Introduction to
Information Retrieval

Lucene: Τι είναι;

- *Open source Java library* for indexing and searching
- Lets you add search to your application
- Not a complete search system by itself
- Written by Doug Cutting
- Used by LinkedIn, Twitter, ...
- ...and many more (see [http://wiki.apache.org/lucene-java/PoweredBy](http://wiki.apache.org/lucene-java/PoweredBy))
- Ports/integrations to other languages
  - C/C++, C#, Ruby, Perl, Python, PHP, ...
Πηγές

  - Code samples available for download
  - Java build system used by “Lucene in Action” code

---

Lucene in a search system

- Acquire content
- Analyze document
- Build document
- Index document
- Index
- Users
- Search UI
- Build query
- Render results
- Run query
Lucene in action

- Command line **Indexer**
  - ../lia2e/src/lia/meetlucene/Indexer.java

- Command line **Searcher**
  - ../lia2e3/src/lia/meetlucene/Searcher.java

How Lucene models content

- A **Document** is the atomic unit of indexing and searching
  - A Document contains **Fields**
- **Fields** have a **name** and a **value**
  - Examples: Title, author, date, abstract, body, URL, keywords, ..
  - Different documents can have different fields

- You have to translate raw content into **Fields**
- Search a field using name:term, e.g., title:lucene
Introduction to Information Retrieval

Parametric and field indexes

- Documents often contain **metadata**: specific forms of data about a document, such as its author(s), title and date of publication.

- Metadata generally include **fields** such as the date of creation, format of the document, the author, title of the document, etc.

- There is one **parametric index** for each field (e.g., one for title, one for date, etc).

Example query: “find documents authored by William Shakespeare in 1601, containing the phrase ‘alas poor Yorick’.”

- Usual postings intersections, except that we may merge postings from standard inverted as well as parametric indexes.

- For ordered values (e.g., year) may support querying ranges -> use a structure like a B-tree for the dictionary of such fields.
Zone indexes

- Zones similar to fields, except the contents of a zone can be arbitrary free text.
  - example, document titles and abstracts

- We may build a separate inverted index for each zone of a document, to support queries such as “find documents with merchant in the title and william in the author list and the phrase gentle rain in the body”.

- Whereas, the dictionary for a parametric index comes from a fixed vocabulary, the dictionary for a zone index whatever vocabulary stems from the text of that zone.

We can reduce the size of the dictionary by encoding the zone in which a term occurs in the postings.

Also, supports weighted zone scoring.
Lucene in a search system

Fields

Fields may

- Be indexed or not
  - Indexed fields may or may not be analyzed (i.e., tokenized with an Analyzer)
    - Non-analyzed fields view the entire value as a single token (useful for URLs, paths, dates, social security numbers, ...)
  - Be stored or not
    - Useful for fields that you’d like to display to users
- Optionally store term vectors
  - Like a positional index on the Field’s terms
  - Useful for highlighting, finding similar documents, categorization
Field construction
Lots of different constructors

```
import org.apache.lucene.document.Field

Field(String name,
     String value,
     Field.Store store, // store or not
     Field.Index index, // index or not
     Field.TermVector termVector);
```

can also be specified with a Reader, a TokenStream, or a byte[]

Field options

- Field.Store
  - NO: Don’t store the field value in the index
  - YES: Store the field value in the index
- Field.Index
  - ANALYZED: Tokenize with an Analyzer
  - NOT_ANALYZED: Do not tokenize
  - NO: Do not index this field
  - Couple of other advanced options
- Field.TermVector
  - NO: Don’t store term vectors
  - YES: Store term vectors
  - Several other options to store positions and offsets
Using Field options

<table>
<thead>
<tr>
<th>Index</th>
<th>Store</th>
<th>TermVector</th>
<th>Example usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT_ANALYZED</td>
<td>YES</td>
<td>NO</td>
<td>Identifiers, telephone/SSNs, URLs, dates, ...</td>
</tr>
<tr>
<td>ANALYZED</td>
<td>YES</td>
<td>WITH_POSITIONS_OFFSETS</td>
<td>Title, abstract</td>
</tr>
<tr>
<td>ANALYZED</td>
<td>NO</td>
<td>WITH_POSITIONS_OFFSETS</td>
<td>Body</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>Document type, DB keys (if not used for searching)</td>
</tr>
<tr>
<td>NOT_ANALYZED</td>
<td>NO</td>
<td>NO</td>
<td>Hidden keywords</td>
</tr>
</tbody>
</table>

Document

import org.apache.lucene.document.Field

- Constructor:
  - Document();

- Methods
  - void add(Fieldable field); // Field implements // Fieldable
  - String get(String name); // Returns value of // Field with given // name
  - Fieldable getFieldable(String name);
  - ... and many more
Multi-valued fields

- You can add multiple Fields with the same name
  - Lucene simply concatenates the different values for that named Field

```java
Document doc = new Document();
doc.add(new Field("author",
  "chris manning",
  Field.Store.YES,
  Field.Index.ANALYZED));
doc.add(new Field("author",
  "prabhakar raghavan",
  Field.Store.YES,
  Field.Index.ANALYZED));
```

...
Basic Application

Document
super_name: Spider-Man
name: Peter Parker
category: superhero
powers: agility, spider-sense

IndexWriter
addDocument()

IndexSearcher
search()

Lucene Index

Hits
(Matching Docs)

1. Get Lucene jar file
2. Write indexing code to get data and create Document objects
3. Write code to create query objects
4. Write code to use/display results

Basic Application: notes

Only a single IndexWriter may be open on an index
An IndexWriter is thread-safe, so multiple threads can add documents at the same time.

Multiple IndexSearchers may be opened on an index
• IndexSearchers are also thread safe, and can handle multiple searches concurrently
• an IndexSearcher instance has a static view of the index, it sees no updates after it has been opened

An index may be concurrently added to and searched, but new additions won’t show up until the IndexWriter is closed and a new IndexSearcher is opened.
Analyzers

Tokenizes the input text

- Common Analyzers
  - WhitespaceAnalyzer
    *Splits tokens on whitespace*
  - SimpleAnalyzer
    *Splits tokens on non-letters, and then lowers cases*
  - StopAnalyzer
    *Same as SimpleAnalyzer, but also removes stop words*
  - StandardAnalyzer
    *Most sophisticated analyzer that knows about certain token types, lowers cases, removes stop words, ...*

Analysis examples

“The quick brown fox jumped over the lazy dog”

- WhitespaceAnalyzer
  - [The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- SimpleAnalyzer
  - [the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- StopAnalyzer
  - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]
- StandardAnalyzer
  - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]
More analysis examples

- “XY&Z Corporation – xyz@example.com”
- WhitespaceAnalyzer
  - \[XY&Z\] [Corporation] [-] [xyz@example.com]
- SimpleAnalyzer
  - \[xy\] [z] [corporation] [xyz] [example] [com]
- StopAnalyzer
  - \[xy\] \[z\] [corporation] [xyz] [example] [com]
- StandardAnalyzer
  - \[xy&z\] [corporation] [xyz@example.com]

What’s inside an Analyzer?

- Analyzers need to return a TokenStream
  - public TokenStream tokenStream(String fieldName,
  Reader reader)

```
public class TokenStream {
  public TokenStream tokenize(String fieldName, Reader reader) {
    // Tokenization logic...
  }
}
```
Tokenizers and TokenFilters

- **Tokenizer**
  - WhitespaceTokenizer
  - KeywordTokenizer
  - LetterTokenizer
  - StandardTokenizer
  - ...

- **TokenFilter**
  - LowerCaseFilter
  - StopFilter
  - PorterStemFilter
  - ASCII FoldingFilter
  - StandardFilter
  - ...

IndexWriter construction

// Deprecated
IndexWriter(Directory d,
            Analyzer a, // default analyzer
            IndexWriter.MaxFieldLength mfl);

// Preferred
IndexWriter(Directory d,
            IndexWriterConfig c);
Creating an IndexWriter

import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
...
private IndexWriter writer;
...
public Indexer(String indexDir) throws IOException {
    Directory dir = FSDirectory.open(new File(indexDir));
    writer = new IndexWriter(
        dir,
        new StandardAnalyzer(Version.LUCENE_30),
        true,
        IndexWriter.MaxFieldLength.UNLIMITED);
}

Core indexing classes

- **Document**
  - Represents a collection of named Fields.
  - Text in these Fields are indexed.

- **Field**
  - Note: Lucene Fields can represent both “fields” and “zones” as described in the textbook
A Document contains Fields

```java
import org.apache.lucene.document.Field;

protected Document getDocument(File f) throws Exception {
    Document doc = new Document();
    doc.add(new Field("contents", new FileReader(f)));
    doc.add(new Field("filename", f.getName(), Field.Store.YES, Field.Index.NOT_ANALYZED));
    doc.add(new Field("fullpath", f.getCanonicalPath(), Field.Store.YES, Field.Index.NOT_ANALYZED));
    return doc;
}
```

Index a Document with IndexWriter

```java
private IndexWriter writer;

private void indexFile(File f) throws Exception {
    Document doc = getDocument(f);
    writer.addDocument(doc);
}
```
Indexing a directory

```java
private IndexWriter writer;
...
public int index(String dataDir,
    FileFilter filter)
    throws Exception {
    File[] files = new File(dataDir).listFiles();
    for (File f: files) {
        if (... &&
            (filter == null || filter.accept(f))) {
            indexFile(f);
        }
    }
    return writer.numDocs();
}
```

Closing the IndexWriter

```java
private IndexWriter writer;
...
public void close() throws IOException {
    writer.close();
}
```
Adding/deleting Documents to/from an IndexWriter

void addDocument(Document d);
void addDocument(Document d, Analyzer a);

Important: Need to ensure that Analyzers used at indexing time are consistent with Analyzers used at searching time

// deletes docs containing term or matching // query. The term version is useful for // deleting one document.
void deleteDocuments(Term term);
void deleteDocuments(Query query);

Index format

- Each Lucene index consists of one or more segments
  - A segment is a standalone index for a subset of documents
  - All segments are searched
  - A segment is created whenever IndexWriter flushes adds/deletes
- Periodically, IndexWriter will merge a set of segments into a single segment
  - Policy specified by a MergePolicy
- You can explicitly invoke optimize() to merge segments
Basic merge policy

- Segments are grouped into levels
- Segments within a group are roughly equal size (in log space)
- Once a level has enough segments, they are merged into a segment at the next level up

Core searching classes

- **IndexSearcher**
  - Central class that exposes several search methods on an index
- **Query**
  - Abstract query class. Concrete subclasses represent specific types of queries, e.g., matching terms in fields, boolean queries, phrase queries, ...
- **QueryParser**
  - Parses a textual representation of a query into a Query instance
Creating an IndexSearcher

import org.apache.lucene.search.IndexSearcher;
...
public static void search(String indexDir,
   String q)
   throws IOException, ParseException {
    Directory dir = FSDirectory.open(
    new File(indexDir));
    IndexSearcher is = new IndexSearcher(dir);
    ...
}

Query and QueryParser

import org.apache.lucene.search.Query;
import org.apache.lucene.queryParser.QueryParser;
...
public static void search(String indexDir, String q)
   throws IOException, ParseException
   ...
   QueryParser parser =
   new QueryParser(Version.LUCENE_30,
   "contents",
   new StandardAnalyzer(
    Version.LUCENE_30));

   Query query = parser.parse(q);
   ...
}
Core searching classes (contd.)

- TopDocs
  - Contains references to the top documents returned by a search

- ScoreDoc
  - Represents a single search result

search() returns TopDocs

```java
import org.apache.lucene.search.TopDocs;
...
public static void search(String indexDir,
    String q)
    throws IOException, ParseException
...
    IndexSearcher is = ...;
    ...
    Query query = ...;
    ...
    TopDocs hits = is.search(query, 10);
}
```
TopDocs contain ScoreDocs

```java
import org.apache.lucene.search.ScoreDoc;
...
public static void search(String indexDir, String q)
    throws IOException, ParseException
    ...
    IndexSearcher is = ...;
    ...
    TopDocs hits = ...;
    ...
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = is.doc(scoreDoc.doc);
        System.out.println(doc.get("fullpath"));
    }
}
```

Closing IndexSearcher

```java
public static void search(String indexDir, String q)
    throws IOException, ParseException
    ...
    IndexSearcher is = ...;
    ...
    is.close();
}
```
IndexSearcher

- Constructor:
  - IndexSearcher(Directory d);
    - deprecated

IndexReader

```
Query  -->  IndexSearcher  -->  TopDocs
         \   /       \       /
          V        \IndexReader
              \    /     \       /
               V     \     \Directory
                  \   /       /
                   \V    /
                    \  /
                     \/
                      |
                       |
                      CSV
```
IndexSearcher

- Constructor:
  - IndexSearcher(Directory d);
    - deprecated
  - IndexSearcher(IndexReader r);
    - Construct an IndexReader with static method
      IndexReader.open(dir)

Searching a changing index

Directory dir = FSDirectory.open(...);
IndexReader reader = IndexReader.open(dir);
IndexSearcher searcher = new IndexSearcher(reader);

Above reader does not reflect changes to the index unless you reopen it. Reopening is more resource efficient than opening a new IndexReader.

IndexReader newReader = reader.reopen();
If (reader != newReader) {
    reader.close();
    reader = newReader;
    searcher = new IndexSearcher(reader);
}
Near-real-time search

IndexWriter writer = ...;
IndexReader reader = writer.getReader();
IndexSearcher searcher = new IndexSearcher(reader);

Now let us say there’s a change to the index using writer

// reopen() and getReader() force writer to flush
IndexReader newReader = reader.reopen();
if (reader != newReader) {
    reader.close();
    reader = newReader;
    searcher = new IndexSearcher(reader);
}

IndexSearcher

- Methods
  - TopDocs search(Query q, int n);
  - Document doc(int docID);
QueryParser

- Constructor
  - QueryParser(Version matchVersion, String defaultField, Analyzer analyzer);

- Parsing methods
  - Query parse(String query) throws ParseException;
  - ... and many more

QueryParser syntax examples

<table>
<thead>
<tr>
<th>Query expression</th>
<th>Document matches if...</th>
</tr>
</thead>
<tbody>
<tr>
<td>java</td>
<td>Contains the term java in the default field</td>
</tr>
<tr>
<td>java junit</td>
<td>Contains the term java or junit or both in the default field (the default operator can be changed to AND)</td>
</tr>
<tr>
<td>java OR junit</td>
<td>Contains both java and junit in the default field</td>
</tr>
<tr>
<td>+java +junit</td>
<td>Contains both java and junit in the default field</td>
</tr>
<tr>
<td>java AND junit</td>
<td>Contains both java and junit in the default field</td>
</tr>
<tr>
<td>title:ant</td>
<td>Contains the term ant in the title field</td>
</tr>
<tr>
<td>title:extreme ~subject:sports</td>
<td>Contains extreme in the title and not sports in subject</td>
</tr>
<tr>
<td>(agile OR extreme) AND java</td>
<td>Boolean expression matches</td>
</tr>
<tr>
<td>title:&quot;junit in action&quot;</td>
<td>Phrase matches in title</td>
</tr>
<tr>
<td>title:&quot;junit action&quot;~5</td>
<td>Proximity matches (within 5) in title</td>
</tr>
<tr>
<td>java*</td>
<td>Wildcard matches</td>
</tr>
<tr>
<td>java~</td>
<td>Fuzzy matches</td>
</tr>
<tr>
<td>lastmodified:[1/1/09 TO 12/31/09]</td>
<td>Range matches</td>
</tr>
</tbody>
</table>
Construct Queries programmatically

- TermQuery
  - Constructed from a Term
- TermRangeQuery
- NumericRangeQuery
- PrefixQuery
- BooleanQuery
- PhraseQuery
- WildcardQuery
- FuzzyQuery
- MatchAllDocsQuery

### Lucene Query Parser

Example: `queryParser.parse("name:Spider-Man");`
- good human entered queries, debugging, IPC
- does text analysis and constructs appropriate queries
- not all query types supported

### Programmatic query construction

Example: `new TermQuery(new Term("name","Spider-Man"))`
- explicit, no escaping necessary
- does not do text analysis for you

Analysis & Search Relevancy

#### Document Indexing Analysis

- LexCorp BFG-9000
  - WhitespaceTokenizer
  - LexCorp BFG-9000
    - WordDelimiterFilter catenateWords=1
      - Lex Corp BFG 9000
        - LowercaseFilter
          - lex corp bfg 9000

#### Query Analysis

- Lex corp bfg9000
  - WhitespaceTokenizer
  - Lex corp bfg9000
    - WordDelimiterFilter catenateWords=0
      - Lex corp bfg 9000
        - LowercaseFilter
          - lex corp bfg 9000

A Match!
TopDocs and ScoreDoc

- **TopDocs methods**
  - Number of documents that matched the search
  - `totalHits`
  - Array of `ScoreDoc` instances containing results
  - `scoreDocs`
  - Returns best score of all matches
  - `getMaxScore()`

- **ScoreDoc methods**
  - Document id
  - `doc`
  - Document score
  - `score`

Scoring

- Scoring function uses basic tf-idf scoring with
  - Programmable boost values for certain fields in documents
  - Length normalization
  - Boosts for documents containing more of the query terms

- **IndexSearcher** provides an `explain()` method that explains the scoring of a document
Based on “Lucene in Action”

- By Michael McCandless, Erik Hatcher, Otis Gospodnetic

ΤΕΛΟΣ 12ου Μαθήματος

Ερωτήσεις?

Χειροτονίστηκαν:
- Pandu Nayak and Prabhakar Raghavan, CS276: Information Retrieval and Web Search (Stanford)