Telcordia LSI Engine: Implementation and Scalability Issues

C. Chen, N. Stoffel, M. Post, C. Basu, D. Bassu and C. Behrens
Applied Research
Telcordia Technologies, Inc.
Morristown, NJ 07960-6438

Proc. of the 11th Int. Workshop on Research Issues in Data Engineering (RIDE 2001): Document Management for Data Intensive Business and Scientific Applications,
Heidelberg, Germany, Apr. 1-2, 2001.

Telcordia LSI Engine: Implementation and Scalability Issues

Chung-Min Chen, Ned Stoffel, Mike Post, Chumki Basu, Devasis Bassu, Clifford Behrens
Telcordia Technologies, Inc.
Morristown, NJ 07960
chungmin@research.telcordia.com

Abstract

Latent Semantic Indexing (LSI), a vector space-based approach to information retrieval , has been proven to be an effective tool in correlating and retrieving relevant documents. While much work has been published on LSI, most of it addresses the algorithmic or theoretical basis of the model. Little, if any, presents implementation issues in practice. In this paper, we describe a production-level implementation of LSI. The system integrates components including document collection and preprocessing, singular value decomposition (SVD), multilingual processing, and a tree-based access method for similarity querying. We discuss implementation issues encountered during the development of the system. In particular, we address scalability issues in the query engine and various components of the system, and present lessons learned.

1. Introduction

Latent Semantic Indexing (LSI) [1] is a vector space-based technique to create associations between conceptually related documents. Since the original paper [3], much work has been published on LSI. Examples of recent work include [10, 4, 7, 2, 1, 8] (also see `http://lsi.research.telcordia.com` for a list of early work). These papers cover various aspects including theoretical properties, semantic interpretation, optimization, extensions and applications of the LSI model. Few papers discuss implementation issues in practice.

In this paper, we report a production-level implementation of LSI at Telcordia. We call it Telcordia LSI Engine (in this paper we will simply refer to it as *the LSI engine*). We will describe the functional components of the product, and discuss the issues and considerations encountered during the implementation of the LSI engine. In particular, we will address scalability issues in various components of the system, including document pre-processing, term-document matrix generation, Singular Value Decomposition (SVD), document folding, and querying. Where appropriate, we will provide scaling rules-of-thumb derived empirically. We pay special attention to query performance, presenting practical implementation techniques to reduce search overhead, as well as a cluster tree-based access method used to prune the multidimensional search space. Preliminary results show substantial improvement in the query response time, with minimal loss in relevance recall.

We note here that while recent advances in Web Information Retrieval techniques have been quite successful in finding relevant Web pages, there exist other IR applications for which these techniques are not adequate. The majority of the Web IR methods rely on two basic techniques: inverted files (for keyword search) and connectivity analysis (ranking Web pages according to their hyper-link structure) [6]. Inverted files have limited utility for finding related documents that contain synonyms of the keywords. Connectivity analysis is useful only for Web pages. A good IR example for which these techniques are of little use is patent processing. When a new patent application is received, the US patent office needs to find out if there are similar patents that already exist. More than often, related patent documents may use different terms to describe similar subjects/ideas and they usually have no hyperlinks. LSI is ideal for such applications.

The rest of the paper is organized as follows: Section 2 reviews the LSI technique. Section 3 gives an overview of the Telcordia LSI Engine. In Section 4, we address the problem of similarity search in LSI vector spaces and describe implementation techniques and a

[1]Latent Semantic Indexing (LSI) is a patent of Bellcore (now Telcordia Technologies), under U. S. Patent No. 4,839,853, June 13, 1989.

© Telcordia Technologies, Inc. 2000. All rights reserved

tree-based access method to improve search speed. Section 5 presents scaling behaviors and considerations in LSI vector space generation. Finally, we discuss future research.

2. Latent Semantic Indexing

We will only give a very brief description of LSI (the details can be found in [3]). Given a collection of d documents and a set of t terms (words) appearing in the documents, we construct a $t \times d$ matrix X (i.e., t rows and d columns), where $X(i, j)$ is the number of occurrences of the i'th term in the j'th document. We call X the *term-document matrix*. It is imperative to compress X as t and d may be large (tens of thousands). Let m be the rank of matrix X, we may decompose X as :

$$X = T_0 \times S_0 \times D_0',$$

where T_0 (size $t \times m$) and D_0 (size $d \times m$) have orthonormal columns and S_0 (size $m \times m$) is diagonal. Note D_0' denotes the transpose of D_0. This is called Singular Value Decomposition (SVD). Without loss of generality, we assume the values in the diagonal of S_0, called the *singular values*, are ordered by size. By keeping only the k largest singular values we reduce S_0 to a $k \times k$ diagonal matrix S. Also, let T (D) be the resulting matrix by keeping only the first k columns of T_0 (D_0). Then the product

$$\hat{X} = T \times S \times D'$$

is a rank k approximation to X. We call k the *number of factors* of the reduced vector space.

We may view the i'th row in the matrix $\mathcal{D} = D \times S$ as the representation of the i'th document (see [3] for more details). Thus, each document is represented by a k-dimensional (k-dim for short) vector. We will use "factor" and "dimension" interchangely in the rest of the paper. Typically, the value of k is chosen to be much smaller than t, so $\mathcal{D}$ (instead of the original X) can be efficiently stored and manipulated. For single-language document collections, a value between 100 and 300 for k is usually required to obtain good query results.

The SVD is a computationally-intensive process. Thus, when a document collection is very large, only a limited subset of the document collection, called the *training set*, is used in the SVD process to compute the matrices T, D and S. This training set is chosen to be representative of the full collection. Once these matrices are obtained, a new document y, represented in its raw term frequency vector $y = (y_1, \ldots, y_t)$, can be "folded" into the LSI vector space and represented by a k-dim vector

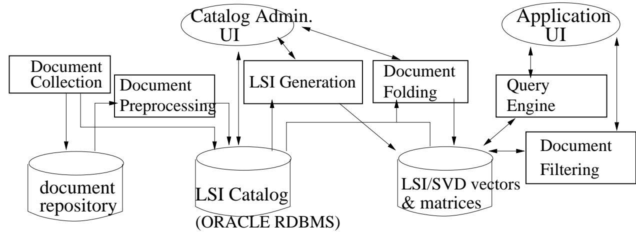


Figure 1. Telcordia LSI Engine Functional Components

$$v_y = y \times T \times S^{-1}.$$

Similarly, by treating a query as a "pseudo document", it can be transformed into a k-dim vector using the same expression above.

Given a query q (in its k-dim vector representation), the similarity between the query and any document v (in its k-dim vector representation) is measured as the cosine between the vectors: $\frac{q \cdot v}{|q||v|}$. If we pre-normalize all vectors to unit length, the similarity score between two vectors can be computed simply as their inner-product. In the following, we assume all vectors are pre-normalized.

One major advantage of LSI over other methods (e.g. inverted files) is its capability of finding documents that contain synonyms of the searched terms. This capability is embedded in the SVD process (which extracts latent semantic concepts) and the elimination of the non-significant singular values (which removes unwanted semantic "noise") [3].

3. Telcordia LSI Engine: Putting the Pieces Together

Figure 1 shows the system architecture of Telcordia LSI Engine, which includes six major functional processes:

Document Collection generates metadata for raw documents and stores them in RDBMS tables. Our current implementation supports ORACLE and MySQL (a public domain RDBMS). The raw documents, however, are placed in a document repository under the UNIX file system in which each document is a separate file. The metadata contain information such as where to locate the document in the repository (access-descriptive metadata), how to regulate the processing of the document (process-descriptive

metadata), and what the document contains (content-descriptive metadata).

Document Preprocessing is a daemon process that periodically checks the LSI catalog for new unprocessed documents. Once unprocessed documents are identified, the daemon retrieves the documents from the repository, extracts terms, tabulates the term frequencies, and populates the database tables. The core of the pre-processor is a general text parsing module that separates data streams into tokens based on the language of the documents. The term frequencies are recorded in three relational tables: `document, term`, and `frequency`. Each row of the `document` (`term`) table represents a document (term). The `frequency` table contains foreign keys to the `document` and `term` tables, and an attribute `count` that records how many times the term appears in the document. This relational representation avoids the space problem of sparse matrices (most entries in the document-term matrix are zero).

LSI Generation takes a subset of the document collection as the training documents and produces the LSI vector space. First, it reconstructs the corresponding term-document matrix from the LSI catalog. The set of terms may be reduced by eliminating common words (e.g. articles and prepositions) from a "stop-word" list or by using global or document frequency thresholds. The thresholds determine how often a term must appear in a document in order to be considered an indexable term. The selection of stop-words and term thresholds may affect the resulting LSI space and thus the search results. Unfortunately, there are no simple rules guiding how these factors should be optimized.

After the term reduction, we further applies global log entropy weighting [11] to the term frequencies. The resultant term-document matrix is stored in a file in the sparse Harwell-Boeing format [5]. Finally, the SVD process is invoked against the matrix. The output of the SVD process includes the term matrix T, the document matrix D, and the singular value matrix S. These matrices are saved in files on the disk.

For multilingual applications, we require parallel corpora for the training set, where each document has a correspondent in each of the languages of interest. The corresponding document in a different language is called a *mate* of the original document. Term frequencies are then collected for each mate (the terms are in their corresponding native language). We then concatenate these term vectors into a single vector. This vector occupies a column of the term-document matrix and allows the SVD process to extract semantic relationships between terms and concepts in different languages. Once the LSI matrices are computed and all the documents (in any of the languages) are folded in, we may search the multilingual collection by composing a query with terms in any of the language. For example, one will be able to express a query in English and receive a related document in Chinese.

Document Folding maps each document in the collection to the corresponding vector representation in the LSI vector space. The matrices T and S generated from the training phase are used for this purpose. The mathematics for the folding procedure is described in Section 2.

The **Query Engine** allows users to find relevant documents by specifying a list of terms. Documents in the collection can also be used as part of a query. The user can specify the number of documents to be returned. The query terms and documents are used to compose a query vector in the LSI space (see Section 2). Once the query vector is formed, the most relevant documents are those with the largest cosine values from the query vector. The implementation of the query engine is discussed further below.

The **Document Filtering** utility enables automatically classification of incoming documents into a number of pre-determined subjects/categories. It works as follows: First, a collection F of n "filter documents" are created, each representing one of the n categories. Let D be the document collection to be classified (filtered). We first draw a small sample $S \in D$ from the document source. We then take $F \cup S$ as the training set and create the LSI vector space. As a result, for each filter document in F we have its vector representation well-positioned in the LSI space created jointly with the sample documents. Finally, for each document $d \in D$, we fold it into the LSI space and let v_d be the resultant vector. The document then belongs to the category represented by the filter document whose vector representation is closest to v_d (based on cosine distance).

4. Query Engine Implementation and Performance

We will not address the precision and recall of LSI, as previous work has demonstrated LSI's effectiveness in these metrics. Instead, we focus on query response time—the time it takes to answer a query. Query response time is crucial to user satisfaction as the document collection and the number of users grow larger. The LSI engine offers two query options: *exact query* and *approximate query*. Exact query returns the actual top-k documents that are closest to the query vector; approximate query returns k documents that are close to the query vector but gives no guarantee that they

are the actual top-k matches. Results include the cosine values and are sorted by these values.

We implemented two search options: *linear scan* and *cluster search*. Linear scan computes the distance between the query vector and every document vector, and returns the top-k closest documents. Linear scan is used solely for exact queries. Cluster search utilizes a pre-built tree-based indexing structure that partitions the vectors into a hierarchy of clusters/buckets. Cluster search can be used for both exact and approximate queries.

In the following subsections, we will describe in greater detail the two search options, along with some useful implementation techniques that help to reduce the computational overhead.

4.1. Linear Search

The query response time of linear search grows linearly with the number of factors (dimensions) and the number of documents. For relatively small collections and factors that fit into the physical memory, the query response time is satisfactory and obeys the linear rule. However, as the size increases beyond the available memory size, the response time degrades drastically and scales worse than linear, due to virtual memory paging/swapping overhead.

We ran an experiment observing the response time as a function of collection size. Figure 2 shows the results. The numbers are measured for top-20 query with a 646-dimensional vector space. The experiment was run on a SunSparc Station with 500MB RAM. The vectors were pre-loaded into virtual memory at start-up time. The query time reported is in user CPU time. It also shows the size of (virtual) memory allocated to the process. It can be seen that the query time increases almost linearly with collection size. The query time, however, grows faster for larger collections (when number of documents $\geq$ 80K), as their space requirement approaches the physical memory size.

We also observed that for smaller collections, the user CPU time is virtually equal to the real elapsed time (wall-clock time). For larger collections, the CPU utilization drops quickly (due to swapping). For example, for the 180K collection, the real time is about 100,000 msec (compared to the 160msec CPU time). Using the UNIX 'ps' command, we get 0.34% CPU usage for the 180K collection, versus 99% for the small collections.

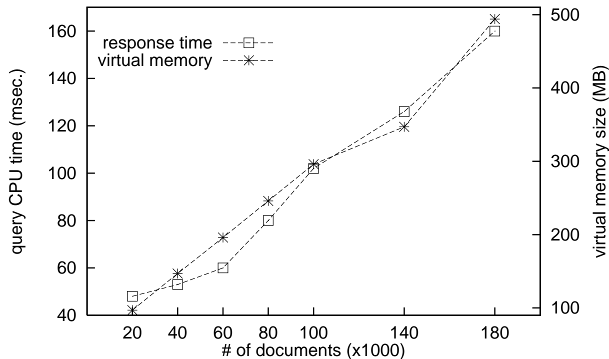


Figure 2. Scaling of linear scan

4.2. Vector Compression and Fast Distance Computation

8-bit Vector Compression The numbers from above indicate that to get efficient query response time, we should try to load as many vectors into the memory as possible. One trick to achieve this is to compress the representation of the vectors. Instead of using the standard "`float`" type which takes 4 bytes, we represent each vector component as a 8-bit number (in "char" type). Since the (pre-normalized) vectors fall on the surface of the m-dimensional sphere with unit radius, the value of each vector component is between -1 and 1. We map each floating number x in $[-1, 1]$ to an integer between 2 and 254 using the following linear encoding:

```
int (x * 126.0) + 128.
```

In the above, we scale x by 126.0 instead of 128.0 to avoid problems with floating point error in C++. The encoded integer is then stored as an 8-bit "char" field. As we will describe later, the data precision loss due to the compression has little impact on the quality of the query results for moderate numbers of factors (dimensions). However, the cumulative error becomes noticeable for larger dimensions (this also depends on the distribution of the vector space).

Quick Look-up Table Using 8-bit vector representation effectively compresses the vector space to one fourth of its original size. But how do we compute the cosine distance using 8-bit representation? Recall that since all vectors are normalized to unit length, the cosine distance can be computed as the inner-product between the two vectors. To facilitate this computation, we use a 256×256 look-up table.

Using the look-up table, no multiplication operations are needed when computing the inner products. Compared to the standard floating point multiplication operations, which consume lots of CPU cycles, the

look-up table is much more efficient in practice.

Truncated Cosine Computation We have also implemented a technique that significantly reduces the number of look-ups needed when computing the inner-products (cosine distances). The idea is as follows: Consider, in a m-dim LSI space, the search of top-k matches of a given query vector q. Let θ be the cosine distance between q and the k'th nearest vector currently found. Then, considering an unvisited vector $v = (v_1, \ldots, v_m)$, we need to compute the inner-product $v \cdot q$: if it is larger than θ then we need to update the current top-k list (and thus θ) by including v. A direct computation of $v \cdot q$ will require m table look-ups using the 8-bit vector implementation (if standard implementation is adopted instead, then m floating point multiplications are required). Fortunately, we often can stop the table look-ups earlier, as soon as we know the possibly maximum value of $v \cdot q$ can not exceed θ. This is elaborated below.

Given vector $v = (v_1, \ldots, v_m)$, define subvector $v(i) = (v_i, v_{i+1}, \ldots, v_m)$, where $1 \leq i \leq m$. Consider a query vector $q = (q_1, \ldots, q_m)$. Assume for a moment that we have a list $UB = (\theta_1, \ldots, \theta_m)$, where θ_i is an upperbound of the partial inner-product $q(i) \cdot v(i)$, for all vector v. That is, $\theta_i \geq \max_{\text{all } v} q(i) \cdot v(i)$. Then, given θ (cosine distance between q and the current k'th nearest vector) and an unvisited vector v, the table look-ups for $v \cdot q$ can be stopped as soon as we know $\sum_{i=1}^{l} v_i q_i + \theta_{l+1} < \theta$ for some l. The following code fragment shows the details:

```
bogey = θ;
for (i=0; i<m; i++) {
   bogey = bogey - product[((int v)<<8) | q];
   // if v can not beat the current k'th
   // nearest vector, return a cosine lowerbound
   if (bogey > UB[i])
      return (-2.0);
}
return (θ - bogey); // this is equal to v·q
```

To compute the list UB, we resort to the following lemma:

Lemma 1 *Fixing q, we compute $\theta_i = \sqrt{\sum_{j=i}^{m} q_j^2}$, for $1 \leq i \leq m$. Then $\theta_i \geq \max_{\text{all } v} q(i) \cdot v(i)$.*

The proof (omitted) is based on the Cauchy-Schwarz Inequality and the fact that $|v(i)| \leq 1$, for all vector v. To make earliest stop of the look up, we have also reordered the components of the query vector according to their absolute magnitude (in descending order). This results in quicker drop of θ_i in the UB list, and thus earlier stop of table look-ups for each vector. In general, we have found that, when the factors(dimensions) are more than 100, more than half of the table look-ups can be saved using this technique. Given 1 million (document) vectors and 100 factors, that translates into saving more than $1M * 100/2 = 50M$ table look-ups per query. The only overhead is the computation of UB, which is insignificant as it needs to be computed only once per query.

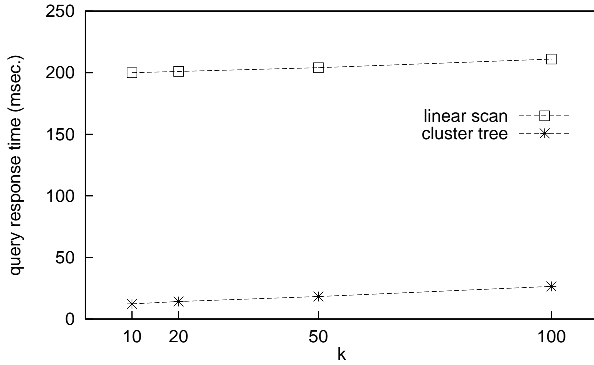
32-bit representation:	8-bit representation:
DocID=19127, cos=0.3918	DocID=19127, cos=0.3910
DocID=19139, cos=0.3257	DocID=19139, cos=0.3273
DocID=19131, cos=0.3013	DocID=19131, cos=0.2989
DocID=19129, cos=0.2501	DocID=19129, cos=0.2472
DocID=19136, cos=0.2214	DocID=19136, cos=0.2206
DocID=19126, cos=0.2029	DocID=19126, cos=0.2019
DocID=19141, cos=0.2012	DocID=19142, cos=0.2007
DocID=19142, cos=0.2012	DocID=19141, cos=0.2007
DocID=19128, cos=0.1928	DocID=19128, cos=0.1929

Figure 3. Query result comparison of 32-bit and 8-bit vector representation

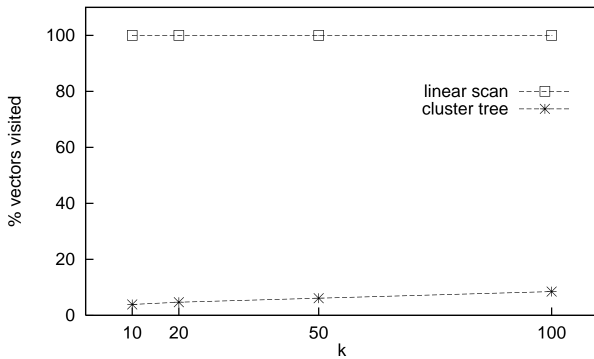
Collectively, the above techniques have improved the response time by at least a factor of 3. For larger collections, the improvement is even more impressive when all compressed vectors can fit into the physical memory. For example, the query response (in real time) for the 180K collection in Section 4.1 was reduced from 100,000 msec to less than 100 msec.

The data loss in the compression, however, does not have a significant impact on the recall of the top matches. Indeed, the process of calculating the cosines averages out a lot of the relatively large errors in the 8-bit numbers. The absolute RMS error in the cosines is much less than 0.01, which is good enough to get most documents ordered correctly in the like-document search. Our own experiences show that the effect of the compression on recall becomes noticeable when the number of dimensions reaches 250. In this occasion, the 8-bit and 32-bit representations begin to differ on the second decimal digit after the floating point. We note, however, that the maximum dimension after which the result quality becomes noticeable also depends on the distribution of the vector space.

Figure 3 compares the document sets returned by a top-9 query, using the 8-bit and the standard 32-bit float representation. The documents in each set are ordered according to their cosine distance to the query. In the 8-bit result set, the documents 19142 and 19141 are miss-ordered (w.r.t. the standard result set), but that is excusable because their cosines are the same up to 4 significant figures.



(a) response time



(b) percentage of vectors visited

Figure 4. Query performance of cluster and linear search options

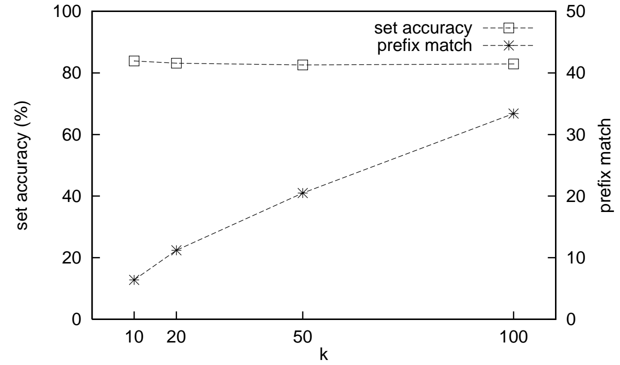


Figure 5. Accuracy of cluster search option

4.3. Cluster Search

While the above techniques help to reduce the vector space size and improve the query speed, they do not change the complexity of linear search, which needs to compute the distance between each document vector and the query vector. The cluster search option avoids this problem by employing a search tree. Note that the techniques described in the previous subsection are adopted as well in the cluster search option.

The search tree we implemented is a variation of R-tree. Like other R-tree variations (e.g., [12, 9]), our tree partitions the vectors into a hierarchy of clusters (buckets), where vectors in each bucket are enclosed by a geometric object. The search tree supports both exact query and approximate query search. For exact query, the tree must be searched until the exact top-k matches are found. For approximate queries, only a small number of branches are searched, based on some stopping heuristics. Due to space limitation, we will not provide details about the construction of the search tree.

The search tree allows the user to specify the fan-out factors of the internal/leaf nodes. This is unlike other R-tree variations where the size of a disk block dictates the fan-out factor. Our purpose is to allow users to configure these parameters for different physical memory capacities. For example, when physical memory is abundant, we can construct a two- or three-level tree and preload the entire tree into memory. This usually gives a better performance (typically sub-15 msec per query) than the disk-based R-tree variations (even when they are cached).

Figures 4 and 5 compare the query performance between the tree-based and linear scan options. The statistics are obtained by averaging among 100 randomly generated top-20 queries, against a collection of 20,382 documents (vectors) with 104 factors (dimensions). The experiment is conducted on a Sun Ultra-Enterprise workstation with fourteen 168MHz CPUs (though the query engine uses only one CPU) and 1.5GB RAM.

Figure 4 (a) and (b) show query response time (in real elapsed time) and percentage of vectors visited, respectively, for k varying from 10, 20, 50 to 100. The linear scan option retrieves all vectors (100%) regardless of k. Its response time, however, increases slightly as k increases, due to the house-keeping overhead of the top-k list. For the cluster search option, both response time and percentage of vectors visited increase as k increases. Overall, the cluster search option achieves an order of magnitude speed-up over the linear search option. The speed-up can be further improved with more sophisticated code optimization. For example, we may avoid computation of Euclidean distance and thus the expensive square root operations. Currently, the implementations of both exact and approximate query options involve Euclidean distance computation, though it is not necessary for the latter.

Figure 5 shows the accuracy of the cluster search option, which is measured in terms of the difference between its returned top-k list and the one returned by the linear scan option. Let C and L be the ranked top-k list returned by the cluster search and linear scan options, respectively. We define two accuracy metrics for the cluster search option as follows: Treating C and L as sets (both of size k), we define *set accuracy* to be the percentage of C that also appears in L. That is,

$$\text{set accuracy} = \frac{|C \cap L|}{|C|} \times 100 = \frac{|C \cap L|}{k} \times 100.$$

Treating C and L as ordered lists, we define *prefix match* to be the length of the longest common prefix of C and L. Formally, let $C(i)$ denote the prefix of C of length i, then

$$\text{prefix match} = \max\{i | C(i) = L(i)\}.$$

Figure 5 shows that the set accuracy is above 80% and remains roughly stable for all k. The prefix match grows as k increases (note that it actually declines if measured in terms of percentage). When $k = 100$, the prefix match is 33, which means that, in average, the top 33 matches returned by the cluster search are the actual top 33 matches. This suggests that to find the exact top-k matches, we can perform a top-k' approximate search, for some sufficiently large $k' > k$. For example, to find the exact top-20 matches, we may issue a top-50 query using the cluster tree (instead of issuing a top-20 query using linear scan). According to figure 5, there is a good chance that the top 20 entries in the returned ranked list (which contains 50 entries) will be the actual top 20 matches.

5. LSI Space Generation: Scaling Behavior and Considerations

We discuss the scaling behaviors in various phases of the LSI generation process and provide (rough) scaling rules. We use the following notations:

T: number of documents in a training set or collection
W: average document length (in number of words)
L: number of languages used in the training set
F: number of factors (dimensions)

5.1. Document Collection and Preprocessing

Document collection is application dependent: it is up to the users to determine which documents are to be included to create the LSI space. Once the documents are collected, text preprocessing/parsing is straightforward: it breaks each document into a sequence of tokens and records the frequency for each distinctive term.

Based on recent experiments for TREC (Annual Text-Retrieval Conference sponsored by NIST), we have collected some measurements on the rate of the preprocessor. For one set of English documents used for filtering experiments, the rate was approximately 400 documents per minute. For another set of Chinese language documents from a multilingual collection, we achieved a processing rate of just over 100 documents per minutes. Differences in processing time can be due to a number of factors including the average length of documents in the collection and the granularity of "term" (e.g., English word vs. Chinese character). A portion of the processing time is spent updating the database tables. In order to make this stage more efficient, batch updating of these tables has been implemented.

We have observed that the preprocessing time per document is sub-linear in the average document length. The database update cost is in proportion to the number of distinct terms in a document. As a rough guide, this number grows logarithmically with the length of the document. Thus, the preprocessing time scaling rule of thumb is

$$T \times \log W.$$

5.2. Term-Document Matrix Generation

The time required to generate the term-document matrix depends on the number of documents and the number of unique terms in the training set. The number of unique terms depends, in turn, on the term thresholds, and the length and content of the training documents. In general, the number of unique terms in a document collection scales logarithmically with the number of documents in the collection. In our experiences, this logarithmic growth saturates for very large collections when there are few words that have not already be seen. In a multilingual training set, the number of unique terms also scales linearly with the number of languages. As a general rule, the time required to generate the term-document matrix scales in

$$L \times T \times \log T.$$

Note the cost includes both the database access cost and the cost internal to the C++ programs.

5.3. SVD Process

The SVD process runs in a time proportional to the number of training documents and the number of terms. It is also a strong function of the number of

factors to be extracted from the LSI space. The SVD times are roughly quadratic in the number of factors to be extracted for reasonable size of training set. Thus, the following rule guides the SVD time:

$$T \times \log T \times F \times F.$$

If we assume F scales as square root of the number of languages (which is the case in our multilingual experiments), then the rule becomes

$$T \times \log T \times L.$$

Our experiences show that, for training sets of the order of 10,000 documents, standard workstations of 1999 vintage (150-200 MHz processor and 0.5-2GB RAM) will not be able to handle more than 1000 factors. This assumes all the data are virtual memory-resident.

5.4. Document Folding

The folding process is simply a matrix multiplication procedure. Thus, it consumes very small memory and is generally not CPU bound. The time to fold in a document depends on the number of index terms in the document, which, according to our experiences, scales as the log of the number of words in the document. The time also depends on the number of factors extracted. Thus, the time spent in folding a single document is dictated by the rule

$$\log W \times F.$$

6. Future Research and Development Issues

The latency of certain LSI processes can be reduced by multithreading the processes and running the threads on multiple processors on a multi-processor server. We have explored this approach for the SVD process. Some tasks, particularly the text preprocessing, are amenable to distributing to multiple hosts.

We have already achieved substantial reductions in the number of CPU cycles consumed by the LSI query processing. This is achieved by a mixture of vector compression, fast table look-ups, short-cut cosine distance computation, and the adoption of a similarity search tree. In some phases of LSI processing, the requisite CPU cycles were reduced by a factor of 100 or more.

Another important issue is the construction of the search tree for the cluster search option. Currently, the tree takes hours to construct. For some extremely large collections (e.g. 600K documents with 400 dimensions), it takes more than 24 hours to build the tree. We plan to explore parallel techniques to speed up the process, which is heavily CPU-bound. In addition, we are exploring self-configuring techniques that will automatically determine the optimal parameters such as the cluster (bucket) size, minimum and maximum fan-out factor, and re-insertion rate (needed when a node is first split). This is of great importance as the traditional trial-and-error approach to determine the parameters will worsen the already slow tree construction process. These parameters should be determined in a manner that balances the time to build the tree and the query performance derived from the resultant tree.

References

[1] R. Ando. Latent semantic-space: iterative scaling improves precision of inter-document similarity measurement. In *Proc. of ACM SIGIR Conf.*, 2000.

[2] D. Baek, H. Lim, and H. Rim. Latent semantic indexing model for boolean query formulation. In *Proc. of ACM SIGIR Conf.*, 2000.

[3] S. Deerwester, S. Dumais, T. Landauer, G. Furnas, and R. Harshman. Indexing by latent semantic analysis. *J. of the Society for Information Science*, 41(6), 1990.

[4] C. Ding. A similarity-based probability model for latent semantic indexing. In *Proc. of ACM SIGIR Conf.*, 1999.

[5] I. Duff and R. Grimes amd J. Lewis. Sparse matrix test problems. *ACM Trans. on Mathematical Software*, 15(1), 1989.

[6] M. Hensinger. Web information retrieval. Tutorial handout, Int. Conf. on Data Engineering, 2000.

[7] T. Hofmann. Probabilistic latent semantic indexing. In *Proc. of ACM SIGIR Conf.*, 1999.

[8] F. Jiang and M. Littman. Approximate dimension equalization in vector-based information retrieval. In *Proc. of Int. Conf. on Machine Learning*, 2000.

[9] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proc. of ACM SIGMOD Conf.*, 1997.

[10] C. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala. Latent semantic indexing: A probabilistic analysis. In *Proc. of ACM PODS Symp.*, 1998.

[11] G. Salton and C. Buckley. Term-weighting approaches in automatic retrieval. *Information Processing & Management*, 24(5), 1988.

[12] D. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. of ICDE Conf.*, 1996.