http://webdam.inria.fr

Web Data Management

Text indexing with LUCENE (Nicolas Travers)

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

`http://webdam.inria.fr/Jorge/`

Contents

LUCENE[1] is an open-source tunable indexing platform often used for full-text indexing of Web sites. It implements an inverted index, creating posting lists for each term of the vocabulary. This chapter proposes some exercises to discover the LUCENE platform and test its functionalities through its Java API.

1 Preliminary: a LUCENE sandbox

We provide a simple graphical interface that lets you capture a collection of Web documents (from a given Website), index it, and search for documents matching a keyword query. The tool is implemented with LUCENE (surprise!) and helps to assess the impact of the search parameters, including ranking factors.

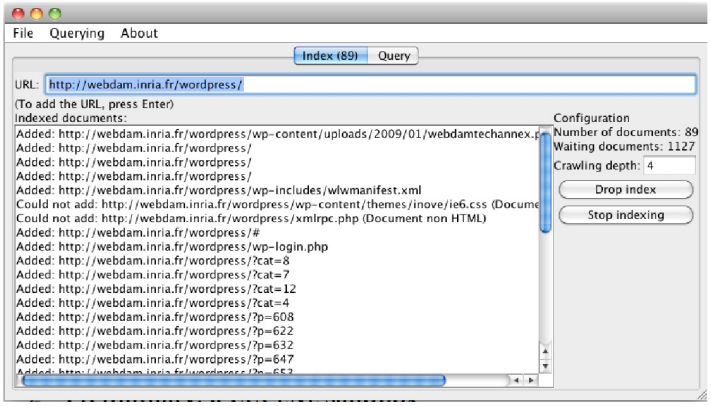

Figure 1: The LUCENE sandbox of WDM

[1] `http://lucene.apache.org/java/docs/`

You can download the program from our Web site. It consists of a Java archive that can be executed right (providing a decent Java installation on your computer). Figure 1 shows a screenshot of the main page. It allows you to

1. Download a set of documents collected from a given URL (including local addresses).
2. Index and query those documents.
3. Consult the information used by LUCENE to present ranked results.

Use this tool as a preliminary contact with full text search and information retrieval. The projects proposed at the end of the chapter give some suggestions to realize a similar application.

2 Indexing plain-text with LUCENE – A full example

We embark now in a practical experimentation with LUCENE. First, download the Java packages from the Web site `http://lucene.apache.org/java/docs/`. The examples and exercises that follow have been tested with version 3.0.2, so check whether something changed if you use another version.

You will find several packages.

- *lucene-core-3.0.2.jar*: the main package; put it right away in your `CLASSPATH` or in your project environment if you use an IDL;
- *lucene-demos-3.0.2.jar*: a set of demonstration programs;
- *luceneWeb.war*: a simple LUCENE Web application installer based on a demonstrator;
- *contrib*, a set of packages that complement the core functions.
  - *analyzers*: main languages analyzers (*lucene-analyzers-3.0.2.jar*); mandatory with most LUCENE indexing and querying applications;
  - *collation*: change LUCENE analyzer to optimize ranking and range queries;
  - *db*: the berkeleyDB database management system
  - *instantiated*: RAM-based LUCENE indexing;
  - *queryparser*: tuning of the query parser
  - *snowball*: stemming package that extract stems over terms within a given language;
  - *spellchecker*: words spell checking; suggests replacement by the nearest valid word;
  - *spatial*: sort LUCENE results with distance based scoring;
  - *wordnet*: the Wordnet API is integrated to LUCENE in order to check words and synonyms in the dictionnary.

The packages *lucene-core-3.0.2.jar* (for indexing and querying) and *lucene-analyzers-3.0.2.jar* (for text analyzer features) are required for the examples and exercises below. We will use them in the following to create our LUCENE application.

2.1 The main program

We will detail in the next pages a simple example that creates a LUCENE index, adds a few documents, and executes some searches. The main java program follows this open-create-query structure:

```
public class Simple {
  String directory = "index";

  public static void main(String[] args) {
    // Name of the directory that holds the index
    String directory = "index";

    // Instantiate a new Lucene tool
    MyLucene lucene = new MyLucene();

    // Open the directory
    lucene.openIndex(directory, true);

    // Add a few documents
    lucene.addDoc("Web Data Management");
    lucene.addDoc("Data on the Web");
    lucene.addDoc("Spatial Databases -- with Application to GIS");

    // Close the index
    lucene.closeIndex();

    // Now, search for some term
    String query[] = {"Web"};
    lucene.search(query);
  }
}
```

Everything is handled by the *MyLucene* class, which is now detailed (the full code can be found on the book's Web site). Note that the program is for illustration purposes, and thus makes a poor job at catching exceptions.

2.2 Create the Index

The *openIndex* method creates an index, or opens an existing index.

```
  public void openIndex(String directory, boolean newIndex) {
    try {
      // Link the directory on the FileSystem to the application
      index = FSDirectory.open(new File(directory));

      // Check whether the index has already been locked
      // (or not properly closed).
      if (IndexWriter.isLocked(index))
        IndexWriter.unlock(index);
      if (writer == null)
```

```
        // Link the repository to the IndexWriter
        writer = new IndexWriter(index, analyzer, newIndex,
            IndexWriter.MaxFieldLength.LIMITED);
    } catch (Exception e) {
      System.out.println("Got an Exception: " + e.getMessage());
    }
  }
```

The LUCENE index repository must be opened prior to any access. When the *newIndex* Boolean value is *true*, LUCENE creates the repository name *directory*. When *newIndex* is *false*, the previously created index is reused. Only one *IndexWriter* at a time can be access the repository. Be careful, each time *newIndex* is set to `true`, the index will be entirely replaced.

The *StandardAnalyzer* object is the document analyzer process (instantiated in the constructor). It takes into account the specifics of the input language. For consistency reasons, the same analyzer must be used for the creation and for searches.

Once an index is generated, you can look at the repository generated by LUCENE, which contains several files:

- *segments_X / segments.gen*. The segment index files.
- *write.lock*. Lock file, modified each time an *IndexWriter* instance works on the repository.
- *_X.cfs*. Compound files. Describe all indexed files.
- *_X.cfx*. Compound files for storing field values and term vectors.

The index must be closed when inserts and updates are finished.

```
public void closeIndex() {
    try {
      writer.optimize();
      writer.close();
    } catch (Exception e) {
      System.out.println("Got an Exception: " + e.getMessage());
    }
  }
```

Each call to *optimize()* applies a compression and store the modified values in the repository.

2.3 Adding documents

Once the index is created, we can populate it with documents. LUCENE defines an abstraction of documents as instances of the *Document* class. Such an instance contains several *Field*s that define which information will be stored, indexed and queried. The following example defines a single field, named *content*. You will be invited to create multi-fields documents in the labs.

```
public void addDoc(String value) {
    try {
      // Instantiate a new document
```

```
      Document doc = new Document();
      // Put the value in a field name content
      Field f = new Field("content", value, Field.Store.YES,
          Field.Index.ANALYZED);
      // Add the field to the document
      doc.add(f);
      // And add the document to the index
      writer.addDocument(doc);
    } catch (Exception e) {
      System.out.println("Got an Exception: " + e.getMessage());
    }
  }
```

Modeling a document as a list of fields is tantamount to defining how the information is analyzed, indexed, and stored.

2.4 Searching the index

We can instantiate the *IndexSearcher* class, giving as a parameter the index repository name. We also provide to the constructor an *Analyzer* object, which must be of the same type as the one used during the indexing process. The *QueryParser* instance applies the analyzer to the the query string, ensuring that the tokenization and other transformations applied to terms is consistent. We must also specify which fields will be queried by default for each query.

```
  public void search(String[] args) {
    // Nothing given? Search for "Web".
    String querystr = args.length > 0 ? args[0] : "Web";

    try {
      // Instantiate a query parser
      QueryParser parser = new QueryParser(Version.LUCENE_30, "content",
          analyzer);
      // Parse
      Query q = parser.parse(querystr);
      // We look for top-10 results
      int hitsPerPage = 10;
      // Instantiate a searcher
      IndexSearcher searcher = new IndexSearcher(index, true);
      // Ranker
      TopScoreDocCollector collector = TopScoreDocCollector.create(
          hitsPerPage, true);
      // Search!
      searcher.search(q, collector);
      // Retrieve the top-10 documents
      ScoreDoc[] hits = collector.topDocs().scoreDocs;

      // Display results
      System.out.println("Found " + hits.length + " hits.");
      for (int i = 0; i < hits.length; ++i) {
        int docId = hits[i].doc;
        Document d = searcher.doc(docId);
```

```
        System.out.println((i + 1) + ". " + d.get("content"));
      }

      // Close the searcher
      searcher.close();
    } catch (Exception e) {
      System.out.println("Got an Exception: " + e.getMessage());
    }
  }
```

2.5 LUCENE querying syntax

The LUCENE querying syntax is almost simple. A query is composed of a set of words, each of which check in the index the posting lists and provide a tf/idf value. The global rank of the query is the sum of these values, as we saw previously. Here is a sketch of query:

```
Web Data Management
```

This query is composed of three words that will be searched into the index. The score of each document depends of the sum of scores for "Web", "Data" and "Management".

In order to complexify queries, LUCENE provides some more features that helps to creates richer queries. All these features follow a specific syntax. For each feature, we will illustrate it by modifying the previous query:

- **Negation:** *NOT xxx*
  The given word must not be present into the document. All documents containing this words will be removed from the result set (*i.e. the score is equal to zero*).
  `Web Data NOT Management` - The document will contain "Web" and "Data" but never "Management"

- **Mandatory keywords:** *+xxx*
  The given word must be present in the document. In fact, although a word is asked in the query and its score in a document is equal to zero, this document could appear (other words bring a good score). This feature forbids documents that do not mention this word.
  `+Web Data Management` - The word *Web* must be contained in the document, *Data* and *Management* may be present

- **Exact matching:** *"xxx yyy"*
  Bringing a sentence into quotes makes an exact matching query, for which the document must contains this sentence:
  `"Web Data" Management` - The document must contain the sentence "Web Data", the word "Management" brings an additional score to the document.

- **Word importance:** *xxx^X*
  A word may be more important than others in a query, for this you can increase the scoring weight of this word in the document. This weight is applied on the score before making the sum of all scores.
  `Web^3 Data Management` - The resulting score of the word "Web" in the documents is three times bigger than "Data" and "Management".

- **Wildcard search:** *xx**
  LUCENE will search for words that matches with given letters, completing the wildcard with existing words in the index. A word must not begin with a wildcard.
  `Web Data Manag*` - All documents that contain a words beginning with "Manag" will be returned (like *Management*, *Manage*, *Managing*, *Manager*...)

- **Querying fields:** *FFF:xxx*
  As we saw during indexing, we can specify fields (title, content, path). By default, we specified that the "content" field is used for queries. By specifying a field, LUCENE searches the given word into it.
  `title:Web Data Management` - The word "Web" will be searched into the *field* "title", while "Data" and "Management" will be searched into the default field.

- **Similarity search:** *xxx ~*
  A similarity search corrects misspelling of a given word, LUCENE will search this words with different spelling. The tilda can optionally be followed by a numeric value, this value gives the distance of similarity between the given word and the proposed one.
  `Web Data ~ Management` - Documents will contain "Web" and "Management", but also words similar to "data" (like *date*, *tata*, *sata*)

- **Range queries:** *xxx TO yyy*
  A query can ask for a range of values for corresponding documents. This range can be numeric values, dates, or words (with lexicographic orders).
  tit title:"Web Data Management" title:{1 TO 3} - All documents must have the exact sentence "Web Data Management" and also a numeric (book's version) value between 1 and 3.

3 Put it into practice!

You should first make our simple example run and examine carefully all the methods. Note that the following improvements would be useful:

1. handle more carefully the exceptions raised by LUCENE classes;
2. instead of printing the result right away in the *search* method, implement an iterator-like mechanism that allows to retrieve the documents in the result one by one;
3. add some optional, yet useful features, such as for instance a management of a list of stop words (this is a first opportunity to look at the LUCENE API).

Once these changes are effective, you should then be ready to implement your first search engine.

3.1 Indexing a directory content

A sample of several files with two "fields", respectively "title" and "content", can be found on the Website (*lucene* directory). Download them in a *files* directory. Next, create a parsing function that takes as input a file path, open this file, and extracts title, content according to the following pattern:

```
title:XXXX
content:YYYY
```

Create a directory extractor to get all the files from the *files* directory and index them with LUCENE (do not forget to call the *closeIndex()* function).

Your index is created. Now, implement a function that considers a list of words given on the standard input stream for querying the index. Here are some possible searches:

- *Information Research*
- *Research NOT Lucene*
- *+Research Information*
- *"Information Research"*
- Research^3 Information
- *Research Info**
- *title:Research*
- *Research Information~2*
- *title:{Research TO Information}*

3.2 Web site indexing (project)

To index the content of a whole Web site, create a class that "crawls" the document belonging to a given domain (e.g., `http://Webdam.inria.fr/`). Index the documents content, extract titles from the appropriate HTML tags (`<title>` or `h1` – *hint*: use *java.util.regex.Matcher*). Once a document is loaded, find the embedded links in order to recursively process the whole Web site (*hint*: look for `href` attributes).

Be careful to not index a document twice, and do *not* process external links (not belonging to the given domain), nor images or generally non-html documents.

4 LUCENE – Tuning the scoring (project)

As previously discussed, LUCENE computes a score values for each document with respect to the query terms. This score is based on the tf-idf measures. Here is the detailed scoring function used in LUCENE:

$$score(q,d) = \sum [tf(t_d) \times idf(t) \times boost(t.field_d) \times lengthNorm(t.field_d)] \times coord(q,d) \times qNorm(q)$$

where q is the query, d a document, t a term, and:

1. $\underline{tf}$ is a function of the term frequency within the document (default: $\sqrt{freq}$);
2. $\underline{idf}$: Inverse document frequency of t within the whole collection (default: $\log(\frac{numDocs}{docFreq+1}) + \overline{1}$);

3. *boost* is the boosting factor, if required in the query with the "^ " operator on a given field (if not specified, set to the default field);
4. *lengthNorm*: field normalization according to the number of terms. Default: $\frac{1}{\sqrt{nbTerms}}$
5. *coord*: overlapping rate of terms of the query in the given document. Default: $\frac{overlap}{maxOverlap}$
6. *qNorm*: query normalization according to its length; it corresponds to the sum of square values of terms' weight, the global value is multiplied by each term's weight.

Only underlined functions can be modified in LUCENE: *tf*, *idf*, *lengthNom* and *coord*. Default functions are given and can be modified by creating a new *Similarity* class with overloaded methods. Specifically:

1. Create a new class that inherits the *org.apache.lucene.search.DefaultSimilarity* class;
2. Overload and implement default similarity functions:
   - **public float** `tf(`**float** `freq);`
   - **public float** `idf(`**int** `docFreq,` **int** `numDocs);`
   - **public float** `lengthNorm(String fieldName,` **int** `numTerms);`
   - **public float** `coord(`**int** `overlap,` **int** `maxOverlap);`
3. Add some parameters to allow changing the similarity functions as follows:
   - tf : $\sqrt{freq}$, 1, $freq$, $\sqrt{(1-freq)}$;
   - idf : $\log(\frac{numDocs}{docFreq+1}) + 1$, 1, $\frac{numDocs}{docFreq+1}$, $\log(1 - \frac{numDocs}{docFreq+1}) + 1$;
   - lengthNorm : $\frac{1}{\sqrt{numTerms}}$, 1, $1 - \frac{1}{\sqrt{numTerms}}$;
   - coord : $\frac{overlap}{maxOverlap}$, 1, $1 - \frac{overlap}{maxOverlap}$;
4. Change the similarity function in the querying class previously created with: *searcher.setSimilarity(similarity);*
5. Compute previous queries with different combinations of similarity functions.