5-30-1997

An Information Retrieval System for Performing Hierarchical Document Clustering

Eric Hagen
Dartmouth College

An Information Retrieval System for Performing Hierarchical Document Clustering

Eric Hagen
Department of Computer Science
Dartmouth College

Abstract

 This thesis presents a system for web-based information retrieval that supports precise and informative post-query organization (automated document clustering by topic) to decrease real search time on the part of the user. Most existing Information Retrieval systems depend on the user to perform intelligent, specific queries with Boolean operators in order to minimize the set of returned documents. The user essentially must guess the appropriate keywords before performing the query. Other systems use a vector space model which is more suitable to performing the document similarity operations which permit hierarchical clustering of returned documents by topic. This allows "post query" refinement by the user. The system we propose is a hybrid beween these two systems, compatibile with the former, while providing the enhanced document organization permissable by the latter.

1. Introduction

   If a long lost friend ever decided to get in contact with me, he might try to find my home page on the World Wide Web. Let's assume that this long lost friend only remembers my nickname, Loki. This assumption is not unreasonable since even my best friends do not use my real name. A web search on the keyword "Loki" using the Infoseek search engine returns 6,808 documents. After milling through the first 100 documents, my long lost friend has probably given up his search. He doesn't bother (I would hope) to check the suggested related topic: "Siberian Husky". Instead, he remembers that my nickname comes from Norse mythology, and he decides to learn more about Loki, the Norse God of mischief. He narrows the search to "loki", "norse", "god" and "mischief". He is slightly comforted that only 547 documents match his query. He assumes that the suggested related topics, "Internet Security" and "Encryption" probably won't help much, so he starts going through pages. He finds everything from gaming organizations, to uninteresting personal pages belonging to individuals with my nickname. He even finds a page with a review of the latest Jim Carrey movie, and another dedicated to a mischievous pet ferret named Loki. He finally comes across a page that looks like it might have something to do with Norse mythology, and is even in English, but it turns out to be a page for a cult of Loki worshipers. Frustrated, my long lost friend turns off his computer and drives to the local library, where they are sure to have an encyclopedia with information about Loki.

   The information age has taken us by storm, and the resulting flood may be overwhelming. The internet is becoming more accessible, computers

are becoming faster, and memory is becoming cheaper and more efficient. Technology exists to keep up with the relentless flood information, but will we, as users, be able to do the same? Where is the profit in the ability to return thousands of documents pertaining to a keyword query in milliseconds if the user wastes hours sorting through all of the information to find what was really needed? Previous information retrieval systems have relied on the ability to generate highly specific queries with Boolean operations, but this is not sufficient. We saw how the highly specific (and even unusual) query about Loki, the Norse God of Mischief, generated hundreds of completely unrelated documents It is next to impossible to formulate highly specific queries with a small set of keywords and Boolean operators.

The task, then, is to create a system of information retrieval which will allow faster lookup time on the part of the *user* by summarizing document content and arranging them into topic clusters which are presented to the user in a visually meaningful manner. We often talk about algorithms that computers can be programmed to run in order to solve certain problems efficiently, but what algorithm does the user employ when analyzing the results? In the case of the aforementioned web search, the algorithm was as follows:

> Step 1) Generate the most specific, and relevant query to my needs as
> possible ("loki", "norse", "god" and "mischief").
> Step 2) Submit the query.

Step 3) Perform a linear search of the query results to find the

  sought after information.

The third step will obviously require the most time. Linear searching is very slow and inefficient, but if it is known that the data is organized in a particular way, the total search time may, in some cases, be improved. Consider a system which groups all of the documents into a finite number of topic clusters. Within each cluster are subclusters (subtopics), and so on. The user is then presented with a hierarchical topic summary which enables him to prune large sections of the retrieved documents. Such a system could significantly reduce the total search time required by the user. The new algorithm will become:

Step 1) Generate the most specific query as possible ("Loki", "Norse",

  "God" and "mischief").

Step 2) Submit the query.

Step 3) For every cluster of documents starting from the highest level:

  Step 3a) Check a few representitive documents in the cluster, or

  ask the system for a topic summary which can be

  performed by extracting the most frequent

  keywords in the set.

  Step 3b) If the documents do not appear to be leading in the

  direction sought after, eliminate the entire

  cluster of documents from the search.

Step 3c) If the documents do seem to relate to the search goal, select the cluster and repeat Step 3 until the information sought is obtained.

The second user algorithm allows entire clusters of documents to be systematically eliminated if a few representitive documents are "going nowhere," permitting the user to drill down to the right information considerably faster.

The clustering can be achieved in several ways; the simplest method involves measuring the similarity between all the documents in the query result and organizing them hierarchically based on these similarities. The Vector Space model of information retrieval allows structuring of information on document characteristics in such a way as to permit similarity measurements between any two documents.

The premise behind the Vector Space model is the reasonable assumption that two documents are similar if they contain the same words. Computationally, this translates into compiling statistical information on the frequency of keywords in a document collection to define the important dimensions in a keyword vector space. Each document is then represented as a vector in this space. The similarity between two documents is computed as a proximity measure between the corresponding vectors. The angle between the two documents has been shown to be a reliable measure of the statistical similaritiy of occurance of all keywords in both documents.

Post-query organization by hierarchical clustering is theoretically possible but inconvenient in practice. The majority of existing systems do not

store the information in a way that easily lends itself to this kind of computation. Existing data organizations must be modified or completely rearranged in order to make use of this powerful tool.

In this thesis we describe a model for an information retrieval system that allows fast, efficient clustering of query results. The model is based on a four layer design which supports modularity and implementation independance between layers. The model is designed specifically for storing the informational content of web-based documents. It has characteristics of both the Boolean and vector space models (described below), and provides the neccesary operations to perform the document clustering.

We begin with a report of previous work regarding the implemention of both Boolean and vector space based systems. We then present the four layer system design and the *desiderata* which were weighed during the design process. A detailed description of the responsibilities of each layer follows the specifications, as well as a specific description and performance analysis of the implementation of the system prototype, which is compared to other possible implementations. Finally, we justify the four layer design in the context of our *desiderata.*

2. Previous Work

The study of information retrieval is not new to computer science. The core technology has not changed significantly over the last twenty years. Most IR systems are based on inverted indices, which, for each keyword in the language, store a list of documents containing that keyword. The Vector

Space model provides a different way of looking at the same information, but is not used as often in practice. A vector space implementation stores a lists of (keyword, frequency) pairs for each document in the data set. This allows a set of documents to be visualized as points in an n dimentional space, where n is the total number of keywords in the language. Both systems have advantages and disadvantages. Our system will attempt to harness the advantages of both, without incurring the disadvantages of either. However, it is first essential to describe more carefully the two models and previous implementations of them.

2.1. *Inverted indices*

The majority of information retrieval systems available commercially are based on an inverted index [6, page 24]. This most likely results from a combination of simplicity and tradition (inverted files go back as far as 1890) [6, page 47]. The existence of so many systems implies considerable variation in implementation, but the core technology of an inverted index remains relatively unchanged (for over 100 years).

The minimum data structure of an inverted index consists of a dictionary of keywords, each containing a list of document identifiers corresponding to the files which contain that keyword. Space requirements and lookup times are obviously implementation dependent. Assuming a simple array of document lists, indexed by a keyword identifier, it is possible to find the list of all documents containing a certain keyword in constant time. If we assume a conversationally dismal world with just three words

("cat", "Siamese" and "Burmese") and two documents (feline.1 and feline.2), we can create an inverted index similar to the following:

```
feline.1 = Burmese cats are not Siamese cats.
feline.2 = Siamese cats are cats.


keywords    | document, frequency pairs
            |
cat         | (feline.1, 2), (feline.2, 2)
Siamese     | (feline.1, 1), (feline.2, 1)
Burmese     | (feline.1, 1)
            |
```

Figure 1: Example of an Inverted Index

The performance of Information Retrieval systems is highly dependent on the content of a collection and is therefore difficult to evaluate objectively. To help evaluate the performance of these systems, the Information Retrieval community has developed two complementary measures: "recall" and "precision". Recall is defined as the fraction of relevant documents in the data set which are returned as results to a given query. Precision is defined as the fraction of documents in the returned set which are actually relevant to the query. In mathematical terms:

Recall = relevant documents returned / all relevant documents   *(2.1.a)*

Precision = relevant documents returned / all returned documents   *(2.1.b)*

The relationship between recall and precision in information retrieval is analogous in many ways to the relationship between space and time found in certain other computer science models. Just as time requirements for certain algorithms cannot be improved beyond a point without sacrificing more memory space, improvements to either recall or precision are typically made at the expensive of the other.

In terms of recall, one would expect the result from a search with an inverted index to contain all documents in which the keyword may be found. However, it is not a given that it will return *all relevant* documents. Consider, for example, two very similar documents about soft drinks. One was written by a "pop" drinker, and another by a "soda" drinker. Although these documents may be identical in every aspect except for the replacement of a single word, an inverted index will not return the "soda" document if a search is performed on "pop".

In terms of precision, one would expect the result from a search with an inverted index to contain many files that have nothing to do with what the user actually wants (though they contain the particular keyword), making precision relatively low.

Various enhancements have been proposed to improve the accuracy of inverted index queries. Most of these enhancements are "labor intensive"; that is, they ultimately require the user to be more specific. One such

improvement is the ability to create sets of documents corresponding to an individual keyword and then to manipulate those sets using Boolean logic.

The DIALOG system developed by Lockheed in 1980 allows the user to select various keywords, and returns the document list corresponding to that keyword in the form of a set of documents with a set identification number [6, page 30]. Sets may then be combined in various ways according to Boolean logic. For example, if the user decided to combine sets A and B with the Boolean AND operator, (A AND B), the result would be the intersection of the two sets (e.g., those documents containing both keywords A and B) [6, page 31]. It is also possible to combine sets using an OR operation; the resulting set would contain the union of the two sets (those documents containing either keyword A or keyword B). The DIALOG system also provides the Boolean NOT operation. The ability to perform set operations can increase both precision and recall (though potentially at the expense of the other). Consider the documents referring to soft drinks described above. Requesting the query "pop OR soda" has the effect of increasing recall, as both document sets will be merged. Precision, however, may decrease, since the new document set will contain files unrelated to both keywords. The query request "pop AND soda" will increase precision since any file containing both keywords will be more likely to refer to soft drinks. At the same time, recall may decrease since documents with only one keyword (pop or soda) are sure to exist within each set.

The DIALOG system permits adjacency operations to further increase the specificity of queries. By using the "ADJ" command, a query can be formulated to return all documents containing two keywords within a

specified distance from each other [6, page 32]. This allows the user to express a relationship between words within a query. For example, specifying a query with the words "Burmese" and "cat" with an adjacency factor of one would increase precision by eliminating all documents dealing with Burmese things unless they happen to be cats, as well as all documents dealing with cats unless they happen to be Burmese.

The STAIRS system developed by IBM attempted to present query results to the user in a more efficient manner by storing the frequency of occurrence of every keyword in each document and then sorting the query results based on various user defined attributes such as: the frequency of the term in the document, the frequency of the term in the retrieved set, or the number of documents in the retrieved set in which the term occurs [6, page 57]. while this ranking system does not improve precision or recall, it does provide a less random starting point from which to begin a search; potentially reducing real search time on the part of the user.

More recent text retrieval systems such as the Context Option for the Oracle 7 database seek to increase precision and provide a means of document organization by attempting to parse the language of the text in order to guess what the document is about. H.L. Mencken once said that for every problem there exists a solution that is simple, elegant and wrong. I would propose the following ammendment to Mencken's observation: for every problem, there also exists a solution which is monolithic, inelegant and wrong. What the Oracle text retrieval system gains in precision and document organization, it loses in elegance. We will show that it is not unnecessary to attempt such

complex and unreliable semantic analyses to categorize documents by context, but we will require a different model to do so.

2.2. *The Vector Space Model*

The Vector Space Model of Information retrieval provides an alternative to the Boolean model which allows more accurate automatic document classification. In linear programming terms, the Vector Space representation can be described as the "dual" of the Inverted Index. Instead of storing a list of documents and frequencies for each keyword, as in the Inverted Index, we store a list of keywords and their frequency for each document. Thus every document becomes a vector in n dimensional space where n is the number of keywords in the language. Returning to our previous example:

Figure 2: Example of a vector space for 3 key words.
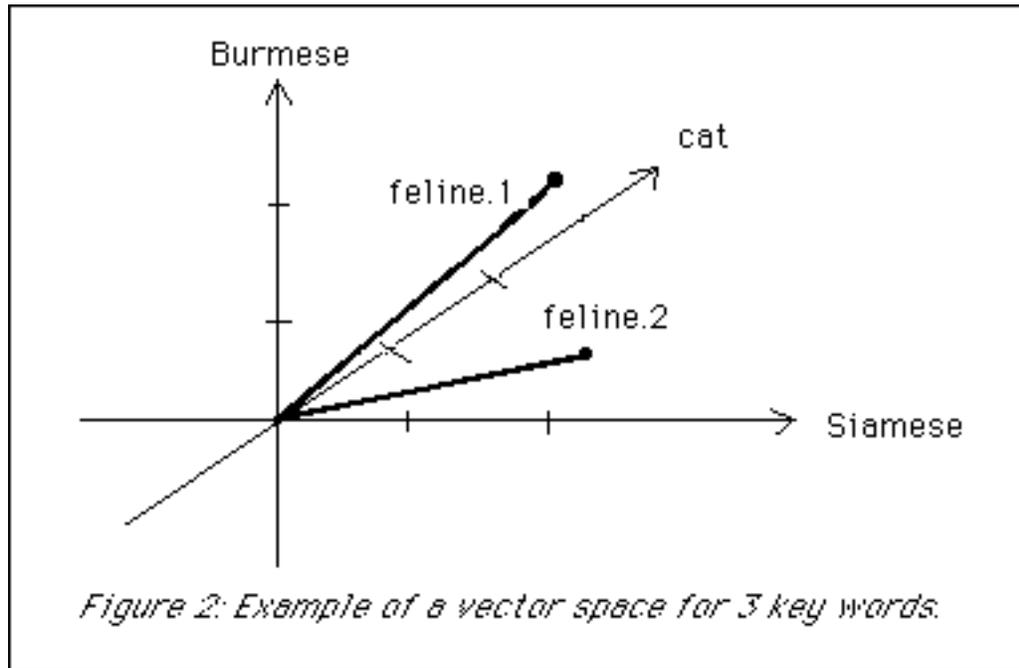
The Vector Space Model is based on the assumption that similar documents will be represented by similar vectors in the *n*-dimensional vector space. In particular, similar documents are expected to have small angles between their corresponding vectors. The cosine of the angle between two document vectors provides a simple scheme for measuring document similarity in a range from zero to one.[1] The cosine correlation is computed as follows:

$$Cosine(\ a,b) = Sum(k= 1,\ t,\ freq(a[k])\ *freq(b[k])\ / \qquad (2.2.a)$$

$$Sqrt(Sum(k = 1,\ t,\ freq(a[k]))\ *$$

$$Sum(k = 1,\ t,\ freq(b[k]))\ )$$

---

[1]A cosine of zero implies that the documents are "statistically" completely dissimilar; a cosine of one implies that the documents are "statistically" identical.

where *a* and *b* are document vectors, *k* is the total number of keywords in the language and freq(a[k]]) refers to the frequency of keyword *k* in document vector *a*.

A query may now be performed by creating a vector containing all of the words in the query, and computing the cosine of the angles between the query vector and every document vector in the database. Those documents whose cosine correlations are "high" with respect to the query will be returned, while those documents whose cosine correlations are "low" will not. A cosine correlation is deemed "high" if it exceeds a specified threshold parameter.

Computing the cosine of the angle between a query vector and every document vector in the database can be a time consuming process (on the order of $k*n$, where *k* is the average length of the document vectors, or the number of keywords in the language, and *n* is the total number of documents in the data set). In theory, it is possible to store the vector information as a tree for which each node points to a certain number of children representing clusters of documents that are sufficiently similar (based on a given threshold) to the centroid vector of the cluster. However, in practice this "proximity tree" indexing scheme is not implemented.

Recall and precision for vector space systems depend on several variables definable by the programmer, as well as faith in the value of statistical analysis to predict document similarity. Modification of the threshold for vector correlation may influence recall and precision. Low thresholds yield good recall but decrease precision by returning more

documents which are potentially irrelevant. Conversely, high thresholds yield good precision at the expense of decreasing recall by not returning some potentially relevant documents. Optimal query results are obtained by choosing the threshold parameter appropriately -- neither too high, nor too low.

The SMART system is perhaps the most widely known experimental information retrieval system based on the Vector Space model. Though various implementations are available, the features unique to the SMART system are as follows: completely automatic indexing of documents, document organization based on statistical similarity, document retrieval and ranking based on vector/query similarity, and automatic procedures for improving search results with relevance feedback [6, pages 120-121].

The SIRE system (Syracuse Information Retrieval Experiment) is another variation on a vector space model which combines the ability to perform Boolean operations (as with inverted indices) and the ability to automatically classify documents (similar to the SMART system). Adding Boolean operations to the vector space model potentially increases both precision and recall, since the user can be more specific in the query generation.

3. The System Design

Our web-based Information Retrieval system was designed with several specifications in mind. The first and foremost was flexibility. We wanted to be able to support different methods for data organization at the

lowest level, and user visualization at the highest. Given the web-based nature of the project, we wanted a system simple and intuitive enough for a global audience. Other considerations included: compatibility, space/time efficiency, maintainability, elegance, automation and effective multi-user capability. The principle inspiration behind the design comes from the layered organization of the SMART system. Our resulting system consists of four major layers: A shell layer for processing input, a control layer for organizing and processing queries and other operations, an interface layer for indexing the database of information, and a database layer for storing all the data. Each layer has its own modules to facilitate its operations, which are described below.

3.1 Motivation for the Layered Design

Before we begin to describe the information retrieval design for this project, it is appropriate to discuss the specifications and requirements that were considered important at the onset of the project. The primary requirement was that the system be capable of performing the cosine correlation computation of document vectors *efficiently* so as to allow the construction of hierarchical clusterings of query results. Since this was the principle computational requirement, and the best way to implement it was not known at the onset of the project, there was considerable room for "artistic license" in the design process. It seemed obvious that the experimental nature of the project implied experimentation with the actual implementation itself. The task then was to create a system largely implementation independent. We could then make major structural

changes in terms of data representation without affecting the rest of the program. For example, if we initially wanted to implement a hybrid system with features of both inverted indices and a vector space, but later chose to implement a pure vector space model, we did not want to be forced to make major changes in the code. Likewise, if we wanted to test the system using various different data structures to store the document information, we would like to be able to do that easily.

3.2. *Desiderata*

In designing the layered architecture for our system, we wished to balance the following important properties.

- Compatibility:

> The core technology behind information retrieval has not changed in many years. There are hundreds of existing systems with thousands of gigabytes of information. Any new information retrieval system incapable of using that information would be useless. The sad reality is that the system would have to be able to handle information stored in an inverted index gracefully.

- Space/Time:

> People have short attention spans. We wanted our system to be able to process a large number of queries "quickly" in terms of real time for the user. The constant lookup time for

most implementations of inverted indices was very attractive. We also needed to strike a balance between speed and storage space; which, given the monumental growth in the amount of information becoming available, would realistically be biased towards efficient use of memory at the possible expense of incredible speed. Some implementations of the SMART system actually store the data in two forms (an inverted file and a list of document vectors). We wanted to avoid having two copies of the same data if at all possible.

- Maintainability:

    We wanted a system that was easy to debug, to fine tune, and to which we could easily add new functionality. We also wanted a system that can survive a massive surgery, should we choose to go in a completely different direction.

- Elegance:

    Looking good has no quitting time. A simple, elegant design can accommodate the high demands on maintainability. It allows others to quickly learn the system and to continue the work should the makeup of the team change for any reason. An object oriented system seemed appropriate to allow the required elegance and maintainability.

- Automation:

> We needed a system capable of automatically indexing large numbers of documents quickly. We also wanted to automatically convert existing data from other systems to a form readable by ours. Given the fact that our system will be used primarily for web-based documents, we also needed a system that could automatically maintain itself by updating indexing information on pages when they are changed or deleted.

- Multi-user:

> Since the system would ultimately be the server end of a web-based application, it needed to handle multiple queries gracefully while "simultaneously" indexing new documents.

3.3. The Resulting System Design

The resulting design given these (sometimes conflicting) considerations is a layered system taking its inspiration from the glory days of the OSI networking model, while supporting control features similar to modern operating systems as well. Each layer interacts with those around it, but the implementation of the other layers remains masked. The topmost layer (Shell Layer) is similar to an operating system's shell, processing input and calling the next layer, (the Control Layer) which then organizes all the queries into jobs and processes them one at a time. The Control Layer makes a lookup request from the Interface Layer which converts the query string

into an integer keyword identification number and performs a lookup request from the Database Layer. The Database Layer finds the relevant documents and returns them to the Interface Layer, which returns the results in a format readable by the Control Layer. Here follows a pictorial representation of the design, and a detailed, top down description of each major layer.
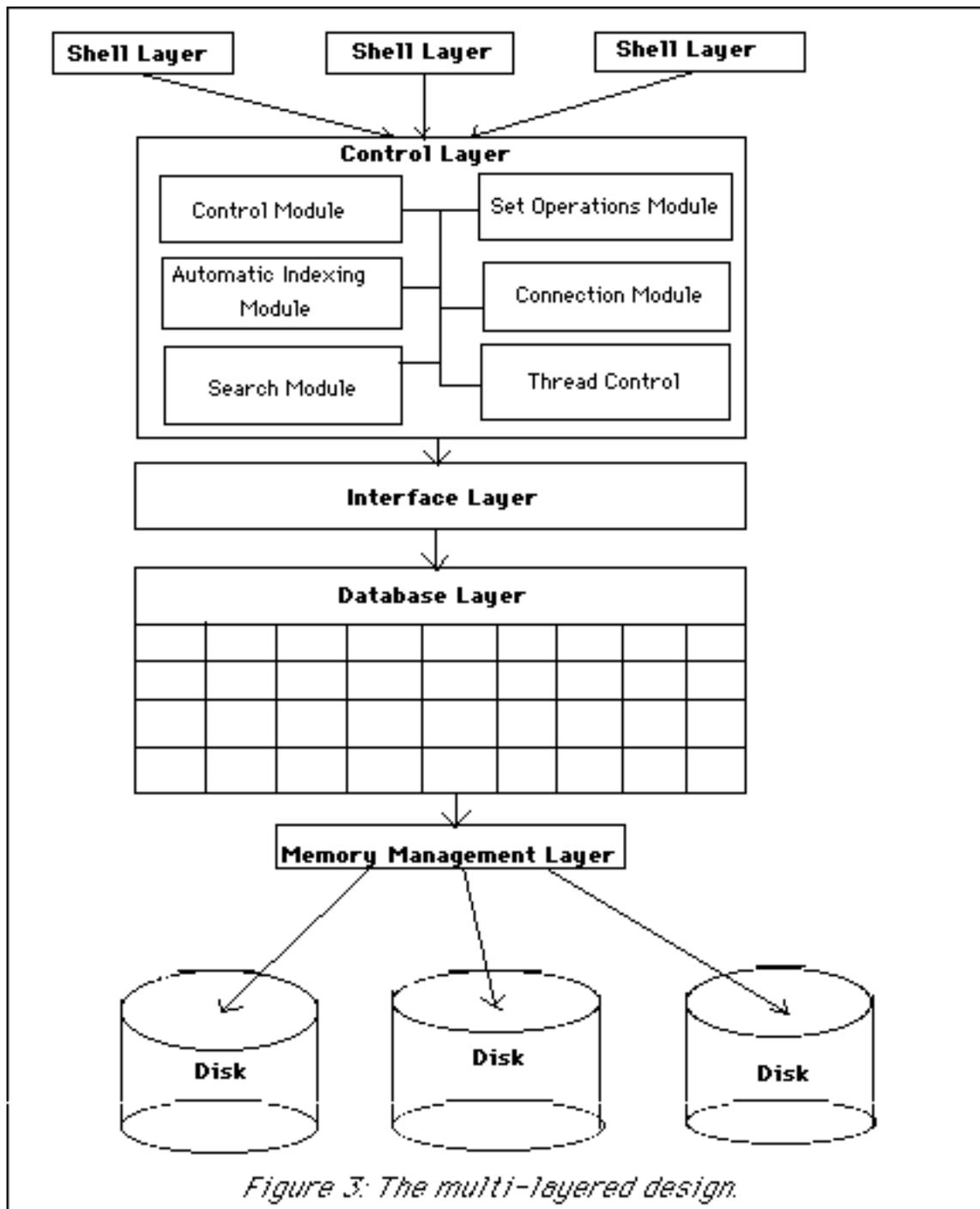
Figure 3: The multi-layered design.

3.3.1 The Shell Layer

The Shell Layer gets its name from the world of operating systems. Just as a UNIX shell provides an interface between the user and the kernel, the shell layer provides an interface between the user and the database, allowing the user to perform queries. Its purpose is to parse queries and provide error checking on user input. There are really two kinds of shell layers: an "administrative shell," created when the system starts up, and a "connection shell" for communicating with remote clients. The administrative shell allows a special user to add new documents to the database and perform operations such as reporting statistics on system performance, memory usage, etc. The connection shell is similar to the administrative shell but does not have the same privileges, and it takes its input from a socket connection accepted by the control layer, rather than from standard input. The connection shell would most likely be limited to making query requests and performing relevance feedback. All shells are created by the control layer and have access to the functions provided by that layer. They are able to call control layer routines for making queries based on one or more keywords, a file or a parse tree containing keywords, files, and Boolean operators such as AND, OR and NOT. The shell also has access to relevance feedback related routines provided by the control layer.

3.3.2 The Control Layer

The Control Layer is the "brain" of the system. It spawns off shells and accepts commands from them. These commands may include: adding a

new file to the database, performing a vector based lookup, searching for a particular keyword, or looking up an entire parse tree for an expression of keywords and Boolean operators. The Control Layer can create one or more Interface Layer objects, which are called as if they were actual "database" portions of the system. Calls to this virtual database are made using the actual document pathnames and keyword strings. Lower layers translate the strings into integer document and keyword identifiers.

The Control Layer can be viewed as several modules which work together. A search module accepts a parse tree of keywords and Boolean operators and then performs lookup commands using functions provided by the Interface Layer. The search module contains an additional module for manipulating sets. This allows the results from individual keyword searches to be combined as designated by the parse tree. The Control Layer also provides a module for performing automatic indexing of documents. This module parses a given input file and extracts the relevant keywords and their relative frequency of occurrence. The indexing module returns a list of (keword,frequency) pairs corresponding to a document vector. This vector can then be added to the database.

The Control Layer might also include a connection module for handling new network connections with shells and storing information relevant to each connection. A statistical module can be used for keeping track of various statistical system variables for debugging and experimentation purposes. Examples of statistical variables might include: the number of times keyword x has been requested, the number of times

document y has been included in a return result, or the number of clock ticks required to process a given querry.

Given the theoretically large number of keyword search requests that are conceivable for a web-based search application, as well as the vast number of new documents that will need to be indexed, it makes sense that the information retrieval server process these requests and perform the necessary additions to the system in an efficient manner. A multi-threaded implementation could help processes all of these demands efficiently. A module for creating, scheduling and killing threads for proccessing queries can easily be added at the Control Layer level.

3.3.3 Interface Layer

The Interface Layer provides a level of abstraction between the Control Layer and the Database Layer. It stores two dictionaries, one containing all of the keywords in the database and their associated integer identification numbers, and another containing all of the document names and their associated integer identification numbers. The Interface Layer simply receives requests from the Control Layer to perform lookups or document additions based on actual keyword or document strings, and then finds the appropriate *id* for these values and calls the Database Layer to perform the neccessary operation. The results returned from the Database Layer are then converted from their identification numbers to their actual string values and passed on to the Control Layer.

3.3.4 Database Layer

The Database Layer is the underlying data structure of the system. Its implementation can vary widely, but it always allows a set of basic operations. The Database Layer performs keyword lookups based on a keyword identification number and returns a list of (document identifier, frequency) pairs. It can also perform the cosine correlation operation on any two document vectors, given their identification numbers.

3.3.5 Memory Management Layer

A layer for managing memory usage was also designed into the system. The assumption was that given any large number of documents, there exists a point at which it is no longer practical (or possible) to store all of the retrieval information located in the Database Layer within physical memory. Furthermore, any system for virtual memory provided by the operating system would not be "fine tuned" for the needs of the system. The role of the Memory Management Layer is to provide a way to simulate the vast amounts of physical memory required to maintain the Database Layer, while providing the neccessary functionality to keep highly correlated information located in positions on disk that would minimize the time required for disk seeks.

4. Layered Design Implementation and Analysis

Great care was taken to design a system comprised of implementation independent layers. For example, the Database Layer should not need to know the actual names of keywords or documents to perform its operations. The Database Layer only requires an integer identification and thus it is blind to the method used to index that keyword to find the appropriate document list. Furthermore, the Control Layer should not need to know how the data is stored, as long as it can perform the operations it needs. It is therefore possible to completely change the implementation of the database itself, without affecting any previous layer. In fact, it is theoretically feasible to have multiple implementations of the Database Layer which are used by the Control Layer as if they were identical. The advantage of this flexibilty is evident, considering that at the onset of this project, we clearly did not know what the best implementation would be. We knew that some experimentation would be required, so we needed a system capable of gracefuly handling massive changes in implementation with minimum effort.

In spite of the all the implementation independencies in the system design itself, it is obvious that different implementations will have varying performance attributes. Here follows a discussion of the implementation issues involved in each layer. We will describe the implementation of the prototype that has been developed as well as other possible implementations. We will also discuss the functionality provided by the prototype, and critique the implementation thereof.

4.1 <u>Shell Layer</u>

Two rudimentary versions of the Shell Layer have been implemented for the system prototype. The first takes input from the command line, and the second from a connection with the web-based client. Both shells accpet two simple commands: the first allows the user to add an additional document to the database while the second allows the user to perform a single keyword query. Administrative commands (such as adding a file), are entered with a preceding pound sign(#); otherwise, the input is expected to be a keyword query request.

The simplicity of the available commands implies that, at present, the shell only minimally takes on its primary role as a command and Boolean expression parser. As only one keyword may be requested at a time, parsing is trivial. Future implementations of the Shell Layer will allow keyword query expressions to be evaluated by generating a syntax tree based on a grammar similar to the following:

Exp   -> Exp OR Term

Term -> Term AND Factor | Term Factor

Factor -> KEYWORD | (Exp) | NOT KEYWORD | NOT(Exp)

This grammar generates expressions based on the standard mathematical precedence for the Boolean operators starting with OR (the lowest precedence), and ending with NOT (the highest precendence). The grammar allows any string of keywords, which may or may not be separated by the

"AND" operator. Any two consecutive keywords with no separating operation are assumed to be joined with the AND operator.

The implementation of a lexical analyzer for these expressions is trivial, as the only tokens are keywords, parentheses, and a small number of operators. Furthermore, the actual expression parser could be implemented easily with any automatic parser generator such as YACC or Bison.

4.2 Control Layer

Like the Shell Layer, the implementation of the prototype Control Layer utilizes only a minimal portion of its theoretically conceivable functionality. In the case of a single keyword query, it passes the request to the Interface Layer. In the case of a command to add a document to the database, the Control Layer invokes an automatic indexing module and passes to it the new filename. The indexer returns a list of (keyword, frequency) pairs which is then sent to the Interface Layer for addition to the database.

A brief description of the implementation of the automatic indexing module is appropriate here. The purpose of such an indexer is to extract the keywords from a document which will help to distinguish the document from any other. Words with too common a frequency in the language, such as "and", "or", "but" and "the", do not help to differentiate any one document from another. These are called "stop words". Furthermore, keywords with a low frequency in any particular document are assumed to be less helpful in describing the content of the document.

The automatic indexing module for the prototype was implemented using a form of B-tree with each node representing a substring of characters in a keyword. Each node contains a pointer to a leaf node (which may or may not be NULL) and a pointer to a hash table of pointers to other B tree nodes indexed by a single alphabetic character in lower case.[2] Leaf nodes contain an integer frequency of the particular keyword described by taking a path from the root to that node. Re-insertion of the same keyword has the effect of incrementing the frequency.

The hash tables were implemented using an expandable array starting with only one entry to conserve space. Conflicts are resolved with linked lists chained to each bucket, but due to the nature of the input keys (characters), conflicts are unlikely; thus lookup time will remain constant for any single character. Adding a word of length w to the B-tree requires at worst w new nodes to be created (in the case of an empty tree). A lookup performed on a word of length w requires at most w table lookups.

After all words of length greater than two have been entered into the B-tree, the entries in the tree are checked against a list of stop words specified at the creation of the indexing module. Words found in both the stop list and the B-tree are removed from the tree. All of the frequencies of the leaves are then modified by dividing each by the total number of useful keywords to obtain a normalized vector. The leaves, which are maintained in a list, are then returned to the caller.

---

[2]All input characters are imediately converted to lower case.

The total running time for indexing a document of n words of average length w, given a stop list of x words of average length w, can be summarized as follows:

-Time to add n words to the tree: $O(n * w)$

-Time to extract x stop words from tree: $O(x * w)$

-Time to format remaining leaves: $O(|N\text{-}X|)$ [3]

-TOTAL: $O(MAX(x, n) * w)$

Various extensions to the automatic indexing module are imaginable. More sophisticated indexers attempt to remove prefixes and suffixes from input strings to record only the root. Another enhancement specific to the web-based nature of the project would be to create an indexer specific to HTML documents, and thus capable of ignoring HTML flags or consecutive sequences (of length greater than some threshold) of the same keyword within an HTML comment.[4]

---

[3] Where $|N\text{-}X|$ is the length of the set containing the difference between the set of all N keywords and all X stop words. This is at most length n, if no stop words were extracted.
[4] Web search engines often organize query results by sorting the documents by the total frequency of occurrence of the keyword in each document. Some pages take advantage of this by padding the HTML text with thousands of copies of the same keywords within an HTML comment. In automatic indexing terms, only a few repetitions are really relevant to sufficiently describe the content of the document.

4.3 <u>Interface Layer</u>

The Interface Layer translates keyword and document name strings into their corresponding integer identification numbers. It passes on the requests from the Control Layer to the Database Layer to insert documents or perform a query lookup by providing functions to translate between the data structures used by the two layers. Keyword and document strings are used as keys to index a hash table of identification values. There are also two arrays containing all keywords and all path names, respectively. This allows constant or near constant translation time between identification number and string value, and vice versa. This particular implementation for the prototype of the Interface Layer was due primarily to the fact that a template for a hash table data structure was coded before the B-tree for automatic indexing. A B-tree with its leaves stored in a hash table indexed by integer id's would allow the bi-directional conversions necessary, but would require less storage space. However, in keeping with the philosophy behind the layered design, a major change in implementation such as this would merely amount to a few minor changes in a single layer.
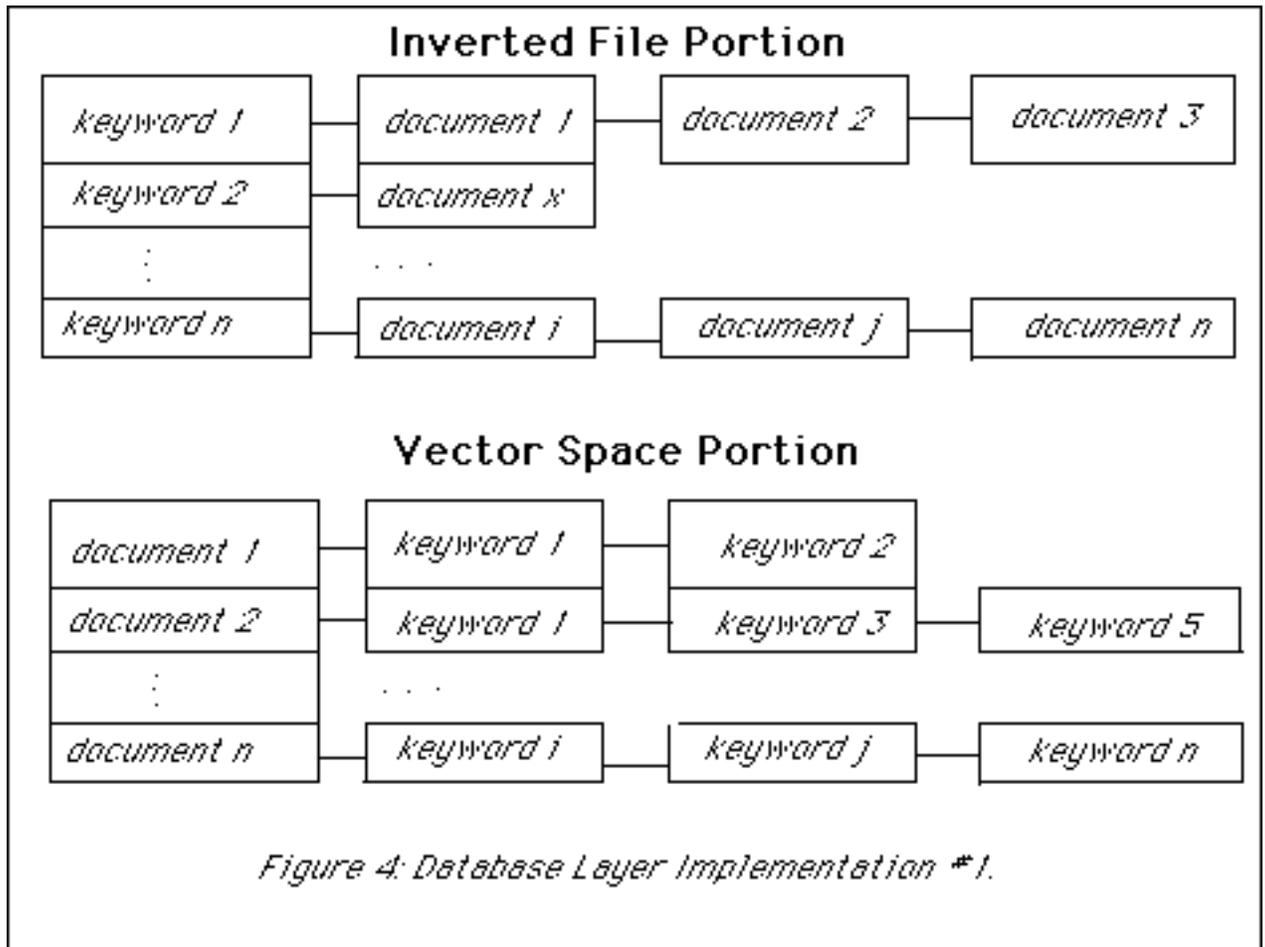
The total time spent in the Interface Layer is bounded by the necessary conversions between the data structures employed by the Database and Control Layers. This time is unavoidable, since passing a copy of the actual data from the Database to the Control Layer would be preferable to passing a pointer to the information, even if the data structures happened to be identical. The time required to transform a list of (keyword, frequency) pairs for addition to the database as a new document vector is linear in the number of entries in the list. Likewise, the time required to convert a list of document

(identifier, frequency) pairs to a list of document strings to be returned as the results of a keyword query is linear in the number of documents.

4.4 Database Layer

The implementation of the Database Layer is the most interesting and has the most profound effects on aggregate system performance. Given the fact that information must be accessed quickly and efficiently in two different ways (inverted files ana vector space), great care must be taken to minimize the total amount of required storage, both in RAM and on disk. For the moment, we have developed the prototype for the Database Layer under the assumption that all the information may be contained in RAM. This assumption will not cause major problems for small test sets of documents, and thus is ideal for a prototype, but we must also consider the larger, more likely case in our implementation.
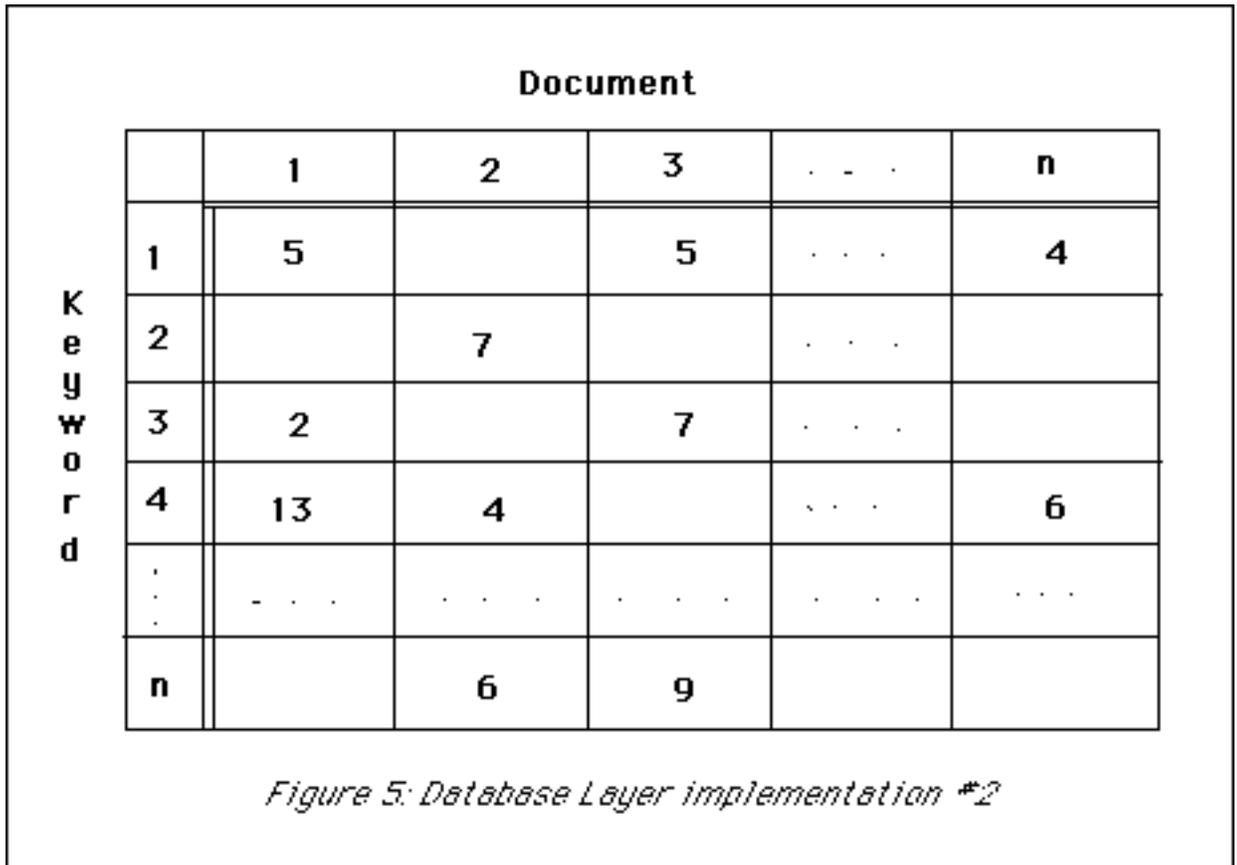
The initial structure for the Database Layer was to maintain two separate copies of the same data. An inverted file portion stored (document, frequency) pairs based a keyword index, and a vector portion stored (keyword, frequency) pairs based on a document index. Both portions were implemented using a linked list of linked lists.

Inverted File Portion

keyword 1	——	document 1	——	document 2	——	document 3
keyword 2	——	document x
⋮		. . .
keyword n	——	document i	——	document j	——	document n

Vector Space Portion

document 1	——	keyword 1	——	keyword 2
document 2	——	keyword 1	——	keyword 3	——	keyword 5
⋮		. . .
document n	——	keyword i	——	keyword j	——	keyword n

Figure 4: Database Layer Implementation #1.

In the context of our first assumption that all information must be stored in RAM, this is quite possibly the worst implementation. Lookup time is linear, and insertion of a document vector requires updating the inverted file portion in addition to changing the list of document vectors.

Another simple and naive implementation would be a two dimensional matrix indexed on one side by keyword *id* and on the other by document id. Any cell [i,j] would contain a floating point frequency of the

keyword *i* in document *j*. If document *i* does not contain keyword *j*, then the cell is empty.

	Document				
	1	2	3	. - .	n
1	5		5	. . .	4
2		7		. . .	
3	2		7	. . .	
4	13	4		. . .	6
. . .	- . .	. . . .	. . .	. . .	. . .
n		6	9		

Figure 5: Database Layer implementation #2

Looking up any document or keyword with this implementation can be done in constant time, but will undoubtedly waste some space. Total space requirements are:

2 * documents * keywords   (4.4.a)

where 2 is the number of bytes required to represent a 16 bit floating point value, and "documents" and "keywords" are the total number of each in the data set. The practicality of this implementation depends on the sparsity of this matrix. In practice, the answer turns out to be approximately 40-1 [5]. That is, for every cell that is used, forty are wasted.

An alternative method for organizing the data which will be used for the second version of the Database Layer prototype is a "mesh" of linked lists.

Figure 6: Database Layer implementation #3.
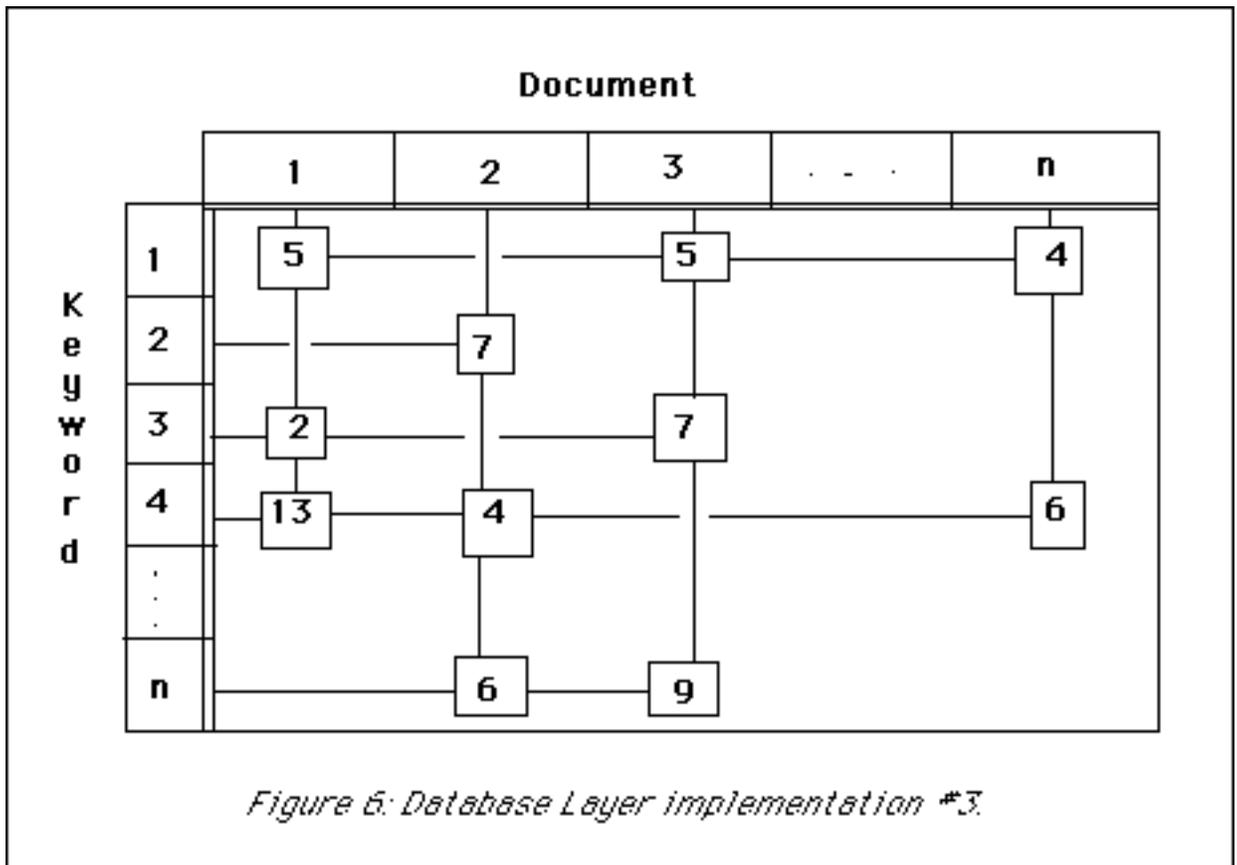
Each node of the mesh will contain the integer identifiers for the document vector and keyword entry to which it belongs. It will also contain a floating point frequency and two pointers, one to the next node in the document vector, and another to the next node in the inverted file entry. The total space requirements[5] for an individual node are:

2 bytes * 2	document/ keyword identifiers
4 bytes * 2	pointers to next nodes in the mesh
2 bytes	frequency
14 bytes	Total

If we consider the "opportunity cost" of the matrix implementation (i.e., the fact that for every cell which is used, forty are wasted), then each cell requires 2*40 + 4 or 84 bytes—over 6 times as much space as the mesh implementation! Lookups for individual keywords will still be constant, since the entire list will be returned to the Interface Layer. The running time of the cosine operation will now be faster than with the matrix representation since we avoid the feasibly large number of unused entries. Running time will now depend on the maximum length of the two vectors being compared, rather than the maximum length of the longest vector in the entire set (the size of the matrix).

---

[5]We assume that the data structure will not need to contain more than 64,000 documents or keywords (the number of integers we can represent with 4 bytes). This assumption is not unreasonable since a data set with so many documents and key words will probably not fit entirely in RAM anyway.

4.5 <u>Example Query Life Cycle</u>[6]

Consider the aforementioned query pertaining to Loki, the Norse God of Mischief. From the initial user request to the clustering and output of the return set, the query will traverse the layers in the following manner:

- The Shell Layer receives the string "loki norse god mischief" from the user (web client).
- The Shell Layer parses the input string and generates a syntax tree corresponding to: loki AND norse AND god AND mischief.
- The Shell Layer calls the *Control::lookup(SyntaxTree s)* function, passing the query to the Control Layer.
  - The Control Layer calls the *Interface::lookup(char *keyword)* function for the first string in the syntax tree "loki".
    - The Interface Layer looks up the keyword identification number for the string, "loki".
    - The Interface Layer calls the *Dbase::k_lookup(int id)* function, transferring control to the Database Layer.
      - The Database Layer looks up the integer id in the inverted index portion of the data structure.
      - The Database Layer returns the address of the list of documents corresponding to that keyword back to the Interface Layer.

---

[6]References to C++ functions in this section are actual function calls implemented by the prototype.

- The Interface Layer creates a copy of the document list while replacing the document integer identification numbers with the actual document names (strings).
- The Interface Layer returns the new list to the Control Layer.
- The Control Layer performs similar lookups on the remaining keywords in the query, and then combines the results into a single document set based on the Boolean operations specified in the syntax tree.
- The Control Layer calls the *Control::cluster(list *l)* function which hierarchically clusters the list of documents, *l*. The document clustering is returned to the Shell Layer.
- The Shell Layer returns the clustering scheme to the web client, which outputs a visual, hierarchical clustering of documents.

5. Conclusions

Many advantages of the layered design for our web-based Information Retrieval system have been clearly illustrated. However, one possible concern with the design is the inherent overhead of maintaining multiple layers. Passing information down through various layers and then back up again can be extremely costly in terms of the time and space required for the increased number of function ccalls (allocation of stack frames, etc.). This concern, however, is poorly merited. We have shown that each layer performs actions to manipulate the user commands in such a way that redundancy of computation is not an issue. Furthermore, every operation

performed at each layer is indispensable to the functionality of the system as a whole. Any clearly written implementation would define the same functions to perform the same operations in some manner, but we have chosen to think of the process hierarchically.

A discussion of our implementation in terms of our original desiderata still remains.

5.1. *Justification of the Layered Design in the Context of the Model Specifications*

- Compatibility:

    The first concern in designing the system was that vast amounts of existing data are presently organized in inverted files. Obviously, the implementation of each inverted index-based system is widely different. The layered organization of our system allows quick installation of modules specific to each of these implementations which can then translate their data into a form readable by ours.

- Space/Time:

    The space/time considerations obviously depend on the implementation we choose for the Database Layer. Each version was shown to perform query lookups in a reasonable amount of time, but the mesh data structure required the least amount of memory. In terms of the previous systems that we have

described, ours requires much less memory to perform more reliable document collection organization. We do not need to store an entire thesaurus to maximize precision and recall, nor do we need to take the time to parse every document in the set against the language in which it was written in order to obtain a very good idea as to its content.

- Maintainability:

    Our layered design is extremely maintainable. It is written in a common language available on almost all platforms. It allows modules to be plugged in or taken out without massive surgery to other modules. Huge changes in implementation can be made painlessly. This approach is ideal in an experimental design environment.

- Automation:

    We have described the automatic document indexing and the automatic document collection clustering supported by our system. Our architecture is designed to support other automatic features as well. Consider, for example, a modification of the Shell Layer to act as a web crawler. Finding new documents and adding them to the database can now be performed automatically as well.

- Multi-user:

    Given the web-based nature of the system, it can be accessed by a large number of users. The multi threaded functionality that the system is capable of supporting will allow many queries from many users to be handled effectively.

5.2 *A Final Example*

The study of computer science has helped to develop algorithms for automatically performing many useful tasks with computers. These algorithms are designed to make the most efficient use of computer resources. Their speed and accuracy is often dependent on the way the input is organized.

Computers are becoming increasingly more affordable, while human effort becomes more expensive. We need ways to present data efficiently so that the user can employ a faster "algorithm" for finding important information quickly. Our Information Retrieval system supports a hierarchical organization of topic clusters which can greatly reduce search time on behalf of the user.

Imagine a new world where this technology is readily available. My long lost friend attempts to lookup the word "loki", but this time he is not only told that 6, 808 documents match his request, he is show several circles of varying size corresponding to topic clusters of varying lengths. After browsing through a few documents of each cluster, he realizes that one contains pages dealing with Norse mythology, and another is dedicated primarily to religious groups. Among the remaining clusters he finds one

which contains several pages with personal information, so he selects the cluster and is shown a new set of circles corresponding to the sub topics for the cluster he chose. He looks at the most frequently appearing keywords in each cluster and notices that one contains a high frequency of the keywords "Oingo" and "Boingo". My long lost friend now remembers my endless tirades about Oingo Boingo being the best rock band to have ever existed, and at this point it is only a matter of seconds before my friend finds "Loki's Oingo Boingo Appreciation Page". Five minutes later my computer plays a brief selection of *My Life* [4] and a window pops up on my screen notifying me that I have just received mail from a good friend with whom I had lost contact years ago.

6. Acknowledgments

References:

[1] Aslam, Javed; Pelekov, Katya and Rus, Daniela, *Generating, Visualizing, and Evaluating High Quality Clusters for Information Organization,* Preprint, 1997.

[2] Cormen, Thomas; Leiserson, Charles and Rivest, Ronald, <u>Introduction to Algorithms</u>. (New York: McGraw-Hill Book Company, 1990).

[3] Jones, Karen Sparck. <u>Automatic Keyword Classification for Information Retrieval</u>. (London: Butterworth and Company, 1971).

[4] Oingo Boingo, *Alive.* MCA Records, 1988.

[5] Pelekov, Katya. Personal Communication.

[6] Salton, Gerard and McGill, Michael, <u>Introduction to Modern Information Retrieval</u>. (New York: McGraw-Hill Book Company, 1981).