Under the Hood of a Web Search Engine

Part of the "Under the Hood Internet Technology Series"

By James Powell
Author of "HTML Plus!" (ISBN: 0-534-51626-2) International Thompson Publishing

Your Web page is invisible to most people who search the Web. You are probably already aware of this. The days of being first in the results set list of any given search engine are over. Someone else beat you to it. You know you should care, but you have no idea how to remedy the situation. And based on some of the junk you've seen search engines throw back at you in reply to what you consider reasonable queries, the situation looks hopeless. And in fact, it may be very difficult for you to claw your way back to the top of any given results set at a site like AltaVista or Go.com.

This paper outlines a few important trends in Web searching technology, and exposes the inner workings of a simple but sophisticated search engine written in Java. It also outlines a standard for describing search engine interfaces, around which it is relatively simple to build a metasearch tool that can be used to classify and query collections of search engines based on the types of content they index.

Hypertext is made for browsing, but increasingly the Web is a search tool as much as it is a hypertext document repository. Functional domain names (such as search.com, personalization.com, etc.) sometimes lead to useful resources, but more often than not they've been appropriated by commercial entities looking to sell a specific product. E-commerce is taking the Web by storm, but it doesn't always result in objective, information-rich resources for the end user. People want information. They don't always have hours to invest in exploring hypertext document collections that might ultimately provide great depth of coverage on a particular topic, but only if you follow the right path. They want to ask a question and get an answer.

Users want to search the Web. Virtually any searchable collection of data can be easily made accessible through a simple HTML form, and many already have. Those that are not are usually unavailable for non-technical reasons such as a lack of a mechanism for protecting the intellectual content or ensuring proper use. While the capabilities and results presentations of different search engines may vary, such services are maturing quickly. Technologies such as Java and JavaScript allow sophisticated, stateful search services to be provided with all the capabilities that were once only available in custom, proprietary clients. Fundamental similarities in the interfaces of search systems are allowing developers to implement mechanisms such as Apple's Sherlock to allow users to perform multiple site searches in parallel. Free search engines are allowing more and more websites to index their content. But like the websites themselves, there is a definite need for someone to help users sort through the options and select good quality resources.

Searching on the Web is still defined as much by what is indexed as by what is not. There are some interesting trends in the field of Web searching and some worrisome trends. With Apple's introduction of Sherlock, metasearching has stepped onto the desktop in a big way. Metasearching and federated searching seem to be the most promising developments in the realm of search engines. Individual search sites all face the same problem: they cannot index everything on the Web. So by applying sometimes secret, proprietary rules to the act of collecting and indexing documents, sites

manage to present the appearance of completeness. But the truth of the matter is that in order to index every site on the Web, at least some information about each site would have to be gathered and indexed on a regular basis. No technology yet has managed to entirely eliminate the need for some type of reconstructible representation of the indexed document to exist on the search server. Capturing the "soul" of a document and applying qualitative attributes as well as author-provided metadata renders the document locatable. The problem with this system is that over-dependence on author-provided metadata can cause problems. Some authors will mis-represent their content in order to ensure higher visibility, even going so far as to "spoof" search engines with keywords describing competitors or unrelated services. Ethical or not, this is a big problem for Web search engines. Human-mediated portal sites such as Yahoo are still the only sure-fire solutions to this problem.

Search Engines

Search engines themselves are available in many different implementations. Many are derived from the vector space search model, initially described by Gerard Salton. In its simplest form, the vector space search engine consists of a word set that encompasses all the words in all of the documents in an indexed collection. This collection represents a vector, which is a line with an n-dimensional endpoint, where n corresponds to the number of documents in the collection. For example, if you had only three words in your document set, the endpoint of the vector for your word set would have an x, y and z coordinate. Each coordinate value corresponds to the number of times each word occurs in all of the documents. A similar vector also represents each document. So searches themselves are converted to vectors, and the most similar vectors to the search vector represent the best matches in the document collection. The beauty of this system is the ease with which relevance searches can be constructed. If you want more documents like one that showed up in your document set, the vector representing that document already exists, so it is simply converted into a query. Below is the equation that you would use to compare two document matrices u and v (we'll use this formula to build a search engine a bit later!):

$$\cos\theta = \frac{u \bullet v}{||u||||v||}$$

Each matrix contains a single column of data. Each row represents a word in the collection of words that make up the documents represented by u and v. The value in each row contains an integer value of 0 or greater that corresponds to the number of occurrences of that particular word in the document. You can then convert each document matrix into a vector. When this vector is plotted, the end point coordinates correspond to the values in the matrix. For example, the integer value in row 1 represents the coordinate position on the x-axis; the value in row 2 represents the coordinate position on the y-axis, etc. The cosine of theta is the angle between those two vectors. And the smaller this value is, the more similar the two documents are!

At the lower end of the search engine spectrum are brute force text scanners that do not build indexes. Many Web sites use these sorts of search engines. They are often based on or derived from system tools originally developed for UNIX such as grep. They tend to work fairly well for small document sets, providing basic single or multi-word case-insensitive query capabilities. But as a

document collection grows, it takes longer for a search to be completed. A further drawback is that it is difficult to search content spanning multiple formats with text scanning software, whereas most programs that build indexes of a document collection usually perform some type of format conversion during the indexing process. In the end, text scanner-style search engines simply do not scale.

By contrast, vector space search engines can scale nicely. Since they represent documents as compact integer matrices, the storage requirements for the index are a fraction of the storage required for the document set. And while vector space search engines do not directly solve the problem of indexing distributed document collections, their efficiency makes it a more easily solvable problem.

So let's take a look at the inner workings of a simple vector space search engine. The following Java application builds an index of a set of documents by representing each document as an n-dimensional vector. To improve performance, it also uses a configurable stop-word list to toss out articles and other common words. The first class we will look at is one representing an indexed document in memory. The class, *docMatrix*, contains the contents of a document, and can compare itself to another matrix (such as a query matrix). Since this comparison involves linear algebra functions to determine the angle between the document represented by this instance and some other document (a query), this class contains the mathematical meat of the application:

```
import java.util.*;
import java.io.*;
import java.lang.*;

/**
 * docMatrix is a vector representing a single document identified
 * by filename, in n-dimensions.  Each column in the matrix represents
 * the number of times a specific word appears in this document.
 * It can extend itself, increment the value in a specific column and
 * perform calculations to compare itself to another vector.
 * Implements serializable interface so that instances of this class can
 * be saved to a file.
 *
 * @author James Powell, jpowell@vt.edu
 * @version 1.0
 */
public class docMatrix implements Serializable {
  private Vector wordVector;
  private String filename;

  /**
   * This constructor creates a new, empty matrix (Vector java type)
   */
  public docMatrix() {
    wordVector=new Vector();
  }



  /**
```

```
     *
     * @param startSize number of elements it should initially accomodate
     * this constructor creates a new, empty matrix (vector java type)
     */
    public docMatrix(int startSize) {
      wordVector=new Vector(startSize);
    }

    /**
     * adds one to the integer value found at element
     *
     * @param    element     element to be incremented
     */
    public void incrementMatrixElementValueAt(int element) {
      Integer currentValue=new
      Integer((String)wordVector.elementAt(element));
      int currentVal=currentValue.intValue();
      String newValue=String.valueOf(currentVal+1);
      wordVector.setElementAt((Object)(newValue), element);
    }

    /**
     * finds and returns integer value at element
     *
     * @param    element     element to be looked up
     * @return an integer representing the value of the element requested
     */
    public int getMatrixElementValueAt(int element) {
      if (element>=wordVector.size()) {
        return 0;
      } else {
        Integer currentValue=new
        Integer((String)wordVector.elementAt(element));
        return currentValue.intValue();
      }
    }

    /**
     * extends the matrix (java Vector) by one element, and inserts a 0
     */
    public void extendMatrix() {
      String newValue=String.valueOf(0);
      wordVector.addElement((Object)newValue);
    }

    /**
     * inspects itself and returns current size of matrix


     * @return an integer representing the size of the matrix


     */
    public int showSize() {
      return wordVector.size();
    }

    /**
     * displays to the console the contents of the matrix (primarily
     * for debugging)
     */
    public void showMatrix() {
      System.out.println(wordVector.toString());
```

```
}
/**
 * set the filename field value for this instance of docMatrix
 * (perhaps this should be performed by constructor?)
 *
 * @param    filename    file being described by this matrix
 */
public void setFilename(String filenameParam) {
  filename=filenameParam;
}

/**
 * inspects itself and returns the filename of the file represented
 * by this matrix
 *
 * @return string containing filename field of this instance of
 * docMatrix
 */
public String showFilename() {
  return filename;
}

/**
 * calculate angle between this vector and another vector
 * angle is inverse cosine of the dot product of two matrices divided
 * by the product of the length of the same matrices
 *
 * @param    vMatrix docMatrix instance describing another document
 * vector
 * @return double numeric value representing the angle in degrees
 * between these two vectors
 */
public double calculateAngle (docMatrix vMatrix) {
  double theta=0.0;
  docMatrix uMatrix=this;
  docMatrix u=null;


  docMatrix v=null;



  if (uMatrix.showSize()>vMatrix.showSize()) {
    u=uMatrix;
    v=vMatrix.padMatrix(vMatrix,uMatrix.showSize());
  } else {

  if (vMatrix.showSize()>uMatrix.showSize()) {
    v=vMatrix;
    u=uMatrix.padMatrix(uMatrix,vMatrix.showSize());
  } else {
    u=uMatrix;
    v=vMatrix;
  }}

  double udotv=calculateDotProduct(u,v);
  // System.out.println("Dot Product was "+udotv);
  double normuv=calculateNorm(u)*calculateNorm(v);
  // System.out.println("Norm was "+normuv);
  if (normuv>0) {
    theta=Math.acos(udotv/normuv);
  } else {
```

```
    }
    theta=convertToDegrees(theta);
    return theta;
  }

  /**
   * convert a value in radians to degrees
   *
   * @param    radianValue double numeric value in radians
   * @return double numeric value converted to degrees
   */
  public double convertToDegrees(double radianValue) {
    double degrees;
    degrees=radianValue*(180/Math.PI);
    return degrees;
  }

  /**
   * calculate dot product of two vectors (sum of the product of
   * each corresponding matrix element
   *
   * @param    u     docMatrix instance describing a document vector
   * @param    v     docMatrix instance describing a document vector
   * @return double numeric value representing dot product of u and v
   */
  public double calculateDotProduct(docMatrix u, docMatrix v) {


    int count, u_n, v_n;


    double dotProduct=0;
    for (count=1; count<u.showSize(); count++) {
      u_n=u.getMatrixElementValueAt(count);
      v_n=v.getMatrixElementValueAt(count);
      dotProduct=dotProduct+(u_n*v_n);
    }
    return dotProduct;
  }

  /**
   * calculate the length of a vector
   *
   * @param    x     docMatrix instance describing a document vector
   * @return double numeric value representing the length of vector x
   */
  public double calculateNorm(docMatrix x) {
    int count, x_n;
    double norm=0;
    for (count=1; count<x.showSize(); count++) {
      x_n=x.getMatrixElementValueAt(count);
      norm=norm+Math.pow(x_n, 2);
    }
    norm=Math.sqrt(norm);
    return norm;
  }

  /**
   * extend this instance of docMatrix so that it has the same number
   * of elements as another vector
   *
   * @param    x     docMatrix instance
```

```
    * @return docMatrix instance reference
    */
   * @param    dimension   size it should be
  private docMatrix padMatrix(docMatrix x, int dimension) {
    for (int count=x.showSize(); count<dimension; count++) {
      x.extendMatrix();
    }
    return x;
  }
}
```

There are two constructors for the *docMatrix* class. If you create a new instance of the class and pass it no arguments, an empty vector field is created for the instance. If you pass it a parameter, then a vector of size startSize is created for the instance. Next is a set of methods that allow you to look at and modify the matrix and the other field that each *docMatrix* instance has – a filename corresponding to the file represented by *docMatrix*. A second set of methods allows you to perform the calculations necessary to compare this matrix to another matrix.

The *index* class is a collection of *docMatrix* objects. As its name implies, it is an index for a document set. It loads the stop word list, and then parses each document in the document set, creating a new instance of *docMatrix* to represent each one. The size of the master word list corresponds to the number of dimensions occupied by the collection of documents. A document is actually converted into a list of integer values. Each integer value corresponds to a word in the master word list, and a point on the axis representing that word. The *index* can reveal a particular matrix, the master word list, or the *docMatrix* instance that corresponds to a particular file. It can also iterate through all documents in the index to perform a query, calling the calculateAngle() function for each.

```
import java.util.StringTokenizer;
import java.io.*;
import java.util.*;

import docMatrix;
import resultSet;

/**
 * the index class contains an array of docMatrix objects and a word list
 * which contains all of the words from the document set represented by
 * this array.  The three main tasks performed by the index are building
 * the word list, performing searches and sorting search results.
 *
 * @author James Powell, jpowell@vt.edu
 * @version 1.0
 */
public class index extends vectorSpaceSearch implements Serializable {
  private Vector wordList=new Vector();
  private Vector stopwordList=new Vector();
  private docMatrix[] docList;
  private transient int fileCount=0;
  private int numbDocs;

  /**
   * this constructor sets the filecount and creates a new array of
```

```
 *
 * @param   numbFiles   integer set to number of files to be indexed
 * docMatrix objects
 */
public index (int numbFiles) {
  numbDocs=numbFiles;
  docList=new docMatrix[numbDocs];
}
```

```
/**
 * this method adds a new file to the index and calls another method to
 * extract the words and build a vector describing this document
 *
 * @param    filename    string containing filename of file to index
 * @param    fileContent string containing contents of this file
 */
public void addFile (String filename, String fileContent) {
  fileCount+=1;

  docList[fileCount]=new docMatrix();
  docList[fileCount].setFilename(filename);
  showTokens(fileContent, fileCount);
}

/**
 * this method loads the stopword list into the field stopwordList
 *
 */
 private void loadstopWords() {
  char [] fileContents;
  String stopwordFile="stopwords.txt";
  try {
    File aFile=new File(stopwordFile);
    FileReader in=new FileReader(aFile);
    int size=(int)aFile.length();
    fileContents=new char[size];
    int chars_read=0;
    while (chars_read<size) {
      chars_read+=in.read(fileContents,chars_read,size-chars_read);
    }
    String contents=String.valueOf(fileContents);
    StringTokenizer st =
      new StringTokenizer(contents, punctuation);
    stopwordList=new Vector();
    while (st.hasMoreTokens()) {
      String token = st.nextToken();
      stopwordList.addElement(token);
    }
  }
  catch (IOException e) {}
}


/**
 * this method breaks the current document into individual words,
 * and keeps track of the number of occurrences of each word.
 * If a new word is found, it is also added to the master word list
 * for this document collection.
 *
```

```
     * @param    dataFile    contents of file to be processed
     * @param    docId filenumber in document collection
     */
    private void showTokens(String dataFile, int docId) {
      StringTokenizer st =
        new StringTokenizer(dataFile, punctuation);
      int location;

      loadstopWords();
      while (st.hasMoreTokens()) {
        String token = st.nextToken();
        String lcToken= token.toLowerCase();
        if (!(stopwordList.contains(lcToken))) {

        if (wordList.contains(lcToken)) {
          location=wordList.indexOf(lcToken);
          while (docList[docId].showSize()<=location) {
            docList[docId].extendMatrix();
          }
          docList[docId].incrementMatrixElementValueAt(location);
        } else {
        // if (!(wordList.contains(lcToken))) {
          System.out.println(lcToken);
          // word was not in master word list, so add it
          wordList.addElement(lcToken);
          location=wordList.indexOf(lcToken);
          if (location==-1) {
            docList[docId].extendMatrix();
          } else {
            while (docList[docId].showSize()<=location) {
              docList[docId].extendMatrix();
            }
          }
          docList[docId].incrementMatrixElementValueAt(location);
        }
        }
      }
      docList[docId].showMatrix();
    }

    /**
     * this method displays the master wordlist on the console
     * it is used mainly for debugging purposes
     */
    public void showMatrix() {
      System.out.println(wordList.toString());
    }

    /**
     * this method returns the master word list when requested by other
   objects


     *


     * @return a vector containing all the words found in a document set
     */
    public Vector showWordList() {
      return wordList;
    }
```

```
  /**
   * this method attempts to find the documents in the docMatrix array
   * that are most similar to the queryMatrix
   *
   * @param    queryMatrix a matrix containing query terms
   * @return a string containing a segment of HTML code with links
   * to the best matches
   */
  public String findSimilarDocuments(docMatrix queryMatrix) {
    int count;
    resultSet [] queryResults=new resultSet[docList.length];
    StringBuffer results=new StringBuffer();
    for (count=1; count<docList.length-1; count++) {
      double score=docList[count].calculateAngle(queryMatrix);
      queryResults[count]=new resultSet();
      queryResults[count].setFilename(docList[count].showFilename());
      queryResults[count].setScore(score);
      queryResults[count].setFilenumber(count);
    }
    selectionSort(queryResults);
    for (count=1; count<docList.length-1; count++) {
      if ((queryResults[count].showScore()<90.0) &&
(queryResults[count].showScore()>0.0)) {
        results.append("<TR><TD>");
        results.append("<INPUT TYPE=\"radio\" NAME=\"RELEVANCE\"
VALUE=\""+queryResults[count].showFilenumber()+"\">");
        results.append(" file <A
HREF=\""+queryResults[count].showFilename()+"\">");
        results.append(queryResults[count].showFilename()+"</A></TD>");
        results.append("<TD> angle " + queryResults[count].showScore() +
"</TD></TR>\n");
      }
    }
    if (results.length()==0) {
      results.append("<TR><TD>");
      results.append("<STRONG>Nothing found</STRONG>");
      results.append("</TD></TR>");
    }
    return results.toString();
  }

  /**
   * this method sorts a set of objects by a specific field
   *
   * @param    A     an array of objects of type resultSet
   */
  static void selectionSort(resultSet[] A) {
    // sort A into increasing order, using selection sort
    // A has values from 1 to A.length
    for (int lastPlace = A.length-2; lastPlace > 0; lastPlace--) {
      // Find the largest item among A[1], A[2], ... A[lastPlace],
      // and move it into position lastPlace by swapping it with
      // the number that is currently in position lastPlace
      int maxLoc = 1;  // location of largest item seen so far
      for (int j = 1; j <= lastPlace; j++) {
        // System.out.println("lastPlace="+lastPlace+" j="+j);
        // System.out.println(A[j].showFilename()+j+A[j].showScore());
        if (A[j].showScore() > A[maxLoc].showScore())
```

```
        }
        resultSet_temp = A[maxLoc];  // swap largest item with A[lastPlace]
        A[maxLoc] = A[lastPlace];
        A[lastPlace] = temp;
      }
    }

    /**
     * this matrix returns the instance of docMatrix that corresponds to
     * the specified file number
     *
     * @param    filenumber  an integer representing an indexed file
     * @return a reference to an instance of the docMatrix class, describing
     * one of the documents in the index
     */
    public docMatrix lookupDoc(int filenumber) {
      return docList[filenumber];
    }
  }
```

The *index* class extends the *vectorSpaceSearch* class, which defines characteristics of this search engine. In this case, the *vectorSpaceSearch* class simply defines the collection of characters that represent word boundaries for the types of documents it can index.

```
/**
 * vectorSpaceSearch is a trivial class that defines a single
 * constant, the tokens that indicate word boundaries.
 *

 * @author James Powell, jpowell@vt.edu

 * @version 1.0
 */
public class vectorSpaceSearch {
    public static final String punctuation=" ,.;:(){}[]<>=!?'\"\t\r\n\\+-
*/";
}
```

The *indexer* class takes care of discovering and loading documents from the file system. It also processes queries against the document set, including relevance feedback queries. Since every query is converted into a matrix, a relevance query simply copies the selected document into a query. So when a user asks to find more documents like a certain document, the selected document becomes a query matrix. The *indexer* class has methods for saving and loading an index (which is why the *index* class implements the Serializable interface). This class has a main method because it is the tool used by the search engine administrator to build an index. It expects a single string parameter representing the directory containing the document that it is indexing.

```
import java.util.StringTokenizer;
import java.io.*;
import java.util.*;
import query;
import index;
import java.util.zip.*;

/**
 * indexer is responsible for indexing a document collection and initiating
searches
 * against the collection.
 *
 * @author James Powell, jpowell@vt.edu
 * @version 1.0
 */
public class indexer {

  private index collectionIndex;

  // You may need to provide full path to Vsindex.zip file on your
system!!!

  /**
   * this method loads the contents of a text file into a character array
and
   * returns a string containing the file's contents
   *
   * @param    filename    name of file to open and load
   * @param    docId a unique numeric value that is generated by a counter
   * @return a string containing the entire contents of the file
   */


  private String loadFile(String filename, int docId) {


    char [] fileContents;
    try {
      File aFile=new File(filename);
      FileReader in=new FileReader(aFile);
      int size=(int)aFile.length();
      fileContents=new char[size];
      int chars_read=0;
      while (chars_read<size) {
        chars_read+=in.read(fileContents,chars_read,size-chars_read);
      }

      return String.copyValueOf(fileContents);
    }
    catch (IOException e) {}
    return null;
  }

  /**
   * this method finds all text files in a specified directory, counts them
and
   * loads the contents of each, then creates a new instance of docMatrix
to
   * contain the vector representing the contents of each document
   *
   * @param    directoryName     directory containing files to be indexed
```

```
  private void findFiles(String directoryName) {
    String[] files;
    File dir = new File(directoryName);
    if (!dir.isDirectory())
      throw new IllegalArgumentException("FileLister: no such directory");
    files = dir.list();
    int numbFiles=files.length;
    collectionIndex=new index(numbFiles+1);

    for (int i=0; i<numbFiles; i++) {
      if ((files[i].endsWith("html")) || (files[i].endsWith("htm")) ||
          (files[i].endsWith("txt")) || (files[i].endsWith("xml"))) {
        System.out.println("Loading #" + i + " " +
directoryName+"/"+files[i]);
        String fullFilename=directoryName+"/"+files[i];
        collectionIndex.addFile(fullFilename, (loadFile(fullFilename, i)));
      }
    }
    collectionIndex.showMatrix();
  }
```

```
  /**
```

```
   * this method creates an instance of the query class which constructs a
matrix
   * that describes a vector that will be compared against each document in
the
   * collection.
   *
   * @param    searchTerms a space delimited list of search terms
   * @return string containing a segment of HTML code with links to
matching docs
   */
  public String doQuery (String searchTerms) {
    query newSearch=new query(searchTerms, collectionIndex.showWordList());
    String
results=collectionIndex.findSimilarDocuments(newSearch.showQueryMatrix());
    newSearch.showQueryMatrix();
    return results;
  }

  /**
   * this method is similar to doQuery, but instead of a string of search
terms,
   * the user has specified that documents similar to a specified document
be
   * found.  So this method uses the contents of the document matrix
specified by
   * the user as the vector to be compared with other documents in the
collection
   * and the instance of query gets the contents of the selected document.
   *
   * @param    docSelected integer representing the document ID to be used
   * @return a string containing a segment of HTML code with links to
matching docs
   */
  public String doRelQuery (int docSelected) {
    docMatrix docsLike=collectionIndex.lookupDoc(docSelected);
    query newSearch=new query(docsLike, collectionIndex.showWordList());
```

```
    return results;
  }
  String
results=collectionIndex.findSimilarDocuments(newSearch.showQueryMatrix());
  /**
   * this method saves the index, which is an array of docMatrix objects.
   * The array is gzipped before it is written to a file called VSindex.zip
   */
  public void saveIndex () {


    try {


      FileOutputStream fos=new FileOutputStream(indexFilename);
      GZIPOutputStream gzos=new GZIPOutputStream(fos);
      ObjectOutputStream out=new ObjectOutputStream(gzos);
      out.writeObject(collectionIndex);
      out.flush();
      out.close();
    }
    catch (IOException e)
      { System.out.println("Error writing index.  Error was "+e); }
  }

  /**
   * this method loads an index, which is an array of docMatrix objects.
   * It looks for a gzipped file called VSindex.zip
   */
  public void loadIndex() {
    try {
      FileInputStream fis=new FileInputStream(indexFilename);
      GZIPInputStream gzis=new GZIPInputStream(fis);
      ObjectInputStream in= new ObjectInputStream(gzis);
      index newIndex=(index)in.readObject();
      in.close();
      collectionIndex=newIndex;
    }
    catch (IOException e)
      { System.out.println("Error reading index. Error was "+e); }
    catch (ClassNotFoundException e)
      { System.out.println("Error reading index. Error was "+e); }
  }

  /**
   * the main of indexer is used to build an index of a set of documents
   *
   * @param    args[]      an array of strings, actually just a directory
name
   */
  public static void main(String args[]) {
    indexer getWords=new indexer();
    getWords.findFiles(args[0]);
    getWords.saveIndex();
  }
}
```

The *query* class converts a user query into an n-dimensional vector that contains the same number of elements as any document in the vector space search engine *docMatrix* instances. It extends

vectorSpaceSearch, since it needs to identify the string tokens using the same delimiters as used by the index class. The stop list is used indirectly, since the query is built by examining the contents of the master word list for the document set. Words that have been eliminated during the indexing process are simply ignored when the query class encounters them. The *query* class has two constructors. One is used when the query is based on one or more words a user keyed into a form. The other is used when a user selects a document in a results set and asks to see more documents like it (a relevance query).

```
import java.util.StringTokenizer;
import java.io.*;
import java.util.*;
import docMatrix;

/**
 * query contains a vector describing a user query
 * It extends vectorSpaceSearch, which is a trivial class
 * that defines a single constant, the tokens that indicate
 * word boundaries.
 *
 * @author James Powell, jpowell@vt.edu
 * @version 1.0
 */
public class query extends vectorSpaceSearch {

  private docMatrix queryMatrix;

  /**
   * this constructor allows a user to specify a document
   * in the document collection as a query - a show me
   * more documents like this one query.
   *
   * @param    docChoice   instance of docMatrix describing
   * the document chosen by the user
   * @param    wordList    master word list for this document collection
   */
  public query (docMatrix docChoice, Vector wordList) {
    int location=wordList.size();
    queryMatrix=docChoice;

    while (queryMatrix.showSize()<=location) {
      queryMatrix.extendMatrix();
    }
    queryMatrix.showMatrix();
  }

  /**
   * this version of the constructor supports the more typical
   * search where a user enters one or more search terms
   * the search terms are used to construct a vector which is
   * then compared to vectors describing documents in the
   * document collection.
   *
   * @param    queryString a string containing the user's search terms
   * @param    wordList    master word list for this document collection
```

```
   */
  public query (String queryString, Vector wordList) {
    StringTokenizer st =
      new StringTokenizer(queryString, punctuation);
    int location;
    queryMatrix=new docMatrix();

    while (st.hasMoreTokens()) {
      String token = st.nextToken();
      String lcToken= token.toLowerCase();
      if (wordList.contains(lcToken)) {
        location=wordList.indexOf(lcToken);
        while (queryMatrix.showSize()<=location) {
          queryMatrix.extendMatrix();
        }
        queryMatrix.incrementMatrixElementValueAt(location);
      } else {
        System.out.println(lcToken);
        wordList.addElement(lcToken);
        location=wordList.indexOf(lcToken);
        if (location==0) {
          queryMatrix.extendMatrix();
        } else {
          while (queryMatrix.showSize()<=location) {
            queryMatrix.extendMatrix();
          }
        }
        queryMatrix.incrementMatrixElementValueAt(location);
      }
    }
    queryMatrix.showMatrix();
  }

  /**
   * this method returns the reference to this instance of
   * a queryMatrix
   *
   * @return a reference to this queryMatrix instance
   */
  public docMatrix showQueryMatrix() {
    return queryMatrix;
  }
}
```

Finally, the *resultsSet* class is responsible for providing various attributes about the result of a query, including the score of each document when compared to the query, and the actual filename of each document in the index.

```
/**
 * an instance of result set describes how a single document in the index
 * compares to a query.
 *
 * @author James Powell, jpowell@vt.edu
 * @version 1.0
 */
```

```
public class resultSet {
  private int size;
  private String filename;
  private int filenumber;
  private double angleScore;
  // Change the value of the following string to correspond to your Web
server doc root!!
  // Here is an example: /usr/local/etc/httpd/htdocs (no trailing slash)
  private static final String webServerDocRoot="/web_server_doc_root";

  /**
   * this method sets the filename field for this result
   *
   * @param     filenameParam   name of file that was compared to query
   */
  public void setFilename(String filenameParam) {
    int filePathLen=webServerDocRoot.length();
    filename=filenameParam.substring(filePathLen+1);
  }
  /**
   * this method sets the score, which is the angle between the vector
   * that describes this document and the vector that describes the query
   *
   * @param    score a double precision numeric of the angle in degrees
   */
  public void setScore(double score) {
    angleScore=score;
  }

  /**
   * this method sets the filenumber for the document that was compared
   * to the query
   *
   * @param    filenum     integer containing unique numeric id of file
   */
  public void setFilenumber(int filenum) {
    filenumber=filenum;
  }
```

```
  /**
```

```
   * this method shows the filename field of this instance of resultSet
   *
   * @return a string containing the filename
   */
  public String showFilename() {
    return filename;
  }

  /**
   * this method shows the score of this document
   *
   * @return a double precision angle value in degrees between vectors
   */
  public double showScore() {
    return angleScore;
  }

  /**
   * this method shows the filenumber for this instance
```

```
    * @return an integer representing file id
    */
   public int showFilenumber() {
     return filenumber;
   }
}
```

This search engine is not fast, and consumes a fair amount of memory. But it is essentially a textbook example of how linear algebra can be applied to the problem of searching collections of documents. At the end of this paper (Appendix A), you will find an implementation that ties the vector space search engine classes together into an application. This implementation is built around a simple Web server, but it would be just as easy to implement the search engine as a servlet, or as part of a larger application.

Formulating Queries

Making text searchable is fairly well understood. What is less well understood is how users want to search the content. In addition to single and multi-word queries, some users want to perform phrase queries, where a string of words must occur exactly as specified in order to constitute a match. Many users expect to be able to perform Boolean queries, which allow them to indicate what combinations of words should be allowed in documents in the results set. More sophisticated users want to be able to perform near, with, and relevance feedback queries against the full text. Still other users prefer searching catalog records that consist of metadata describing each item in the collection. All of these options and more can be found on the Web.

The vector space search engine example above supports multiword queries and relevance feedback queries. When a user presents one or more words as search terms, it creates a matrix that represents all of the words that are not in the stop word list. This matrix is then used in calculations that compare it to each document matrix in the index. When a user performs a relevance query, the document they select is used as a query, and the same calculations are performed. While not mutually exclusive, it would be difficult to support Boolean and proximity searching in a search engine that supports relevance queries using this vector model.

More recently, users have begun to become aware of the shortcomings of various search engines. They are beginning to realize that the AltaVistas and HotBots of the Web do not index every single document ever published, or even all documents currently available on the Web. So they have been seeking alternatives that allow them to search multiple sites in parallel. In fact, it is very possible that more Web queries in the not too distant future will be parallel, distributed queries than will be queries targeted at one specific search engine.

Is this an indicator that current search engines are not doing a good job? Maybe. But search engines have improved in many respects. They've gotten faster. They've remained free. They've combined searching with high-level categories to form "portal" sites, which have become very popular with users who've lost patience with Boolean searching, stemming operators, and "natural language" queries that seem to utilize some obscure, stilted notion of natural language designed by chattering

parrots. Areas where they've not improved include displaying result sets that list documents that no longer exist. They've made very little real progress in the area of multi-lingual searching. Most boast of multilingual interfaces but that usually means someone translated the text of the search page into another language. However, both AltaVista and Yahoo have made significant contributions in this field. Yahoo has provided geographically and linguistically distinct search sites like Yahoo Japan and Yahoo Italia. AltaVista has collaborated with machine translation services to provide simple translations of documents between various languages. But no search site yet provides targeted query translation, and this really seems like a metasearch-level service touching upon many areas of linguistic computation that are still being researched. Search engines are sluggish in terms of updating their content. Changes in documents indexed by a site might not be reflected in a search engine's index for weeks, months or ever.

But the lack of standards is perhaps the most frustrating characteristic of Web searching. It is true that some standards having the potential to allow interoperability between search engines, and defining uniform search interfaces have been developed. But there is no widely supported protocol for search engines or search clients and search engines to exchange information about breadth of content coverage, search features, or metadata. Projects like the Dienst system, which defines an architecture and a protocol for distributed digital libraries, (http://www.cs.cornell.edu/cdlrg/dienst/protocols/DienstProtocol.htm) and the Stanford Protocol Proposal for Internet Retrieval and Search, which is also known as STARTS (http://www-db.stanford.edu/~gravano/starts.html), both describe mechanisms for distributed heterogeneous search systems. Yet, no major search engine vendors have implemented support for the architecture or protocols defined by these projects, nor even for tried and true standards (at least according to libraries) such as Z39.50. The only near-term solution for allowing searches to span multiple search engines seems to be in heuristic solutions that take advantage of some inherent similarities among search engines. As for interface design, the best practices are emerging from the field of Human Computer Interaction (such as Ben Schneiderman's four-phase framework for search: see http://ijhcs.open.ac.uk/shneiderman/shneiderman-t.html) but the closest thing to a standard is the simple search, which consists of one fill-in field and one button. There is plenty of work left to be done to determine how to present advanced search features in a standard, universally understood way.

So the immediate future of Web searching will be defined by various heterogeneous meta- or federated search systems. Such systems allow software to programmatically conceal and resolve differences between search engines, using heuristics that have been encoded in some manner. This allows such systems to even support pre-query processing such as use of a thesaurus to improve a query or the translation of a query into the primary language used by the collection before it is submitted to a search engine. It also allows for post-query processing to merge and possibly translate results.

Metasearching

What is metasearching all about? Simply put, it is the act of delivering query terms to more than one search engine, in the format expected by each search engine. Some pseudo-metasearch systems present a set of pre-filled forms at an intermediate stage, and then expect the user to submit each query as desired. The user can achieve a similar effect by relying on a set of search engine

bookmarks and a few open browser windows. True metasearching involves some form of query reformulation and automatic delivery. Beyond that, there are many additional tasks that can be performed.

Sherlock was added to the Macintosh operating system starting with version 8.5 (http://www.apple.com). It was the first integrated, extensible Web metasearch system ever provided as part of the services of an operating system. Sherlock employs a simple markup language that allows anyone with basic knowledge of HTML to build descriptions of almost any Web-accessible search engine. These descriptions can then be installed into a special folder, which Sherlock consults whenever a user attempts to perform a find action on their local hard drive's contents or for Web content. The user selects a set of search engines to which their query should be sent, and Sherlock takes care of mapping between search engines, delivering queries, and merging results.

Several Web-based metasearch systems employ similar solutions. A similar system called the Federated Searcher (http://jin.dis.vt.edu/fedsearch/), is a Web-based system that utilizes XML site descriptions to allow users to search descriptions of search engines, and deliver queries to multiple sites. The site description markup language is called SearchDB-ML (the DTD is included in Appendix B). The following is an example of a search site description for the AltaVista search engine:

```
<?XML version="1.0">

<!DOCTYPE searchdb SYSTEM "searchdb.dtd">
<SEARCHDB>
  <ENGINE>
    <NAME>Alta Vista</NAME>
    <URL>http://www.altavista.digital.com/</URL>
    <TYPE>Full text</TYPE>
      </ENGINE>
      <INSTANCE>
        <CONTENT>
          <TITLE>Alta Vista</TITLE>
          <CREATOR>Yahoo! Inc.</CREATOR>
     <DESCRIPTION>Full text search of WWW that supports
          multiple languages, newsgroup searching, natural
          language searches, dynamic categorization, phrase and
          boolean searches, search by host, title, images, links,
          date ranges, anchors and domains.
          A GFS top_ten site.</DESCRIPTION>
          <IDENTIFIER>http://www.altavista.digital.com/</IDENTIFIER>
     <LANGUAGE TYPE="ISO6391988">en</LANGUAGE>
     <RIGHTS>Publicly accessible</RIGHTS>
   </CONTENT>

   <INTERFACE TYPE = "web">
     <URL>http://www.altavista.digital.com/cgi-bin/query</URL>
     <FORMTYPE>get</FORMTYPE>
         <QUERYFIELD>
               <FORMNAME>q</FORMNAME>
         </QUERYFIELD>
                <QUERYFIELD>
                        <FORMNAME>pg</FORMNAME>
```

```
                </QUERYFIELD>
                <QUERYFIELD>
                        <FORMNAME>what</FORMNAME>
                      <DEFAULT>web</DEFAULT>
                </QUERYFIELD>
                        <FORMNAME>kl</FORMNAME>
                      <DEFAULT>XX</DEFAULT>
        </QUERYFIELD>

        <LANGUAGE TYPE="ISO6391988">en
        </LANGUAGE>
    </INTERFACE>



    <RESPONSE TYPE="html"/>



      </INSTANCE>
</SEARCHDB>
```

The markup is divided into two sections. The first section describes the characteristics of the search engine and the document set it indexes. This information is used to help users select the best search engines to send a query to from a catalog of XML descriptions. The second section describes the Web interface to the search engine. This information is used by an application to create and submit a query to the search engine just as if a user had visited the search engine's website and submitted the query directly.

The implementation details are quite simple. An application that uses SearchDB-ML descriptions to perform metasearches first provides some mechanism for the user to select which search engines to search. Next, it accepts a query (one or more words) from the user. Finally, it constructs a CGI request using the information it finds in the <INTERFACE> section of each site description. The request targets the URL contained within the <URL> tag, it sets a number of form element values contained within <DATABASE> or <CONTROL> elements, using their <DEFAULT> value. Finally, it assigns the user's query to the first <QUERYFIELD> element, which has no <DEFAULT> value. Then it submits it as a GET or POST request, depending on the value of the <FORMTYPE> element. Our implementation also checks the <LANGUAGE> element value, and communicates with another network application, called the TRP (Translation Request Protocol) Server. Prior to submitting a query to a search engine, it tries to translate the individual query terms into the language of the target server.

An additional feature of the Federated Searcher system was the ability of the metasearch software to identify the primary language of each search engine targeted by the user. It then employed a mechanism to communicate with a dictionary-based word translation service to request translations of queries from, for example, English to Japanese, before delivering the query to a Japanese search engine. It is easy to underestimate the importance of such a feature. But if you have ever tried to search a search engine in a language you do not speak well, particularly one that uses different

characters and employs different techniques for query entry, encoding, and parsing, then you will probably appreciate being able to bypass these hurdles. This feature merely hints at the possibilities as networks and processors become faster, and applications and developers become smarter.

One big question in the emerging field of metasearching is what will be the reaction of the major search engine sites? Until Sherlock appeared on the Macintosh, the threat to individual search engines was fairly small. But imagine if later version of the Windows operating system supports a Sherlock-like mechanism that eliminates ads, conceals the complexity of a search engine and seamlessly merges results from different search engines into a single set of links. Who will pay for search engines? Will there be royalties associated with the use of such tools? Will this be decided in the courts? This will probably be the case. What could be a big win for end users will likely cost search engines in lost revenue and legal fees, so the battle over metasearches hasn't yet begun.

Intranet-based metasearches will likely lead the pack in terms of functionality. Linking large multinational companies will require that various barriers to communication will have to be overcome in a secure, seamless manner. There is a great opportunity for libraries to lead the unification of the scattering of Web-search engines that inevitably crop on campuses, in corporations and around the Web. Schemes for classifying and describing search engines will require experts to evaluate and catalog search engines.

Conclusion

What does this mean for your site? First of all, if you haven't made your website searchable, do it. There are a number of free search engines available for indexing content stored on a single Web server. You could even use the vector space search engine described in this article, although it is intended to be used as an instructional tool rather than as a production Web search engine. For high traffic sites, or for indexing multiple servers, you may want to consider purchasing software such as Infoseek's Ultraseek search engine, or contracting with a vendor such as Inktomi. Or you can establish indexes for each site and build a page pointing users to the collection of indexes. Or you can try out Javascript or Perl-based metasearch systems that provide simple, transparent metasearching without a lot of bells and whistles. By providing your own Web-based keyword query, you can reach a wider audience than ever before.

Next, prepare your site so that it is search-engine friendly. Danny Sullivan, of "The Search Engine Report" recommends that you "think of search engines as the third major type of Web browser." You should carefully consider what words and phrases you expect users might use to find your site, and then make sure those words actually appear in the documents you would want them to be directed to by search engines. You can also use a robots.txt file to control which documents are indexed by external search engines. If you first establish the collection of documents you do want search services to focus on, then you can devote extra time to designing them for all three types of Web browsers! Keep in mind that it does you little good to show up in results sets unexpectedly. You want your content to show up in the right results sets.

Appendix A
Pulling Together the Search Engine Example

One way the vector space search engine described in this paper could be implemented is as a Java servlet. The code below defines a servlet called vsServlet. It can be used along with the HTML document that follows it to provide a Web interface to your vector space search engine. Please note that you will need to make a few changes to this file and to a couple of the class files in the paper in order to customize the application for your environment.

```
import java.net.*;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import indexer;

public class vsServlet extends HttpServlet {

    private static indexer webCollectionIndexer;

      public void doGet (HttpServletRequest request,
           HttpServletResponse response)
      throws ServletException, IOException
      {
          PrintWriter  out;
          String       title = "vsSearch Output";

          // set content type and other response header fields first
          response.setContentType("text/html");

          String searchRequest=request.getParameter("query");
          String searchType=request.getParameter("RELEVANCE");
          String results;

          // creates an instance of indexer class to load index, do searches
          webCollectionIndexer=new indexer();
          webCollectionIndexer.loadIndex();

          // then write the data of the response
          out = response.getWriter();

        if (searchType!=null) {
          int docSelected=Integer.parseInt(request.getParameter("RELEVANCE"));
          // get doc matrix for this document

results=createResultsPage(webCollectionIndexer.doRelQuery(docSelected));
        } else {
          results=createResultsPage(webCollectionIndexer.doQuery(searchRequest));
        }
          out.println(results);

   }

  /**
   * this method builds the results page that is delivered in response to
   * a query
   *
   * @param     results string containing HTML stub of sorted, matching
   * document links
   * @return a string containing the complete results page to be sent
```

```
   * to browser
   */
  public String createResultsPage(String results) {
    StringBuffer resPage=new StringBuffer();
    resPage.append("<HTML><HEAD><TITLE>Results</TITLE>");
    resPage.append("<BASE HREF=\"http://your.server/\"></HEAD><BODY>");
    resPage.append("<H3>Results of search</H3>");
    resPage.append("<EM>The smaller the angle, the better the match</EM><P>");
    resPage.append("<FORM METHOD=\"GET\"
ACTION=\"http://your.server/servlets/vsServlet\">");
    resPage.append("<TABLE>");
    resPage.append(results);
    resPage.append("</TABLE>");
    resPage.append("<INPUT TYPE=\"submit\" VALUE=\"Find similar\">");
    resPage.append("</BODY></HTML>");
    return resPage.toString();
  }

  /**
   * the main of vsServlet is used for test purposes
   *
   * @param     args[]  a string to search for
   */
  public static void main(String args[]) {
    webCollectionIndexer=new indexer();
    webCollectionIndexer.loadIndex();

    String searchRequest=(args[0]);

    StringBuffer resPage=new StringBuffer();
    resPage.append("<HTML><HEAD><TITLE>Results</TITLE>");
    resPage.append("<BASE HREF=\"http://your.server/\"></HEAD><BODY>");
    resPage.append("<H3>Results of search</H3>");
    resPage.append("<EM>The smaller the angle, the better the match</EM><P>");
    resPage.append("<FORM METHOD=\"GET\"
ACTION=\"http://your.server/servlets/vsServlet\">");
    resPage.append("<TABLE>");
    resPage.append(webCollectionIndexer.doQuery(searchRequest));
    resPage.append("</TABLE>");
    resPage.append("<INPUT TYPE=\"submit\" VALUE=\"Find similar\">");
    resPage.append("</BODY></HTML>");
    System.out.println(resPage.toString());

  }

}
```

A simple HTML query page for the vector space search servlet:

```
<HTML>
<HEAD>
  <TITLE>Vector Space Search Engine Query Page</TITLE>
</HEAD>
<BODY>
  <H3>Vector Space Search Engine</H3>
  <H4>Test Query Page</H4>
  <FORM METHOD="get"
      ACTION="http://your.server/servlets/vsServlet">
    Enter your search terms: <INPUT NAME="query">
    <P>
    <INPUT TYPE="submit">
  </FORM>
</BODY>
</HTML>
```

A basic stop word list. Stop words should be listed one per line (save as stopwords.txt):

a
an
any
the
this
then
that

You can add any words you want to omit from the index to this file.

Appendix B
The SearchDB-ML DTD

The SearchDB markup language is one example of a methodology for describing search engines. It actually predates the Apple Sherlock system, but resembles it in some ways.

```
<!-- SearchDB Lite DTD version 1.0                              -->
<!-- Markup for text-based Internet searchable archives         -->
<!-- Conforms to XML 1.0                                        -->
<!-- Lite version is suitable for describing simple, or         -->
<!-- basic search capabilities of a site, esp. Web sites        -->
<!-- Created February 18, 1998                                  -->
<!-- James Powell, jpowell@vt.edu                               -->

<!ENTITY % formelems "formname+, size*,  default*, option*, mandatory*">
<!ENTITY % dbelems "formname+, default*, option*, mandatory*">
<!ENTITY % types "TYPE CDATA">

<!ENTITY amp "&">

<!-- Top level elements -->
<!ELEMENT searchdb         - -   (engine, instance)>

<!-- Markup elements for describing the type of software -->
<!-- by the site being described                         -->
<!ELEMENT engine     - -   (name, url, type)>
<!ELEMENT name       - -   (#PCDATA)>
<!ELEMENT url        - -   (#PCDATA)>
<!ELEMENT type          - -   (#PCDATA)>

<!-- Elements for the specific site described -->
<!ELEMENT instance         - -   (content, interface+, response)>

<!-- Metadata describing the type of content indexed by this site -->
<!-- Slightly abbreviated version of the Dublin core elements     -->
<!-- Documentation at http://purl.oclc.org/metadata/dublin_core   -->
<!ELEMENT content    - -   (title+, creator*, subject*, description*,
  abstract*, publisher*, date*, type*, format*, identifier*, language*,
  relation*, coverage*, rights*)>
<!ELEMENT title         - -   (#PCDATA)>
<!ELEMENT creator          - -   (#PCDATA)>
<!ELEMENT subject       - -   (#PCDATA)>
<!ELEMENT description   - -   (#PCDATA)>
<!ELEMENT abstract      - -   (#PCDATA)>
<!ELEMENT publisher     - -   (#PCDATA)>
<!ELEMENT date          - -   (#PCDATA)>
<!ELEMENT format     - -   (#PCDATA)>
<!ELEMENT identifier - -   (#PCDATA)>
<!ELEMENT relation   - -   (#PCDATA)>
<!ELEMENT coverage   - -   (#PCDATA)>
<!ELEMENT rights     - -   (#PCDATA)>

<!-- Elements for describing the interface to this search engine -->
<!ELEMENT interface  - -   (url+, formname*, formtype*, database*, (control |
queryfield)*, language+)>
<!ATTLIST interface TYPE CDATA #REQUIRED
                    VERSION (simple | complex) #REQUIRED>
<!ELEMENT formtype   - -   (#PCDATA)>

<!-- Elements for describing an input field on a form -->
```

```
<!ELEMENT queryfield - -   (%formelems;)>
<!ELEMENT control    - -   (%formelems;)>
<!ELEMENT formname   - -   (#PCDATA)>

<!-- Elements belonging to %formelems that are used to describe -->
<!-- standard Web and windowing system form elements including  -->
<!-- field name, size, default value, and required status       -->
<!-- also content modifiers such as search attribute name,      -->
<!-- boolean operators, proximity and stemming operators        -->
<!-- Maximum query/field content size -->
<!ELEMENT size          - -   (#PCDATA)>
<!-- default value, other allowed values -->
<!ELEMENT default    - -   (#PCDATA)>
<!ELEMENT option     - -   (#PCDATA)>
<!-- is field required -->
<!ELEMENT mandatory  - -   EMPTY>

<!-- Element for describing database reference(s) -->
<!ELEMENT database   - -   (%dbelems;)>

<!-- Element for recording language information-->
<!ELEMENT language      - -   (#PCDATA)>
<!ATTLIST language TYPE (Z3953 | ISO6391988) #REQUIRED>

<!-- Response elements -->
<!ELEMENT response      - -   EMPTY>
<!ATTLIST response TYPE (html | vrml | xml | text) #REQUIRED>
```