.

To define the classification categories, Borko and Bernick constructed a term-document matrix for their experimental document collection, using the frequencies of the terms in each document as the elements of the matrix. They then computed correlation coefficients for each term against all the other terms in the matrix. After the correlation matrix was subjected to factor analysis [Har47] to reduce its dimensionality, a set of orthogonal factors was extracted, rotated, and interpreted as classification categories.

Once the classification categories were determined, a prediction formula, based on term frequency and the normalized factor loading of terms in the given category, was devised to automatically classify the documents in the validation set. The prediction formula was applied to each document and category, and the category with the highest predicted result was selected as the most probable category for the document.

Results indicated that approximately half of the documents were correctly classified by the system. The authors, somewhat disappointed by their results, attributed their less than optimal performance to the generalizing ability of factor analysis. They believed the factor analysis kept them from exploiting the documents' individual variances. However, they were encouraged that factor analysis performed comparably to other automatic classification systems.

Like Borko and Bernick's classification system, LSI uses a reduced-dimension space to represent the terms and documents. However, LSI uses the singular value decomposition (SVD), a two-mode factor analysis technique, rather than one-mode factor analysis to reduce the dimensionality of the space. The SVD will be discussed in Section 2.3.3.

2.2.3 Salton and the SMART Information Retrieval System

The SMART information retrieval system, initiated by Salton *et al* at Harvard University in 1964 and continued at Cornell University in 1968 [FBY92], embodied many of the information retrieval techniques found in modern vector-space systems. By using stemming to remove suffices from terms, and a variety of dictionaries and thesauri to increase the effectiveness of the many different information analysis techniques incorporated in SMART, SMART became an important research vehicle as well as an effective information retrieval system [SL65].

Salton and other researchers used SMART to study term-weighting, relevance feedback, clustering, stemming, synonyms, and the use of phrases to improve the performance of the retrieval system [FBY92]. In particular, SMART used syntactic phrase matching to recognize similar concepts in the documents and to allow documents with similar concepts to be retrieved. It also used statistical phrase matching to detect the co-occurrence of concepts in sentences. SMART attempted to adapt to the query being performed by allowing the user to choose from a variety of automatic analysis procedures to process the query. In addition, the different analyses could be automatically compared to determine their relative performance and to ensure most of the documents matching the query were found [SL65].

By comparing SMART to a conventional information retrieval system (MEDLARS), Salton found that SMART's standard configuration performed worse than the standard MEDLARS search. However, by eliminating common words, words that occurred only once in the document collection, and words that increased the inter-document similarity in the document collection, SMART performed almost as well as MEDLARS. Salton also found that by building a thesaurus to aid SMART and using one or two relevance feedback iterations, SMART commanded an 18 to 30 percent advantage over MEDLARS [Sal71].

2.2.4 Sparck Jones and Global Weighting

In 1972, Sparck Jones [Jon72] examined the use of global weighting schemes to improve the performance of information retrieval systems. The author argued that terms occurring frequently in a collection, while skewing the results of queries that include those terms, were essential for effective information retrieval. However, to better differentiate between documents, non-frequently occurring terms in a collection should be given greater value than frequently occurring terms when determining which documents in a collection were relevant to a query.

The author used the function $f(n) = m$ such that $2^{m-1} < n \leq 2^m$ to globally weight three standard test collections. For a collection containing N documents, a term occurring n times in the collection would receive the weight $f(N) - f(n) + 1$. On each of the three document collections, the performance improvements provided by global weighting surpassed that of any other single improvement to information retrieval systems suggested in the literature. Together with local weighting, the global weightings substantially increased the retrieval performance of even simple retrieval systems. Commonly used local and global weighting schemes as they relate to LSI will be discussed in Section 2.3.2.

2.2.5 Koll and Reduced-Space Models for Information Retrieval

Koll, in 1979, expanded Borko and Bernick's idea of a reduced-dimension term-document space by creating WEIRD [Kol79], an information retrieval system that tried "to advance the form of the retrieval decision from word-matching to concept-similarity." WEIRD attempted to capture similarities in word usage across documents by placing terms and documents at points in space relative to each other. That is, Koll placed documents (which he called "contexts") at the average of the locations of its constituent terms (which he called "concepts"). The average of the locations of the documents in which a term occurred determined the location of the terms.

Koll began by selecting seven documents from his test collection whose terms almost, but not completely, spanned the term-space of the document collection. He used those seven documents as the orthogonal axes of the term-document space. Then, he examined each of the remaining documents, removed common words, and stemmed the terms in the documents to remove common suffices. By using the terms in the documents that had already been placed in the term-document space, he temporarily placed the documents at the average position of those terms. The existing terms in the space were then relocated to reflect their usage in the new document. The terms in the new document that didn't previously exist in the space were assigned the document's location as their location in space. After all the documents were processed, Koll reassigned the locations of all the documents to the average of their constituent terms since the location of documents placed in the space early in the process were determined by relatively fewer terms than documents placed later in the process.

Queries were treated as documents to determine their location in the space. The Euclidean distance between the query and each document was used to rank the nearest neighbors.

Koll found that, in general, relevant documents were ranked higher than irrelevant documents, although WEIRD's performance was approximately similar to standard lexical searching systems. On longer queries, though, WEIRD fared somewhat better, outperforming the lexical models. Koll discussed the number of dimensions required for effective searching and proposed methods of selecting documents to serve as the axes of the space, although he reached no definite conclusions for either problem. LSI's term-document representation will be discussed in Section 2.3.1.

2.2.6 Salton and Buckley on Relevance Feedback

In 1990, Salton and Buckley [SB90] reexamined the automatic relevance feedback process introduced in the mid-1960's and used in Salton's SMART information retrieval system (Section 2.2.3). By testing several methods of relevance feedback on five standard test collections, they found average performance improved 90% when relevance feedback was used to augment a user's query. In addition, they found that improvements up to 100% were possible when several iterations of relevance feedback were employed.

Specifically, they found:

- weighting the terms in the document collection improved the effectiveness of relevance feedback,
- using the full text of the relevance feedback documents (rather than using only the most commonly occurring terms in the relevance feedback documents) for a query slightly improved the performance of the query, and
- better performance is achieved with relevance feedback when all the relevant documents retrieved by a query are used to augment the relevance feedback query, but only the highest-ranking non-relevant document is used.

In addition, they found:

- insufficiently stated queries (for example, queries consisting of only a few terms) benefit more from relevance feedback than larger, more detailed queries,
- document collections that are initially difficult to search benefit more from relevance feedback than document collections that are at first easy to search, and
- technical collections fared better with relevance feedback queries than general collections. The authors speculated that technical collections are confined to a smaller term-document space than more general collections, allowing relevance feedback to help better differentiate the documents.

Dumais [Dum91] reported that relevance feedback improved the performance of LSI between 33 and 67 percent. Relevance feedback, as used in the LSI model, will be discussed in Section 2.3.5.

2.3 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) [GL89] 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 analyses 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 [Dum91].

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 (Section 2.2.5), 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 (Section 2.2.2), 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 [DDF+90].

Compared to other information retrieval techniques, LSI performs surprisingly well. In one test, Dumais [Dum91] found 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 LSI performed an average of 20% better than lexical retrieval techniques. In addition, LSI is fully automatic and easy to use, requiring no complex expressions or syntax to represent the query. Because terms and documents are explicitly represented in the space, relevance feedback (Section 2.2.6) 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 [BDO95].

2.3.1 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 are assigned a row in the matrix, while each of the n documents in the collection are assigned a column in the matrix. A non-zero 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 [BDO95].

2.3.2 Weighting

The benefits of weighting are well-known in the information retrieval community (see Section 2.2.4). LSI typically uses both a local and global weighting scheme to

$$
\begin{aligned}
&\text{Binary:} & L(i,j) &= \begin{cases} 0 & \text{if} \quad tf_{ij} = 0 \\ 1 & \text{if} \quad tf_{ij} > 0 \end{cases} \\
&\text{Term-Frequency:} & L(i,j) &= tf_{ij} \\
&\text{Log:} & L(i,j) &= \log_2(tf_{ij} + 1)
\end{aligned}
$$

Where:

tf_{ij} = the frequency of term i in document j

Figure 2.1: Local weighting functions.

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 non-zero 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 Figures 2.1 and 2.2 from [Dum91]), Dumais found the log-entropy weighting scheme provided a 40% advantage over raw term frequency on several standard document test collections [Dum91].

2.3.3 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) [GL89]. 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

Normal: $G(i) = \sqrt{\dfrac{1}{\sum\limits_j tf_{ij}^2}}$

GfIdf: $G(i) = \frac{gf_i}{df_i}$

Idf: $G(i) = \log_2\left(\frac{ndocs}{df_i}\right) + 1$

Entropy: $G(i) = 1 - \sum\limits_j \frac{p_{ij}\log_2(p_{ij})}{log_2(ndocs)}, \quad p_{ij} = \frac{tf_{ij}}{gf_i}$

Where:

tf_{ij}	=	the frequency of term i in document j
gf_i	=	the global frequency of term i
df_i	=	the number of documents in which term i appears
$ndocs$	=	the number of documents in the collection

Figure 2.2: Global weighting functions.

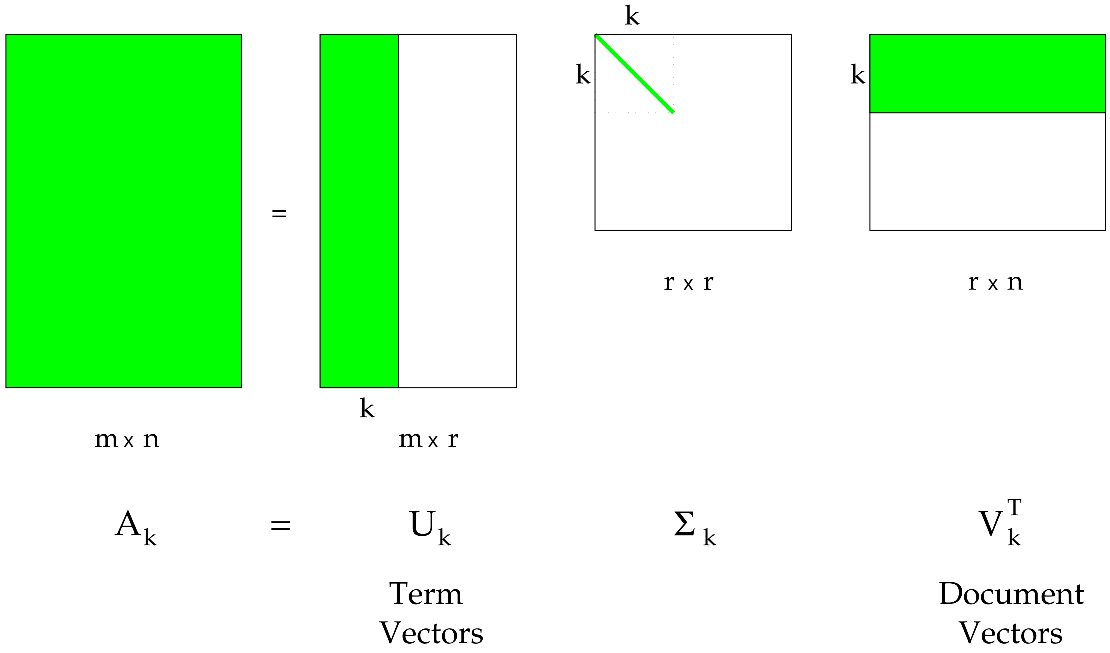

Figure 2.3: A pictorial representation of the SVD. The shaded areas of U and V, as well as the diagonal line in Σ, represent A_k, the reduced representation of the original term-document matrix A.

are called the singular values of the matrix A. As illustrated in Figure 2.3, 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 [GR71] suggests that A_k, constructed from the k-largest singular triplets* of A, is the closest rank-k approximation (in the least squares sense) to A [BDO95].

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 "noise" that causes poor retrieval performance is thought to be eliminated. Thus, although a high-dimensional repre-

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

sentation 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 won't be eliminated, resulting in poor retrieval performance [BDO95].

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 [BDO95].

2.3.4 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 non-zero elements contain the weighted (using the same local and global weighting schemes applied to the document collection being searched; see Section 2.3.2) 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 [BDO95].

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 [BDO95].

2.3.5 Relevance Feedback

Relevance feedback is an effective method for iteratively improving a query without increasing the computational requirements to perform the query (see Section 2.2.6). 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.

Chapter 3

The Design and Implementation of a Latent Semantic Indexing Server

The LSI model described in the previous chapter 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 $q =< b, r, n, t, d, f >$, 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 is used to to refine the query through relevance feedback (Section 2.3.5), 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 frequency 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 [FBY92]. 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 can't 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. See Sections 2.3.4 and 2.3.1 for a more complete explanation of the pseudo-document and the term-document space.

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. See Section

2.3.4 for a complete explanation of projection and matching in the term-document space.

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 3.1 summarizes the process.

3.2 Original Implementation

The initial implementation of LSI [DDF+90] 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. It's design was heavily influenced by PARAFAC [HL84], a system used to find the numerical solution of multi-mode factor-analysis problems. When PARAFAC was replaced by SVDPACK [Ber92], remnants of the PARAFAC version remained in the resulting code.

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* 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. Figure 3.2 shows the data pathway for a typical relevance feedback (Section 2.3.5) query.

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

*UNIX is a registered trademark of AT&T

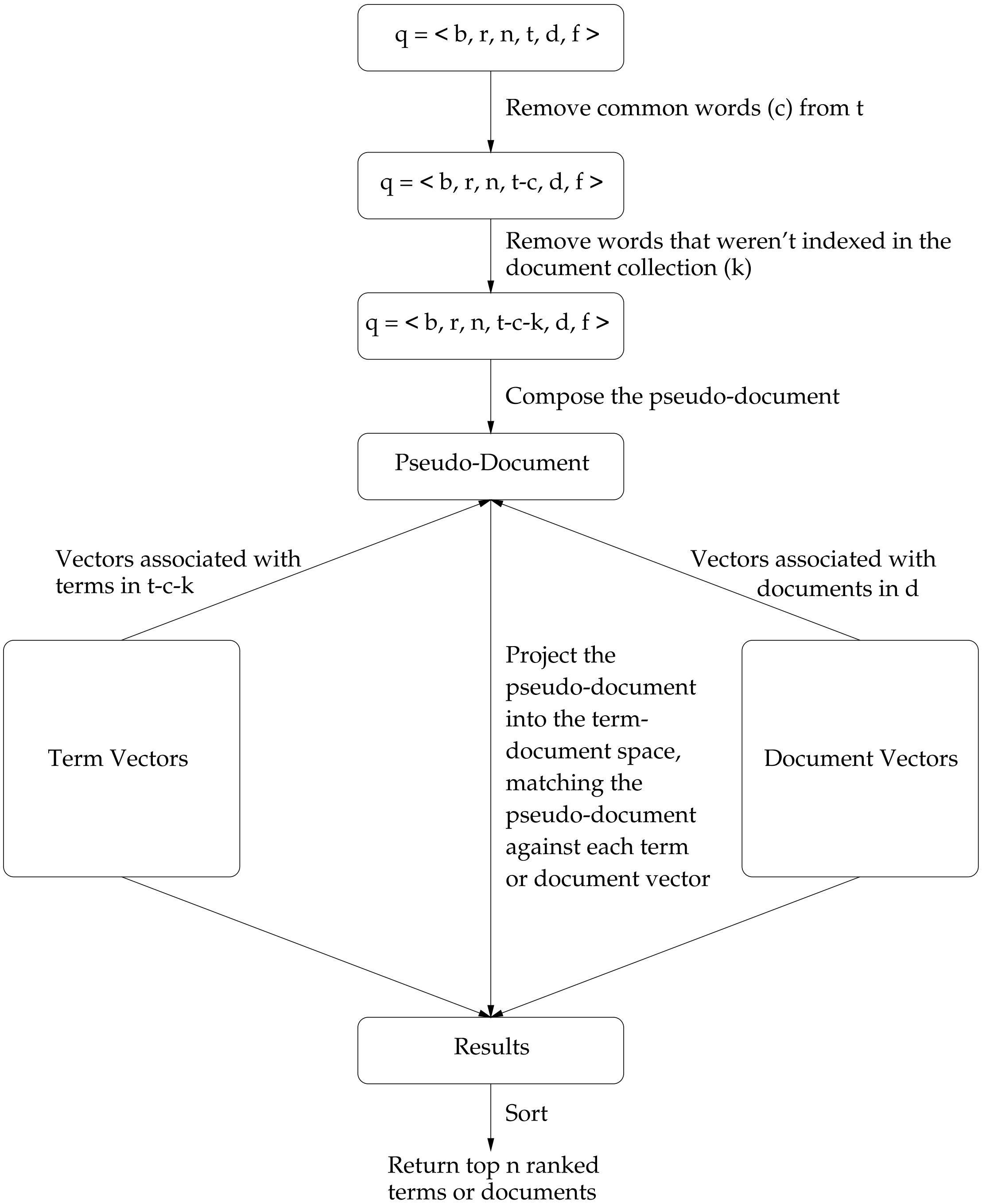

Figure 3.1: The query phase of the LSI model.

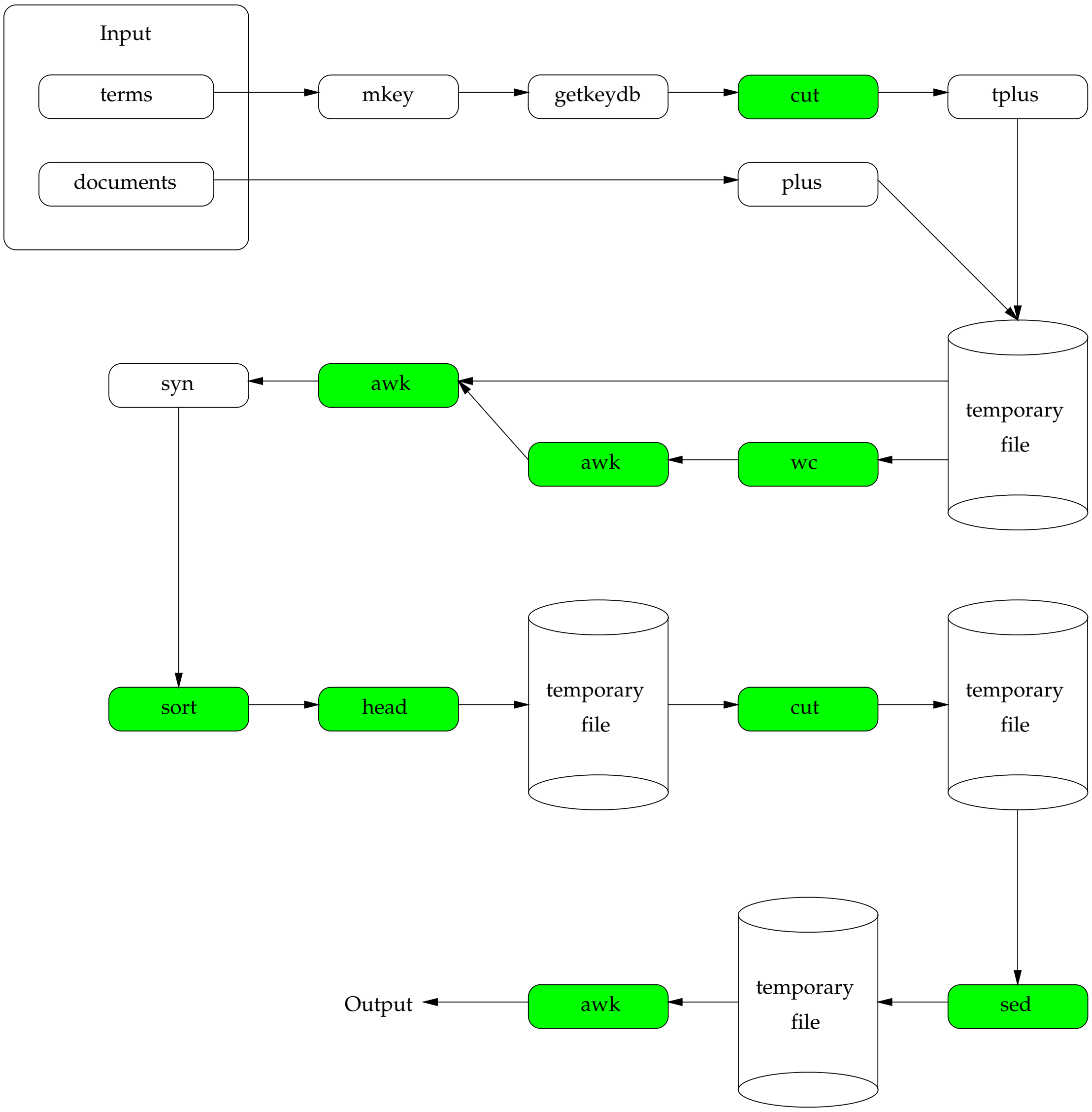

Figure 3.2: A typical relevance feedback query in the Bellcore implementation. Each box represents a separate program that is started from an encompassing script. The unshaded boxes represent programs specific to LSI while shaded boxes represent UNIX utilities used to convert data formats between stages of computation.

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. Although pipes could be simulated on non-UNIX platforms, their use only adds to the portability problems of the rest of the system.

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 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, multi-user 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,

$$\underbrace{\underbrace{2q_t f + q_d f}_{\substack{\text{Build the}\\ \text{Pseudo-}\\ \text{Document}}} + \underbrace{t(6f+1)}_{\text{Cosine}} + \underbrace{O(t \log t)}_{\text{Sort}}}_{\substack{\text{Time Complexity (FLOPS) when}\\ \text{Matching Terms}}} \qquad \underbrace{\underbrace{2q_t f + q_d f}_{\substack{\text{Build the}\\ \text{Pseudo-}\\ \text{Document}}} + \underbrace{d(6f+1)}_{\text{Cosine}} + \underbrace{O(d \log d)}_{\text{Sort}}}_{\substack{\text{Time Complexity (FLOPS) when}\\ \text{Matching Documents}}}$$

Where:

$$\begin{aligned} f &= \text{the number of factors,} \\ t &= \text{the number of term vectors,} \\ d &= \text{the number of document vectors,} \\ q_t &= \text{the number of terms in the query, and} \\ q_d &= \text{the number of documents in the query.} \end{aligned}$$

Figure 3.3: The serial time complexity. The complexity is given in terms of the number of floating point operations (FLOPS) required to perform the query phase of LSI using the cosine similarity measure.

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. 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. Figure 3.3 describes the time complexity of the serial query phase.

An efficient serial implementation of the query phase should minimize memory usage by loading from disk each term or document vector as it is needed, performing all the computation that involves that vector, and then discarding 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 backend 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 (see Section 5.4 for a discussion of the time required to start a distributed backend server on several sample document collections). 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 is completely independent of the operations performed on another term or document vector. Even a simple blocked data distribution (Figure 3.4) 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

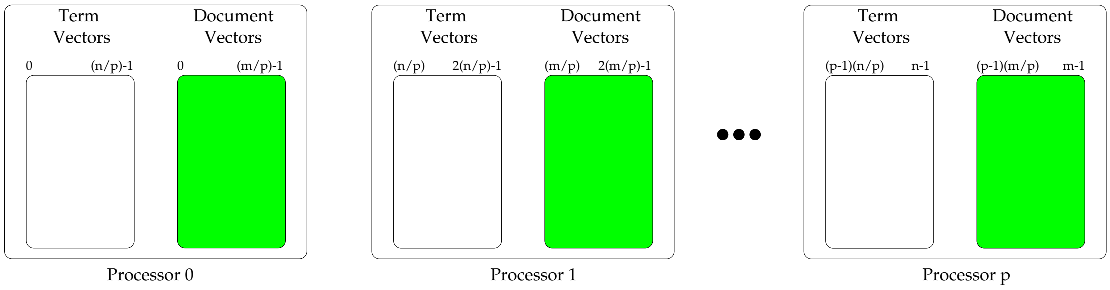

Figure 3.4: A simple blocked data distribution. Each of the p processors of the parallel machine receives a portion of the n term vectors and m document vectors.

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. Figure 3.5 describes the time complexity of the distributed query phase.

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 3.6 shows the inheritance relationship between the LSI++ classes. Most

Time Complexity (FLOPS) when Matching Terms:

$$\left\{ \underbrace{2q_t f + q_d f}_{\substack{\text{Build the}\\ \text{Pseudo-}\\ \text{Document}}} + \underbrace{\frac{t}{p}(6f+1)}_{\text{Cosine}} + \underbrace{O\left(\frac{t}{p}\log\frac{t}{p}\right)}_{\text{Local Sort}} + \underbrace{O\left(n\log p\right)}_{\substack{\text{Global}\\ \text{Merge}}} \right.$$

Time Complexity (FLOPS) when Matching Documents:

$$\left\{ \underbrace{2q_t f + q_d f}_{\substack{\text{Build the}\\ \text{Pseudo-}\\ \text{Document}}} + \underbrace{\frac{d}{p}(6f+1)}_{\text{Cosine}} + \underbrace{O\left(\frac{d}{p}\log\frac{d}{p}\right)}_{\text{Local Sort}} + \underbrace{O\left(n\log p\right)}_{\substack{\text{Global}\\ \text{Merge}}} \right.$$

Where:

$$\begin{aligned} f &= \text{the number of factors,}\\ t &= \text{the number of term vectors,}\\ d &= \text{the number of document vectors,}\\ q_t &= \text{the number of terms in the query,}\\ q_d &= \text{the number of documents in the query,}\\ p &= \text{the number of processors, and}\\ n &= \text{the number of terms or documents to retrieve.} \end{aligned}$$

Figure 3.5: The distributed time complexity. The complexity is given in terms of the number of floating point operations (FLOPS) required to perform the query phase of LSI using the cosine similarity measure.

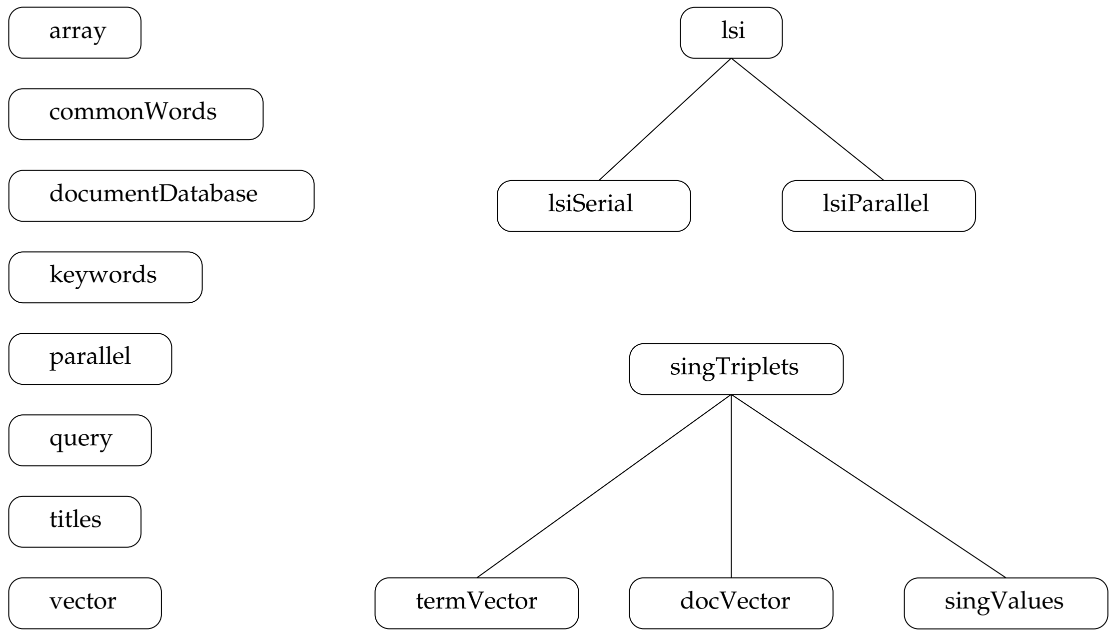

Figure 3.6: The LSI++ inheritance hierarchy. The classes on the left side of the diagram are independent while the *lsiSerial* and *lsiParallel* classes are derived from the *lsi* class. The *termVector*, *docVector*, and *singValues* classes are derived from the *singTriplets* class.

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 GNU version of C++, g++, includes an excellent library [Fre92] 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++.

Figure 3.7 shows the interaction among LSI++ classes in the serial implementation. Since much of the code from the serial implementation is reused in the distributed implementation, the interaction among the classes in the distributed implementation (Figure 3.8) is very similar to the serial case. Both diagrams roughly correspond to Figure 3.1.

In the following sections, each class is described in greater detail and its interaction with other classes is discussed.

The lsi class

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 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.

The member functions given in Table 3.1 allow a query to be specified and performed. Table 3.2 lists the member functions used to get the results of a query, and Table 3.3 shows several member functions that provide miscellaneous information about the query.

The lsiSerial and lsiParallel classes

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

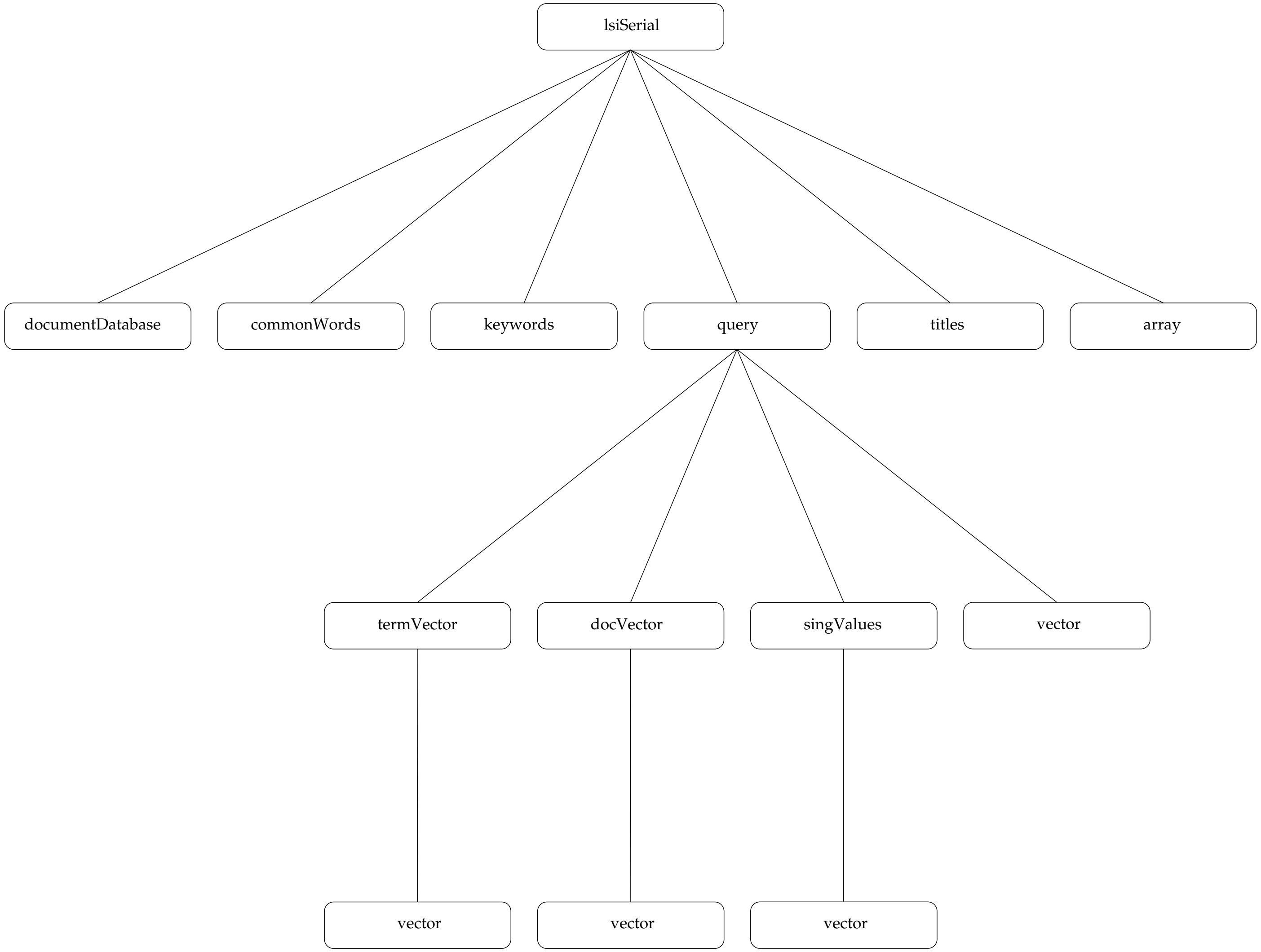

Figure 3.7: The LSI++ serial part-of hierarchy.

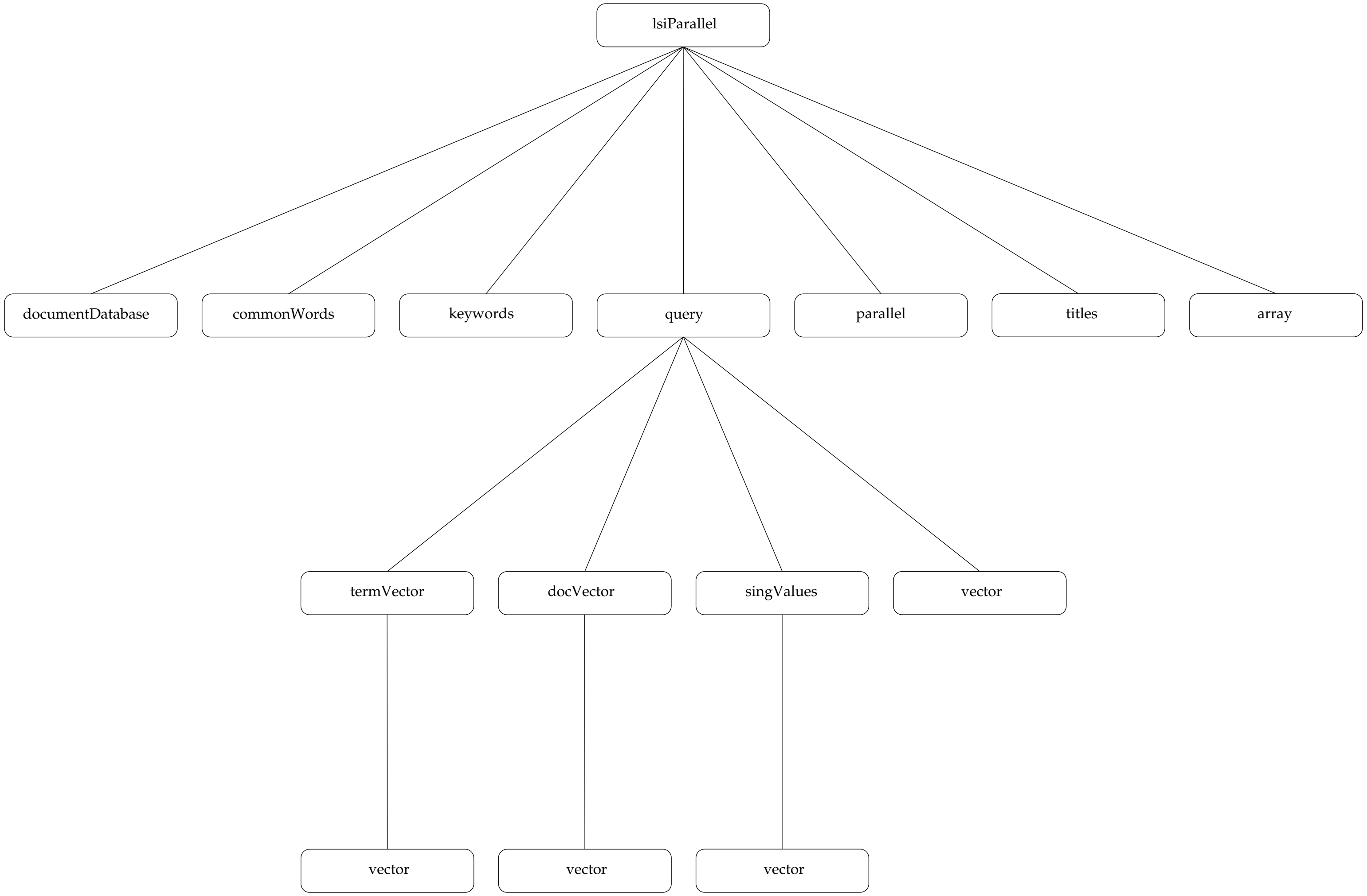

Figure 3.8: The LSI++ distributed part-of hierarchy.

Table 3.1: API member functions used to set the parameters of a query.

Member Function		Explanation
void	addDoc(const int docNum)	adds document with ID docNum to the query
void	addTerm(const int termNum)	adds term with ID termNum to the query
void	addTerm(const String& termString)	adds the terms given in termString to the query
void	setReturnType(const int rt)	sets whether to return terms, documents, or both
int	getReturnType()	retrieves the type of objects to be returned
void	setNumDocsToReturn(int numDocs)	sets the number of documents to be returned to numDocs
int	getNumDocsToReturn()	retrieves the number of documents to be returned
void	setNumTermsToReturn(int numTerms)	sets the number of terms to be returned to numTerms
int	getNumTermsToReturn()	retrieves the number of terms to be returned
void	setNumberFactors(int numFacts)	sets the number of factors to use for this query
int	getNumberFactors()	retrieves the number of factors used in this query
void	performQuery()	projects the pseudo-document into the term-document space

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) [SOHL+96], making it highly portable. Since the time required to start a parallel application tends to be large (see Section 5.4), an application using the *lsiParallel* class will probably be started as a continuously-running backend server, with one or more clients passing queries to it and receiving the results.

Table 3.2: API member functions used to retrieve the results of a query.

Member Function		Explanation
int	getNumQueryTerms()	returns the number of terms submitted for the query
int	getNumQueryDocs()	returns the number of documents submitted for the query
int	getQueryTerm(int n)	returns the ID of the nth term submitted for the query
int	getQueryDoc(int n)	returns the ID of the nth document submitted for the query
int	isQueryDoc(const int docNum)	returns true if the document with ID docNum was submitted as part of the query
int	isQueryTerm(const int termNum)	returns true if the term with ID termNum was submitted as part of the query
void	getResultTerm(const int rt, int& id, float& score)	returns the ID of the rtth ranked term and its relevance score
void	getResultDoc(const int rd, int& id float& score)	returns the ID of the rdth ranked document and its relevance score
int	wasFound(const String& key)	returns true if key was actually used in the query (i.e., it wasn't eliminated as a common word and was a word found in the document collection)
int	numberFound()	returns the number of terms actually used in the query

Table 3.3: Informational API member functions.

Member Function		Explanation
int	getMaxNumberTerms()	returns the number of unique terms in the document collection
int	getMaxNumberDocs()	returns the number of documents in the document collection
int	getMaxNumberFactors()	returns the number of factors set during the preprocessing phase
int	getBookNum()	returns the number of the book being searched
String	getBookName()	returns the name of the book being searched
String	getKeyword(const int id)	returns the term identified by id
int	getKeywordId(const String& key)	returns the ID associated with term key
String	getTitle(const int id)	in the current book, returns the title of the idth document
int	myId()	returns the processor number of this node (*lsiParallel* only)
int	numProcs()	returns the total number of active processors (*lsiParallel* only)

The documentDatabase class

Directory and filename information for each document collection is kept in a file called the document database. When a document collection is created, it is assigned a unique number to identify the collection during searches. The document database lists

- the number of the document collection (a "book" in LSI terminology),
- the title of the book,
- the directory that contains the book,
- the name of the stoplist file for the book,
- the name of the file that contains all the terms in the document collection and the weightings associated with those terms,
- the name of the file that contains the term vectors, the document vectors, and the singular values,

- the name of the file that contains the documents themselves and the name of the index file used to extract individual documents from the collection,
- the name of the file of document titles and the index file used to extract specific titles from the file, and
- the port number and machine that should be used as a backend server to search that book.

The *documentDatabase* class, given a book number, is responsible for extracting each of the above fields for that book from the document database. The member functions of the *documentDatabase* class provide access to the various fields of the document database.

The commonWords class

The *commonWords* class is responsible for eliminating words on the stoplist from the query. On instantiation, the *commonWords* class reads the appropriate stoplist for the given document collection, querying the *documentDatabase* object for the filename. The class provides member functions for other objects to ask whether a given word is on the stoplist. Currently, the stoplist for the document collection being searched is read into a memory-based hash table for quick access.

The keywords class

The *keywords* class provides access to the list of all words (except words on the stoplist) in the document collection, the identity of the term vectors associated with those words, and their weightings.

After the common words are eliminated from a query, the term vectors and weightings associated with the remaining terms must be retrieved in order to build the pseudo-document. The weighting associated with each word is used to scale the term vector (making the term either more or less important relative to other terms in the query) before the term vector becomes part of the projection.

The *keywords* class currently uses a gdbm file [Nel] (the Free Software Foundation's version of the dbm and ndbm UNIX database management utilities) to store the terms, identifiers, and weights. The gdbm library provides a disk-based associative array that allows efficient storage and retrieval of byte streams. Gdbm was selected over the standard dbm utilities since it is more portable, is freely available, and, unlike standard dbm, doesn't have arbitrary limits on bucket size.

The query class

The *query* class performs most of the computation in LSI++. It adds terms and documents to the pseudo-document, projects the pseudo-document into the term-document space, computes the similarity between the pseudo-document and each term or document vector, and returns the results to higher-level classes for sorting and further processing. It performs all of the computation at a high level, dealing with vectors of data rather than individual elements. It receives the term vectors, document vectors, and singular values from the *termVector*, the *docVector*, and the *singValues* classes, respectively.

The singTriplets class

The *singTriplets* class serves as a base class for the *termVector*, *docVector*, and *singValues* classes. It is responsible for providing term vectors, document vectors, and singular values to the query object. Since the term vectors, documents vectors, and singular values are stored in secondary storage, and reading from secondary storage is very expensive, the *singTriplets* class provides the option of caching vectors in memory for its derived classes. When the *query* class requires a specific vector to perform a computation, the *termVector*, *docVector*, and *singValues* classes can determine whether the *singTriplets* object has cached that vector in memory. If the vector has been cached, the memory-based vector is returned. Otherwise, the vector must be loaded from disk. Since term vectors, document vectors, and singular values are only read during a query operation, and never written, the *singTriplets* class does not have to handle cache coherency problems, even in parallel applications.

The termVector class

The *termVector* class is derived from the *singTriplets* class and is responsible for providing term vectors to the *query* object. If the *termVector* object determines that a given vector has been cached by the *singTriplets* class, it returns the cached vector. If the vector has not been cached, it determines the location of the vector in the vectors file, loads the vector, and returns it to the *query* class.

The docVector class

The *docVector* class, also derived from the *singTriplets* class, operates exactly like the *termVector* class, except it provides document vectors to the *query* object. Like the *termVector* class, the *docVector* class first determines whether the document vector has been cached by the *singTriplets* class, loading the vector only if it has not already been cached.

The singValues class

The third class derived from the *singTriplets* base class is the *singValues* class. The *singValues* class is responsible for providing the singular values to the *query* object. Since the number of singular values is limited, the vector containing the singular values is always cached by the *singTriplets* object.

The parallel class

The *parallel* class is used only by the *lsiParallel* class. It initializes MPI and provides the necessary information for executing LSI++ on a parallel processing machine. On instantiation, the *parallel* class initializes the message passing library, determines the total number of processors that will participate in the parallel computation, and finds the processor number for the particular node on which it is executing. It also computes the starting term and document vector numbers for the node and determines the number of vectors to cache in memory.

The titles class

Given a document number, the *titles* object loads the title of that document. To allow efficient access to specific titles in the document collection, the *titles* class manages an index into the file of titles.

The vector class

The *vector* class hides the implementation of vectors from the other objects and allows them to manipulate the vectors as a single entity. For example, if the query object wishes to multiply two vectors a and b, and place the result in the vector c, the statement

$$c = a * b$$

multiplies vectors a and b, placing the element-wise multiplicative result in c. The *vector* class overloads most of the arithmetic operators and also provides dot product, square root, and inverse operations.

The array class

The *array* class stores homogeneous collections of objects. It allows access to individual elements of the array, provides a member function to access the number of elements in the array, and allows the array to be resized.

Chapter 4

A Client-Server Interface

From a user's perspective, an effective information retrieval system requires an intuitive, flexible interface. Although command-line interfaces are adequate for advanced users, their arcane syntax is often difficult for the casual user to grasp. Users typically want an interface that is easy to understand the first time they use it, without reading a lot of instructions or consulting a reference guide. A graphical user interface (GUI) can provide the visual clues necessary to guide the user through the searching process without requiring him or her to know a great deal about the system.

From a programmer's perspective, graphical user interfaces are difficult to write and maintain. In one survey, researchers found that an average of 47.6% of the code in a typical application is devoted to the interface. In addition, the survey states "respondents spent an average of 44.8% of design time on the user interface, 50.1% of implementation time, and 37.0% of maintenance time [MR92]." To a programmer writing an application that requires a GUI, developing and maintaining a useful, portable GUI becomes as enormous and time-consuming as the application itself.

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 interface for LSI, based on LSI++, is described. A World-Wide Web interface was selected as the first application of LSI++

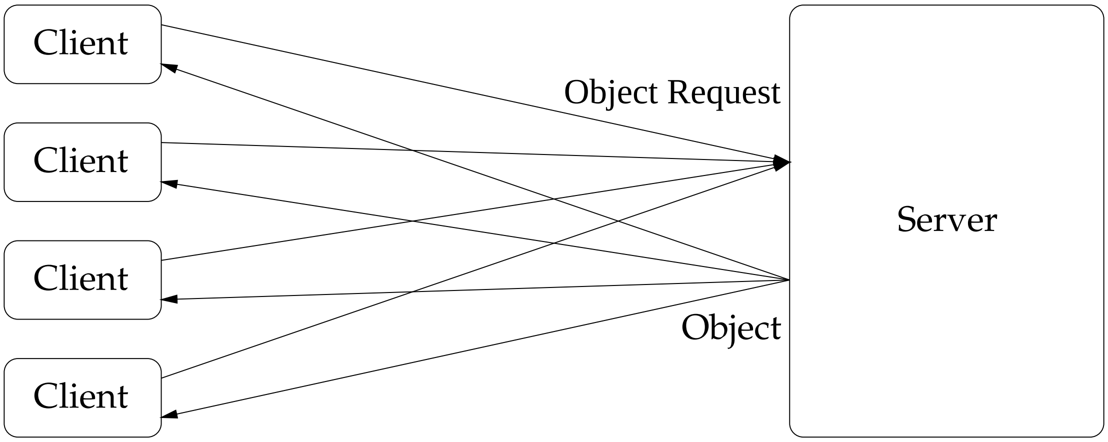

Figure 4.1: Client-server communication. The server listens at a well-known address, receives a request for an object from the client, and sends the object back to the client.

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. The LSI World-Wide Web interface permits even a naive user to utilize LSI efficiently and effectively.

4.1 The World-Wide Web

The World-Wide Web (WWW) was originally developed by a team led by Tim Berners-Lee in 1989 at the Conseil Europeen pour la Recherche Nucleaire (CERN), the European Laboratory for Particle Physics, in Geneva, Switzerland. The WWW consists of a vast collection of publicly-available, interlinked documents, images, and other media [LPJ+94].

The WWW uses a client-server approach to access data. The server binds to a well-known port and waits for incoming connections. When a client, called a browser, connects to that port on the server and requests a file, the WWW server retrieves the file and passes it back to the browser (Figure 4.1). If the file is not available, the server informs the browser, and the browser displays an error message [LPJ+94].

The location of each object on the WWW is specified by a Uniform Resource Locator

$$\underbrace{\text{http}}_{\text{Transfer Protocol}} \; :// \; \underbrace{\text{www.cs.utk.edu}}_{\text{HTTP Server}} \; / \; \underbrace{\sim \text{lsi/lsi.html}}_{\text{Directory and Filename}}$$

Figure 4.2: The format of a Uniform Resource Locator (URL).

(URL). The URL for a file consists of the transfer protocol required to access the file, the name of the WWW server on which the file is located, the directory where the file can be found, and the name of the file (Figure 4.2). To access a file, a user directs his or her browser to retrieve the file by specifying the URL at which the file can be found [LPJ+94].

Typically, browsers are capable of using a variety of transfer protocols to access information on the World-Wide Web. A browser can use the Gopher protocol to access a Gopher server, act as a File Transfer Protocol (FTP) client to access files available via FTP, and use the Z39.50 protocol to access a Wide Area Information Server (WAIS) [LPJ+94].

To access the World-Wide Web, browsers use the HyperText Transfer Protocol (HTTP). HTTP is a lightweight protocol developed specifically for the World-Wide Web. Before an HTTP server sends a file to the browser, it first sends a header that describes the type of data being sent. The header tells the browser how to interpret the incoming data stream. For example, if the header specifies "text/html," the browser knows it should display the document using the HyperText Markup Language (HTML) [Ste95] formatting commands embedded in the document. On the other hand, a file with "image/gif" specifies that the incoming data stream is an image in the Graphics Interchange Format (GIF), a standard image format [LPJ+94].

4.1.1 The HyperText Markup Language (HTML)

The HyperText Markup Language (HTML) is derived from SGML [PC86], the Standard Generalized Markup Language, used in print publications as well as online publica-

$$\underbrace{\text{<A HREF=``http://www.cs.utk.edu/} \sim \text{lsi/lsi.html''>}}_{\text{Anchor and URL Address}} \quad \underbrace{\text{LSI}}_{\text{Hypertext Link}} \quad \text{</A>}$$

Figure 4.3: An HTML anchor. By activating the anchor associated with the word "LSI," the user is taken to the document specified in the anchor's URL.

tions. HTML consists of a variety of formatting commands that are embedded in a document to alter its appearance when it is displayed. However, the markup tags placed in a document merely suggest to the browser how a document should be displayed. The browser and the graphical capabilities of the platform on the which the browser is executing ultimately determine the final appearance of the document [LPJ+94].

HTML markup commands resemble the markup commands of other popular text formatting tools. Each markup command consists of two tags. The beginning tag tells the browser to render the text up to the ending tag in a special way. For example, the . . . markup commands in

Enter a keyword

tell the browser to emphasize the text "Enter a keyword." Since various browsers render emphasis differently, one browser may display the text in italics while another browser may display the text in boldface. A list of markup commands and a description of their effects can be found in [Ste95].

The anchor tag is particularly important in HTML. An anchor tag (Figure 4.3) allows an object to link to another object. When the link is activated, the browser accesses the URL associated with the anchor and loads the object. Both images and text can be enclosed by anchor tags, making them hypertext links to other documents, images, and sounds.

4.1.2 HTML Forms and the Common Gateway Interface (CGI)

Most of the time, information on the WWW flows in only one direction. When a user opens a URL, the browser contacts the WWW server specified in the first part of the URL and asks the server to send the object specified in the second part of the URL.

HTML provides a set of markup tags, called forms, that, together with the Common Gateway Interface (CGI), allow information to flow in both directions. The browser and the server hold a dialog where the browser passes some information to the server, the server reacts to the information, and then passes new information, usually an HTML document, back to the browser.

HTML forms are used to collect information from a user (such as a query) and submit it to the WWW server for processing. An HTML form is an HTML document that contains the <form>. . . </form> markup tags. Within the form, several types of markup tags allow the user to enter information and submit the form to the server for processing. Table 4.1 lists the HTML markup tags currently supported in forms. The HTML form must include a URL that not only specifies the HTTP server that will handle the transaction, but also the program or script that will be started by the server to handle the information submitted by the form [LPJ+94].

When an HTML form is submitted to the server, the browser packages the information contained in the form and sends it to the server. The server spawns the program specified in the URL and uses CGI to pass the information from the server to the program [LPJ+94].

The HTML form specifies how CGI should pass the information from the server to the spawned program. If the form specifies the GET method, the server, before starting the program, sets the REQUEST_METHOD environment variable to GET and sets the QUERY_STRING environment variable to the string of key-value pairs received from the browser. If the form specifies the POST method, the server sets the REQUEST_METHOD environment variable to POST and sets the CONTENT_LENGTH environment variable to the length of the string of key-value pairs received from the browser. Then, the server starts the program and passes the string of key-value pairs

Table 4.1: HTML markup tags currently supported in forms.

Markup Tag	**Example Use**
Text entry field	<input type="text" name="keywords" size="50">
Password field	<input type="password" name="passwd" size="50">
Checkbox	<input type="checkbox" name="queryTerm" value="1">
Radio Button	<input type="radio" name="newBook" value="3">
Hidden	<input type="hidden" name="queryString" value="parallel">
Scrollable text area	<textarea name="keywords" rows="4" cols="50">
Pop-up Menu	<select name="returnNumber"> <option>10 <option>20 </select>
Submit Button	<input type="submit" value="Submit Query">
Reset Button	<input type="reset" value="Reset">

through standard input. An example of a CGI-encoding string, passed to the spawned program using either the GET or POST method, and the key-value pairs it contains is shown in Figure 4.4. The POST method is preferred over the GET method since the lengths of environment variables are limited on some systems. Environment variables that exceed the limit may be truncated, causing a loss of information [LPJ+94].

When the server starts a program that is the target of a submitted request, the program must examine the REQUEST_METHOD environment variable to determine whether its should receive its input from the QUERY_STRING environment variable or standard input. After processing, the spawned program writes its output to standard output, which is captured by the server and sent to the browser. When the program exits, the server closes the connection to the browser and again listens for connections.

The HTTP server does not preserve any state information between connections from a browser. Each time a browser connects to the server, it is as if the browser and the server had never communicated before. If state information must be preserved between connections, the browser is responsible for storing the state information and transmitting it to the server each time the form is submitted. HTML provides the

bookNum=4&returnNumber=10&keyword=parallel
&keyword=distributed&keyword=computing

key	value
bookNum	4
returnNumber	10
keyword	parallel
keyword	distributed
keyword	computing

Figure 4.4: A CGI-encoded string and its corresponding key-value pairs. The keys act as symbolic names to reference the values.

HIDDEN markup tag within HTML forms to preserve state information. Values within the HIDDEN tag are not displayed, but are sent to the server when the form is submitted [Ste95].

4.2 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.

4.2.1 Features

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 2.3.5) to be seamlessly integrated with the rest of the interface.

4.2.2 A Look at the Interface

The LSI WWW interface consists of two HTML forms. The first form, entitled *LSI:Book*, allows the user to select a book to search. The second form, called *LSI:Query*, lets the user enter the text of a query and adjust the parameters that affect the search.

LSI:Book

Before performing a search, a user first must choose the book to search. The *LSI:Book* form (Figure 4.5) 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.

LSI:Query

The *LSI:Query* form actually consists of two similar, yet separate forms. The initial *LSI:Query* form (Figure 4.6) 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

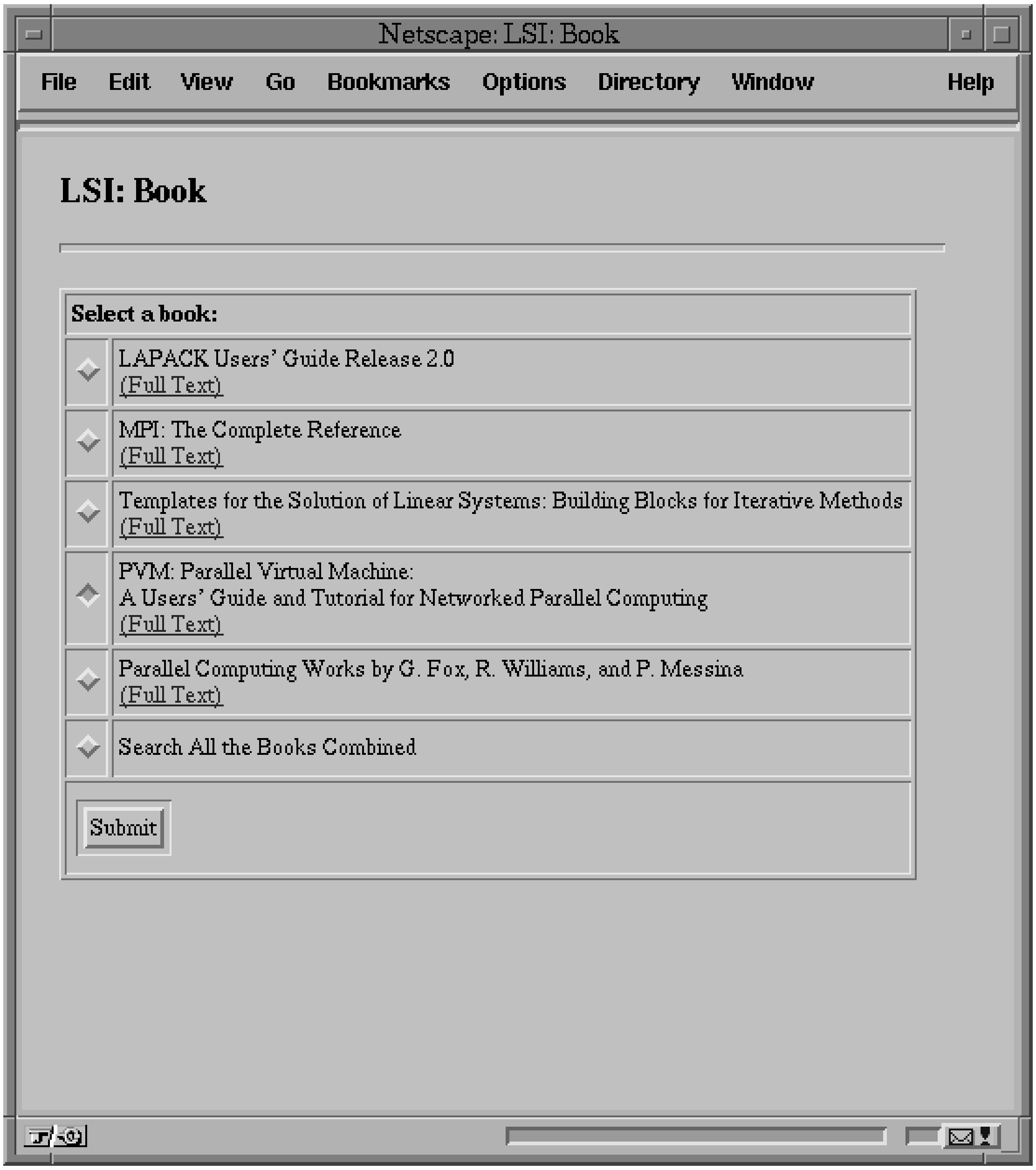

Figure 4.5: The *LSI:Book* form.

Figure 4.6: The initial *LSI:Query* form. The user has elected to find documents related to "fault tolerance" in the PVM User's Guide.

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 (split across Figures 4.7, 4.8, & 4.9), 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. 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.

The initial LSI:Query form

The initial *LSI:Query* form consists of five regions:

1. **The name of the document collection.** The name of the document collection is centered at the top of the form.

2. **A text box for user input.** The text box is used to collect keywords from the user. Keywords typed in the box are used to compose the query. Keywords may be separated by either punctuation or spaces.

3. **A slider to adjust the number of factors.** Ideally, the number of factors to use in a search would be continuously adjustable from the lowest possible setting (1) to the highest possible setting, which is determined during the preprocessing phase of LSI. However, because HTML does not provide a slider mechanism, a series of ten radio buttons are used to simulate a slider. The radio buttons provide discrete steps from the lowest number of factors (10% of the maximum number of factors set during the preprocessing phase of LSI) on the left to the highest number of factors (100% of the maximum number of factors) on the right. Each radio button represents a 10% increase in the number of factors from the previous button. Since only one radio button may be selected at a time, a single value for the number of factors will be passed to the server when the form is submitted.

   As discussed in Section 2.3.3, given a large enough number of factors, the original

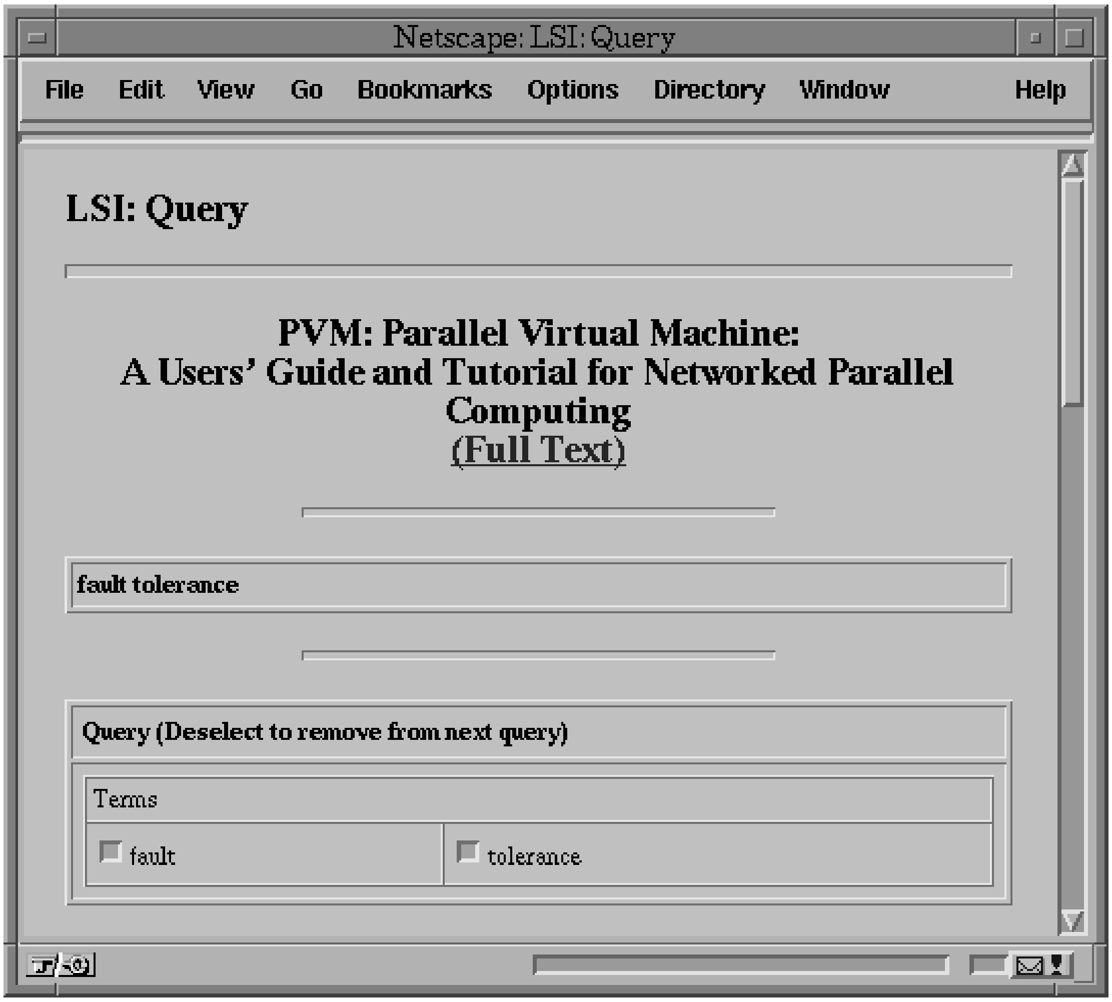

Figure 4.7: The subsequent *LSI:Query* form (first region). While searching for "fault tolerance," both "fault" and "tolerance" were found in the document collection.

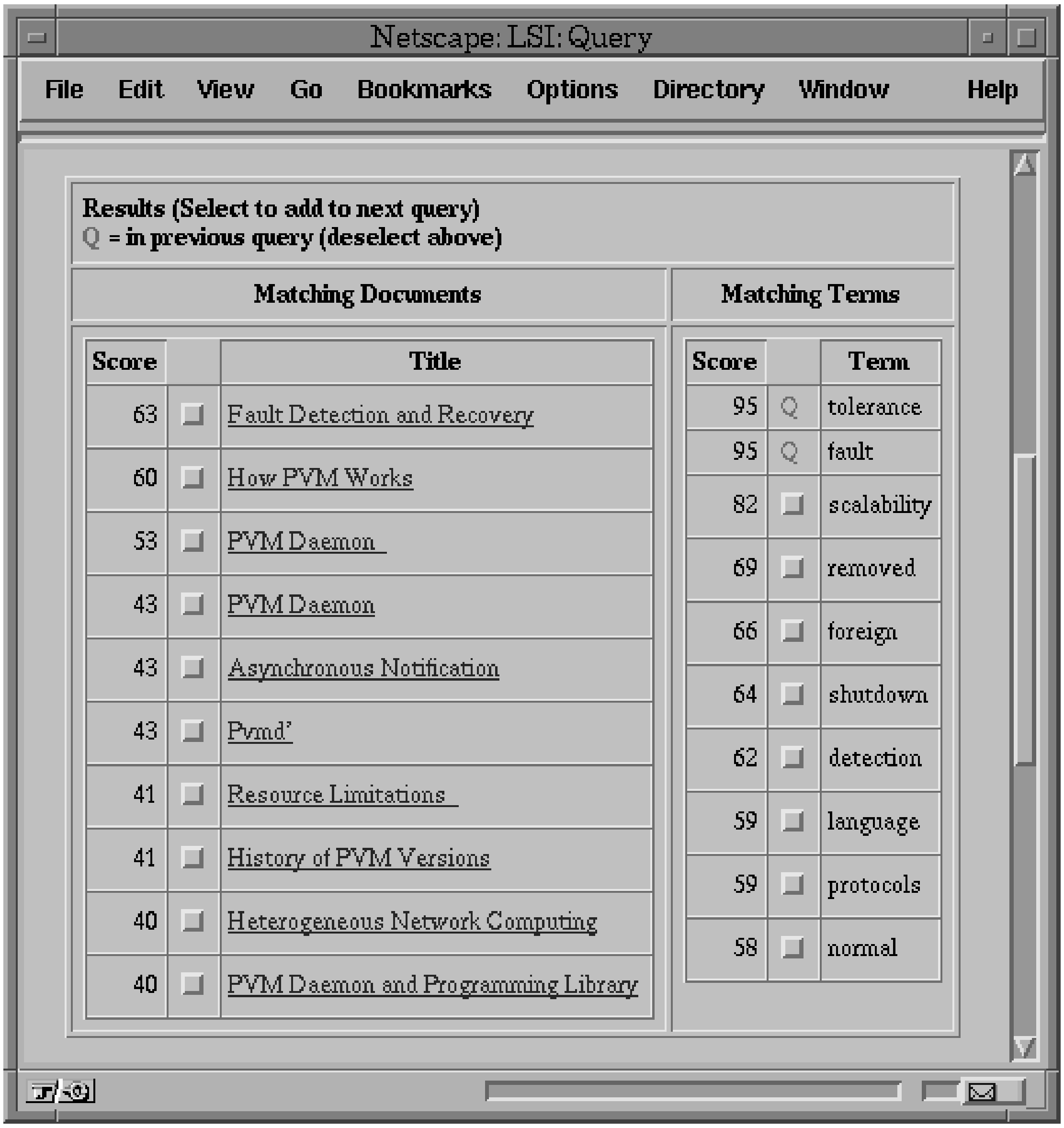

Figure 4.8: The subsequent *LSI:Query* form (second region). The terms and documents related to "fault tolerance." By selecting the checkbox next to a term or document, the term or document is added to the query on the next search. Both "fault" and "tolerance" lack checkboxes in the Matching Terms section since they were part of the previous query.

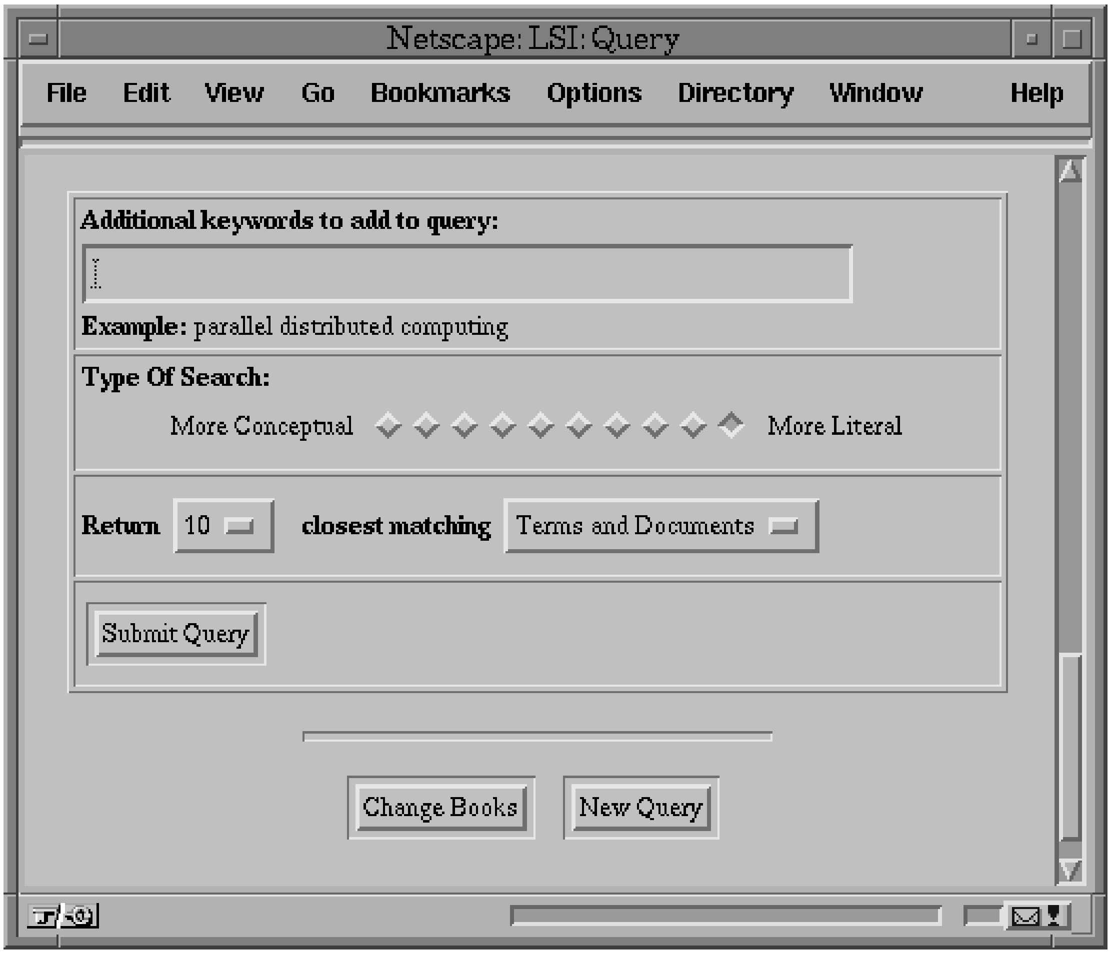

Figure 4.9: The subsequent *LSI:Query* form (final region). Additional keywords can be added to the next search. In addition, the user can adjust the various parameters of the search.

term-document matrix will be nearly reconstructed when computing the SVD. A fully reconstructed term-document matrix provides almost literal matching since a non-zero cosine between the vector representing the pseudo-document and the term and document vectors would occur only if the term existed in the query or the document contained a term in the query. The label "more literal" to the right of the radio buttons suggests that the highest-ranking terms and documents returned when a query is projected into the term-document space with a higher number of factors more literally matches the query since a higher-dimensional representation of the original term-document matrix is used. The label "more conceptual" to the left of the radio buttons signifies that projecting the query into the term-document space with fewer numbers of factors uses a lower-dimensional representation of the original term-document matrix, essentially collapsing dimensions and pushing previously separated terms and documents closer to each other, providing a less literal search.

4. **Selectors for the number and type of objects to be returned.** Selectors allow the user to choose whether to return the 10, 20, 30, 40, or 50 highest-ranked terms, documents, or both.

5. **The "Submit Query" button.** When pressed, the "Submit Query" button submits the contents of the form to the server for processing. The results of the search are returned in the subsequent *LSI:Query* form.

The subsequent LSI:Query form

When the initial *LSI:Query* form is submitted, the server performs the search based on the keywords entered in the text box and returns the results of the search as another, more complicated HTML form. Like the initial *LSI:Query* form, the subsequent *LSI:Query* form consists of five regions:

1. **The name of the document collection.** The name of the document collection is once again centered at the top of the form.

2. **The user input.** The keywords entered in the text box on the previous form are displayed as they were entered. The keywords that were not on the stoplist and were found in the document collection are highlighted.

3. **The keywords and documents used in the query.** Each keyword and document (if it was a relevance feedback search) used in the query is listed individually, accompanied by a selected checkbox. By deselecting the checkbox next to a term or document, that term or document will be removed from the query the next time the form is submitted.

4. **The results of the query.** Depending on whether the user chose to return terms, documents, or both, the table may contain a ranked list of terms, a ranked list of documents, or both lists. Both terms and documents are listed in order of decreasing score, the score being 100 times the cosine between the vector representing the pseudo-document and that representing the term or document. To add one of the ranked terms or documents to the next query, the user can select the checkbox next to the term or document. Terms and documents with a "Q" next to them instead of a checkbox were used in the previous search. They will automatically be used as part of the next query unless they are deselected in the third region.

   If a ranked list of documents is generated, the title of each document is a hypertext link to the text of the document. When the link is activated and the document is displayed, words that correspond to keywords used in the search are highlighted.

5. **The initial LSI:Query form.** The final region is similar to the initial *LSI:Query* form. It allows the user to add additional keywords to the query, adjust the number of factors, and set the number of terms, documents, or both to return to the user the next time the form is submitted.

4.3 Implementation

Both the *LSI:Book* and the *LSI:Query* HTML documents are created by programs executing under the Common Gateway Interface (CGI) (Section 4.1.2). In the following sections, the creation of each document is considered, and the interactions between their generating programs and the LSI++ Application Programming Interface is discussed. Also, a few auxiliary C++ classes, used mainly for input and output, are examined.

4.3.1 LSI:Book

The *LSI:Book* form is created by a program called *lsiBook*. On instantiation, *lsiBook* retrieves the information passed to it from CGI, extracting the values for the *newBook* and *bookNum* keys, if they are present. It then uses the *documentDatabase* class (Section 3.3.3) to iterate through the document database. For each entry in the document database, *lsiBook* prints the name of each book after a radio button whose value is the unique book number for that document collection as it appears in the document database. If *newBook* is negative and *bookNum* contains a number that matches the book number for an entry in the document database, the radio button for that book is set to be selected when the HTML form is displayed by the browser. If *newBook* is non-negative, the radio button for the first document collection is selected. When the *LSI:Book* form is submitted, the *newBook* variable is set to the number of the book that was selected by the user, and its value is passed to the program that creates *LSI:Query*.

4.3.2 LSI:Query

The *LSI:Query* form can be created in several different ways, depending on the configuration of the LSI search engine. 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

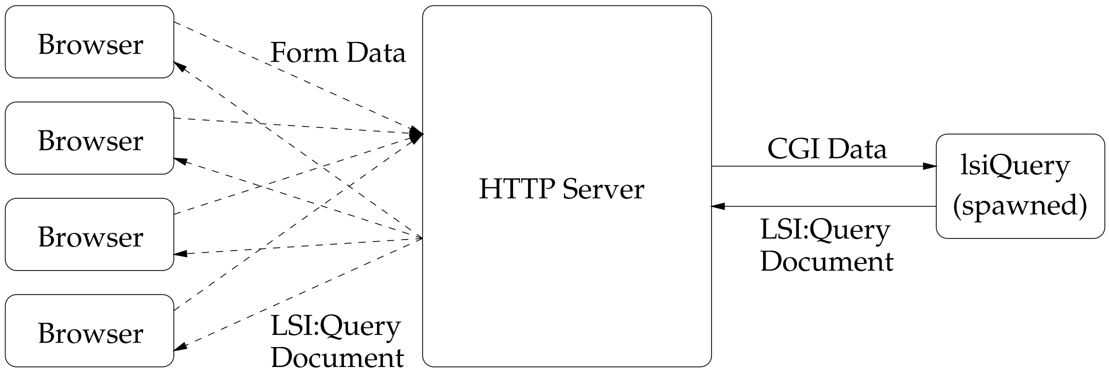

Figure 4.10: *LsiQuery* executing on the HTTP server. The HTTP server spawns *lsiQuery* on the same machine as the server, using CGI to pass the HTML form input to *lsiQuery*. The resulting *LSI:Query* document, created by *lsiQuery*, is relayed to the browser by the HTTP server. The solid lines between 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.

document to the server, who then passes it to the browser (Figure 4.10).

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 backend processors. The HTTP server, acting as a frontend, 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 backend 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 4.11).

Although the computation is done at different locations (in the first configuration, on the server itself; in the second configuration, on a remote backend processor), both search engines perform the same computation and produce equivalent *LSI:Query* forms. Since the configurations differ only in how they receive their input and write their output, only the production of the *LSI:Query* document, regardless of where the document is produced, is considered here.

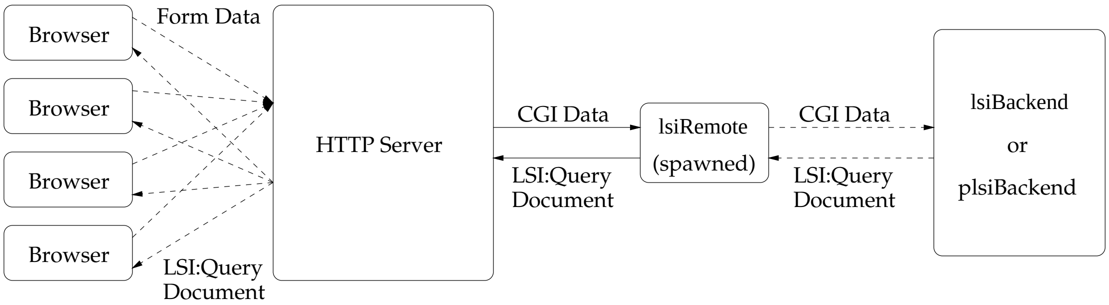

Figure 4.11: *LsiBackend* or *plsiBackend* executing on a remote machine. The HTTP server spawns *lsiRemote*. *LsiRemote* gathers the CGI data and passes it to the backend server. After the backend server processes the query, the resulting *LSI:Query* document is sent to the browser through *lsiRemote* and the HTTP server. The dashed lines represent a connection to a remote machine.

On instantiation (or, upon receiving a connection from an lsiRemote client), the program that produces the *LSI:Query* document accesses the CGI information passed to it, reading the CGI information into an internal data structure. It then extracts the *newBook*, *bookNum*, and *numFactors* values from the CGI input. If *newBook* is non-negative, the *LSI:Query* page is being accessed from the *LSI:Book* form. The initial *LSI:Query* page (Figure 4.6) is sent to the HTTP server and then sent to the browser. On the other hand, if *newBook* is negative, the CGI information contains a query for the book identified by the value of *bookNum* in the document database. The value of *numFactors* is sent to the query engine to control the number of factors used when processing the query.

A total of five keys are extracted from the CGI key-value pairs to perform the query:

- **keywords** – the string entered into the keywords text box by the user. The complete string of words is added to the query by calling the LSI++ *addTerm()* member function.

- **queryTerm** – two parts of the *LSI:Query* form contain terms accompanied by checkboxes. Any checkboxes selected when the form is submitted are referenced in the CGI input using the *queryTerm* key. Since several checkboxes may be selected, the *queryTerm* key may appear many times in the CGI input. Each *queryTerm* key is associated with the numerical identifier of the term displayed next to the checkbox. Each term whose checkbox is selected is added to the query by calling the LSI++ *addTerm()* member function, passing it the numerical identifier of the term.

- **queryDoc** – similarly, two parts of the *LSI:Query* form contain documents accompanied by checkboxes. When the form is submitted, documents whose checkboxes are selected are added to the query by calling the LSI++ *addDoc()* member function, passing it the document number of the selected document.

- **returnWhat** – the *returnWhat* key describes whether the search engine should return terms, documents, or both terms and documents. The value of the *returnWhat* key is passed to the *setReturnType()* member function of LSI++.

- **returnNumber** – the *returnNumber* key provides the number of terms, documents, or both to return as the result of the query. The value of the *returnNumber* key is passed to both the *setNumTermsToReturn()* and the *setNumDocsToReturn()* LSI++ member functions.

Once all the parameters for the query are set, the LSI++ *performQuery()* member function is called to compute the similarity between the pseudo-document representing the query and each term or document vector. When *performQuery()* is finished, a new HTML document is created. Producing the new HTML form consists of five steps, one step for each region of the document:

1. **Write the name of the document collection.** The name of the document collection is provided by the *getBookName()* member function of the LSI++ API.

2. **Write the user input.** A hidden CGI key, *queryString*, is used to store the input entered on previous queries. Write the words in *queryString*, highlighting each word that was indexed in the document collection. Also, write the input entered in the current query, again highlighting words actually used to perform the search.

The *wasFound()* member function determines whether a term was actually used to perform the search.

3. **Write the keywords and documents used in the query.** Iterate through the values of the *queryTerm* CGI key, looking up each term ID using the *getKeyword()* LSI++ member function to find the term associated with the term ID. For each term ID, write a checkbox and the term, associating the term ID with the checkbox. Similarly, iterate through the values of the *queryDoc* CGI key, writing a checkbox and the title of the document for each document ID. Given a document ID, the title of a document can be found by calling the LSI++ *getTitle()* member function. The hypertext link between the document title and the text of the document is established by creating an HTML anchor and specifying the document ID of that document. Since all the terms and documents were part of the query, set their checkboxes to be selected by default.

4. **Write the results of the query.** Depending on whether the user wishes terms, documents, or both terms and documents to be returned, one or two tables may be generated.

   To return a table of terms, iterate through the ranked list of terms created by *performQuery()*, calling *getResultTerm()* and writing the term's score (100 times the cosine between the vector representing the pseudo-document and the vector representing the term), a checkbox, and the term itself for each entry. If, given the term ID, a call to *isQueryTerm()* returns true, a checkbox for the term already exists in third region of the *LSI:Query* document. Instead of creating another checkbox, write a "Q" to signify that a checkbox for the term exists elsewhere.

   Similarly, to return a table of documents, iterate through the ranked list of documents by calling *getResultDoc()* and writing the document's score (100 times the cosine between the vector representing the pseudo-document and the vector representing the document), a checkbox, and the title of the document. If a call to *isQueryDoc()*, passing the document ID as a parameter, returns true, write a

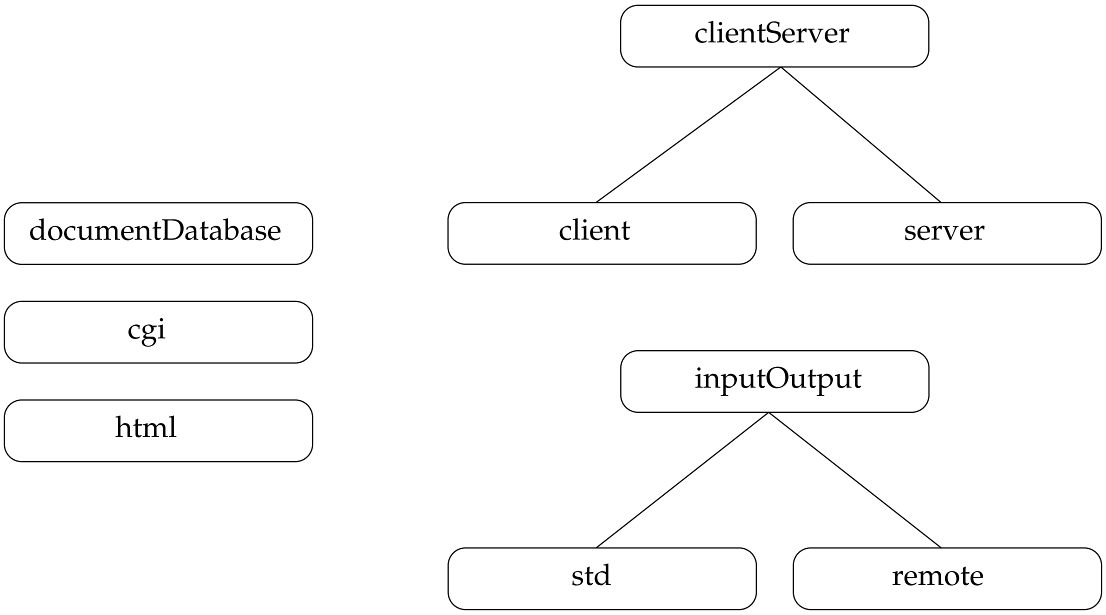

Figure 4.12: The WWW interface inheritance hierarchy.

"Q" in place of the checkbox to signify that a checkbox for the document exists elsewhere.

5. **Write the parameters of the query.** The final region is similar to the initial *LSI:Query* form, except any parameters changed from the default values on the previous *LSI:Query* form become the default values on the new *LSI:Query* form.

4.3.3 Supporting C++ Classes

Several C++ classes are used to produce the *LSI:Book* and *LSI:Query* HTML forms. The classes are used to access the document database, receive input from CGI, create the HTML documents themselves, and send the information back to the HTTP server.

Figure 4.12 shows the inheritance relationship among the classes. The *documentDatabase*, *cgi*, and *html* classes are independent. The *std* and *remote* classes inherit from the *inputOutput* base class while the *client* and *server* classes inherit from the *clientServer* class.

Figure 4.13 describes the interaction among the classes. *LsiBook* uses the *document-*

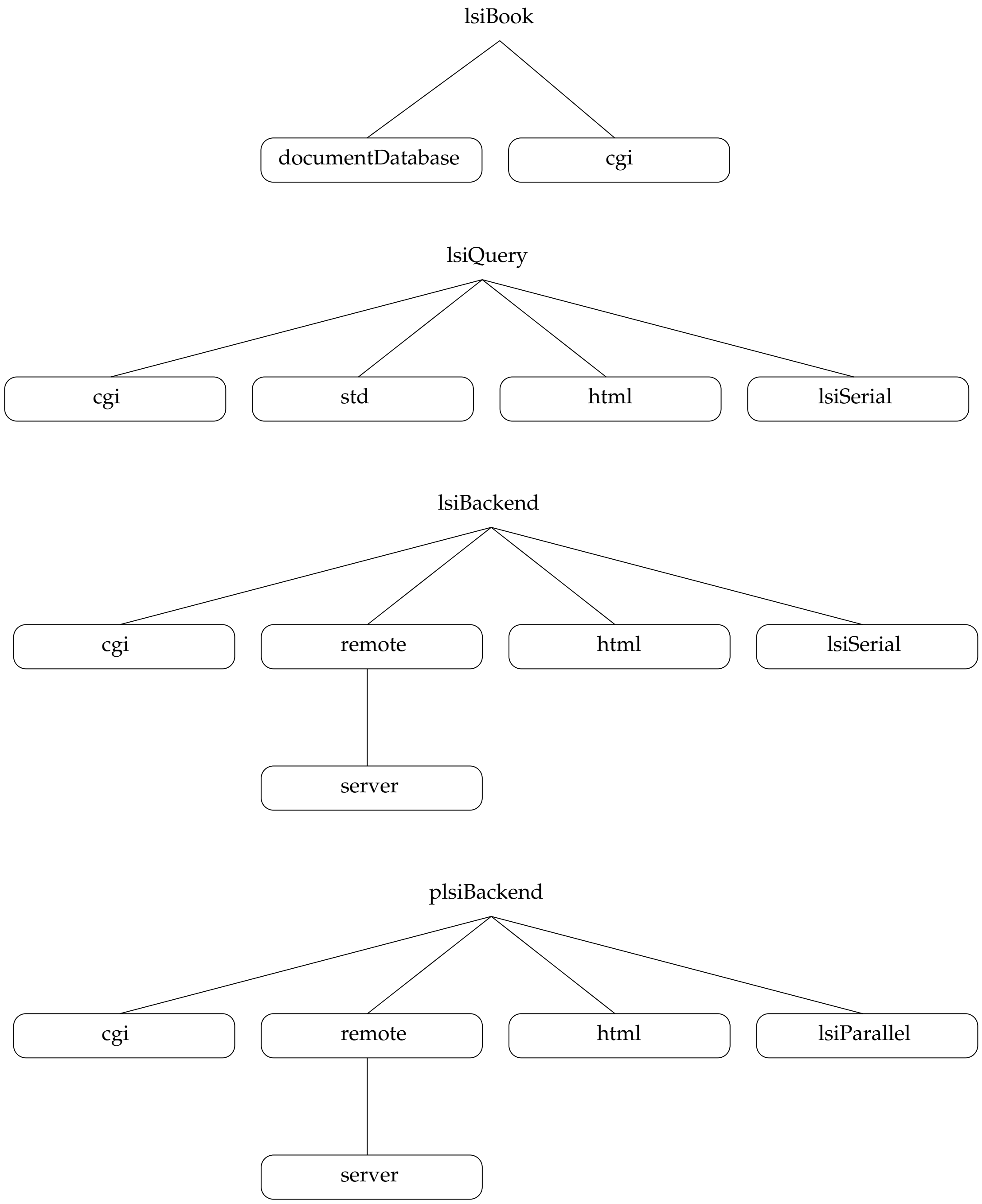

Figure 4.13: The WWW interface part-of hierarchy. Objects within boxes are C++ classes. *LsiBook* and *lsiQuery* are both directly started by the HTTP server. *LsiBackend* and *plsiBackend* are continuously-running, remote search engines. They receive their input from and send their output to *lsiRemote*, the client program started by the HTTP server.

Database and *cgi* classes to write the *LSI:Book* form. *LsiQuery*, the serial code that executes on the HTTP server, uses the *cgi*, *std*, *html*, and *lsiSerial* classes (from LSI++) to perform the search and write the resulting *LSI:Query* HTML form. *LsiBackend*, the serial code that executes as a continuously-running backend processor, uses the *cgi*, *remote*, *html*, and *lsiSerial* classes to read its input from *lsiRemote*, perform the search, and write the *LSI:Query* HTML form back to *lsiRemote*. *PlsiBackend*, a continuously-running backend processor that performs query matching in parallel, uses the *cgi*, *remote*, *html*, and *lsiParallel* classes to receive its input from *lsiRemote*, perform the search, and send the *LSI:Query* HTML form to *lsiRemote*. The *remote* class uses the services of the *server* class to communicate with the *lsiRemote* client.

The lsiSerial class

The *lsiSerial* class is used by both *lsiQuery* and *lsiBackend* to perform standard, sequential query matching. It is described in Section 3.3.3.

The lsiParallel class

The *lsiParallel* class is used by *plsiBackend* to perform the query matching in parallel. It is described in Section 3.3.3.

The documentDatabase class

The *documentDatabase* class is used by *lsiBook* to extract records from the *document-Database*. It is described in Section 3.3.3.

The cgi class

The *cgi* class is responsible for reading the input provided by the Common Gateway Interface and storing the key-value pairs in an internal data structure. Depending on the value of the REQUEST_METHOD environment variable, the *cgi* class either reads its input from the QUERY_STRING environment variable or standard input. Given a key, member functions allow other objects to request the value associated with that particular key. If a key has multiple values, iteration functions allow all the values associated with that key to be extracted.

The html class

The *html* class writes the *LSI:Query* HTML forms described in Section 4.2.2. Two member functions, *initialQuery()* and *subsequentQuery()*, write the initial *LSI:Query* HTML form and the subsequent *LSI:Query* HTML forms, respectively.

The inputOutput class

The *inputOutput* class serves as a base class for the *std* and *remote* classes. The *inputOutput* class provides virtual *print()* and *read()* member functions to allow any class to receive input and write output independently of whether the output is being written to standard output (*std*) or a socket (*remote*). It also overloads the extraction and insertion operators to allow objects to read from and write to the *inputOutput* class.

The *html* object writes its output to the *inputOutput* object. By writing to the *inputOutput* object, the *html* class can generically write output to either standard output or a socket without needing a separate version of the code for each type of output.

The std class

The *std* class is derived from the *inputOutput* class. If the *std* class is instantiated as the *inputOutput* object, any output produced by the *html* object is written to standard output. For efficiency reasons, the *std* class buffers output and writes to standard output only when the buffer is full or the *flush()* member function is called.

The remote class

The *remote* class is also derived from the *inputOutput* class. To send output to a socket instead of standard output, the *html* object writes to the *remote* object. Like the *std* class, the *remote* class buffers its output. When the buffer is filled or the *flush()* member function is called, the buffer is sent to the socket.

The clientServer class

The *clientServer* class serves as a base class for the *client* and *server* classes. It provides virtual send and receive member functions to allow an object containing either a *client* or a *server* object to communicate regardless of which object it contains.

The client class

The *client* class is derived from the *clientServer* class. It acts as the agent through which a program can communicate with a continuously-running server. Given the name of a machine and a port number, the *client* object establishes a connection with the server at that machine. The *client* class provides send and receive member functions to communicate with the server. In addition, it overloads the insertion and extraction operators to allow an object to read from and write to the *client* object.

The server object

The *server* class is also derived from the *clientServer* class. Given a port number, a *server* object binds to that port and listens for an incoming connection. When an incoming connection is detected, it accepts the connection and provides facilities for communicating with the client at the other end of the connection. Like the *client* class, the *server* class provides send and receive member functions to communicate with the client. It also overloads the insertion and extraction operators to allow an object to read from and write to the *server* object.

Chapter 5

Results and Conclusions

Both retrieval effectiveness (how reliably a system returns relevant documents) and execution efficiency (the average time required to search a document collection) of an interactive information retrieval system are very important when discussing the overall usefulness of the system [FBY92]. 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 2.3. 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++ has proven to be an efficient implementation of the LSI model described in Section 2.3. 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.

5.1 Computing Environment

All performance timings were obtained on a homogeneous collection of 24 dedicated 70 MHz Sun SPARCstation 5 computers, each with $97,968$K of main memory. Each machine was connected to both a 10 Mb/second and a 100 Mb/second Ethernet network. The 100 Mb/second 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/second network. In addition, communication between lsiRemote and both lsiBackend and plsiBackend (Section 4.3.2) took place over the 10 Mb/second Ethernet network.

5.2 Methodology

Four LSI search engines were tested to determine their relative execution efficiency while processing queries. Each search engine received its input from CGI (Section 4.1.2) and wrote an HTML document as output. The LSI search engines tested included:

lsiFinder – *lsiFinder* is written in the Perl programming language [WS91] and has approximately the same functionality as the original LSI WWW interface written by Loren Shih at Bellcore in 1994. It is essentially a wrapper around *mlsisearch*, a command-line interface to the original Bellcore implementation of LSI shown in Figure 3.2. Given a query, *lsiFinder* calls *mlsisearch* 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* was described in Section 4.3.2 and shown in Figure 4.10. *LsiQuery* receives the query via CGI, performs the search using LSI++, and writes the results as an HTML document.

lsiBackend – *lsiBackend* (described in Section 4.3.2 and shown in Figure 4.11) is a continuously-running, serial backend server that receives a query from *lsiRemote* (also described in Section 4.3.2), performs the search using LSI++, and passes the resulting HTML document back to *lsiRemote*.

Table 5.1: 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 [GBD+94]	PVM	2547	170	55
All of the following combined: PVM: Parallel Virtual Machine [GBD+94] LAPACK Users' Guide Release 2.0 [ABB+95] MPI: The Complete Reference [SOHL+96] Templates for the Solution of Linear Systems [BBC+94] Parallel Computing Works! [FWM94]	PLMTP	9842	1154	122
The Concise Columbia Encyclopedia [cce89]	CCE	31110	15486	104
Usenet News Archives	USENET	534493	100000	100

plsiBackend – *plsiBackend* is a continuously-running, distributed backend server (see Section 4.3.2). Like *lsiBackend*, *plsiBackend* receives its input from *lsiRemote*, performs the search using LSI++, and passes the resulting HTML document back to *lsiRemote*. It can use any number of processors and uses the data distribution shown in Figure 3.4.

To measure the performance of *lsiFinder*, *lsiQuery*, *lsiBackend*, and *plsiBackend*, twenty-five random queries were selected for each of four document collections (see Table 5.1 for a description of each document collection). Each query consisted of one to five terms known to be in the document collection and zero to four terms not in the document collection. In addition, approximately half of the queries were relevance feedback queries,

Table 5.2: A summary of the 25 random queries selected for each document collection.

Document Collection	Terms Not in Collection Min.	Max.	Ave.	Terms in Collection Min.	Max.	Ave.	Documents 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

with one to five documents added to the query. Each set of twenty-five queries was posed to the search engines twice, once to find related terms and again to find related documents. Table 5.2 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. Table 5.3 describes the time period measured for each search engine.

5.3 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*,

Table 5.3: The time period measured for each search engine.

Query Engine	Timing Began	Timing Ended
lsiFinder	upon instantiation of *lsiFinder*	after the HTML document was written
lsiQuery	upon instantiation of *lsiQuery*	after the HTML document was written
lsiBackend	upon instantiation of *lsiRemote*	after *lsiRemote* received the HTML document
plsiBackend	upon instantiation of *lsiRemote*	after *lsiRemote* received the HTML document

which is designed to execute directly on the HTTP server, was between 5 and 10 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*.

5.3.1 The Problems of Timing on the UTK CS 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,

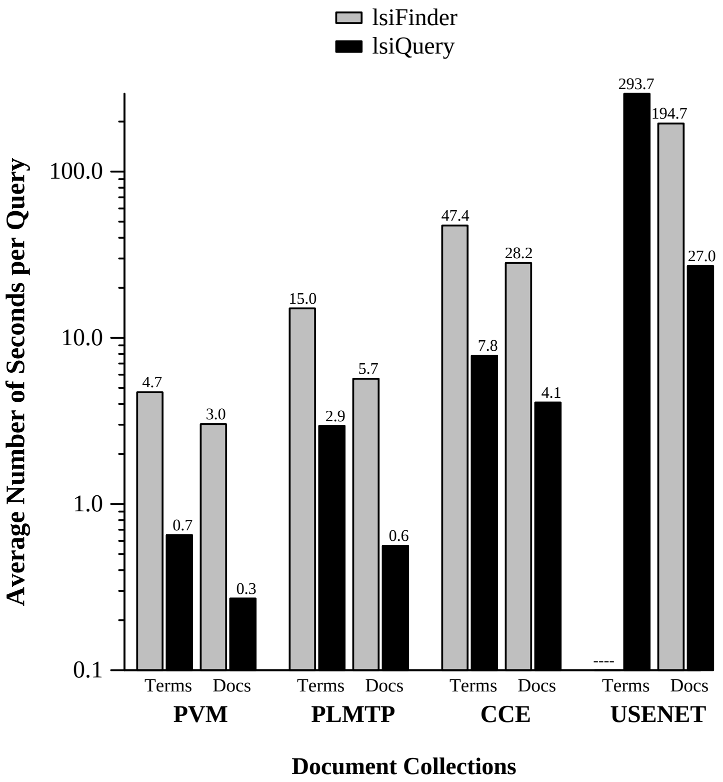

Figure 5.1: *lsiFinder* vs. *lsiQuery*. The average number of seconds required by *lsiFinder* and *lsiQuery* to search for both terms and documents in each of the four document collections.

the times reflect natural variances as the network became more and less congested.

5.3.2 LsiQuery versus LsiFinder

Figure 5.1 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 seconds 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 suggests 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, doesn't 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.

5.3.3 LsiBackend versus LsiFinder

The execution efficiency of *lsiBackend* was very similar to the efficiency of *lsiQuery* (see Figure 5.2). 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 5.3.1.

5.4 Distributed Results

PlsiBackend, the continuously-running, distributed LSI++ backend 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 not have a significant effect, but larger collections were able to take advantage of the greater parallelism and decrease the overall search time. Figure 5.3 provides the time required to initialize *plsiBackend* to serve each of the document collections on 2, 4, 8, 12, 16, 20, and 24 processors. The initialization time includes the one-time costs of starting each node and loading the term and document vectors into memory.

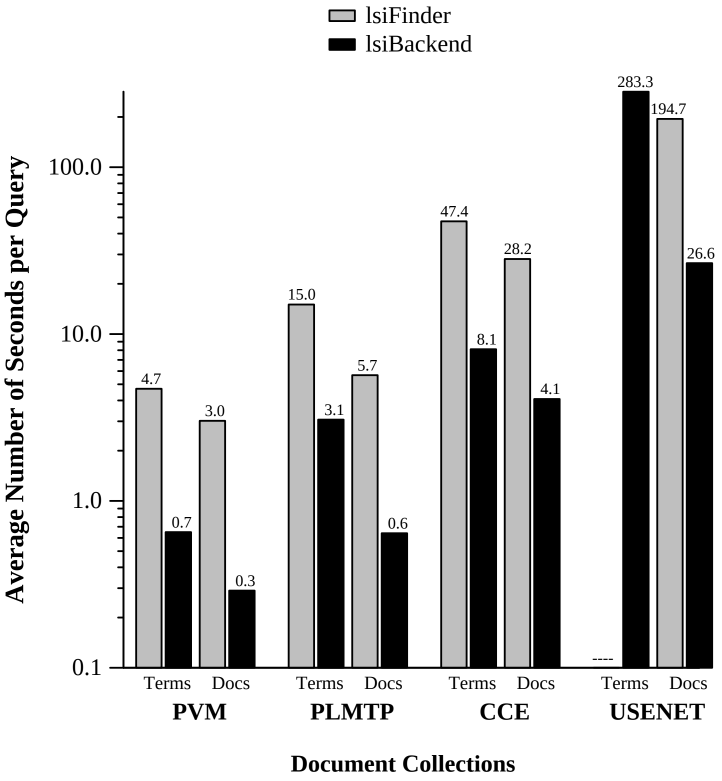

Figure 5.2: *lsiFinder* vs. *lsiBackend*. The average number of seconds required by *lsiFinder* and *lsiBackend* to search for related terms and documents in each of the four document collections.

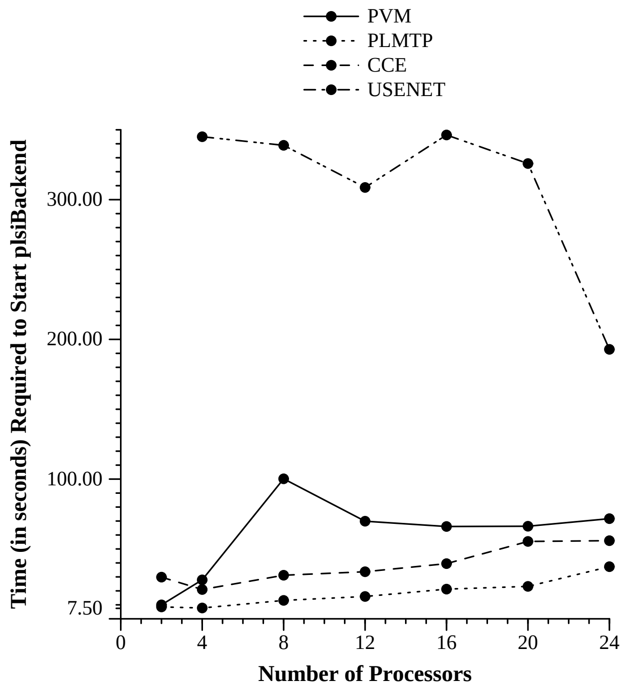

Figure 5.3: Starting *plsiBackend*. The time required to initialize *plsiBackend* in each of the configurations tested. The initialization time includes the one-time costs of starting each node and loading the term and document vectors into memory.

5.4.1 Execution Time

Figure 5.4 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 5.5 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 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.

5.4.2 Speedup, Efficiency, and Scalability

When computing the speedup and efficiency 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.

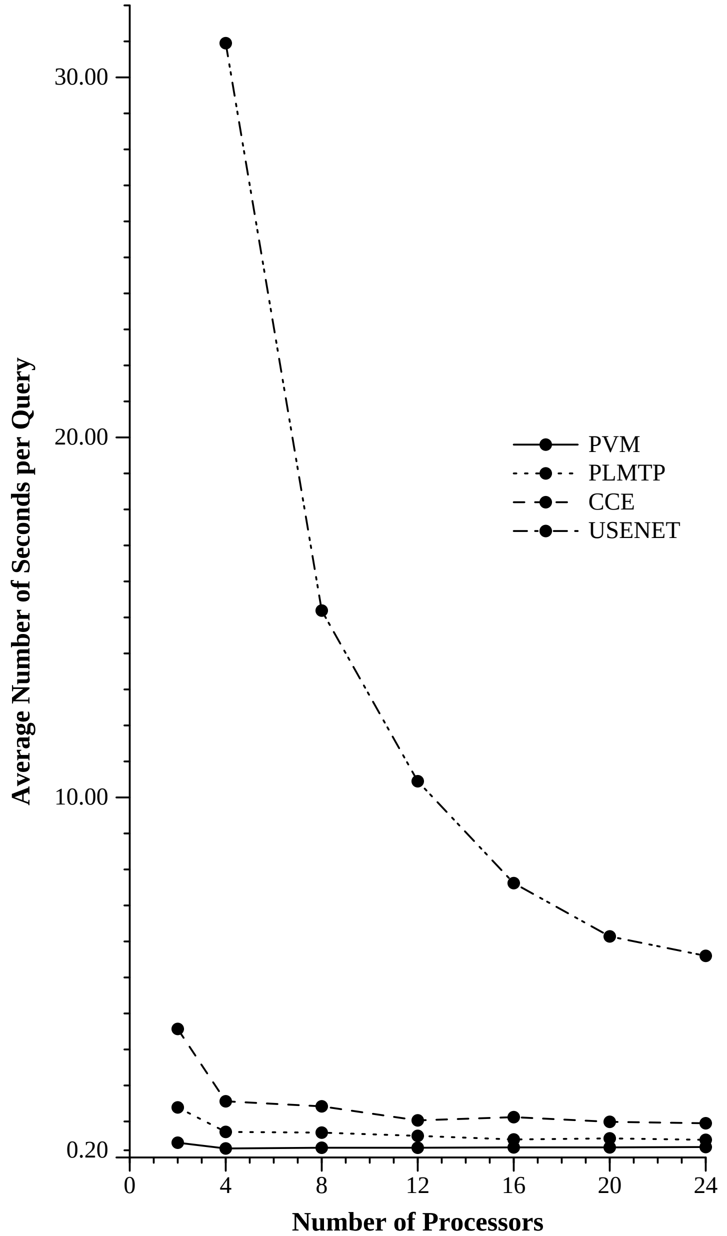

Figure 5.4: The average search time of *plsiBackend* (Terms). The average number of seconds required by *plsiBackend* to search for related terms in each document collection on 2, 4, 8, 12, 16, 20, and 24 processors.

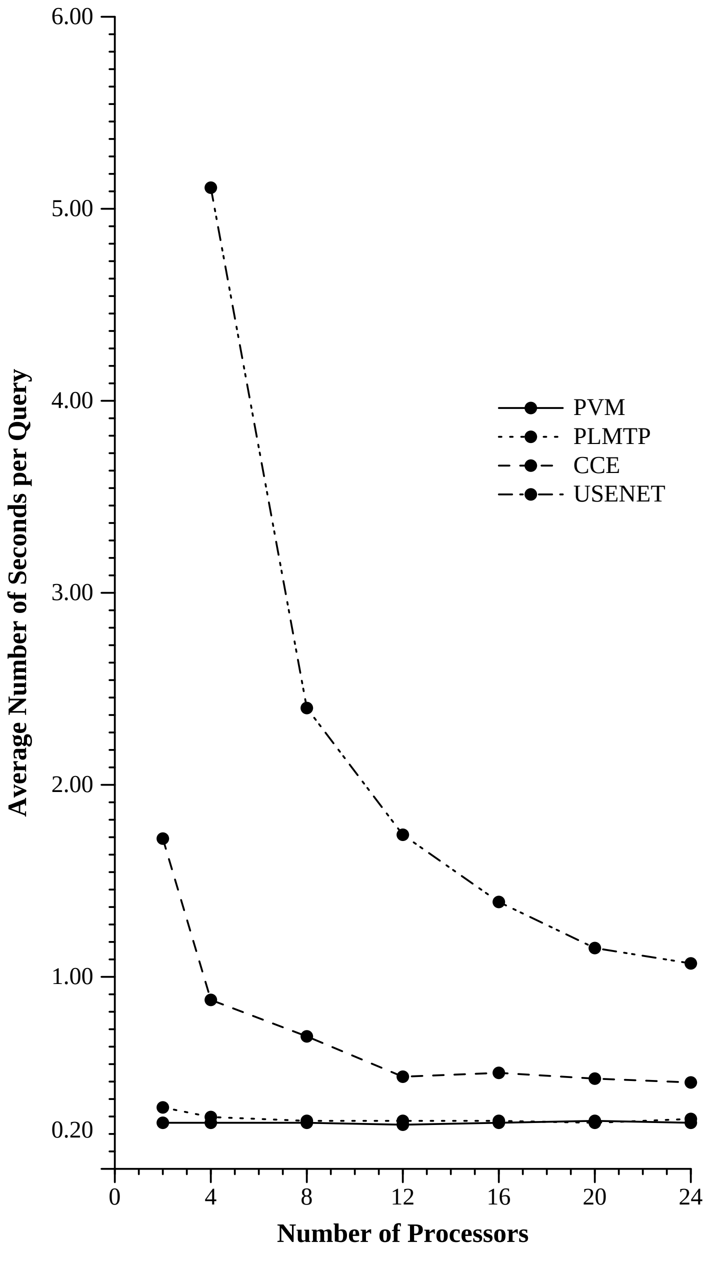

Figure 5.5: The average search time of *plsiBackend* (Documents). The average number of seconds required by *plsiBackend* to search for related documents in each document collection.

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 [KGGK94]. 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 5.6 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 5.7 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 document 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.

Efficiency

The efficiency of a parallel algorithm on a p-processor system is given by the ratio of the speedup using p processors, S_p, to the number of processors, p:

$$E_p = \frac{S_p}{p}.$$

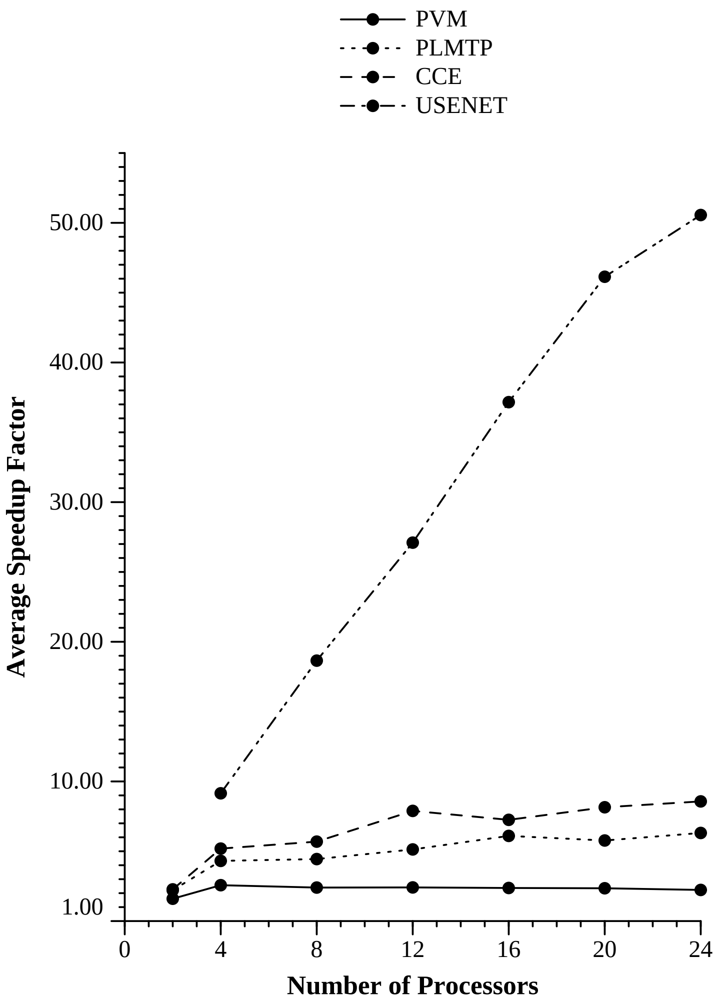

Figure 5.6: The average speedup of *plsiBackend* (Terms). The average speedup achieved by *plsiBackend* while searching for terms related to the test queries using 2, 4, 8, 12, 16, 20, and 24 processors.

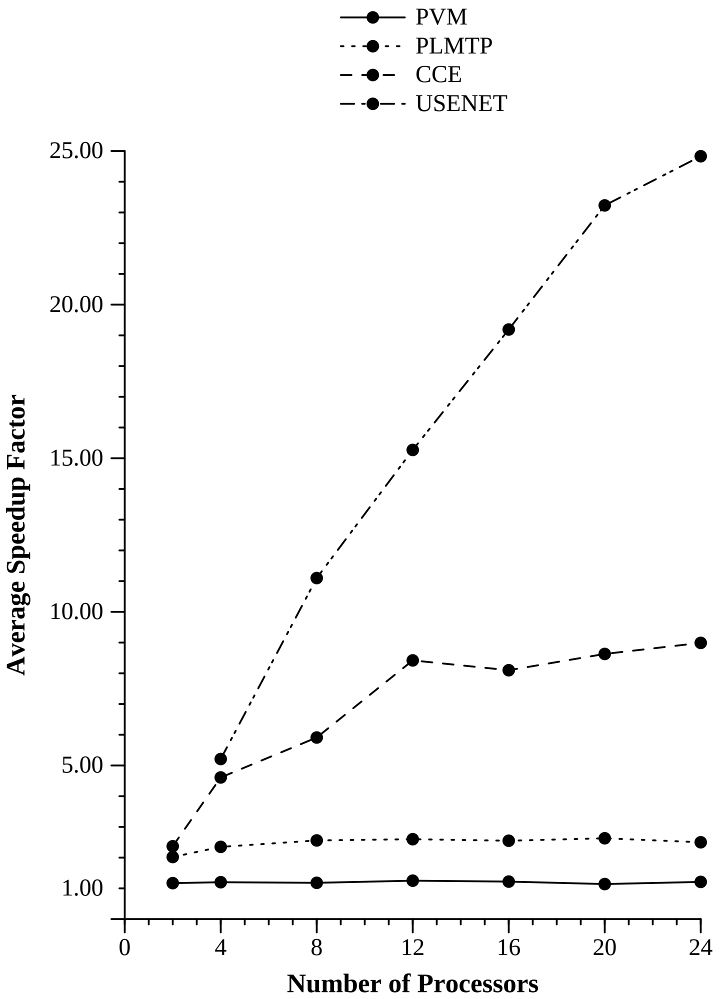

Figure 5.7: The average speedup of *plsiBackend* (Documents). The average speedup achieved by *plsiBackend* while searching for documents related to the test queries.

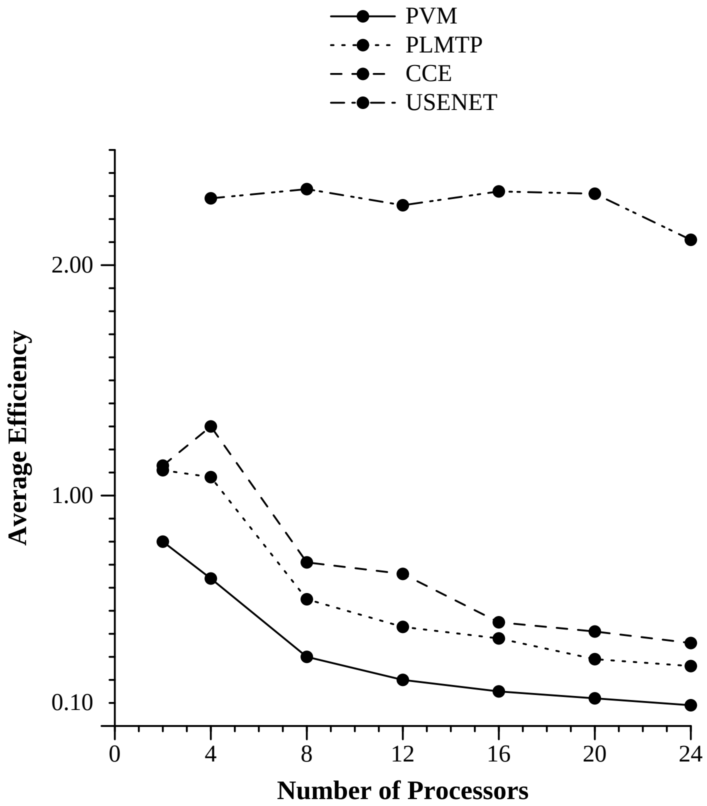

Figure 5.8: The average efficiency of *plsiBackend* (Terms). The average efficiency of *plsiBackend* when searching each of the document collections for related terms.

Efficiency normally varies between 0 and 1, with an efficiency of 1 representing an ideal parallel system [KGGK94].

Figures 5.8 and 5.9 give the average efficiency of *plsiBackend* when searching for related terms and documents, respectively. Because *lsiBackend* and *plsiBackend* differ in the locality of their data (*lsiBackend* loads each vector from disk while *plsiBackend*'s data is resident in memory), efficiencies greater than 1.0 are achieved.

The efficiency of *plsiBackend* decreases dramatically between 4 and 8 processors for the PVM, PLMTP, and CCE document collections. However, the CCE collection is reasonably efficient up to 16 processors. Because of its size, the USENET collection allows *plsiBackend* to remain efficient up to, and perhaps past, 24 processors.

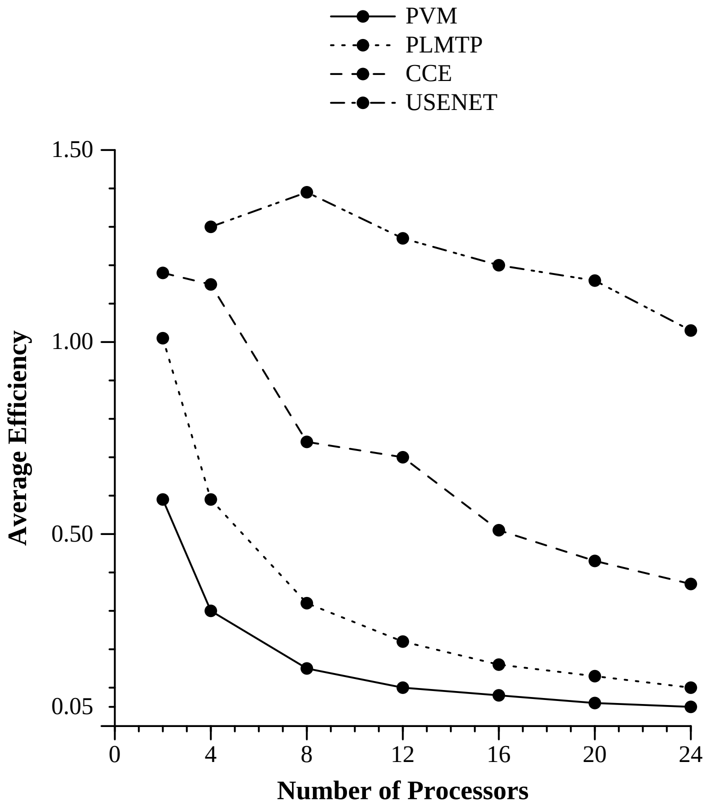

Figure 5.9: The average efficiency of *plsiBackend* (Documents). The average efficiency of *plsiBackend* when searching each document collection for documents related to the test queries.

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 is 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 both the speedup and efficiency 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 6-8 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 is distributed evenly across the nodes, *plsiBackend* should scale to almost any number of processors.

5.5 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 [O'B94], 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.

Even with updating and downdating, the memory and disk space required for an extremely large document collection (for example, several large collections spread across the World-Wide Web) would overwhelm a single machine or even a small set of machines a given site might dedicate to an LSI information retrieval system. In addition, the network bottlenecks associated with transporting a large number of documents to a single site for indexing would greatly impact users attempting to access the collection. A distributed information gathering and dissemination system, like the Harvest system developed at the University of Colorado [HSW96], would spread the network, disk space, and memory load to several different sites, allowing efficient gathering and storage of information as well as a reduction in the network traffic going to and from any one site. LSI may be able to use the Harvest system to gather documents and provide an LSI broker to search those documents. Even if LSI cannot be integrated into the Harvest system, a distributed gathering system is needed so large collections of documents can be indexed by LSI without consuming excessive resources at any one site.

5.6 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 [Fol90] and cross-language information retrieval [You94]. 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.

Bibliography

Bibliography

[ABB+95] 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*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, second edition, 1995.

[BB63] H. Borko and M. Bernick. Automatic document classification. *Journal of the Association for Computing Machinery*, 10:151–162, 1963.

[BBC+94] 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, Pennsylvania, 1994.

[BC87] N. Belkin and W. Croft. Retrieval techniques. In M. Williams, editor, *Annual Review of Information Science and Technology (ARIST)*, volume 22, chapter 4, pages 109–145. Elsevier Science Publishers B.V., 1987.

[BDO95] M. Berry, S. Dumais, and G. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

[Ber92] M. W. Berry. SVDPACK: A fortran-77 software library for the sparse singular value decomposition. Technical Report CS–92–159, University of Tennessee, Knoxville, TN, June 1992.

[cce89] *The Concise Columbia Encyclopedia*. Columbia University Press, New York, second edition, 1989.

[DDF+90] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[DF95] K. Decker and S. Focardi. Technology overview: A report on data mining. Technical Report CSCS TR-95-02, CSCS-ETH Swiss Scientific Computing Center, 1995.

[Dum91] S. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, 1991.

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[FD92] P. Foltz and S. Dumais. Personalized information delivery: An analysis of information filtering methods. *Communications of the ACM*, 35(12):51–60, December 1992.

[Fol90] P. Foltz. Using latent semantic indexing for information filtering. In *Proceedings of the ACM Conference on Office Information Systems (COIS)*, pages 40–47, 1990.

[Fox95] E. Fox. World-wide web and computer science technical reports. *Communications of the ACM*, 38(4):43–44, April 1995.

[Fre92] The Free Software Foundation. *The GNU G++ Class Library*, 1992.

[FWM94] G. Fox, R. Williams, and P. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, Massachusetts, 1994.

[GL89] G. Golub and C. Van Loan. *Matrix Computations*. Johns-Hopkins, Baltimore, Maryland, second edition, 1989.

[GR71] G. Golub and C. Reinsch. *Handbook for Automatic Computation II, Linear Algebra*. Springer-Verlag, New York, 1971.

[Har47] H. Harman. *Modern Factor Analysis*. University of Chicago Press, Chicago, Illinois, 1947.

[HL84] R. Harshman and M. Lundy. Data preprocessing and the extended PARAFAC model. In H. Law, Jr. C. Snyder, J. Hattie, and R. McDonald, editors, *Research Methods for Multimode Data Analysis*. Praeger, New York, 1984.

[HSW96] D. Hardy, M. Schwartz, and D. Wessels. Harvest user's manual. Technical Report CU-CS-743-94, University of Colorado, Boulder, CO, January 1996.

[Jon72] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[Kol79] M. Koll. WEIRD: An approach to concept-based information retrieval. *SIGIR Forum*, 13:32–50, 1979.

[LPJ+94] C. Liu, J. Peek, R. Jones, B. Buus, and A. Nye. *Managing Internet Information Services*. O'Reilly & Associates, Inc., Sebastopol, California, 1994.

[Luh53] H. Luhn. A new method of recording and searching information. *American Documentation*, 4(1):14–16, 1953.

[Luh57] H. Luhn. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal*, 1(4):309–317, October 1957.

[MR92] B. Myers and M. Rosson. Survey on user interface programming. In P. Bauersfeld, J. Bennett, and G. Lynch, editors, *CHI '92 Conference Proceedings*, pages 195–202, Monterey, California, May 1992. Association for Computing Machinery, Addison-Wesley Publishing Company, Inc.

[Nel] P. Nelson. *The GNU DBM Library*. The Free Software Foundation.

[O'B94] G. O'Brien. Information management tools for updating an SVD-encoded indexing scheme. Master's thesis, University of Tennessee, Knoxville, Tennessee, December 1994.

[PC86] JTC 1 / SC 18 Document Processing and Related Communication Committee. *ISO 8879:1986 Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*. International Organization for Standardization (ISO), Geneva, Switzerland, 1986.

[Sal71] G. Salton. A new comparison between conventional indexing (MEDLARS) and automatic text processing (SMART). Technical Report TR71-115, Cornell University, Ithaca, New York, 1971.

[SB90] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Science*, 41(4):288–297, 1990.

[SL65] G. Salton and M. Lesk. The SMART automatic document retrieval system – an illustration. *Communications of the ACM*, 8(6):391–398, June 1965.

[SOHL+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.

[Ste95] L. Stein. *How to Set Up and Maintain a World Wide Web Site*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[Wie95] G. Wiederhold. Digital libraries, value, and productivity. *Communications of the ACM*, 38(4):85–96, April 1995.

[WS91] L. Wall and R. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, California, 1991.

[Wur89] R. Wurman. *Information Anxiety*. Doubleday, New York, 1989.

[You94] P. Young. Cross-language information retrieval using latent semantic indexing. Master's thesis, University of Tennessee, Knoxville, Tennessee, December 1994.

Vita

Todd Alan Letsche was born September 8, 1969 in Cherokee, Iowa. He graduated from Cherokee Washington High School in 1988 and received a Bachelor of Arts degree from Wartburg College, Waverly, Iowa, in May 1992, majoring in Computer Science and Mathematics. During his third year at Wartburg, he spent eight months as an intern at International Business Machines, Inc., in Rochester, Minnesota, working on networking subsystems for their mid-range line of business computers. Following graduation from Wartburg, he spent a year interning at Cray Research, Inc., Eagan, Minnesota, in the Scientific Libraries group. In August 1993, he entered graduate school at the University of Tennessee – Knoxville, receiving, in August 1996, a Master of Science degree in Computer Science.