Downdating the Latent Semantic Indexing Model for Information Retrieval

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Dian Irene Witter

December 1997

Acknowledgments

I thank my advisor, Dr. Michael Berry, for his patience, encouragement, support, and advice throughout this project. I would also like to thank Dr. David Straight and Dr. Brad Vander Zanden for serving on my thesis committee.

I thank Todd Letsche for all his insight and guidance on the original LSI++ code. I am grateful to my husband, John Martin, for his patience, love, and support through all aspects of this project. Also, many thanks go to my parents, Dr. and Mrs. Alan F. Witter, for supporting me through graduate school. Most importantly, I thank God for helping me complete this project.

This research has been supported by the National Science Foundation under Grant Nos. ASC-94-11394 and CDA-95-29459.

Abstract

Due to the growth of large data collections, information retrieval or database searching is of vital importance. Lexical matching techniques may retrieve irrelevant or inaccurate results because of synonyms and polysemous words, thus, effective concept based techniques are needed. One such technique is Latent Semantic Indexing (LSI) which uses a vector-space approach by identifying documents whose content is related to the user's query in order of similarity. LSI uses the singular value decomposition (SVD) of term-by-document matrix to encode the terms and documents in a vector-space model. The existing methods for removing terms or documents from the term-document space are either time consuming or do not sufficiently change the term-document relationships. This thesis presents a new method for downdating, **Downdating the Reduced Model** (or DRM) method, and discusses its implementation into the LSI++ software environment. The DRM method can be used to assess the effect that a term or document has on the clustering of relevant information in a collection and for the incorporation of user feedback in the existing LSI model. Implementing the DRM method within LSI++ not only provides downdating functionality, but is less time consuming than recomputing the SVD when removing a term, document, or both. The DRM method is a viable algorithm for dynamic information modeling and retrieval.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Statement of Problem and Goals 3

1.3 Overview 4

2 Overview of LSI and Downdating **5**

2.1 Latent Semantic Indexing 6

2.2 Querying 11

2.3 Updating 12

2.3.1 Updating Terms 14

2.3.2 Updating Documents 14

2.3.3 Updating Term Weights 15

2.4 Downdating 16

2.4.1 Downdating Terms 17

2.4.2 Downdating Documents 22

2.4.3 Updating Term Weights 24

2.5 An LSI Example with Downdating 25

3 The Design and Implementation of the DRM Method 33

3.1 The Functionality of Downdating the Reduced Model 34

3.2 DRM Implementation 36

3.2.1 Serial Implementation 36

3.2.2 C++ Classes 39

3.3 The LSI++ User Interfaces involving Downdating 52

3.3.1 World-Wide Web Interface 52

3.3.2 Command-Line Interface 56

4 Evaluating the Performance and Effectiveness of Downdating 58

4.1 Performance Evaluation 59

4.1.1 Procedure 59

4.1.2 Results 62

4.2 Effectiveness Evaluation 71

4.2.1 Procedure 71

4.2.2 Results 72

5 Conclusions 76

5.1 Future Work 77

Bibliography 78

Vita **82**

List of Tables

2.1 The labels and titles for topics on baking and music. Keywords are in boldface. 26

2.2 The 10×9 term-by-document matrix corresponding to the titles in Table 2.1. 26

2.3 Results for the query **Rock** and **Roll** using a cosine threshold of .80. . . 27

2.4 DRM results from the query **Rock** and **Roll** using a cosine threshold of .80. 29

2.5 Recomputing the SVD results for the query **Rock** and **Roll** using a cosine threshold of .80. 29

3.1 The API member functions added to incorporate downdating. 44

3.2 The required and optional features of the command-line interface for the LSI++ search engine. 57

4.1 Document collections used to examine the performance of downdating. . 59

4.2 The number of LSI factors used for each document collection. 61

4.3 The elapsed wall-clock times in seconds for downdating a document from the INTERF and MPI document collections using the in-memory (IN) implementation, the out-of-memory (OUT) implementation, and recomputing the SVD (RC). 65

4.4 The elapsed wall-clock times in minutes for downdating a document from the PCW document collection using the in-memory (IN) implementation, the out-of-memory (OUT) implementation, and recomputing the SVD (RC). 65

4.5 The elapsed time in seconds for downdating and recomputing the SVD for the MPI document collection. 66

4.6 The elapsed time in minutes for downdating (DD) and recomputing (RC) the SVD for the PCW document collection. 67

List of Figures

2.1 Local Weight Functions 7

2.2 Global Weight Functions 8

2.3 A schematic diagram of the truncated SVD with the shaded regions representing A_k [Let96]. 10

2.4 Nonzero chasing procedure based on Givens rotations for determining a 5×5 bidiagonal matrix $\tilde{B}$. 20

2.5 A rank-2 LSI model of the music/baking titles collection along with the query, **Rock** and **Roll**. 28

2.6 The rank-2 LSI model obtained from removing the term **Roll** using the DRM method. 30

2.7 Rank-2 LSI model obtained from removing the term **Roll** and recomputing the SVD. 31

3.1 The query phase with downdating [LB97]. 37

3.2 The LSI++ inheritance hierarchy with the independent classes on the left and inheritance relationships on the right [LB97]. 39

3.3 The LSI++ out-of-memory serial implementation. 41

3.4 The LSI++ in-memory serial implementation. 42

3.5 The initial query form. 53

3.6 The bottom of the second form showing the results of the initial query. . 55

3.7 The downdated term and document attached to the bottom of the query results. 56

4.1 The time complexity for the DRM method given in terms of FLOPS. . . 63

4.2 The elapsed time for downdating and recomputing the SVD for the INTERF document collection. 67

4.3 The amount of time elapsed for the worst case of downdating compared with recomputing the SVD. 69

4.4 The elapsed time compared to the number of FLOPS for the PCW collection using 45 LSI factors. 70

4.5 The initial query of **white collar crimes**. 72

4.6 The results of the search performed on the initial query along with the decision to downdate the term **white**. 73

4.7 The results of the search with downdating the term **white**. 75

Chapter 1

Introduction

Today there is an increasing flood of information from a growing number of sources. The data is virtually useless without the ability to organize, manipulate, and search this wealth of data. For this reason, the development of information retrieval techniques are of vital importance.

1.1 Motivation

As an information retrieval (IR) technique, simple lexical matching is insufficient, often producing inaccurate and irrelevant results. This is due to synonymy (multiple ways to express a concept) and polysemy (multiple meanings for a word or words) within the set of information to be searched [BDO95]. Conceptual-based IR techniques are needed to deal with these problems. Latent Semantic Indexing (LSI) [DDF+90], a vector-space approach to IR, is one such technique. LSI assumes that an underlying semantic structure of word usage exists in a document collection and uses this to estimate the semantic content of documents in the collection [LB97]. This estimation is accomplished through the truncated singular value decomposition (SVD) of a term-by-document matrix [BDO95]. The coordinates of k-space produced by the singular vectors and corresponding singular values are used to encode both terms and documents along with a user's query in the same k-dimensional vector space. Using LSI, relevant terms and/or documents can be matched and retrieved even when the query contains no words in common with the relevant documents.

The LSI model for IR has been implemented in a useful, efficient, and maintainable system known as LSI++ [LB97]. Using LSI++, a user can search large document collections quickly without knowledge of the underlying details and concepts of LSI or its implementation. Along with LSI++, methods and tools exist to perform operations within the LSI model such as parsing document collections, producing the term-by-document matrix, adding or updating terms and documents to an existing document collection, etc. [BDO95]. Multiple techniques are available to update terms and/or documents. One technique reflects the effects of adding a new term or document on the existing term-document space, but requires a huge amount of time and storage. Another technique is inexpensive in time and storage, but ignores the effects a term or document can have on the existing term-document space. The updating technique described in [O'B94] allows new terms and documents to perturb the existing LSI model without having to recompute the SVD of a new term-by-document matrix.

The opposite of updating, *downdating* refers to the removal of a term or document from the existing document collection. The incorporation of downdating is beneficial for a number of reasons. Over time, terms or documents in a collection may become obsolete. A term that once was useful for dissociating clusters (term-document relationships) in the collection may appear more frequently with addition of new documents, causing unwanted clustering. A specific document may become out-dated and its effect on the term-document space undesirable. Downdating can be used to improve the effectiveness of query matching by removing terms or documents that adversely affect retrieval, breaking apart clusters of semantically related terms and/or documents. Finally, downdating can be used to examine the effect of a term or document on the clustering within a collection. This provides knowledge about the sensitivity of the database and has implications for the hierarchical storage (caching) of large text collections.

1.2 Statement of Problem and Goals

Prior to the downdating algorithm developed in this thesis, there were two methods available for removing a term and/or document from a collection: *folding-out*, and *recomputing* the SVD to form a new LSI-generated database. Folding-out consists of simply ignoring terms or documents that are not relevant to a query. This method can misrepresent the *true* term-document relationships that result from the actual removal of those terms or documents from the original term-by-document matrix. In contrast, recomputing the SVD provides a more accurate low-rank approximation to the proper term-by-document matrix (less any specified terms or documents), but can be a time-consuming process or impossible if the term-by-document matrix cannot be constructed.

This thesis presents a new technique for downdating, **Downdating the Reduced Model** (DRM), that models the recomputation of the term-document space without destroying the original LSI model. This allows the effects of removing a term or document to be exploited with the aim of improving information retrieval and is less time-consuming than recomputing the SVD. Because this technique is implemented within the LSI++ software environment, the user can invoke downdating to help improve the precision (the proportion of the set of documents returned relevant to the query) and recall (the proportion of the relevant documents retrieved by the system from the database) [BDO95] of his or her query.

1.3 Overview

The following chapters discuss the theory, implementation, and evaluation of the DRM method for downdating within the LSI model. Chapter 2 reviews the underlying concepts of LSI and discusses the DRM algorithm. Chapter 3 introduces the design and implementation of downdating within the existing LSI++ software environment. Chapter 4 examines the execution efficiency of downdating and retrieval effectiveness of a query with downdating. Finally, chapter 5 provides a summary of the usefulness and effectiveness of the DRM model for downdating.

Chapter 2

Overview of LSI and Downdating

Latent Semantic Indexing (LSI) [BDO95] has become a widely known method for information retrieval. LSI is a mathematical-based information retrieval model which relies upon the singular value decomposition of a term-by-document matrix [BDO95]. Whereas query matching and the updating of terms and documents are available LSI tools [Let96, O'B94], downdating is yet another important tool of LSI for inclusion in the LSI++ software environment.

This chapter provides a brief overview of LSI along with a brief explanation of query matching and the updating of terms and documents. The mathematical algorithm for downdating is explored in detail followed by an example.

2.1 Latent Semantic Indexing

LSI is a vector space model based on the singular value decomposition of a term-by-document matrix A. Suppose A is $m \times n$ where $m \gg n$, i.e. there are more unique terms than unique documents [BDO95], and the rank of A is r. Each element of A is represented by

$$A = a_{ij},$$

where a_{ij} specifies the frequency with which term i appears in document j. The matrix A is usually sparse since the number of terms appearing in a document is far less than the total number of terms [Let96].

Local and global weighting functions are typically used to define each a_{ij}, so that

$$a_{ij} = L(i,j) * G(i), \tag{2.1}$$

where $L(i,j)$ is the local weighting function and $G(i)$ is the global weighting function. The local weighting function $L(i,j)$ is used to emphasis or deemphasize the weight of term i within document j whereas the global weighting function G(i) is used to emphasize or deemphasize the weight of term i across the entire document collection. Local and global weighting functions [Dum91] that have been used in LSI are shown in Figure 2.1 and Figure 2.2.

Once a weighting scheme has been applied to each element a_{ij} of A, the singular

Binary	$L(i,j) = \begin{cases} 0, \text{ if } tf_{ij} = 0 \\ 1, \text{ if } tf_{ij} = 1 \end{cases}$
Term-frequency	$L(i,j) = tf_{ij}$
Log	$L(i,j) = \log(tf_{ij} + 1)$

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

Figure 2.1: Local Weight Functions

Normal	$G(i) = \sqrt{\dfrac{1}{\sum_j L(i,j)^2}}$
GfIdf	$G(i) = \dfrac{gf_i}{df_i}$
Idf	$G(i) = \log_2\left(\dfrac{ndocs}{df_i}\right) + 1$
Entropy	$G(i) = 1 + \sum_j \dfrac{p_{ij}\log_2(p_{ij})}{\log_2(ndocs)}$ where $p_{ij} = \dfrac{tf_{ij}}{gf_i}$

Definitions:

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, and
ndocs = the number of documents in the collection.

Figure 2.2: Global Weight Functions

value decomposition of A is computed as,

$$A = U \Sigma V^T, \tag{2.2}$$

where the $m \times m$ U matrix and $n \times n$ V matrix are orthogonal, i.e., $U^T U = V^T V = I_n$, where I_n is the identity of order n. The columns of U and V define the left and right singular vectors of matrix A, which are the orthonormal eigenvectors associated with AA^T and $A^T A$, respectively. Σ is a diagonal matrix representing the singular values of A, which are the nonnegative square roots of the eigenvalues of $A^T A$ [BDO95]. As illustrated in Figure 2.3, the first k columns of U and V and the first k diagonal elements of Σ can be used to construct a rank-k approximation to A defined as $A_k = U_k \Sigma_k V_k^T$ [BDO95]. As discussed in [LB97], A_k is the closest (in the least squares sense) rank-k matrix to A. Much of the *noise* that leads to poor information retrieval is lost with the reduced dimensionality of A_k, thus allowing documents and terms similar in content and/or meaning to be related [Let96].

Using the rank-k model A_k, the associated vector space represents a semantic structure for the term and documents in the LSI-generated database. Each term vector i in the vector space is in the i-th row of U_k whose columns are scaled (or multiplied) by the k singular values of Σ_k. The same is true for each document vector except the i-th row of V_k is scaled by the k singular values of Σ_k [DDF$^+$90].

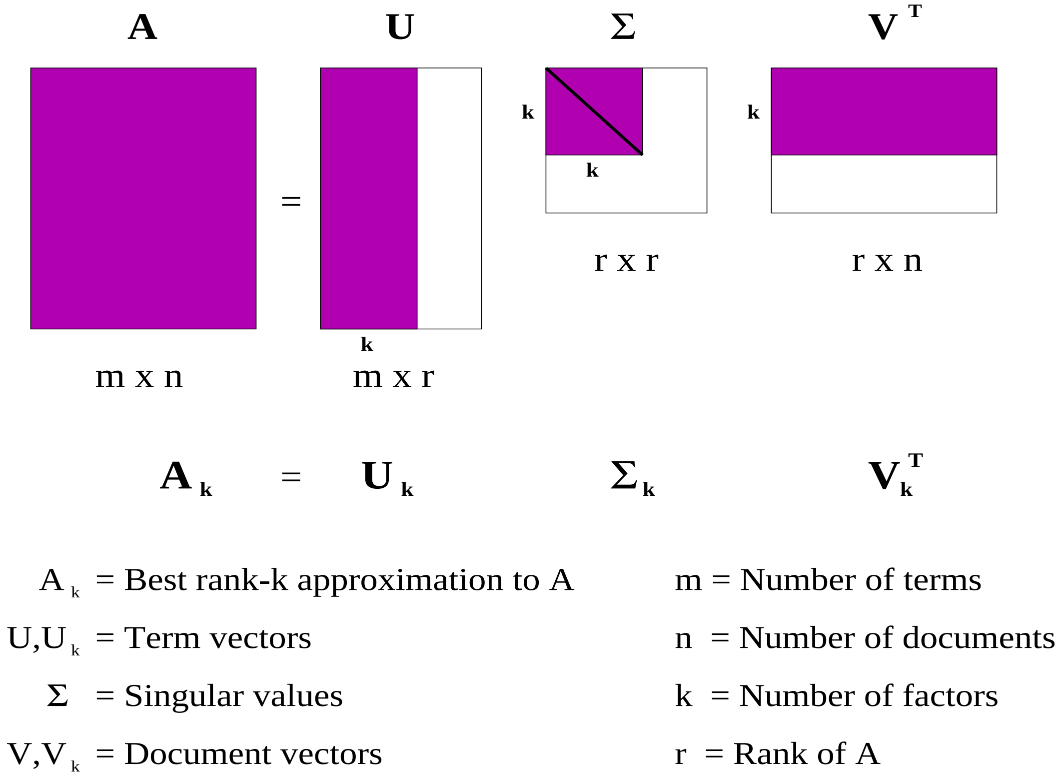


Figure 2.3: A schematic diagram of the truncated SVD with the shaded regions representing A_k [Let96].

2.2 Querying

In the LSI model implemented in LSI++, there are two parts to a query vector which represents the user's query: the search terms and the documents selected in relevance feedback [SB90]. As stated in [Let96], relevance feedback helps improve a user's query because the terms in the selected documents are added to the initial query terms, thus empowering the query to get better precision and recall during retrieval. To specify the location of a query in the reduced term-document space, the query vector is formed into a *pseudo-document*. First, the querying terms are represented by a $m \times 1$ vector q whose elements are either zero or correspond to the frequency of a valid query term or keyword in the LSI-generated database. The appropriate local and global weighting functions (see Figures 2.1 and 2.2) for the document collection are applied to each nonzero element (term frequency) in vector q. Represented as a *pseudo-document* without relevance feedback, the query vector $\hat{q}$ is

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

where $q^T U_k$ is the sum of term vectors specified by vector q scaled by Σ_k^{-1} [Let96]. To add relevance feedback, a $n \times 1$ vector d must be constructed whose nonzero elements are indices specifying the relevant document vectors. The representation of the query vector $\hat{q}$ as a *pseudo-document* with relevance feedback is

$$\hat{q} = q^T U_k \Sigma_k^{-1} + d^T V_k, \tag{2.3}$$

where $d^T V_k$ is the sum of the document vectors specified by d. The query vector $\hat{q}$ is then compared to all existing term and document vectors using an appropriate similarity measure (e.g., the cosine), and a ranked list of terms and/or documents is returned [BF96]. The highest ranked terms or documents are assumed to be most similar to the query.

When comparing $\hat{q}$ to documents, the representation specified by Equation (2.3) is used. In computing the cosine between the query vector, $\hat{q}$, and each document vector, the query vector must be appropriately scaled (or multiplied by Σ_k) if the document vectors have been scaled in the vector space model (see Section 2.1). When comparing the query vector to terms, the representation for $\hat{q}$ is no longer applicable and any document d specified by relevance feedback must be represented in light of terms. The query vector representation in this case becomes

$$\hat{q} = q^T U_k + d^T V_k \Sigma_k^{-1}. \tag{2.4}$$

Again, when computing the cosine between the query vector and the term vectors, each vector is appropriately scaled (or multiplied by Σ_k) [Let96].

2.3 Updating

One approach for the addition of terms and documents into the vector space has been examined and implemented for LSI [BDO95, O'B94]. Three processes used to add terms and documents to the LSI-generated database are folding-in, recomputing the SVD, and SVD-updating. Folding-in consists of adding terms and documents to the existing vector space. To fold-in a new $m \times 1$ document vector d whose elements have been appropriately weighted, d is projected onto the span of the current term vectors (i.e., the columns of U_k). Let d_p represent the document projection of d, then

$$d_p = d^T U_k \Sigma_k^{-1}. \tag{2.5}$$

Similarly, to fold-in a $1 \times n$ term vector t whose elements specify the documents associated with the term, t is projected onto the span of the pre-existing document vectors (i.e., the columns of V_k). Let t_p represent the term projection of t, then

$$t_p = t V_k \Sigma_k^{-1}. \tag{2.6}$$

Folding-in does not alter the representation of the pre-existing terms and documents in the term-document space; therefore, any influence of the additional terms or documents is not detected. Since folding-in has no effect on the semantic structure of the database, recomputing the SVD is always an option for updating. However, it can be expensive in memory and computationally time consuming [O'B94]. To save time and memory and the ability to examine the effects of new terms and documents on the current vector space model, SVD-updating described in [BDO95] is a suitable alternative. SVD-updating consists of the following three steps: updating terms, updating documents, and updating term weights.

2.3.1 Updating Terms

To add t term vectors to an existing LSI database, let T denote the $t \times n$ term vectors where n is the number of documents in the collection. T is essentially appended to the rows of the rank-k matrix A_k (see Figure 2.3). Let

$$B = \left(\frac{A_k}{T}\right), \tag{2.7}$$

where B is a $(m+t) \times n$ matrix. As discussed in [BDO95], the truncated SVD of B is used to obtain new term vectors U_B, new document vectors V_B, and new singular values Σ_B for scaling purposes. $B_k = U_B \Sigma_B V_B^T$ is the rank-k model reflecting the change in semantic structure of the database and in the term-document space due to the addition of terms [BDO95].

2.3.2 Updating Documents

Adding d document vectors to an LSI-generated database is similar to adding t term vectors. Let D denote the $m \times d$ document vectors where m is the number of terms. D is appended to the columns of the original A_k matrix so that

$$B = (A_k \mid D), \tag{2.8}$$

where B is a $m \times (n+d)$ matrix. As noted in the previous section, the truncated (rank-k) SVD of B given by $B_k = U_B \Sigma_B V_B^T$ is then used to reflect the change in the term-document space with the addition of documents [BDO95].

2.3.3 Updating Term Weights

Once all valid terms or documents have been updated, a correction step must be performed to reflect the change in local and global frequencies for each affected term. If j terms have been updated and need a change in weighting, a $(m+t) \times j$ matrix Y_j is constructed whose rows are comprised of zeros or rows of the j-th order identity matrix I_j. In addition, a $(n+d) \times j$ matrix Z_j is determined (columnwise) to specify the differences between the old and new weights for the j terms [BDO95]. If A_k is a $(m+t) \times (n+d)$ rank-k matrix with updated terms and documents, then it follows that

$$B = A_k + Y_j Z_j^T, \tag{2.9}$$

is the appropriate matrix to represent the term weight corrections for A_k. As demonstrated in [BDO95], the truncated SVD of B denoted by $B_k = U_B \Sigma_B V_B^T$ can then be used as the rank-k LSI model reflecting the change in the term and document vectors with term weight corrections [BDO95].

Once these three steps have been performed, the resulting terms or term vectors (rows) of U_B, resulting documents or document vectors (rows) of V_B, and the singular values Σ_B reflect changes in the vector space due to the addition of terms and documents.

SVD updating may improve retrieval because the addition of terms or documents can help define clusters of related terms and documents or perhaps dissociate unrelated terms and documents [O'B94].

2.4 Downdating

As stated in Chapter 1, downdating constitutes the removal of terms and/or documents from the vector space model. There are three methods which can be used to remove a term or document from the existing LSI-generated database: folding-out, recomputing the SVD, and downdating the Reduced Model (or DRM).

Folding-out essentially means ignoring the term or document in the vector space, ignoring the corresponding row of U_k or V_k, respectively. Therefore, during query matching, those terms and documents removed or *folded-out* are simply not used in similarity comparisons. Folding-out does not compensate for any effect that removing a term or document might have on term-document associations, clustering, or meanings in the collection [BW97].

As discussed in Section 2.3, recomputing the SVD offers the same complications as it does in updating. While recomputing the SVD will model explicit changes in the vector space model, it is expensive in memory and computational time.

Downdating the Reduced Model models the effects of removing a term or document from the vector space using the rank-k model A_k rather than the original matrix A. Models for downdating the SVD are described in [GE95] and [AB90]. The method presented in [GE95] is based upon algorithms to reduce the downdating problem to an eigenvalue problem of a matrix with simple structure whose solution can be determined from the roots of a secular equation. For LSI++, the method described by Björck in [AB90] is used. This procedure constitutes a backward stable algorithm developed by Park and Van Huffel for computing the SVD of (e_1, A) where e_1 is an added dummy column [PH95].

Downdating the Reduced Model consists of three steps: downdating terms, downdating documents, and updating term weights. The first two steps are applied to semantically remove either a term or document, respectively, while the third step follows from the removal of a document. Changes in the global weightings of the term vectors associated with the terms are needed when any document is removed.

2.4.1 Downdating Terms

Downdating a term refers to the removal of a term from the database, or in the case of LSI, removing the row associated with that term from the existing term vectors (rows) in U_k. As discussed in Section 2.1, the k largest singular values (Σ_k) and the corresponding singular vectors (columns of U_k and V_k) are used to construct the reduced rank matrix A_k which can be written as

$$A_k = \begin{pmatrix} z^T \\ \tilde{A}_k \end{pmatrix} = U_k \Sigma_k V_k^T, \tag{2.10}$$

where z^T is the first row of A_k which is to be downdated. Define e_1 as the $m \times 1$ column vector of all zeros except the first element whose value is 1, and define the $m \times (k+1)$ orthogonal matrix $\hat{U}$ given by

$$\hat{U} = [U_k \mid s],$$

where s is a $m \times 1$ vector orthogonal to U_k (i.e., $U_k^T s = 0$). Then,

$$\hat{U}^T (e_1, A) \begin{pmatrix} 1 & 0 \\ 0 & V_k \end{pmatrix} = \begin{pmatrix} U_k^T e_1 & \Sigma_k \\ q_1 & 0 \end{pmatrix}, \text{where } \mathrm{q}_1 = \mathrm{s}^{\mathrm{T}} \mathrm{e}_1. \tag{2.11}$$

The first column of the right-hand-side matrix of Equation (2.11), referred to as H, is essentially the first row of U_k followed by the first element of s. Accumulated Givens rotation matrices G_l and G_r can be constructed [AB90] so that

$$G_l H G_r = G_l \begin{pmatrix} U_k^T e_1 & \Sigma_k \\ q_1 & 0 \end{pmatrix} G_r = \begin{pmatrix} 1 & w^T \\ 0 & \tilde{B} \end{pmatrix}, \tag{2.12}$$

where $\tilde{B}$ is a $k \times k$ upper bidiagonal matrix. Using the last two nonzero entries in the first column of H, a Givens rotation [GL96] can be determined so that pre-multiplication by the orthogonal rotation matrix effectively zeros out the last nonzero entry in the first column. This Givens rotation constitutes just one rotation which defines the series of rotations accumulated in the $(k+1) \times (k+1)$ orthogonal matrix G_l. Left multiplication by a Givens rotation, of course, may introduce a nonzero element above the superdiagonal of the modified H matrix. Such entries can be zeroed out by an appropriate Givens rotation applied to the right side of H (or to the columns of H). Post-multiplication of H by a Givens rotation (accumulated in the $(k+1) \times (k+1)$ orthogonal matrix G_r), in turn, may cause a nonzero element to appear below the diagonal of matrix H, thus requiring left multiplication by another Givens rotation for removal. This sequence of left and right multiplication by Givens rotations continues until the all nonzero elements appearing below the diagonal or above the superdiagonal entries of H are *chased* out the lower-right corner of H to produce the resulting bidiagonal matrix $\tilde{B}$ [AB90] (see Figure 2.4).

Using Equation (2.12), it can be shown [BW97] that the singular values of the $k \times k$ matrix $\tilde{B}$ are the same as those of $\tilde{A}_k$ from Equation (2.10). If the $(k+1) \times (k+1)$ orthogonal matrices G_l and G_r are defined such that

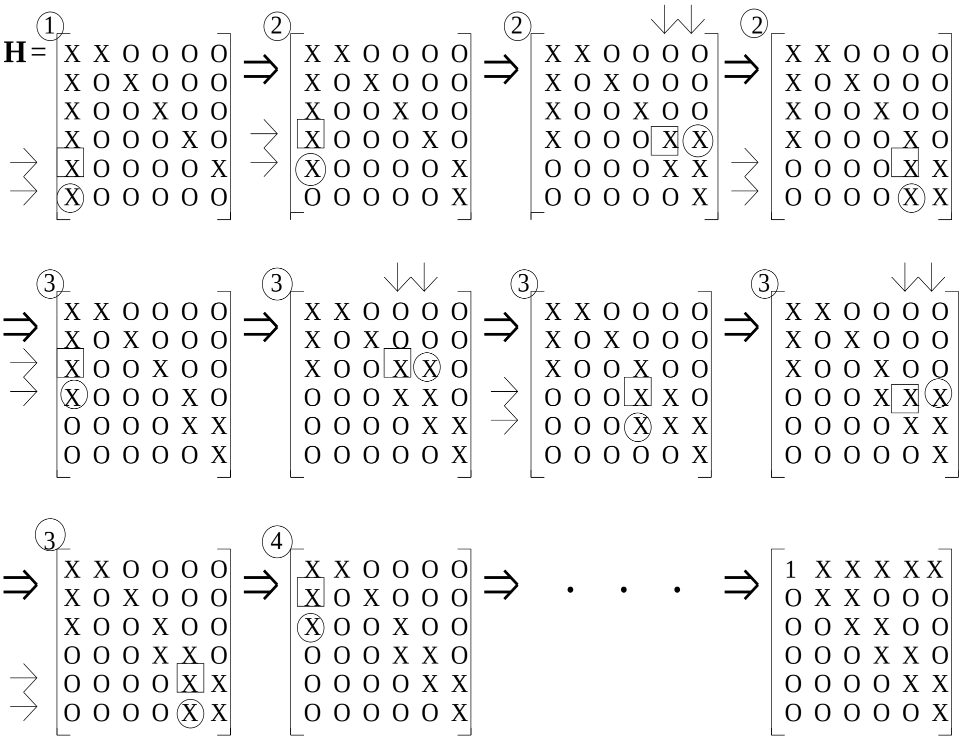
$$G_l \hat{U}^T = \begin{pmatrix} 1 & 0 \\ 0 & \bar{U}_k^T \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 \\ 0 & V_k \end{pmatrix} G_r = \begin{pmatrix} 1 & 0 \\ 0 & \bar{V}_k \end{pmatrix},$$

it follows that

$$\begin{pmatrix} 1 & 0 \\ 0 & \bar{U}_k^T \end{pmatrix} \begin{pmatrix} 1 & z^T \\ 0 & \tilde{A}_k \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \bar{V}_k \end{pmatrix} = \begin{pmatrix} 1 & w^T \\ 0 & \tilde{B} \end{pmatrix}. \tag{2.13}$$

Consequently, the second row of Equation (2.13) yields the downdated bidiagonal form of $\tilde{A}_k$, i.e.,

$$\bar{U}_k^T \tilde{A}_k \bar{V}_k = \tilde{B}. \tag{2.14}$$



Symbols:

X is the entry that needs to be zeroed

X is the entry used to define the Givens rotation to zero out the X entry

\# indicates the step number

Figure 2.4: Nonzero chasing procedure based on Givens rotations for determining a 5×5 bidiagonal matrix $\tilde{B}$.

The $k \times k$ bidiagonal matrix $\tilde{B}$ can be diagonalized using the Golub-Kahan diagonalization method described in [GL96, pages 452–456]. This procedure will produce a diagonal matrix composed of the singular values equivalent to those singular values of $\tilde{B}$ and of the downdated $(m-1) \times k$ matrix $\tilde{A}_k$ from Equation (2.14). Hence,

$$U_B^T \tilde{B} V_B = \Sigma_B, \tag{2.15}$$

where U_B and V_B are $k \times k$ orthogonal matrices constructed from Givens rotations [GL96]. From Equations (2.14) and (2.15) it then follows

$$\tilde{A}_k = (\bar{U}_k U_B)\Sigma_B(V_B^T \bar{V}_k^T). \tag{2.16}$$

The term represented by z in Equation (2.10) has been downdated and the new LSI model is given by

$$\tilde{A}_k = \tilde{U}_k \tilde{\Sigma}_k \tilde{V}_k^T, \tag{2.17}$$

where $\tilde{U}_k = \bar{U}_k U_B$, $\tilde{\Sigma}_k = \Sigma_B$, and $\tilde{V}_k = \bar{V}_k V_B$. Any row of the matrix A_k in Equation (2.10) can be downdated using the process described in this section. A selected row can be permuted to the first row to conform to the representation of $\tilde{A}_k$ in Equation (2.10). To downdate more than one term simply requires that the process be applied iteratively. The resulting $\tilde{A}_k$, $\tilde{U}_k$, $\tilde{\Sigma}_k$, and $\tilde{V}_k$ matrices reflect a change in the semantic model used by LSI for subsequent query matching.

2.4.2 Downdating Documents

Similar to downdating a term, downdating a document involves removing a document (row vector) from the existing document vectors in V_k. In this case, the reduced rank A_k matrix is cast as

$$A_k = [z \ \tilde{A}_k] = U_k \Sigma_k V_k^T, \tag{2.18}$$

where z is the first column of A_k which is to be downdated. Since any column can be permuted to the first column of A_k, the procedure described in this section can be used to downdate any document. Following the steps for downdating a term, define e_1 as a $n \times 1$ column vector of all zeros except the first element whose value is 1, and define $\hat{V}$ as an $n \times (k+1)$ orthogonal matrix of the form

$$\hat{V} = [V_k \,|\, s],$$

where the $n \times 1$ vector s is orthogonal to V_k (i.e., $V_k^T s = 0$). Then

$$\begin{pmatrix} 1 & 0 \\ 0 & U_k^T \end{pmatrix} \begin{pmatrix} e_1^T \\ A_k \end{pmatrix} \hat{V} = \begin{pmatrix} e_1^T V_k & q_1 \\ \Sigma_k & 0 \end{pmatrix} \text{where } \mathrm{q_1 = e_1^T s}. \tag{2.19}$$

The first row of the right-hand-side of Equation (2.19), referred to as W, is composed of the first row of V_k followed by the first element of s. Note that W^T has the same form as the matrix H for downdating terms (see Section 2.4.1). Using the nonzero chasing scheme based on Givens rotations illustrated in Figure 2.4, one can obtain

$$G_l W^T G_r = G_l \begin{pmatrix} V_k^T e_1^T & q_1 \\ \Sigma_k & 0 \end{pmatrix}^T G_r = \begin{pmatrix} 1 & 0 \\ w & \tilde{B} \end{pmatrix}^T, \tag{2.20}$$

where $\tilde{B}$ is a lower bidiagonal matrix [AB90]. As discussed in [BW97], the singular values of the $k \times k$ $\tilde{B}$ matrix are the same as those of $\tilde{A}_k$ in Equation (2.18). If G_l and G_r are $(k+1) \times (k+1)$ orthogonal matrices constructed from Givens rotations so that

$$G_r^T \begin{pmatrix} 1 & 0 \\ 0 & U_k^T \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \bar{U}_k^T \end{pmatrix} \text{ and } \hat{\mathrm{V}}\mathrm{G}_\mathrm{l}^\mathrm{T} = \begin{pmatrix} 1 & 0 \\ 0 & \bar{V}_k \end{pmatrix},$$

then it follows that

$$\begin{pmatrix} 1 & 0 \\ 0 & \bar{U}_k^T \end{pmatrix} \begin{pmatrix} 1 & 0 \\ z & \tilde{A}_k \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \bar{V}_k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ w & \tilde{B} \end{pmatrix}. \tag{2.21}$$

As with Equation (2.13), the second row of Equation (2.21) yields the downdated $\tilde{A}_k$ given by

$$\bar{U}_k^T \tilde{A}_k \bar{V}_k = \tilde{B}. \tag{2.22}$$

$\tilde{B}$ in this case, however, is a lower bidiagonal matrix. Applying the Golub-Kahan diagonalization method [GL96, pages 452-456] to $\tilde{B}^T$ produces

$$G_l \tilde{B}^T G_r = G_r^T \tilde{B} Gl^T = U_B^T \tilde{B} V_B = \Sigma_B, \tag{2.23}$$

where the singular values of $\tilde{B}$ are the diagonal elements of Σ_B. Combining Equations (2.22) and (2.23) yields

$$\tilde{A}_k = (\bar{U}_k U_B)\Sigma_B(V_B^T \bar{V}_k^T), \text{ or } \tilde{A}_k = \tilde{U}_k \tilde{\Sigma}_k \tilde{V}_k^T, \tag{2.24}$$

where $\tilde{U}_k = \bar{U}_k U_B$, $\tilde{\Sigma}_k = \Sigma_B$, and $\tilde{V}_k = \bar{V}_k V_B$. The document represented by z in Equation (2.18) is then considered downdated in the reduced rank model $\tilde{A}_k$.

2.4.3 Updating Term Weights

The final phase of downdating, referred to as updating term weights, is applicable when a document is downdated. If a term is removed from the text collection, the weights of the other terms are not affected, locally or globally. If a document is removed, however, the global weights (not the local weights, see Equation (2.1)) of the terms spanned by that removed document will change. The term frequencies across the collection have changed; therefore, updating the term weights of those terms corresponding to the terms in the removed document vector is essential.

To update the weights of those terms affected by the removed documents, the technique for updating weights described in Section 2.3.3 for updating can be applied. The $(m-t) \times j$ matrix Y_j in this case denotes exactly which terms are affected, and the $(n-d) \times j$ matrix Z_j reflects the difference of subtracting the old weights from the new weights for the terms affected. The truncated (rank-k) SVD of $B = A_k + Y_j Z_j^T$ from Equation (2.9) is then computed, where A_k is a $(m-t) \times (n-d)$ matrix. Since the singular vectors stored in U_k and V_k, and the corresponding singular values (Σ_k) reflecting all term or document removals are available, the SVD of $R = U_k^T B V_k$ can be computed. An approximation to SVD(R) as described in [BDO95] can be obtained from the SVD of $\tilde{R} = \Sigma_k + U_k^T Y_j Z_j^T V_k = U_B \Sigma_B V_B^T$. If the rank of B is the same as that of A_k, then the resulting orthogonal matrices U_B and V_B will be $(m-t) \times k$ and $(n-d) \times k$, respectively.

After applying the term weight update procedure, the final U_k, V_k, and Σ_k, where $U_k = U_B$, $V_k = V_B$, and $\Sigma_k = \Sigma_B$, reflect a new LSI model A_k with removed terms and/or documents. As a result of the procedures from Section 2.4.1 through Section 2.4.3, terms and documents as represented by the new A_k may form new clusters or fuse into larger clusters depending on the effects the downdated terms or documents may have had on the underlying semantic structure of term-document associations.

2.5 An LSI Example with Downdating

The following example is used to show the effectiveness of the Downdating the Reduced Model (DRM) method in comparison to recomputing the SVD of A_k for removing a term or document from a database.

The titles listed in Table 2.1 reflect document titles for articles describing rock music and baking. Keywords or indexing terms for this collection of titles are chosen as words that appear more than once in the collection. The corresponding term-by-document matrix A for this small document collection is provided in Table 2.2. For simplicity,

Table 2.1: The labels and titles for topics on baking and music. Keywords are in boldface.

Label	Titles
m1	**Rock** and **Roll Music** in the 1960's
m2	**Recipe** for a **Drum Roll,** a **Demonstration** of Techniques
m3	**Drum** and Bass **Composition**
m4	A Perspective of **Rock Music** in the 90's
m5	**Music** and **Composition** of Popular Bands
b1	How to make **Bread** and **Rolls**, a **Demonstration**
b2	**Ingredients** for Crescent **Roll Dough**
b3	A **Recipe** for Sour**dough Bread**
b4	A Quick **Recipe** for Pizza **Dough** with Natural **Ingredients**

Table 2.2: The 10×9 term-by-document matrix corresponding to the titles in Table 2.1.

Terms	**Documents**								
	m1	m2	m3	m4	m5	b1	b2	b3	b4
rock	1	0	0	1	0	0	0	0	0
drum	0	1	1	0	0	0	0	0	0
music	1	0	0	1	1	0	0	0	0
composition	0	0	1	0	1	0	0	0	0
demonstration	0	1	0	0	0	1	0	0	0
roll	1	1	0	0	0	1	1	0	0
bread	0	0	0	0	0	1	0	1	0
dough	0	0	0	0	0	0	1	1	1
ingredients	0	0	0	0	0	0	1	0	1
recipe	0	1	0	0	0	0	0	1	1

Table 2.3: Results for the query **Rock** and **Roll** using a cosine threshold of .80.

Terms	**Cosine**	**Documents**	**Cosine**
roll	0.9462	m3	0.9994
drum	0.9285	m1	0.9810
demonstration	0.8770	m4	0.8814
rock	0.8666	m5	0.8755
composition	0.8498	b1	0.8293
music	0.8470		

local weights (see Section 2.1) are simply the term frequencies and no global weights (i.e., $G(i) = 1$) are applied. Given matrix A, the truncated SVD of A is computed to produce the rank-2 approximation (A_2) illustrated in Figure 2.5. The query **Rock** and **Roll** is then projected into the 2D space (see Section 2.2), and ranked lists of the largest cosines between the query vector and the term and document vectors are provided in Table 2.3. Given a cosine threshold of .80, the query **Rock** and **Roll** matches related terms and documents. Note that documents/titles m3 and m5 do not even contain the query terms. This ability to retrieve relevant documents which do not necessarily span the literal query terms is a strength of LSI. Clearly document/title b1 which refers to baking bread is not relevant. The polysemous term **Roll** forces an association between the bread and music titles. Notice the outlier position of the term **Roll** with respect to the intersecting clusters of bread documents (b1 - b4) and music documents (m1-m5) in Figure 2.5.

An attempt to *fold-out* the term **Roll** will not change the position of any other term or document, and therefore will not change the associations of baking to rock music.

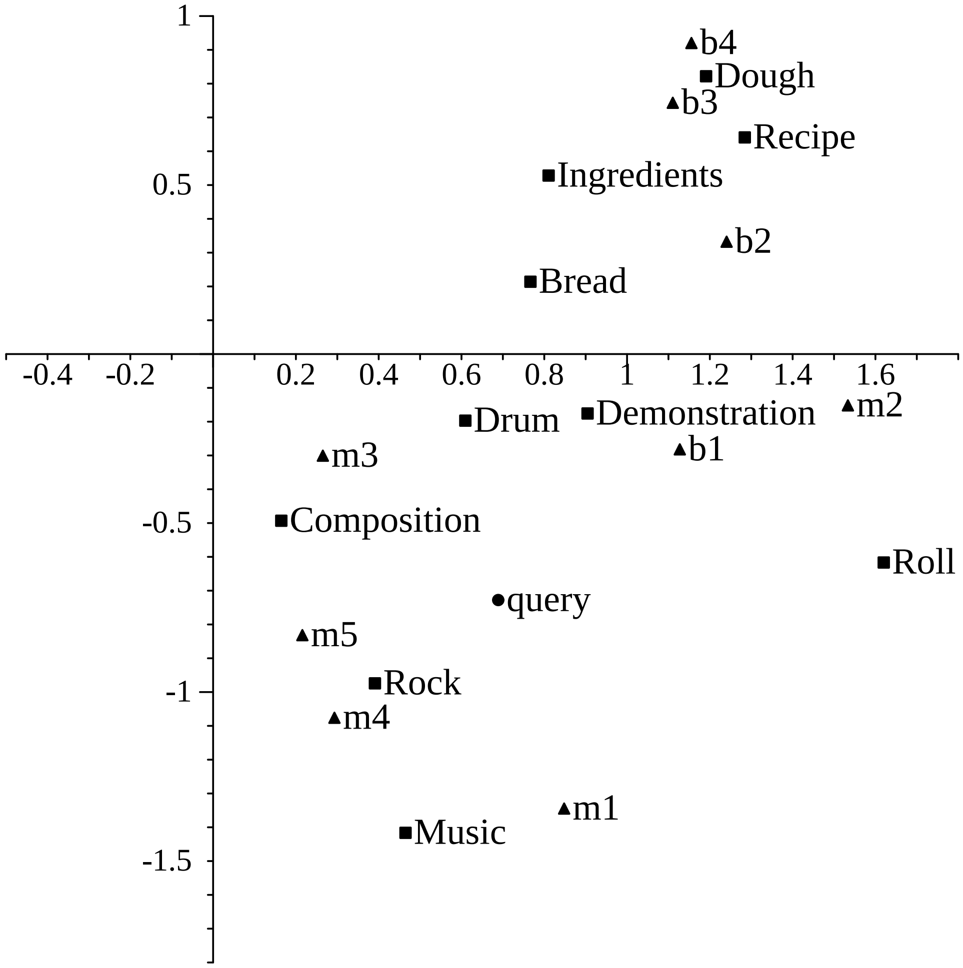


Figure 2.5: A rank-2 LSI model of the music/baking titles collection along with the query, **Rock** and **Roll**.

Table 2.4: DRM results from the query **Rock** and **Roll** using a cosine threshold of .80.

Terms	**Cosine**	**Documents**	**Cosine**
rock	1.0000	m1	0.9989
composition	0.9995	m3	0.9742
music	0.9993	m4	0.9683
		m5	0.9653

Table 2.5: Recomputing the SVD results for the query **Rock** and **Roll** using a cosine threshold of .80.

Terms	**Cosine**	**Documents**	**Cosine**
rock	1.0000	m1	1.0000
music	1.0000	m4	1.0000
composition	0.9905	m5	0.9992
		m3	0.8986

The effect of applying the DRM method to downdate the term **Roll** is illustrated in Figure 2.6. The rank-2 LSI model based on the recomputed SVD of a $(m-1)\times n$ matrix $\tilde{A}$ after the term **Roll** was removed is illustrated in Figure 2.7. For both Figures 2.6 and 2.7, the terms and documents associated with music are aligned along the y axis while the terms and documents associated with baking are now aligned along the x axis.

Tables 2.4 and 2.5 list the cosines between the query vector, **Rock** and **Roll**, and the most similar terms and documents for the DRM and recomputing the SVD methods, respectively. Using the cosine threshold of .80, the terms and documents returned by either LSI model are similar and do not encompass any intrusion by baking terms or documents. Although a fairly simplistic example, this baking/music title collection demonstrates that

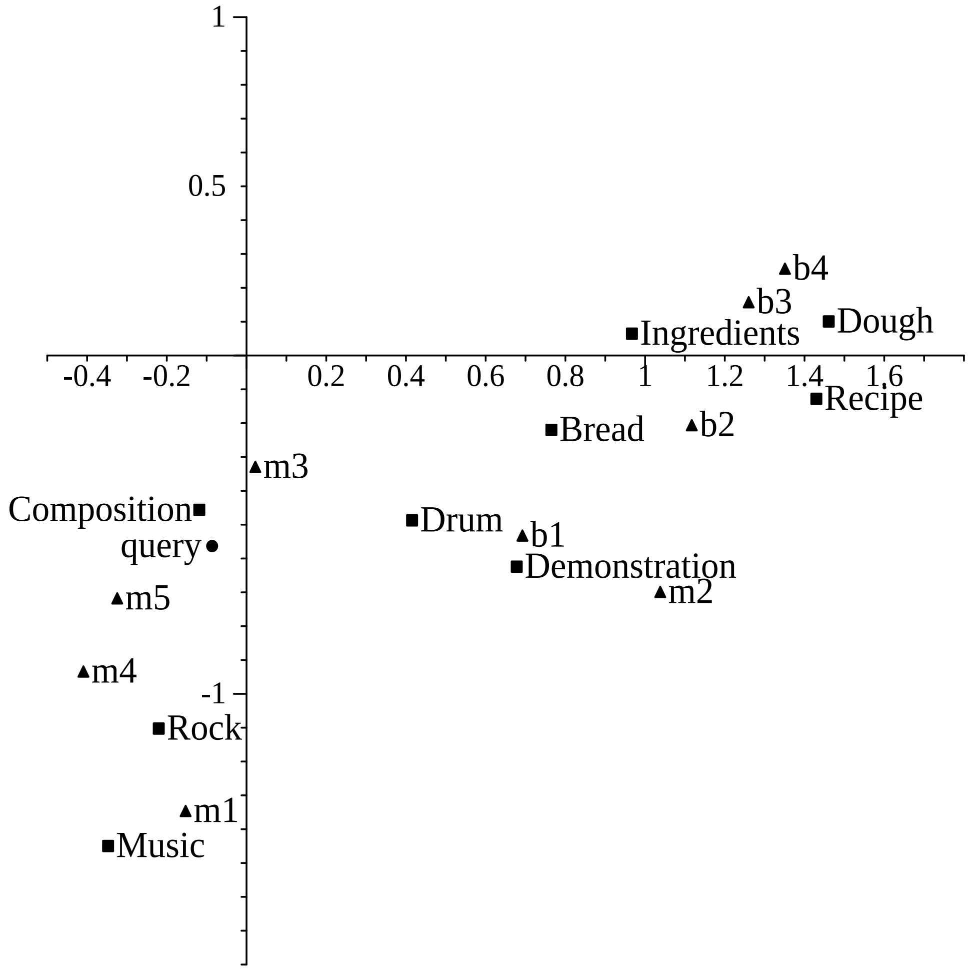


Figure 2.6: The rank-2 LSI model obtained from removing the term **Roll** using the DRM method.

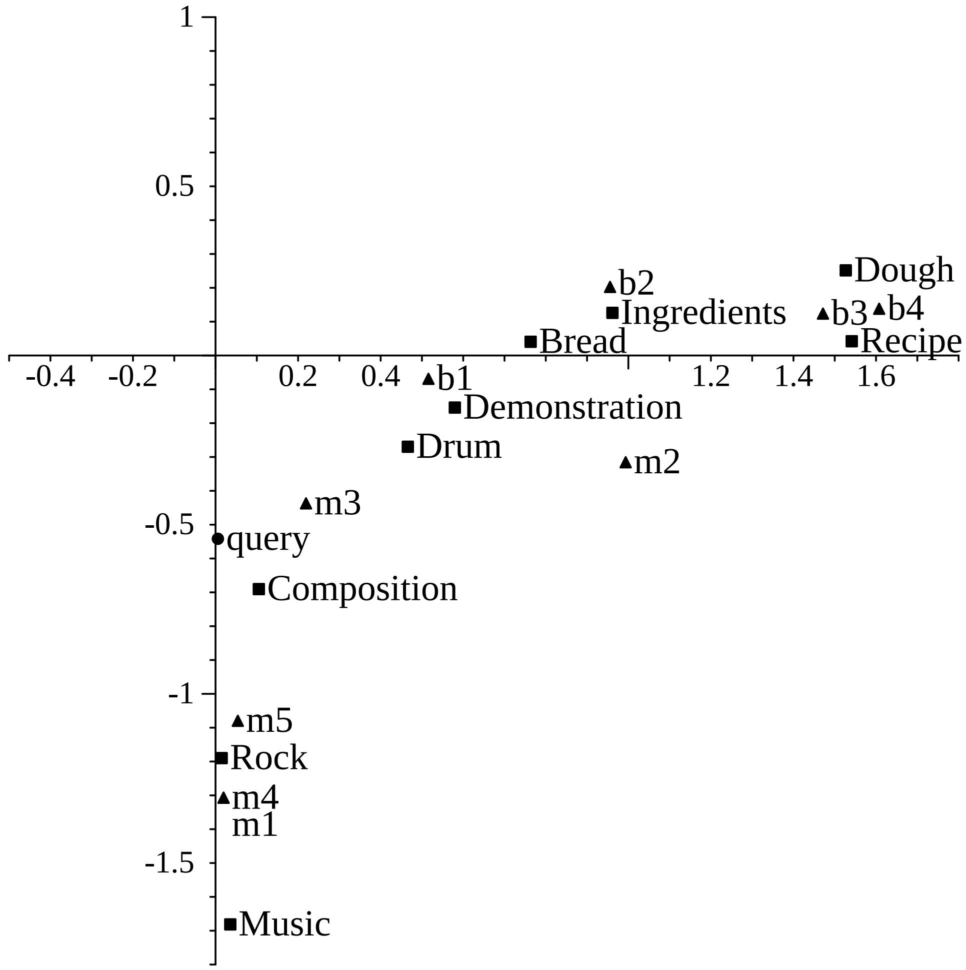


Figure 2.7: Rank-2 LSI model obtained from removing the term **Roll** and recomputing the SVD.

1. removing a term or document can change the underlying semantic space, thus, possibly providing better precision and recall (see Introduction) for the query, and

2. the DRM method produces a vector space encoding similar to that obtained by a procedure based on recomputing the SVD of a matrix with certain terms or documents removed.

The DRM method is especially useful when the matrix A is unavailable or has been discarded so that recomputing the SVD is no longer possible.

Chapter 3

The Design and Implementation of the DRM Method

Prior to the DRM method, the only implementations of downdating for the LSI model were folding-out and recomputing the SVD. In folding-out, the user essentially ignores certain terms or documents when querying. This method does not reflect the effect a term or document removal may have on term-document associations. Recomputing the SVD is not invoked by the user. Instead, the LSI administrator or software manager recomputes the SVD with certain terms and documents removed to produce a new term-document space. Since neither implementation is ideal, a more effective downdating technique is required.

The LSI++ software environment described in [LB97, Let96] provides an interface that is easy to use. The user is able to enter a query in a simple manner, and quickly receive understandable results. Given these features and the need for downdating, an implementation of downdating within LSI++ would be beneficial for the user. As illustrated by the encouraging results in Section 2.5, the DRM method would be a valuable downdating algorithm to integrate with the LSI++ software environment. To this end, a downdate option has been implemented as a feature with the LSI++ software. This chapter comprises a detailed discussion of downdating functionality, integration with the LSI++ software, and the user interface.

3.1 The Functionality of Downdating the Reduced Model

In the LSI++ software environment, a query is defined by the the eight-tuple

$$q =< s, k, h, b, t, d, r, c >$$

where

s denotes the directory of the document collection to be searched,

k indicates the number of LSI factors to use in representing the query, term, and document vectors,

h denotes how many terms, documents, or both to return,

b indicates whether or not to return a ranked list of terms and/or documents,

t identifies the terms that describe the query,

d identifies the documents in relevance feedback to be used in the query,

r indicates the terms to downdate from the existing document collection, and

c indicates the documents to downdate from the existing document collection.

A query requires query terms t or relevance feedback documents d in addition to user-specified or default values for s, k, h, and b [LB97]. Unless the user wishes to downdate a particular term or document before the query is performed, terms r and documents c are not required.

The LSI model performs the following steps to downdate a term or document:

1. **Examine r and c to ensure that each term and/or document to be downdated is valid for the specified document collection.**

   Any term or document to be downdated must reside in the document collection. Hence, each term must be a valid keyword.

2. **Examine t and d for validity.**

   Once a term or document is downdated, it is conceptually removed from the document collection; therefore, query terms and relevance feedback documents cannot be selected from the list of downdated terms or documents.

3. **Downdate any r or c by performing the following the steps described in Section 2.4.**

4. **Form the query vector or *pseudo-document* with the query terms and relevance feedback documents as illustrated in Section 2.2.**

5. **Project the query vector into the term-document space and return a ranked list of h terms, documents, or both.**

From the list above, Steps 2, 4, and 5 are always performed in accordance with the pre-existing specifications for LSI++ [LB97]. The remaining steps are performed only if the user wishes to downdate a term or document. A representation of the query phase with the incorporation of downdating is illustrated in Figure 3.1.

3.2 DRM Implementation

By exploiting the C++ language, LSI++ strives to improve the *maintainability, portability,* and *efficiency* [LB97] of the original Bellcore LSI implementation [DDF+90]. LSI++ provides an Application Programming Interface (API) that allows the construction of multiple user interfaces which are independent of the underlying implementation. Currently there are two user interfaces available for LSI++: a client-server interface accessible through the World-Wide-Web, and a command-line interface [LB97, Let96].

3.2.1 Serial Implementation

Whereas the underlying implementation of LSI within the LSI++ software environment includes a serial and distributive version, the DRM method has been integrated with the serial version only. Efficient use of time and memory were concerns in designing a downdating step during the query phase due to the number of possible users on a single machine. If memory is abundant then a serial implementation is available which

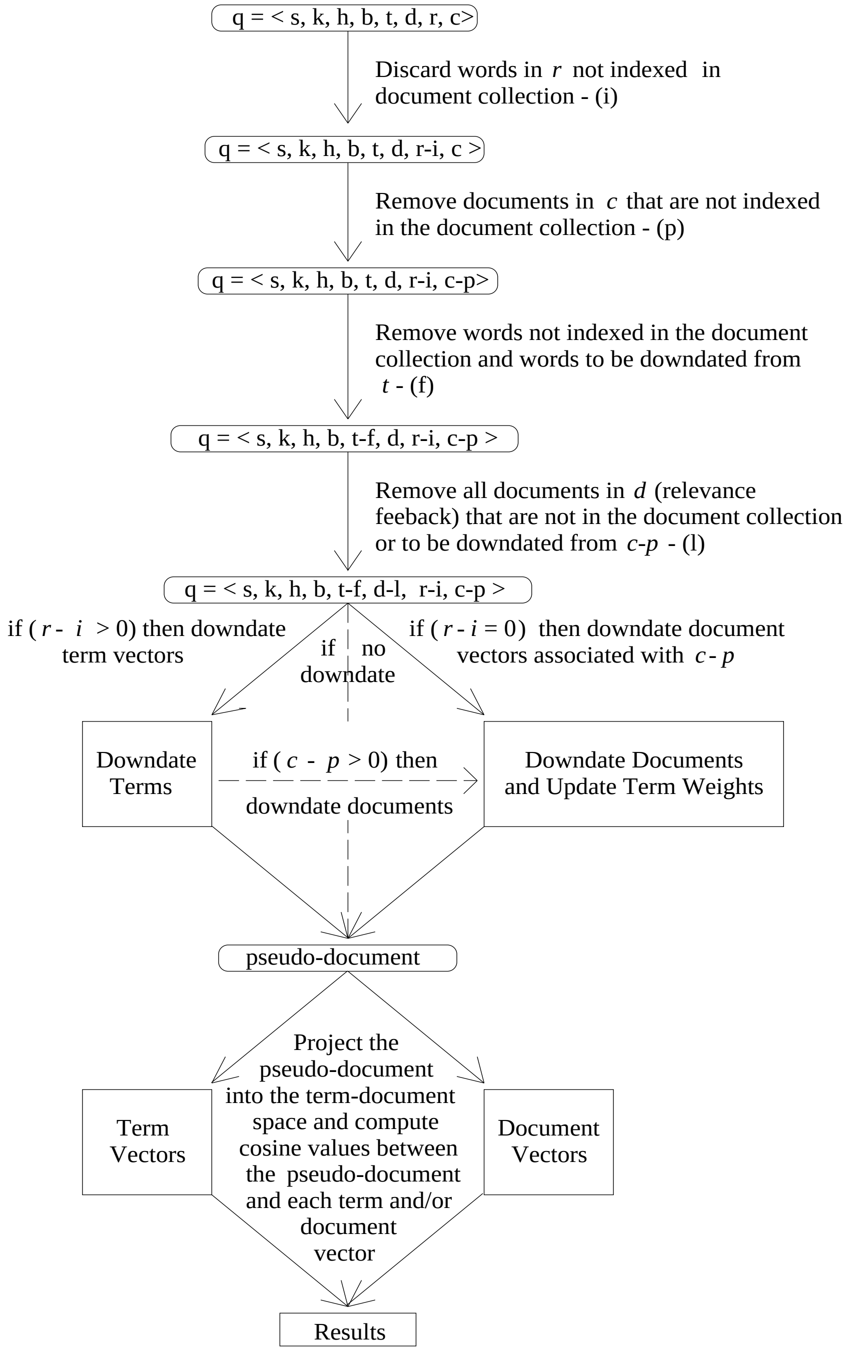


Figure 3.1: The query phase with downdating [LB97].

loads all the term and document vectors (i.e. singular vectors) and singular values into memory before downdating and query matching. Otherwise, a serial implementation that minimizes the use of memory is employed.

Out-Of-Memory Serial version

When available memory is limited, the LSI++ serial implementation must be able to perform downdating and query matching using reduced memory while maintaining time efficiency. Assuming memory cannot be used to store large sections of the document collection, operations on term or document vectors must be done incrementally and discarded upon completion [LB97].

Memory usage is also minimized by computing all Givens rotations prior to any post- or pre-multiplication by a term or document vector. To reflect a downdate, each term and/or document vector must be multiplied by Givens rotations generated by the bidiagonalization and diagonalization procedures described in Section 2.4.1. Each term or document vector is loaded into memory and multiplied by the appropriate Givens rotations, the modified term or document vector is saved to disk, and the original term or document vector is removed from memory.

In-Memory Serial version

If sufficient memory is available, all term and document vectors (and singular values) are loaded into memory thereby producing much faster processing times for both downdating and query matching. As with the out-of-memory version, Givens rotations are calculated to produce both bidiagonal and diagonal forms before any updating of the term and document vectors. In this case however, all the vectors remain in memory.

3.2.2 C++ Classes

LSI++ was written in the C++ programming language to exploit both encapsulation and inheritance [LB97]. Whereas most LSI++ classes or objects are independent, some inheritance relationships exist as illustrated in Figure 3.2. The *lsiSerial* and the *lsiServer*

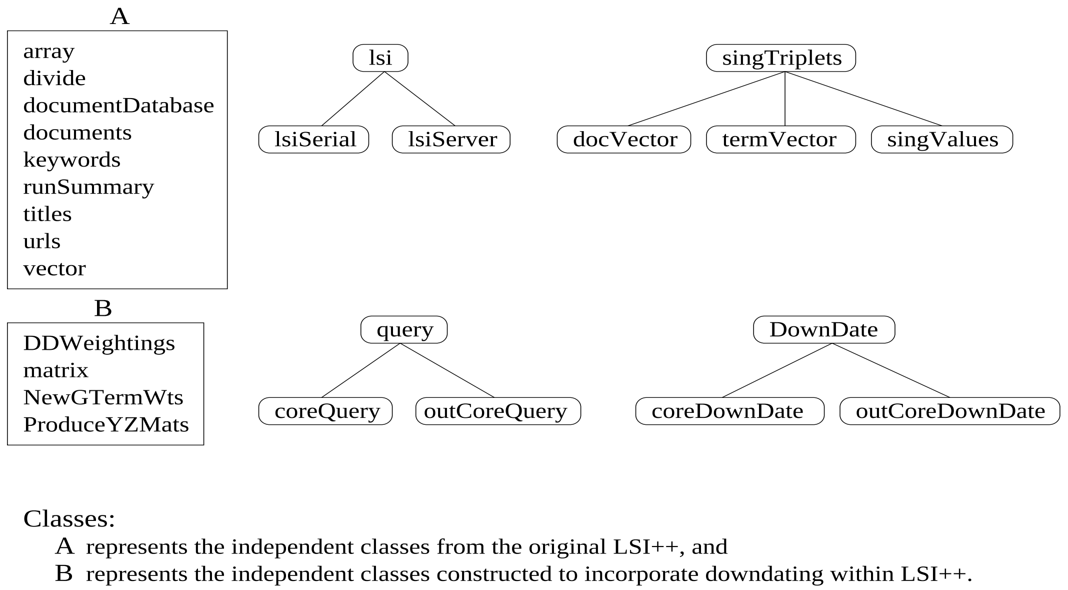


Figure 3.2: The LSI++ inheritance hierarchy with the independent classes on the left and inheritance relationships on the right [LB97].

classes represent the out-of-memory and in-memory versions, respectively, of the serial implementation of the LSI model. The base class *lsi* contains the common characteristics of the *lsiSerial* and *lsiServer* classes, and together the three classes comprise the API for LSI++'s serial search engine. The *query* class contains the common characteristics of the *coreQuery* and *outCoreQuery* classes which create the query vector, project the pseudo-document into the term-document space, and compute the cosine between the query vector and each of the term and document vectors. The *coreQuery* class is an object within the *lsiServer* class, while the *outCoreQuery* class is an object within the *lsiSerial* class. The *coreDownDate* and the *outCoreDownDate* classes are also objects within the *lsiServer* and *lsiSerial* classes, respectively, which perform downdating within each serial version. The common functions of the *coreDownDate* and *outCoreDownDate* classes are contained in the *DownDate* class. The *termVectors*, *docVectors*, and *singValues* classes provide the term vectors, document vectors, and the singular values used in downdating and query matching, and the base class *singTriplets* provides effective retrieval of the vectors [Let96]. Along with LSI++'s independent classes in Figure 3.2, a library, libg++ [Fre92], of low-level classes from the GNU version of C++, g++, are incorporated to provide string manipulation, data storage (such as associative arrays), and other useful data structures [Let96] within the serial implementation of the LSI model.

Figures 3.3 and 3.4 illustrate the associations between the LSI++ classes for the out-of-memory and in-memory serial implementations. Although most of the classes are described in [Let96], new classes associated with downdating and any classes critical to the implementation of downdating are discussed in detail below.

The lsi class

As mentioned above, the *lsi* class and its derived classes, *lsiSerial* and *lsiServer*, serve as Application Programming Interface(API) for the LSI++ search engine [Let96]. Using

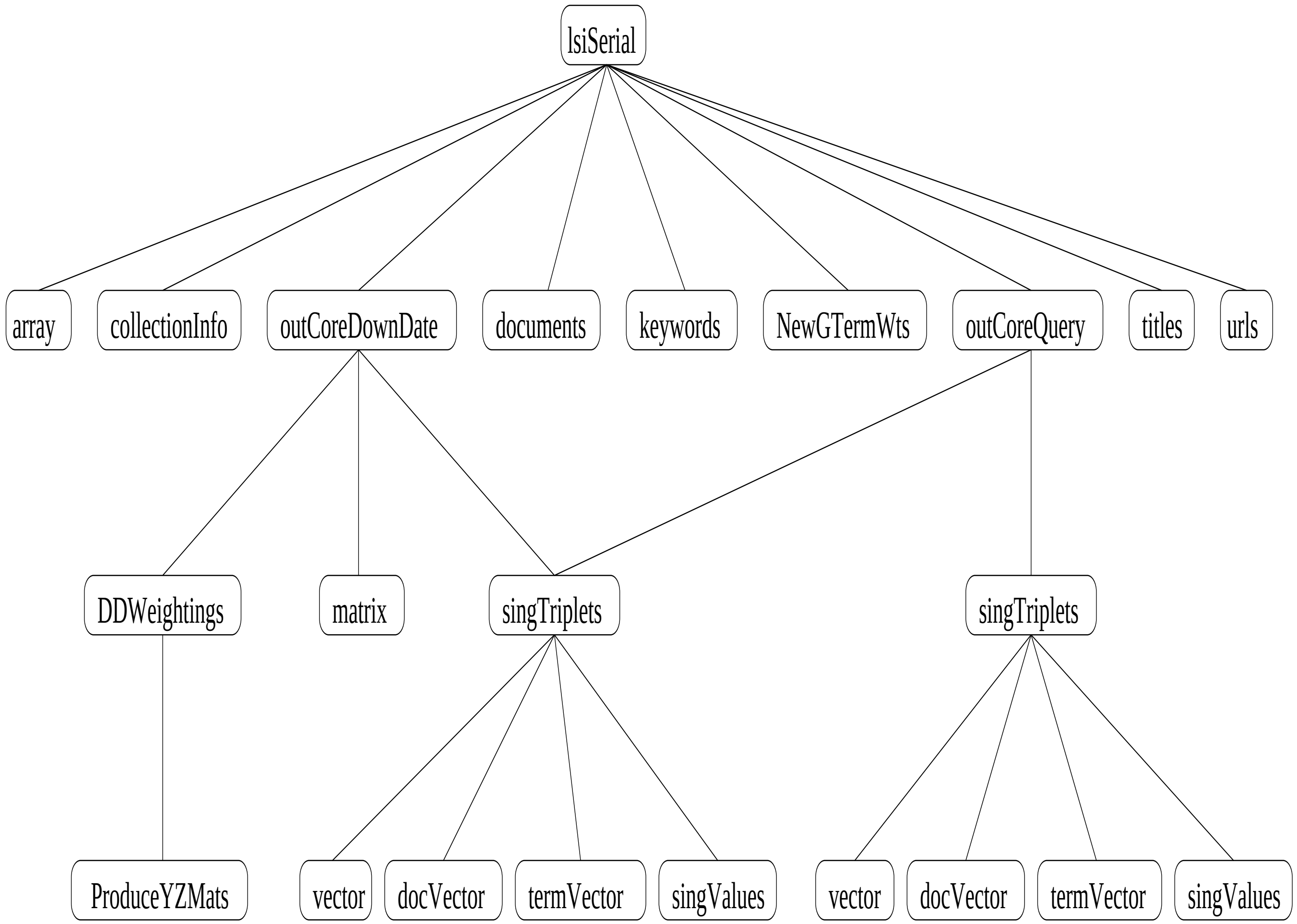


Figure 3.3: The LSI++ out-of-memory serial implementation.

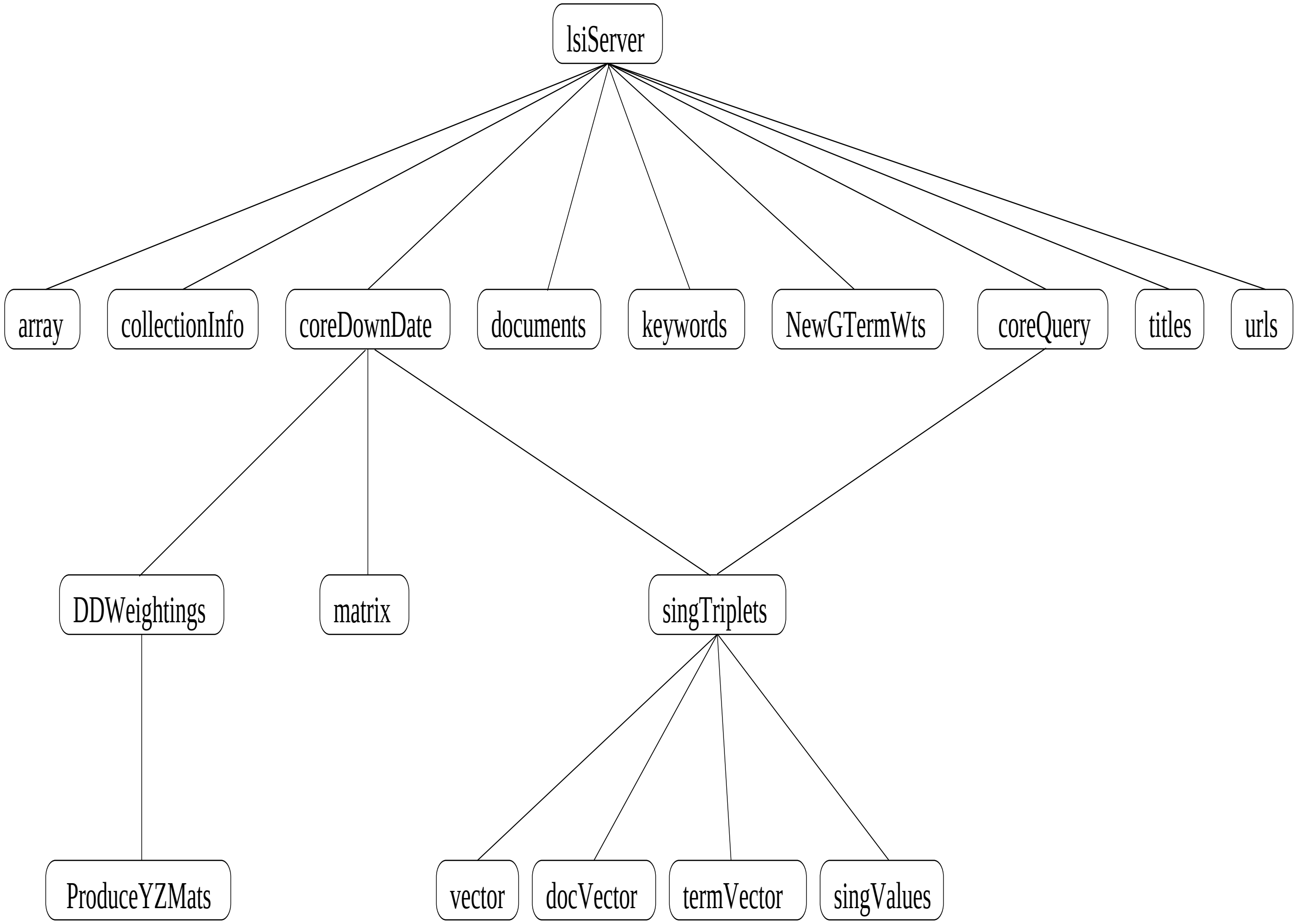


Figure 3.4: The LSI++ in-memory serial implementation.

this API, the user can pose a query with chosen terms or documents to be downdated, and receive the results of the query (reflecting the downdate) without any knowledge of the underlying LSI implementation. Member functions were added to the *lsi* class and its derived classes to invoke downdating within a query and to display the downdated terms and documents within the results of the query. Table 3.1 lists the added API functions for downdating.

The lsiSerial and lsiServer classes

Derived from the *lsi* class, the *lsiSerial* and *lsiServer* classes support the API by invoking query matching and downdating through the following four procedures:

1. **performDownDate** - with term weight updates.

   This procedure is called when documents are to be downdated, regardless whether or not a term is to be downdated, since the term weights must be updated. Since term weight updating occurs only once, for the purpose of efficiency all singular vectors (and corresponding singular values) of the matrix with the updated term weights (see Section 2.4.3) are retained in memory. Therefore, upon completion of this procedure, the term and document vectors (stored on disk or in memory) have not been modified to reflect the downdate.

2. **performQuery** - with update to vectors.

   If documents have been downdated, this procedure is invoked to formulate the query vector and perform query matching. The term and document vectors as

Table 3.1: The API member functions added to incorporate downdating.

Member Function	**Description**
int addDDDoc(const int docNum)	adds document with index docNum to the downdate document list
int addDDTerm(const int termNum)	adds term with index termNum to the downdate term list
int addDDTerm(const String& termString)	add terms given in termString to the downdate term list
int isDDDoc(const int docNum)	returns true (1) if the document with index docNum was inputed to be downdated
int isDDTerm(const int termNum)	returns true (1) if the term with index termNum was inputed to be downdated
int getNumDDDocs()	returns the number of documents downdated before query
int getNumDDTerms()	returns the number of terms downdated before query
int getDDDoc(int i)	returns the index of the i^{th} downdated document
int getDDTerm(int i)	returns the index of the i^{th} downdated term
void performDownDate()	remove terms or documents from the term-document space

well as the singular values, in this case, do not reflect the term weighting update. During query formulation and matching, the query vector, the term and document vectors, and the singular values are then scaled appropriately to reflect the downdating.

3. **performDownDate** - without term weight updates.

   This procedure is called only when terms are to be downdated. Hence, term weight updates are not required. Upon return, the term and document vectors have been appropriately modified to reflect the downdate.

4. **performQuery** - without update to vectors.

   If downdating was not invoked or downdating was invoked with only terms, this procedure is called to build the query and invoke query matching. Since all the term and document vectors have either been modified to reflect downdating or have not been touched, the query formulation and matching can proceed as normal without performing any additional scaling.

The coreQuery and outCoreQuery classes

The *coreQuery* and *outCoreQuery* classes are invoked by the *lsiServer* and *lsiSerial* classes, respectively, within one of the **performQuery** functions described above. These classes contain functions to compose the query vector and to perform the query matching. The *coreQuery* class accesses the appropriate term and document vectors and singular values from memory, applying the term weight update corrections if appropriate (see above class description). The *outCoreQuery* class loads the term and document vectors and singular values from disk, applying the term weight update corrections if appropriate (see above class description).

The Downdate class

The *Downdate* class is the base class for the classes *coreDownDate* and *outCoreDownDate* and contains functions used by both classes to aid in the downdating of terms and/or documents. The *Downdate* class contains functions to produce a Givens rotation, to perform the SVD of the dense matrix produced during the updating of term weights, and to perform the SVD of a bidiagonal matrix (see Section 2.4). Algorithms to compute the SVD of dense matrices are provided by *SGESVD* and *SBDSQR* routines from LAPACK [ABB+95].

The coreDownDate and outCoreDownDate classes

When performing downdating, the function **performDownDate** in the *lsiServer* class activates functions in the *coreDownDate* class which perform all the steps for downdating described in Section 2.4 while manipulating the already in-memory term and document vectors and the singular values. Those same vectors and values are referenced by the *coreQuery* class during the **performQuery** function. Upon completion of a query with downdating, the *coreDownDate* class reloads the original term and document vectors and singular values into memory in preparation for the next query. When performing downdating, the function **performDownDate** in the *lsiSerial* class invokes functions in the *outCoreDownDate* class which perform all the steps for downdating while reading and writing the term and document vectors and singular values to another file on disk to preserve the original singular vectors and values (stored on disk) before the downdate.

The keywords class

Prior to the integration of downdating, the task of the *keywords* class, described in [Let96], was to obtain the keywords in the document collection, the index of each keyword in the term vectors, and the appropriate local and global weightings applied to each keyword. One function was added to the *keywords* class to aid in downdating, **getLWeight**. This function extracts the value of the local weight applied to a specific keyword frequency.

The vector class

The *vector* class implements a data structure that enables other objects to manipulate and alter a vector as a single entity [Let96]. In addition to the existing vector functions, a number of operations were introduced to facilitate downdating: **cat** which concatenates two vectors together, **before** which returns the portion of the vector before the input index, **after** which returns the portion of the vector after the input index, **getvecDD** which returns an integer value indicating whether or not the vector has been downdated, and **oplength** which indicates the operation length of the vector.

The singTriplets class

Serving as a base class for the *singValues*, *termVector*, and *docVector* classes, the *singTriplets* class fetches the appropriate term or document vector or the vector of singular values from memory or disk for the query or downdating object. Loading a vector from disk is more expensive than accessing memory; therefore, the *singTriplets* class provides the caching of vectors in memory for its derived classes [Let96]. The original functions provided by the *singTriplets* class were sufficient for in-memory querying and downdating. However, for out-of-memory querying and downdating, other functions were added in its derived classes to compensate for I/O to alternative secondary storage.

The singValues class

The *singValues* class is responsible for providing the singular values for query formulation and downdating objects. Due to the limited number of singular values needed (5 to 100), they are cached into memory by the base class, *singTriplets* class, except when downdating with the out-of-memory serial version. In this case, the *singValues* class was adapted to read from and write to a new file instead of corrupting the original singular values before the downdate.

The termVector class

The chief responsibility of the *termVector* class is to provide the appropriate term vector, or row of U_k, for querying matching or downdating. If downdating is invoked with the out-of-memory serial implementation, a new term vectors file is created and used during query formulation and matching. When referencing a term vector, a check to see if the vector is cached in memory (by the caching function of the *singTriplets* class) is made. If the vector is not cached, the *termVector* class loads and returns a term vector from the original vectors file or the file created during downdating. Additional code was supplied to support the reading and writing of a vector to a new term vectors file associated with out-of-memory downdating and query processing. Using the new term vectors file allows downdating changes to be made without overwriting the original file.

The docVector class

Derived from the *singTriplets* class, the *docVector* class is similar in function to the *termVector* class. The main task of the *docVector* class is to provide the appropriate document vector, or row of V_k, referenced during query matching or downdating. If the desired document vector is cached in memory (by the *singTriplets* class), that vector is returned from the cache. Otherwise, the *docVector* class returns a document vector from either the original document vectors file or the file created during downdating. As with the *termVector* class, code has been added to facilitate the use of a new document vectors file during downdating, thereby preserving the original document vectors on disk.

The matrix class

Similar to the *vector* class, the *matrix* class hides the implementation of matrix operations. For example, if the query or downdating object wishes to compute $r^T = v^T M$ the following statement

$$M.MultVecMat(v, r)$$

computes the dot product between the vector v and each column of the matrix M and stores the resulting row vector in r. The *matrix* class provides functions to perform matrix multiplication, pre- and post-multiplication of a matrix by a vector, the retrieval and storage a submatrix within a matrix, and matrix transposition.

The ProduceYZMats class

The *ProduceYZMats* class produces the Y_j and the Z_j matrices described in Sections 2.3.3 and 2.4.3. By reading a raw matrix file containing each document along with the index and frequencies of its corresponding terms, the *ProduceYZMats* class extracts the terms affected by downdating a particular document, thus producing Y_j. To facilitate the production of Z_j, a *DDWeightings* class object is created for each affected term and is used to store all the associated document numbers that contain the corresponding term. These objects are then used to produce the old and new term weights for each affected term. When producing Z_j, the *ProduceYZMats* class computes the new global weight for each term affected by a document downdate, and the new global weights are stored in a *NewGTermWts* class object for use during querying.

The DDWeightings class

The *DDWeightings* class is used strictly by the *ProduceYZMats* class. Each term affected by a downdated document is associated with a *DDWeightings* class object. Each *DDWeightings* class object has a list of the documents not downdated containing the affected term along with the term's frequency. The global frequency (total number of times the affected term appears in the document collection after downdating all terms and documents) and the document frequency (total number of documents not downdated that contain the affected term) are calculated for use in producing the new global weights. With the local and global weighting functions for a particular document collection, the *DDWeightings* class produces the old and new term weights for the affected term within each document to produce matrix Z_j. A *NewGTermWts* class object is created to store the global weight values produced by the *DDWeightings* class.

The NewGTermWts class

The primary responsibility of the *NewGTermWts* class is to maintain the new global weights of the terms affected by a document downdate. Information recorded in the *NewGTermWts* class is necessary for query matching when a document is downdated. When producing the *pseudo-document*, each query term appearing in the query vector that is affected during downdating is scaled by the appropriate local and global weight values (see Section 2.2) defined in the *NewGTermWts* class object.

3.3 The LSI++ User Interfaces involving Downdating

Serial user interfaces have been constructed using the LSI++ API including a client-server interface accessible through the World-Wide-Web (WWW) and a command-line interface [LB97, Let96]. The World-Wide-Web interface allows a novice user to interact with the LSI model without having any knowledge of the interface or any detailed reference material [LB97]. A more concise command-line interface is available for the advanced user. Downdating has been implemented as an added feature available through both of these interfaces.

3.3.1 World-Wide Web Interface

The WWW interface provides the user with useful, flexible, and simple options. Initially, the WWW interface allows the user to

- select a document collection for searching,
- input words for a query,
- select the number of factors to use during the search,
- choose whether to have terms, documents, or both returned, and
- choose the number of highest-ranked terms, documents, or both to return.

An example of the initial WWW interface is illustrated in Figure 3.5. This initial query form has selector boxes for choosing the document collection, the number of factors, the

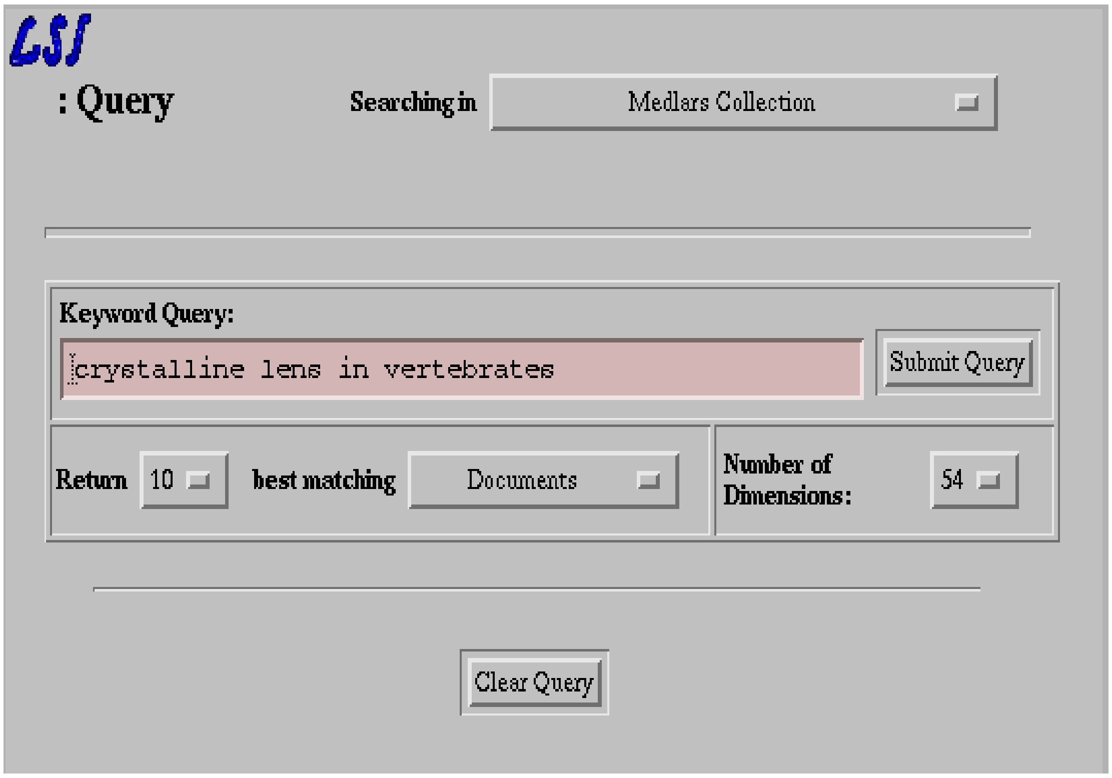
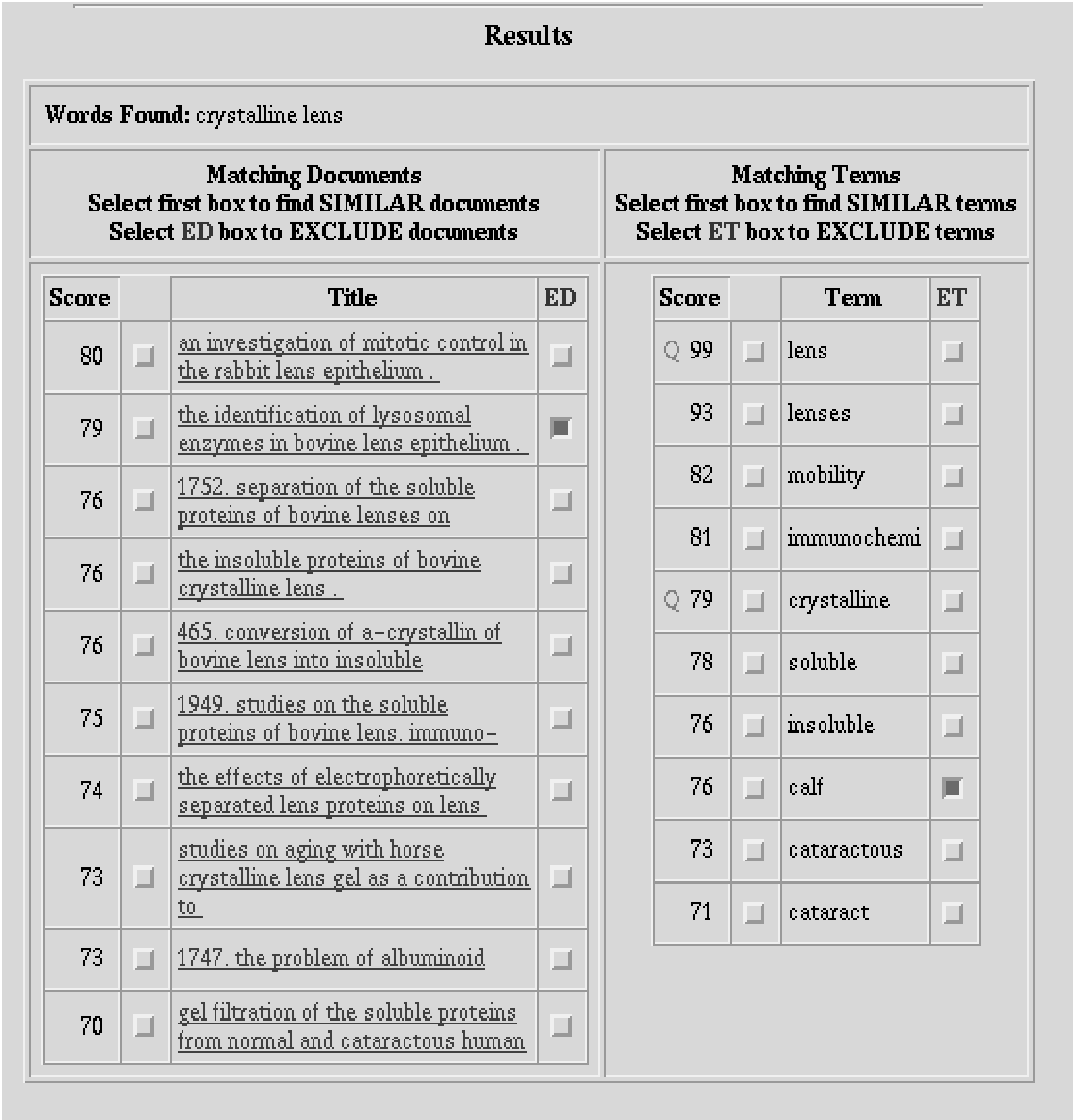


Figure 3.5: The initial query form.

number of terms, documents, or both to be returned from the ranked list, and exactly what to return: terms, documents, or both. A text box is available for the user to input the query along with a **clear** query button to clear the query term(s) and a **submit** button to submit the query and the form to the server. The server calls the appropriate script to process the form, perform the query, and return the results.

Once the initial query has been entered and results returned to the client, a second form is displayed to the user containing the results of the initial query with the addition of relevance feedback and downdating options. The results are attached to the bottom of the initial form in Figure 3.5. Figure 3.6 is an example of the bottom portion of the second form, where each document and term returned has a relevance feedback and a downdate checkbox. These checkboxes allow the user to add a document or term to the query (relevance feedback) or remove a term or document from the term-document space (downdating) in hopes of achieving better recall and precision for the query. If relevance feedback or downdating is checked in the second form, the same form is returned with the specific downdated terms and documents attached to the bottom of the form as illustrated in Figure 3.7. The attached downdated terms and documents have a checkbox which is activated; therefore, those terms and/or documents will continue to be downdated in subsequent queries unless the user deactivates those checkboxes.

The implementation of the client-server interface in C++ using the API provided by LSI++ is described in [Let96]. To incorporate downdating, the serial code which



Results

Words Found: crystalline lens

Matching Documents
Select first box to find SIMILAR documents
Select ED box to EXCLUDE documents

Score		Title	ED
80	☐	an investigation of mitotic control in the rabbit lens epithelium .	☐
79	☐	the identification of lysosomal enzymes in bovine lens epithelium .	■
76	☐	1752. separation of the soluble proteins of bovine lenses on	☐
76	☐	the insoluble proteins of bovine crystalline lens .	☐
76	☐	465. conversion of a-crystallin of bovine lens into insoluble	☐
75	☐	1949. studies on the soluble proteins of bovine lens. immuno-	☐
74	☐	the effects of electrophoretically separated lens proteins on lens	☐
73	☐	studies on aging with horse crystalline lens gel as a contribution to	☐
73	☐	1747. the problem of albuminoid	☐
70	☐	gel filtration of the soluble proteins from normal and cataractous human	☐

Matching Terms
Select first box to find SIMILAR terms
Select ET box to EXCLUDE terms

Score		Term	ET
Q 99	☐	lens	☐
93	☐	lenses	☐
82	☐	mobility	☐
81	☐	immunochemi	☐
Q 79	☐	crystalline	☐
78	☐	soluble	☐
76	☐	insoluble	☐
76	☐	calf	■
73	☐	cataractous	☐
71	☐	cataract	☐

Figure 3.6: The bottom of the second form showing the results of the initial query.

Figure 3.7: The downdated term and document attached to the bottom of the query results.

executes on the server, *lsiQuery*, uses the classes *cgi*, *html*, *std*, and *lsiSerial* from LSI++ to perform the downdate and query search if a backend processor does not exist. If the serial code *lsiServ* is continuously running on a backend processor referencing the particular document collection being searched, then *lsiQuery* sends the user's input taken from the forms to *lsiServ*. *lsiServ* performs downdating and query matching by using *lsiServer* and passes the results back to *lsiQuery* using the *text* and *remote* classes.

3.3.2 Command-Line Interface

The command-line interface allows the user to access the LSI++ search engine both locally and remotely, and provides the same options as the WWW interface. For the local command-line interface, *lsiQuery* is called and a particular directory is specified. For the remote command-line interface, the user *telnets* to the continuously running backend processor that accesses one particular document collection. Table 3.2 lists the parameters and options for the command-line interface.

Table 3.2: The required and optional features of the command-line interface for the LSI++ search engine.

Option	**Explanation**	**Required or Optional**
-n	number of LSI factors	optional - if not specified the maximum number of factors is used
-k #	number of terms, documents, or both to return	optional - if not specified then a default value of 10 is used
-b or -t	return both documents and terms (-b) or return just terms (-t)	optional - if not specified only documents are returned
term1, term2, ...	one or more terms specifying the query	required
+document1, +document2, ...	one or more document numbers specified for relevance feedback	optional
^term1, ^term2, ...	one or more terms specifying which terms to downdated	optional
^^document1, ^^document2, ...	one or more document numbers specifying documents to be downdated	optional

Chapter 4

Evaluating the Performance and Effectiveness of Downdating

An information retrieval system or algorithm is useful if it is both effective and time efficient [FBY92]. A desirable system is one in which relevant information is quickly returned to the user. The LSI model for information retrieval, as implemented in the LSI++ software, provides accurate retrieval in a time efficient manner and has been shown to be a useful system [LB97]. The focus of this chapter is to evaluate the usefulness of downdating as it has been incorporated into the LSI++ software. The complexity and observed execution time for downdating operations are discussed and compared with those required for recomputing the SVD. This chapter also examines the retrieval effectiveness of the downdating operation, evaluating the relevancy of documents returned from a query involving downdating.

Table 4.1: Document collections used to examine the performance of downdating.

Document Collection	Number of Terms	Number of Documents	Max Number of Factors
Interface 1997: List of Talks (INTERF)	230	28	28
MPI: The Complete Reference (MPI)	1952	212	54
Parallel Computing Works! (PCW)	7003	485	100

4.1 Performance Evaluation

The downdating implementation incorporated into LSI++ was compared with recomputing the SVD after the items to be downdated were removed from the original term-by-document matrix.

4.1.1 Procedure

All performance timings were done on a Sun Ultra1 SPARCstation with a 167 MHz processor, 32 KB on-chip cache (16 KB Instruction, 16 KB Data), 512 KB external cache, and 256 MB of main memory. As illustrated in Table 4.1, three document collections were used in testing. The INTERF document collection consists of a set of presentation titles from the *Interface* 1997 conference program announcements [BW97]. The MPI document collection is a reference manual for the Message Passing Interface [SOHL+96]. The PCW document collection is a textbook on parallel computing [FWM94]. Three implementations of term and/or document removal within the LSI model were tested to determine their execution efficiency for each document collection.

- **lsiQuery** - *lsiQuery* is the driver for the out-of-memory serial version for the LSI model with downdating using LSI++ described in Section 3.2.1 which invokes the *lsiSerial* class described in Section 3.2.2. *lsiQuery* accepts a query requesting that a term or document be removed, performs the downdating and subsequent query matching using LSI++, and writes the results either to standard output in simple text or as an HTML document [Let96].

- **lsiServ** - *lsiServ* is a continuously-running backend server for the in-memory serial implementation of LSI which incorporates downdating (see Section 3.2.1). *lsiServ* receives a query with a request for downdating from a remote source, performs the downdating and search using LSI++, and then passes the results back to the remote source.

- **pindex** - *pindex* is the original Bourne shell script written at Bellcore that can be used to recompute the SVD of a term-by-document matrix with some term(s) and/or document(s) removed. The output of this script includes the file of singular vectors and singular values, a searchable keywords file, and a summary file. Query matching can then be implemented by either the *outCoreQuery* class or the *coreQuery* class.

The elapsed wall-clock time observed for the out-of-memory LSI++ serial implementation of downdating term(s)/document(s) is compared with the amount of time

Table 4.2: The number of LSI factors used for each document collection.

	5	15	25	35	45	55	65	75	85	95
INTERF	X	X	X							
MPI	X	X	X	X	X					
PCW	X	X	X	X	X	X	X	X	X	X

observed for the in-memory LSI++ serial implementation, and with recomputing the SVD. It is important to note that the in-memory LSI++ implementation with downdating has the same time complexity in terms of floating-point operations as that of the out-of-memory LSI++ implementation (see Section 4.1.2). The difference in elapsed time observed during the tests between the two implementations can be attributed to the overhead associated (e.g., disk latency) with out-of-memory processing.

For each document collection, the elapsed wall-clock time for both downdating and recomputing the SVD was calculated with the removal of one term, one document, one term and document pair, two terms, two documents, two term and document pairs, and three terms and three documents with different numbers of LSI factors. For each set of tests with a particular document collection, the initial number of factors was chosen to be 5 and then incremented by 10 up to the maximum number of factors computed for that document collection. The number of factors used for each document collection is shown in Table 4.2.

4.1.2 Results

Evaluating observed execution times for a IR system requires an understanding of the time complexity for each process. Given the time complexities, the actual timings can be checked for validity.

Complexity

For comparing recomputing the SVD and the DRM method of downdating, time complexities have been expressed in terms of the number of floating-point operations (FLOPS) used. In the case where a term or a document is removed, Equation (4.1) illustrates the number of FLOPS required to recompute the SVD of a sparse matrix using a Lanczos procedure [BDO+93]. The time complexity of this Lanczos procedure is

$$I \times (4z - m - n) + 2kz - m + I^2 n, \tag{4.1}$$

where I is the number of iterations required to achieve the desired precision of the singular values and vectors, m is the number of terms, n is the number of documents, k is the desired number of dimensions or factors, and z is the number of nonzero elements in the term-by-document matrix. In the case of removing a term or document using the DRM method, the time complexity is shown in Figure 4.1. The time complexity for downdating is divided into three phases: downdating a term, downdating a document, and updating the term weights. Each phase has a series of steps (see Section 2.4) and the time complexity for each step is provided in Figure 4.1. The complexities for

Time complexity (FLOPS) when downdating TERMS

$$\underbrace{3k + 2km + 2m + 1}_{\text{Find an orthonormal vector}} + \underbrace{6k^3 - k^2 + 12}_{\text{Reduce to bidiagonal matrix}} + \underbrace{12k^2 + 32k}_{\text{Reduce to Diagonal matrix}} + \underbrace{4k^3 - k}_{\text{Matrix Multiplication}} + \underbrace{2k^2m + km - 2k^2 - k + 2nk^2 - nk}_{\text{Update term and document vectors}}$$

Time complexity (FLOPS) when downdating DOCUMENTS

$$\underbrace{3k + 2kn + 2n + 1}_{\text{Find an orthonormal vector}} + \underbrace{6k^3 - k^2 + 12}_{\text{Reduce to bidiagonal matrix}} + \underbrace{12k^2 + 32k}_{\text{Reduce to Diagonal matrix}} + \underbrace{4k^3 - k}_{\text{Matrix Multiplication}} + \underbrace{2k^2n + kn - 2k^2 - k + 2mk^2 - mk}_{\text{Update term and document vectors}}$$

Time complexity (FLOPS) when updating TERM WEIGHTS

$$\underbrace{k}_{\text{Add in Singular values}} + \underbrace{2kjm + 2kjn - 2jk + 2jk^2 - k^2}_{\text{Matrix Multiplications}} + \underbrace{12k^3}_{\text{Perform SVD on dense matrix}} + \underbrace{2k^2m + 2k^2n - mk - nk}_{\text{Update term and document vectors}}$$

Variables:

m = Number of terms,
n = Number of documents,
k = Number of LSI factors, and
j = Number of terms affected by downdate.

Figure 4.1: The time complexity for the DRM method given in terms of FLOPS.

the reduction of a bidiagonal matrix to diagonal form and the SVD of a dense matrix are extracted from [GL96]. The other complexities are actual FLOP counts from the LSI++ software. The time to perform the downdating of a term and document certainly depends on the number of LSI factors (k) involved.

Apart from the number of FLOPS required, another factor in the time needed for downdating is the bookkeeping costs. These costs include updating the term and document vectors and singular values and producing the Y and Z matrices for updating term weights (see Section 2.4.3). When producing the Z matrix, a file containing a *raw* form of the term-by-document matrix is read to determine which terms are associated with the downdated documents and which documents correspond to the affected terms.

Timings

Tests were conducted to determine the elapsed time difference between in-memory and out-of-memory downdating within the LSI++ software. A comparison of the elapsed time for out-of-memory downdating and recomputing the SVD was also examined.

The elapsed wall-clock times for the two LSI++ serial implementations of downdating one document (with a varying number of LSI factors) for the INTERF and MPI document collections are illustrated in Table 4.3. Times for downdating a document for the PCW document collection are shown in Table 4.4. These tables show a small speedup for the in-memory downdating compared to the out-of-memory downdating, with both performing faster than recomputing the SVD. This same speedup can be seen in the other tests described in Section 4.1.1 involving different numbers of terms and

Table 4.3: The elapsed wall-clock times in seconds for downdating a document from the INTERF and MPI document collections using the in-memory (IN) implementation, the out-of-memory (OUT) implementation, and recomputing the SVD (RC).

	INTERF			MPI		
Factors	IN	OUT	RC	IN	OUT	RC
5	0.04	0.13	4.80	1.88	2.40	46.20
15	0.14	0.21	4.70	2.87	3.40	52.80
25	0.69	0.90	4.60	4.24	5.00	48.60
35				6.49	7.60	49.30
45				9.61	12.00	55.1

Table 4.4: The elapsed wall-clock times in minutes for downdating a document from the PCW document collection using the in-memory (IN) implementation, the out-of-memory (OUT) implementation, and recomputing the SVD (RC).

	PCW		
Factors	IN	OUT	RC
5	0:04.6	0:05.4	8:23.1
15	0:05.2	0:06.7	8:30.3
25	0:07.6	0:10.1	8:26.1
35	0:11.3	0:14.5	8:35.1
45	0:16.3	0:20.8	8:09.5
55	0:24.0	0:28.8	9:04.7
65	0:32.3	0:38.6	8:17.5
75	0:41.8	0:50.6	8:19.8
85	0:54.4	1:07.0	8:22.3
95	1:11.5	1:25.4	8:41.0

Table 4.5: The elapsed time in seconds for downdating and recomputing the SVD for the MPI document collection.

Remove	15 Factors Downdating	15 Factors Recomputing the SVD	35 Factors Downdating	35 Factors Recomputing the SVD
1 Term	0.50	39.1	2.75	48.2
1 Document	3.4	52.8	7.6	49.3
1 Term/Document Pair	4.6	42.6	11.8	48.2
2 Terms	**2.04**	44.0	**6.5**	49.7
2 Documents	5.7	51.9	14.6	48.7
2 Term/Document Pairs	8.8	48.4	23.3	59.7
3 Terms and 3 Documents	11.5	41.2	31.2	52.0

documents for removal. Of course, the difference in time between the in-memory and out-of-memory downdating would increase for larger collections.

Whereas the in-memory and out-of-memory implementations of downdating have the same time complexities with in-memory downdating performing slightly better, the out-of-memory can be considered an upper bound for downdating. Therefore, the remaining tests only compare out-of-memory downdating with recomputing the SVD. The elapsed times for downdating and recomputing the SVD when different numbers of term(s) and/or document(s) removed were recorded for each of the three documents collections (see Tables 4.5 and 4.6 and Figure 4.2). Downdating never exceeds the time for recomputing the SVD even though the time for recomputing the SVD is constant while time increases with downdating. This factor of reduction makes the DRM method for downdating a viable algorithm. In Tables 4.5 and 4.6 and Figure 4.2, a decrease in time occurs when downdating two terms. Downdating a term is much faster than downdating a document since updating term weights is not necessary in that case. Depending on the

Table 4.6: The elapsed time in minutes for downdating (DD) and recomputing (RC) the SVD for the PCW document collection.

	25 Factors		55 Factors		85 Factors	
Remove	DD	RC	DD	RC	DD	RC
1 Term	0:06.1	8:12.0	0:13.2	8:07.0	0:34.5	8:19.1
1 Document	0:10.1	8:26.4	0:28.8	9:04.7	1:07.0	8:22.3
1 Term/Document Pair	0:18.3	8:00.8	0:50.9	8:47.0	1:58.5	8:36.1
2 Terms	**0:10.2**	7:48.5	**0:33.7**	8:09.0	**1:23.0**	8:10.9
2 Documents	0:36.0	8:11.1	1:27.0	9:25.8	2:53.1	9:11.2
2 Term/Document Pairs	0:51.9	8:07.8	2:27.9	8:21.7	4:30.0	8:01.9
3 Terms and 3 Documents	1:15.2	8:08.7	3:11.6	8:10.7	6:28.7	8:18.9

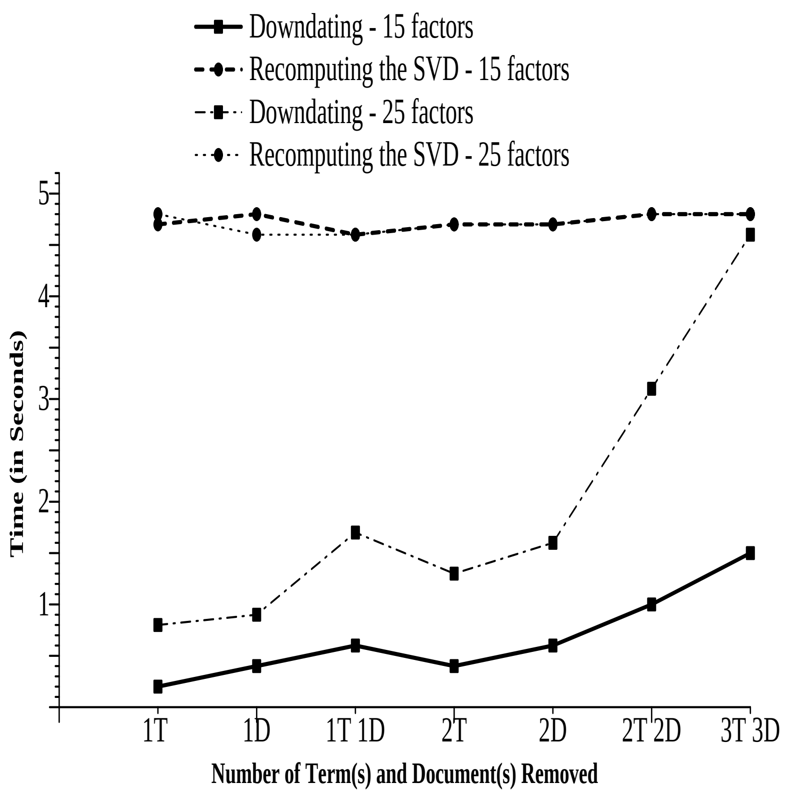


Figure 4.2: The elapsed time for downdating and recomputing the SVD for the INTERF document collection.

document collection, eliminating the updating of term weights can save a substantial amount of time since the original term-by-document matrix need not be reloaded from file and the Y and Z matrices do not need to be generated (see Section 2.4.3). As illustrated in Figure 4.3, downdating performs faster than recomputing the SVD for each document collection when a term and a document are removed.

To solidify the performance analysis of downdating and recomputing the SVD for term and/or document removal(s), the complexities presented in the above section can be compared to the actual elapsed wall-clock times observed for both approaches. An examination of downdating with and without the term weight updating step is illustrated in Figure 4.4. Using 45 LSI factors, the elapsed wall-clock times along with the number of FLOPS required for the different term and/or document removal(s) are plotted. The actual time for recomputing the SVD involves more than the floating-point computational time. There is considerable overhead involved in producing the singular vectors and values file, the searchable keywords file, and the summary file as well as in parsing the document collection to produce the original term-by-document matrix (see pindex from Section 4.1.1). In light of computational complexities, downdating performs better only in the case of removing 1 term and 1 document (when the term weight updating step is not performed) than recomputing the SVD. Removing the term weight updating step reduces the amount of elapsed time and computational complexity significantly. When term weight corrections are ignored, downdating documents is faster than downdating terms since there are typically more terms than documents in

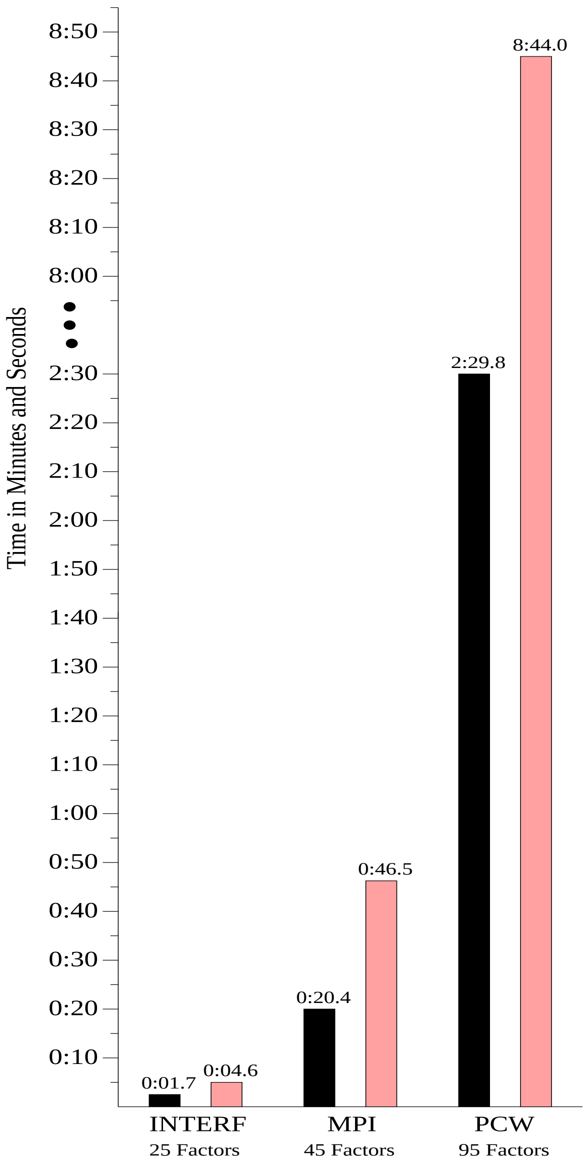


Figure 4.3: The amount of time elapsed for the worst case of downdating compared with recomputing the SVD.

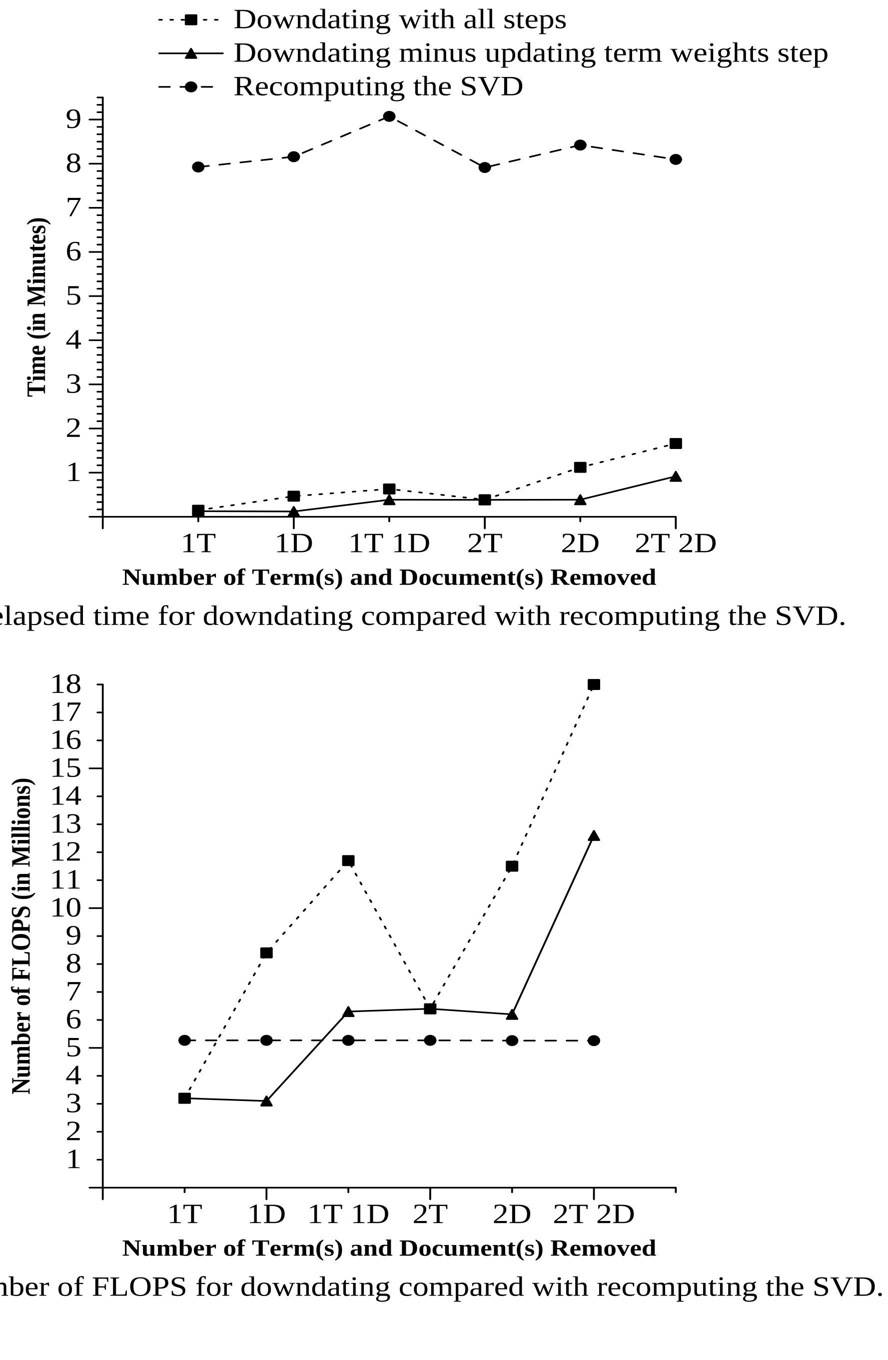


Figure 4.4: The elapsed time compared to the number of FLOPS for the PCW collection using 45 LSI factors.

the collection. Updating term weights accounts for approximately 64% of the elapsed time and computational complexity when removing just one document or one term. For increasing numbers of terms and/or documents for downdating, the percentage of time and complexity for term weight updating (applied only once for all terms and documents) may decrease.

4.2 Effectiveness Evaluation

Retrieval effectiveness was examined using an online version of the the Concise Columbia Encyclopedia (CCE) [cce89]. The CCE document collection is a much larger collection composing of 33,115 terms and 15,486 articles (documents) which better typifies the performance of downdating within a *real* information retrieval system.

4.2.1 Procedure

The procedure used for evaluating retrieval effectiveness was to simply enter a query in the LSI search engine with downdating, examine the results, decide whether to downdate a term or document, and evaluate the results to see if downdating helped retrieve more relevant terms and/or documents. Due to the size of the document collection, term weight updating which consumes the majority of the elapsed time, was not performed.

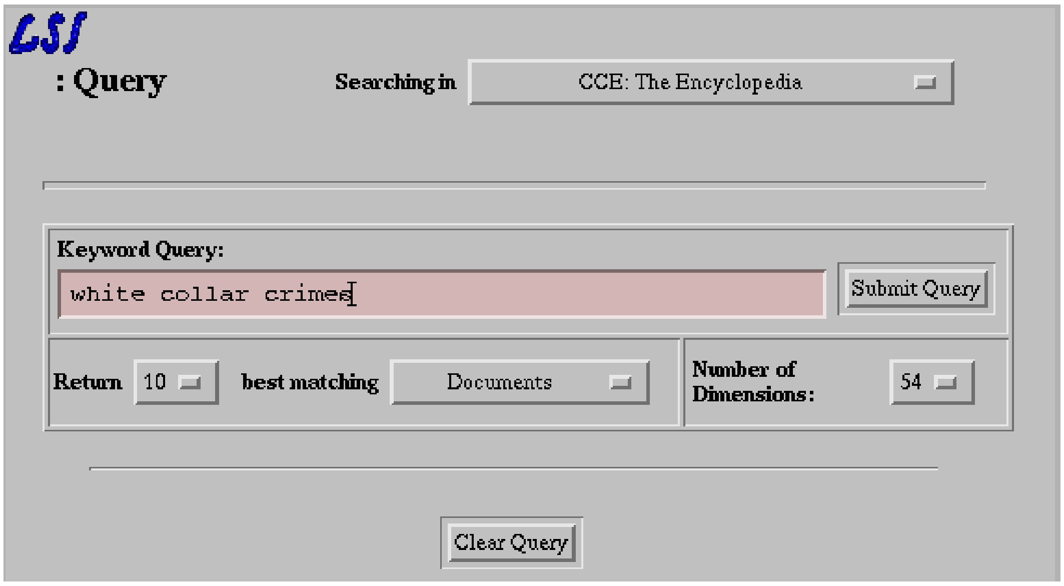


Figure 4.5: The initial query of **white collar crimes**.

4.2.2 Results

Using the client-server interface on the World-Wide-Web, the query **white collar crimes** was entered into the HTML form shown in Figure 4.5. This query demonstrates polysemy as the terms **white collar** could have multiple meanings. One meaning for **white collar** is the white color of a collar on a shirt while another meaning for **white collar** is the description or adjective given to someone who does not partake in manual labor in the workforce. Ideally, the query search should return terms and documents related to **white collar** workers who commit **crimes**. The results of the initial query are illustrated in Figure 4.6 and the terms and documents returned do not pertain to **crime** or criminals. Instead, they pertain to birds and flowers that have colors associated with them. The term **white** seems to dominate the query directing the search to

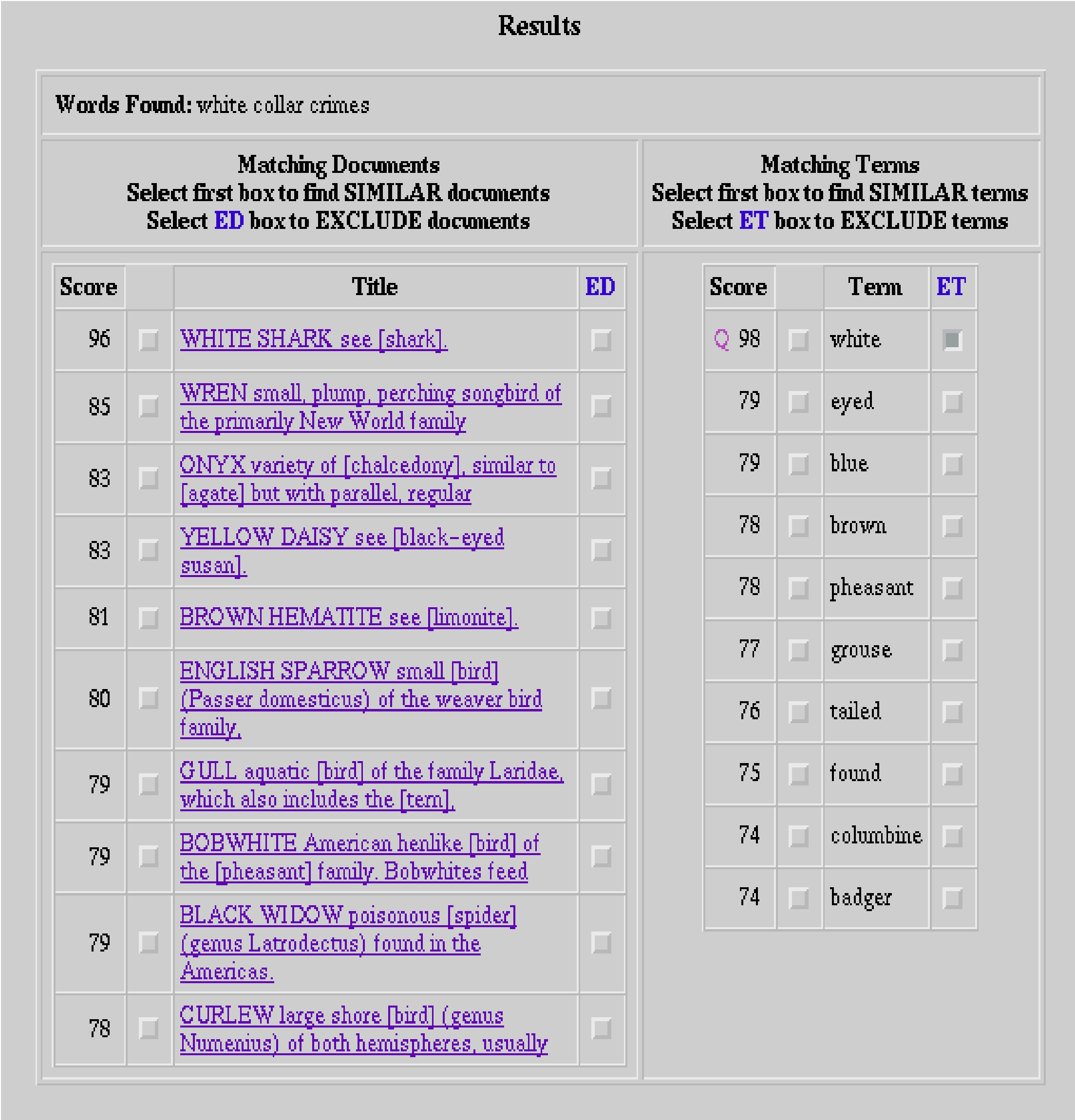


Figure 4.6: The results of the search performed on the initial query along with the decision to downdate the term **white**.

a cluster of articles (documents) in the term-document space pertaining to birds and flowers (as identified by color). Therefore, the term **white** would be a good candidate for downdating. The query search is re-entered with downdating invoked by the **ET** button (see Figure 4.6). The new rank-ordered list of documents is shown in Figure 4.7, and term and documents dealing with **crimes** and criminals are now returned which are more relevant to the query. Of course, just removing **white** from the initial query would produce results similar to Figure 4.7, however, this is not always obvious to the user. Downdating helps the user define his or her current query more precisely in order to obtain relevant terms and documents and improve both recall and precision.

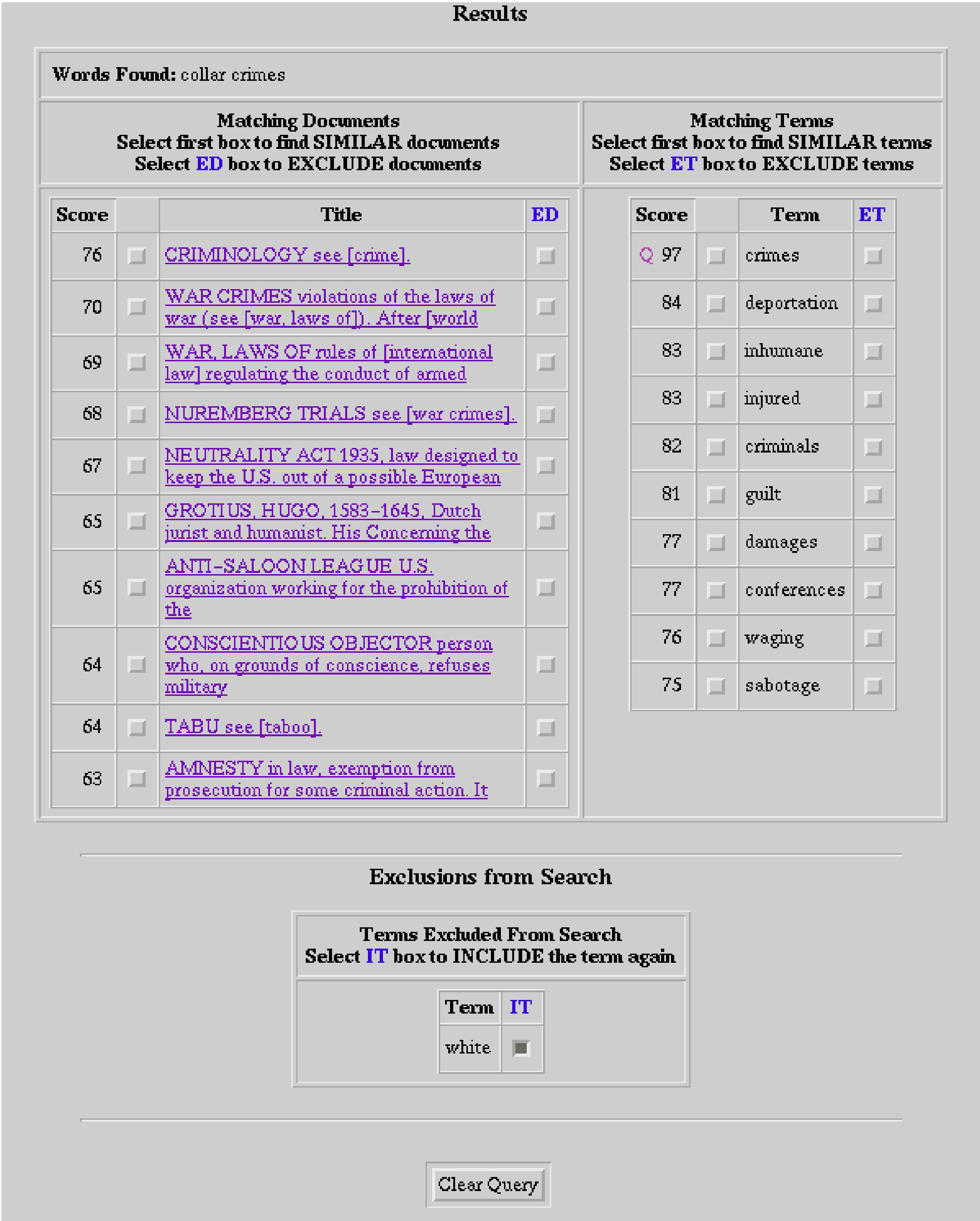


Figure 4.7: The results of the search with downdating the term **white**.

Chapter 5

Conclusions

The **Downdating the Reduced Model** (or DRM) method for downdating is a usable and viable algorithm for removing terms and documents. While recomputing the SVD produces the most accurate method for downdating, it is time consuming and is not available when the original term-by-document matrix is destroyed. Folding-out is a fast method for downdating, but is inaccurate and does not help improve retrieval. The DRM method strikes a balance between computing time and retrieval accuracy providing better retrieval effectiveness than folding-out and faster results than recomputing the SVD.

Given that LSI++ is an efficient and flexible implementation for LSI, the addition of downdating enhances the usability and retrieval effectiveness of the LSI++ search engine. The LSI++ Application Programming Interface allows downdating to be used in a variety of applications and interfaces. The WWW interface provides a simple and easy to use interface for downdating within the LSI++ search engine. Users are able to conceptually eliminate terms and/or documents from their query search, thus helping the query search return relevant documents and terms.

5.1 Future Work

Although downdating is now fully implemented in LSI++, further work remains to make the DRM method an optimal algorithm. The implementation of downdating in LSI++ can be optimized in a number of ways: parallelizing dense matrix multiplication, reducing the number of FLOPS needed during bidiagonalization, and constructing a distributive version of downdating within LSI++.

Research in updating term weights would provide insight into what circumstances require updating term weights and when the step can be eliminated. Also, more research needs to be done in constructing more accurate (but not computationally constrained) approximations of the SVD required in the the term weight updating step (see Section 2.4.3).

Aside from downdating, the LSI++ software environment could be extended in a few functional areas. As described in [Let96], a distributive gathering system for LSI is needed. With the increase in data and the enlargement of document collections, a single machine or a few machines at one site maintaining the LSI-based information retrieval system could no longer hold the document collections. Providing incremental updating of terms and documents to the LSI++ search engine would be quite beneficial.

Bibliography

Bibliography

[AB90] Åke Björck. *Numerical Methods for Least Squares Problems.* Linköping University, Linköping, Sweden, 1990.

[ABB+95] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Cruz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 1995.

[BDO+93] Michael Berry, Theresa Do, Gavin O'Brien, Vijay Krishna, and Sowmini Varadhan. SVDPACKC: (Version 1.0) User's Guide. Technical Report CS-93-194, Computer Science Department, University of Tennessee, Knoxville, Tennessee, April 1993.

[BDO95] M. Berry, S. Dumais, and G. O'Brien. Using Linear Algebra for Intelligent Information Retrieval. *SIAM Review*, 37(4):573–595, 1995.

[BF96] Michael W. Berry and Richard D. Fierro. Low-Rank Orthogonal Decompositions for Information Retrieval Applications. *Numerical Linear Algebra* *with Applications*, 3(4):301 – 327, 1996.

[BW97] Michael W. Berry and Dian I. Witter. Intelligent Information Management using Latent Semantic Indexing. *Computing Science and Statistics*, 9, 1997. Proceedings of Interface '97, Interface of North America Foundation.

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

[DDF+90] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(2):391–407, 1990.

[Dum91] Susan T. 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.

[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.

[GE95] Ming Gu and Stanley C. Eisenstat. Downdating The Singular Value Decomposition. *SIAM J. Matrix Analysis Application*, 16(3):793–810, July 1995.

[GL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[LB97] Todd A. Letsche and Michael W. Berry. Large-Scale Information Retrieval with Latent Semantic Indexing. *Information Sciences - Applications*, 100:105 – 137, 1997.

[Let96] Todd A. Letsche. Toward Large-Scale Information Retrieval Using Latent Semantic Indexing. Master's thesis, University of Tennessee, Knoxville, TN, 1996.

[O'B94] Gavin W. O'Brien. Information Management Tools for Updating an SVD-Encoded Indexing Scheme. Master's thesis, University of Tennessee, Knoxville, Tennessee, 1994.

[PH95] H. Park and S. Van Huffel. Two-way bidiagonalization scheme for downdating the singular value decomposition. *Linear Algebra Applications*, 222:23–40, 1995.

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

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

Vita

Dian Irene Witter was born in Fredericksburg, Virginia on December 16, 1971. She graduated from James Monroe High School in 1990, and received a Bachelor of Science degree in Computer Science and a Bachelor of Arts degree in Elementary Education in 1994 from the College of William and Mary in Williamsburg, Virginia. In August 1994, she entered the Computer Science graduate program at the University of Tennessee in Knoxville, Tennessee and received a Master of Science degree in Computer Science in December 1997.