# The Lucene Search Engine

## Adding search to your applications

*by Thomas Paul*

The Lucene search engine is an open source, Jakarta project used to build and search indexes. Lucene can index any text-based information you like and then find it later based on various search criteria. Although Lucene only works with text, there are other add-ons to Lucene that allow you to index Word documents, PDF files, XML, or HTML pages. Lucene has a very flexible and powerful search capability that uses fuzzy logic to locate indexed items. Lucene is not overly complex. It provides a basic framework that you can use to build full-featured search into your web sites.

The easiest way to learn Lucene is to look at an example of using it. Let's pretend that we are writing an application for our university's Physics department. The professors have been writing articles and storing them online and we would like to make the articles searchable. (To make the example simple, we will assume that the articles are stored in text format.) Although we could use google, we would like to make the articles searchable by various criteria such as who wrote the article, what branch of physics the article deals with, etc. Google could index the articles but we wouldn't be able to show results based on questions such as, "show me all the articles by Professor Henry that deal with relativity and have superstring in their title."

## What's inside?

Let's take a look at the key classes that we will use to build a search engine.

* ***Document*** - The *Document* class represents a document in Lucene. We index *Document* objects and get *Document* objects back when we do a search.
* ***Field*** - The *Field* class represents a section of a *Document*. The *Field* object will contain a name for the section and the actual data.
* ***Analyzer*** - The *Analyzer* class is an abstract class that used to provide an interface that will take a *Document* and turn it into tokens that can be indexed. There are several useful implementations of this class but the most commonly used is the *StandardAnalyzer* class.
* ***IndexWriter*** - The *IndexWriter* class is used to create and maintain indexes.
* ***IndexSearcher*** - The *IndexSearcher* class is used to search through an index.
* ***QueryParser*** - The *QueryParser* class is used to build a parser that can search through an index.
* ***Query*** - The *Query* class is an abstract class that contains the search criteria created by the *QueryParser*.
* ***Hits*** - The *Hits* class contains the *Document* objects that are returned by running the *Query* object against the index.

## Indexing a Document

The first step is to install Lucene. This is extremely simple. Download the zip or tar file from the [Jakarta binaries download page](http://jakarta.apache.org/site/binindex.cgi) and extract the **lucene-1.3- final.jar**. Place this file in your classpath or in the lib directory of your web application. Lucene is now installed.

We will assume that you have written a program that the professors can use to upload their articles. The program might include a place for them to enter their name, a title for the article, and select from a list of categories that describe the article. We will also assume that this program stores the article in a place that is accessible from the web. To index this article we will need the article itself, the name of the author, the date it was written, the topic of the article, the title of the article, and the URL where the file is located. With that information we can build a program that can properly index the article to make it easy to find.

Let's look at the basic framework of our class including all the imports we will need.

**Skeleton class including imports**

import org.apache.lucene.document.Document;

import org.apache.lucene.document.Field;

import org.apache.lucene.analysis.Analyzer;

import org.apache.lucene.analysis.standard.StandardAnalyzer;

import org.apache.lucene.index.IndexWriter;

import org.apache.lucene.search.IndexSearcher;

import org.apache.lucene.search.Query;

import org.apache.lucene.queryParser.QueryParser;

import org.apache.lucene.search.Hits;

import java.util.Date;

public class ArticleIndexer {

}

The first thing we will need to add is a way to convert our article into a *Document* object.

**Method to create a *Document* from an article**

private Document createDocument(String article, String author,

String title, String topic,

String url, Date dateWritten) {

Document document = new Document();

document.add(Field.Text("author", author));

document.add(Field.Text("title", title));

document.add(Field.Text("topic", topic));

document.add(Field.UnIndexed("url", url));

document.add(Field.Keyword("date", dateWritten));

document.add(Field.UnStored("article", article));

return document;

}

First we create a new *Document* object. The next thing we need to do is add the different sections of the article to the *Document*. The names that we give to each section are completely arbitrary and work much like keys in a *HashMap*. The name used must be a *String*. The *add* method of *Document* will take a *Field* object which we build using one of the static methods provided in the *Field* class. There are four methods provided for adding *Field* objects to a *Document*.

* ***Field.Keyword*** - The data is stored and indexed but not tokenized. This is most useful for data that should be stored unchanged such as a date. In fact, the *Field.Keyword* can take a *Date* object as input.
* ***Field.Text*** - The data is stored, indexed, and tokenized. *Field.Text* fields should not be used for large amounts of data such as the article itself because the index will get very large since it will contain a full copy of the article plus the tokenized version.
* ***Field.UnStored*** - The data is not stored but it is indexed and tokenized. Large amounts of data such as the text of the article should be placed in the index unstored.
* ***Field.UnIndexed*** - The data is stored but not indexed or tokenized. This is used with data that you want returned with the results of a search but you won't actually be searching on this data. In our example, since we won't allow searching for the URL there is no reason to index it but we want it returned to us when a search result is found.

Now that we have a *Document* object, we need to get an *IndexWriter* to write this *Document* to the index.

**Method to store a *Document* in the index**

String indexDirectory = "lucene-index";

private void indexDocument(Document document) throws Exception {

Analyzer analyzer = new StandardAnalyzer();

IndexWriter writer = new IndexWriter(indexDirectory, analyzer, false);

writer.addDocument(document);

writer.optimize();

writer.close();

}

We first create a *StandardAnalyzer* and then create an *IndexWriter* using the analyzer. In the constructor we must specify the directory where the index will reside. The boolean at the end of the constructor tells the *IndexWriter* whether it should create a new index or add to an existing index. When adding a new document to an existing index we would specify false. We then add the *Document* to the index. Finally, we optimize and then close the index. If you are going to add multiple *Document* objects you should always optimize and then close the index after all the *Document* objects have been added to the index.

Now we just need to add a method to pull the pieces together.

**Method to drive the indexing**

public void indexArticle(String article, String author,

String title, String topic,

String url, Date dateWritten)

throws Exception {

Document document = createDocument(article, author,

title, topic,

url, dateWritten);

indexDocument(document);

}

Running this for an article will add that article to the index. Changing the boolean in the *IndexWriter* constructor to true will create an index so we should use that the first time we create an index and whenever we want to rebuild the index from scratch. Now that we have constructed an index, we need to search it for an article.

## Searching an Index

We have added our articles to the index and we want to search for them. Assuming we have written a nice front-end for our users, we just need to take the user's request and run it against our index. Since we have added several different types of fields, our users have multiple search options. As we will see, we can specify which field is the default to use for searching but our users can search on any of the fields that are in our index.

The code to do the search is presented here:

**Code to search an index - *searchCriteria* would be provided by the user**

IndexSearcher is = new IndexSearcher(indexDirectory);

Analyzer analyzer = new StandardAnalyzer();

QueryParser parser = new QueryParser("article", analyzer);

Query query = parser.parse(searchCriteria);

Hits hits = is.search(query);

Although there are a lot of classes involved here, the search is not overly complicated. The first thing we do is create an *IndexSearcher* object pointing to the directory where the articles have been indexed. We then create a *StandardAnalyzer* object. The *StandardAnalyzer* is passed to the constructor of a *QueryParser* along with the name of the default field to use for the search. This will be the field that is used if the user does not specify a field in their search criteria. We then parse the actual search criteria that was specified giving us a *Query* object. We can now run the *Query* against the *IndexSearcher* object. This returns a *Hits* object which is a collection of all the articles that met the specified criteria.

Extracting the *Document* objects from the *Hits* object is done by using the doc() method of the *Hits* object.

**Extracting *Document* objects**

for (int i=0; i<hits.length(); i++) {

Document doc = hits.doc(i);

// display the articles that were found to the user

}

is.close();

The *Document* class has a get() method that can be used to extract the information that was stored in the index. For example, to get the author from the *Document* we would code doc.get("author"). Since we added the article itself as *Field.UnStored*, attempting to get it will return null. However, since we added the URL of the article to the index, we can get the URL and display it to the user in our result list. We should always close the *IndexSearcher* after we have finished extracting all the *Document* objects. Attempting to extract a *Document* after closing will generate an error:

java.io.IOException: Bad file descriptor

## Specifying Search Criteria

Lucene supports a wide array of possible searches including AND OR and NOT, fuzzy searches, proximity searches, wildcard searches, and range searches. Let's take a look at a couple of examples:

**Find all of Professor Henry's articles that contain relativity and quantum physics:**

*author:Henry relativity AND "quantum physics"*

**Find all the articles that contain the phrase "string theory" and don't contain Einstein:**

*"string theory" NOT Einstein*

**Find all the articles that contain Kepler within five words of Galileo:**

*"Galileo Kepler"~5*

**Find all the articles that Professor Johnson wrote in January of this year:**

*author:Johnson date:[01/01/2004 TO 01/31/2004]*

If we don't specify a field, then the default is to use the field specified in the constructor of the *QueryParser*. In our example, that would be the article field. You can search on any field in the *Document* unless it was added as *Field.UnIndexed*. Another example of a field that you might wish to store but not index might be a short summary of the article that you wish to display to the user along with the other results.

## Conclusion

Lucene is a highly sophisticated and yet simple to use search engine. It does not automatically search your documents but it provides a framework for writing your own search. Using Lucene you could easily build a web spider for any web site. Although Lucene only supports simple text, there are Java classes that are available that can convert HTML, XML, Word documents, and PDF files into simple text. Many of these classes are available from the Lucene web site. Like many of the Jakarta projects, the documentation for Lucene is not very good, but with a little trial and error you should be able to get Lucene working.

The Lucene web site: <http://jakarta.apache.org/lucene>