

EPL660: Information Retrieval and Search Engines – Lab 3



**University of Cyprus
Department of
Computer Science**

Παύλος Αντωνίου

Γραφείο: B109, ΘΕΕΕ01


Apache Lucene



- Extremely rich and powerful **full-text search library** written in Java
- Makes it easy to add search functionality to an application or website
- Requires 2 steps:
 - **creating a lucence index** on a given set of documents
 - **parsing the user query** and **looking up** the prebuilt **index** (instead of searching the text directly) to **answer the query**
- Content added to Lucene can be from **various sources and formats** like SQL/NoSQL dBs, filesystem (text/pdf/MS Office files), or websites (html files)

Apache Lucene Overview



- Scalable, High-Performance Indexing 
 - Over [500-600GB/hour on modern hardware](#)
 - Small RAM requirements
 - Incremental indexing as fast as batch indexing
 - Index size roughly 20-30% the size of text indexed
 - Cross-Platform Solution
 - Available as Open Source software under the [Apache License](#) which lets you use Lucene in both commercial and Open Source programs
 - 100%-pure Java
 - Implementations [in other programming languages available](#) that are index-compatible
-

Apache Lucene: heart of search engines



- Lucene is the information retrieval software library under the hood of top-ranked search engines
 - Elasticsearch search engine
 - Apache Solr search engine

include secondary database models

21 systems in ranking, September 2020

Rank			DBMS	Database Model	Score		
Sep 2020	Aug 2020	Sep 2019			Sep 2020	Aug 2020	Sep 2019
1.	1.	1.	Elasticsearch	Search engine, Multi-model	150.50	-1.82	+1.23
2.	2.	2.	Splunk	Search engine	87.90	-2.01	+0.89
3.	3.	3.	Solr	Search engine	51.62	-0.08	-7.35

Source: <https://db-engines.com/en/ranking/search+engine>

Apache Lucene Features



- Many powerful query types:
 - phrase queries,
 - wildcard queries,
 - proximity queries,
 - range queries and more
- Ranked searching -- best results returned first
- Pluggable ranking models, including the [vector space model](#) (documents and queries are represented as vectors, find similarity between documents/query) and [okapi BM25](#) (ranking function to rank matching docs according to their relevance to a given query)

Apache Lucene Features



- Fielded searching (e.g. title, author, contents)
- Sorting by any field
- Multiple-index searching with merged results
- Allows simultaneous update and searching
- Fast, memory-efficient and typo-tolerant suggesters (suggesting query terms or phrases based on incomplete user input)
- Flexible faceting (information classified into categories – users explore information by applying filters), highlighting, joins and result grouping

wagner ri|

wagner ring cycle

wagner ribeiro

wagner ring cycle 2018

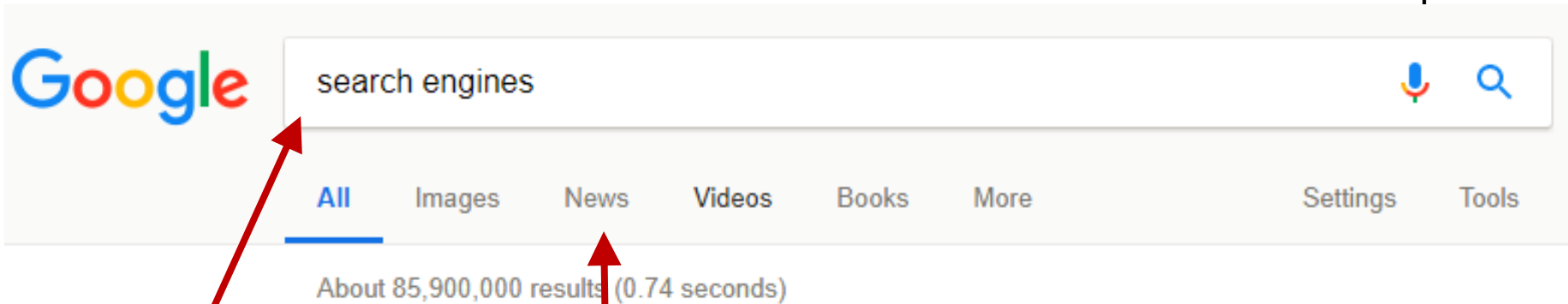
wagner ride of the valkyries

wagner ring cycle 2019

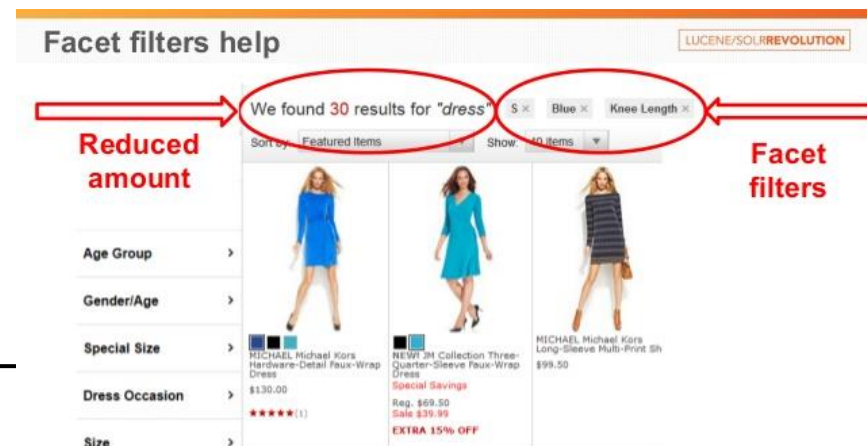
wagner rienzi

wagner richard

Faceted searching



After searching for a term, you can then see that result in the following categories: Images, News, Videos, Books, etc.



Reduced amount

Facet filters

Lucene Indexing



- First step when building a Lucene app is to create the index
 - Lucene index consists of Lucene document objects
- Document objects are stored in the index
 - programmer's job to "convert" data (files) into Document objects

1. Create a Lucene document

- Document consists of one or more fields
- Each field has name and value
- Fields are optionally stored in index, so that they may be returned with hits on the document after searching.
- `doc.add(new TextField(name, value, TextField.Store.YES))`

2. Add the document to the index (indexing)

- Using `IndexWriter` class
- `indexWriter.addDocument(doc)`

But how are the documents are processed prior indexing?

- Tokenization, stop words removal, stemming, etc.

Lucene Analyzer



- An analyzer **extracts index terms from text**
- Lucene comes with a default analyzer which works well for unstructured English text (language-specific analyzers exist)
 - `StandardAnalyzer` → tokenize, lowercase normalization, stop words removal
- Lucene has many built-in analyzers:
 - `SimpleAnalyzer`: A sophisticated general-purpose analyzer.
 - `WhitespaceAnalyzer`: A very simple analyzer that just separates tokens using white space.
 - `StopAnalyzer`: Removes common English words that are not usually useful for indexing.
 - `SnowballAnalyzer`: An interesting experimental analyzer that works on word roots (transforms a word into its root form: a search on *rain* should also return entries with *raining*, *rained*, and so on).
- Lucene also makes it easy to build custom Analyzers

Stemming! No Stemming!

Lucene Searching



- Once the index has been built you can create queries to retrieve documents from the index using the steps:
 1. Create the query
 - Using `QueryParser` which turns Google-like search expressions into Lucene's API representation of a Query
 - Or using the built-in Query subclasses of Lucene API
 2. Search the index
 - Using `IndexSearcher`
 - `searcher.search(query, collector)`
 - Collector object gathers raw results from a search, and implements sorting or custom result filtering
- **Sorting:** By default, Lucene will sort results by the relevance score of the field that is being searched. This behavior can be customized at query time.

Lucene Query Syntax



- Keyword matching
 - title: foo --- search for the word foo
 - title: "foo bar" --- search for the phrase foo bar
- Wildcard matching
 - title: "foo*bar"
 - title: "te?t"
- Proximity matching
 - "foo bar"~4
 - search for a "foo" and "bar" within 4 words of each other in a document
- Range search
 - date: [20110101 TO 20120101]
 - title: {Aida TO Carmen}
 - Inclusive range queries denoted by [] Exclusive range denoted by { }
- Boolean operators
 - "foo bar" AND "quick fox"
- Boosts (specify which terms/clauses are more important)
 - (title:foo OR title:bar)^1.5 (body:foo OR body:bar)

Prepare* for Lucene Applications



1. Download Apache Lucene 5.4.0 binaries (.zip for Windows, .tgz for Unix) from:
<http://www.us.apache.org/dist/lucene/java/5.4.0/>
2. Extract the zip file to a folder
3. Open Eclipse
4. Add 4 JAR files (see next slide), located in the folder of step 2, into CLASSPATH
 - In Eclipse Menu: Window → Preferences → Java → Installed JREs → double click on java-7-oracle → Add External JARs (choose files to add)
5. Finish → Apply → OK

* Apache Lucene is already downloaded and associated with Eclipse in your VM

Setting your CLASSPATH



- You need **four (4)** JARs:
 - Core Lucene JAR,
 - Queryparser JAR,
 - Common analysis JAR,
 - Lucene demo JAR.
- The Core Lucene JAR file is located under **core/** directory, created when you extracted the archive; it should be named **lucene-core-{version}.jar**.
- The other 3 JARs, called **lucene-queryparser-{version}.jar**, **lucene-analyzers-common-{version}.jar** and **lucene-demo-{version}.jar** are located under **queryparser**, **analysis/common/** and **demo/**, **respectively**.

Lucene: Example 1



- Create a new Class named “**HelloLucene**” with a main() function and import the following libraries

```
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.TextField;
import org.apache.lucene.document.StringField;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.queryparser.classic.ParseException;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopScoreDocCollector;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.RAMDirectory;
```

Lucene: Example 1



- Add the following exceptions to the main() function

```
throws ParseException, IOException
```

- Specify the analyzer that you will use for indexing/searching

```
StandardAnalyzer analyzer = new StandardAnalyzer();
```

- Create the Index

```
Directory index = new RAMDirectory\(\); // index in RAM  
IndexWriterConfig config = new IndexWriterConfig(analyzer);
```

```
IndexWriter w = null;
```

```
int id = 0;  
w = new IndexWriter(index, config);  
addDoc(w, "Lucene in Action", ++id);  
addDoc(w, "Lucene for Dummies", ++id);  
addDoc(w, "Managing Gigabytes", ++id);  
addDoc(w, "The Art of Computer Science", ++id);  
w.close();
```

Lucene: Example 1



- Create the query String and parse it with the query parser
- Use the same analyzer for both indexing and searching!

```
String querystr = "lucene";  
// Use the same analyzer for both indexing and  
searching  
QueryParser parser = new QueryParser("title",  
analyzer);  
Query q = null;  
q = parser.parse(querystr);
```

↑
DEFINE THE DEFAULT FIELD
FOR QUERY TERMS

Lucene: Example 1



- Make the search in the index for the specified query String and store the top 10 results

```
int hitsPerPage = 10;
IndexReader reader = null;
TopScoreDocCollector collector = null;
IndexSearcher searcher = null;
reader = DirectoryReader.open(index);
searcher = new IndexSearcher(reader);
collector =
TopScoreDocCollector.create(hitsPerPage);
searcher.search(q, collector);
ScoreDoc[] hits = collector.topDocs().scoreDocs;
```

Lucene: Example 1



- Display the results and close the index reader

```
System.out.println("Found " + hits.length + "
hits.");
for (int i = 0; i < hits.length; ++i) {
    // get ids of the returned documents
    int docId = hits[i].doc; // hit[i].score
    Document d;
    // retrieve document with given id from index
    d = searcher.doc(docId);

    System.out.println((i + 1) + ". " +
        d.get("title"));
}
reader.close();
```

Lucene: Example 1



- Create the addDoc() function

```
private static void addDoc(IndexWriter w, String
title, int id) throws IOException {
    Document doc = new Document();
    doc.add(new StringField("id",
Integer.toString(id), StringField.Store.YES));
    doc.add(new TextField("title", title,
TextField.Store.YES));
    w.addDocument(doc);
}
```

↑
TO BE TOKENIZED
FOR INDEXING

StringField vs TextField: In the above example, the "id" field contains the ID of the document, which is a single atomic value.

In contrast, the "title" field contains an English text, which should be parsed (or "tokenized") into a set of words for indexing.

Use StringField for field with atomic value that is INDEXED BUT NOT TOKENIZED.

Use TextField for a field that is INDEXED AND TOKENIZED into a set of words.



Indexed and Stored



- There are two basic ways a document field can be written into Lucene.
 - Indexed - The field **is indexed** and can be searched (applies for both StringField and TextField)
 - Stored - The **field's full text is stored in index** and will be returned with search results (Store.YES, Store.NO)
- If a field is indexed but not stored, you can search for it, but it won't be returned with search results.
- A common practice is to have an ID field being stored which can be used to retrieve the full contents of the document /record from, for instance, a SQL database, a file system, or an web resource.
- You might also opt not to store a field when that field is just a search tool, but you wouldn't display it to the user, such as a soundex/metaphone, or an alternate analysis of a field.



Lucene: Example 1



- Run the application
- You should see the following in the Console tab:

```
Found 2 hits.
```

1. Lucene in Action
 2. Lucene for Dummies
-

Lucene: Example 1



- Now modify the application to return only the documents whose title contains both “lucene” AND “action”

- 1st approach: Use the special syntax of QueryParser

```
String querystr = "title:lucene AND title:action";
```

- 2nd approach: Construct manually the query

```
TermQuery tq1 = new TermQuery(new Term("title",  
    "lucene"));
```

```
TermQuery tq2 = new TermQuery(new Term("title",  
    "action"));
```

```
BooleanQuery bq = new BooleanQuery();
```

```
q.add(tq1, BooleanClause.Occur.MUST);
```

```
q.add(tq2, BooleanClause.Occur.MUST);
```

- More about query syntax at

<http://www.lucenetutorial.com/lucene-query-syntax.html>

http://lucene.apache.org/core/6_4_0/core/org/apache/lucene/search/package-summary.html#package_description

Lucene: Example 1



- Recall that index contains the titles of 4 docs:

Lucene in Action

Lucene for Dummies

Managing Gigabytes

The Art of Computer Science

- What results do you expect to get when you submit the following query:
 - title:c? OR title:d*
 - title:c* OR title:d*
 - title:lucene AND title:for
-

Lucene: Example 2



- Use Lucene to index all the text files in a directory and its subdirectories
- Additionally to the previous example the following library is required:

```
import java.io.File;
```

- In main() (which throws IOException) add the following code:

```
File dataDir = new File("LuceneFiles");

// Check whether the directory to be indexed exists
if (!dataDir.exists() || !dataDir.isDirectory()) {
    throw new IOException(dataDir
+ " does not exist or is not a directory");
}

Directory indexDir = new RAMDirectory();
// Specify the analyzer for tokenizing text.
StandardAnalyzer analyzer = new StandardAnalyzer();
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter writer = new IndexWriter(indexDir, config);

// call indexDirectory to add to your index
// the names of the txt files in dataDir
indexDirectory(writer, dataDir);
writer.close();
```


Lucene: Example 2



- Create the `indexDirectory()` method:

```
private static void indexDirectory(IndexWriter
writer, File dir) throws IOException {
    File[] files = dir.listFiles();

    for (int i = 0; i < files.length; i++) {
        File f = files[i];
        if (f.isDirectory()) {
            indexDirectory(writer, f); // recurse
        } else if (f.getName().endsWith(".txt")) {
            // call indexFile to add the title of the txt
            file to your index
            indexFile(writer, f);
        }
    }
}
```

Lucene: Example 2



- Create the `indexFile()` method:

```
private static void indexFile(IndexWriter
writer, File f) throws IOException {
    System.out.println("Indexing file \t\t" +
        f.getName());

    Document doc = new Document();
    doc.add(new TextField("filename", f.getName(),
        TextField.Store.YES));
    writer.addDocument(doc);
}
```

Example 2 - Submission



- With the code given your index is ready!
- Don't forget to create "LuceneFiles" directory to your workspace under your project's folder using files in the dataset.zip
- Query your index for the word `rep*rt*` (on the field "filename") using the QueryParser and print results on console as follows:

```
Problems @ Javadoc Declaration Console
<terminated> FileIndexer [Java Application] /usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java
Indexing test.txt
Indexing report.txt
Indexing republic.txt
Found 1 hits.
1. report.txt
```

- Implement Example 2 and submit the .java file and the results in a .txt file to Moodle by 15th of October @ 15:00

Useful Links



- <http://lucene.apache.org/core/>
 - <http://www.lucenetutorial.com/>
 - http://lucene.apache.org/core/8_6_0/index.html
-

Class RAMDirectory



- A memory-resident [Directory](#) implementation.
- This class is optimized for small memory-resident indexes. It also has bad concurrency on multithreaded environments.
 - **Warning:** This class is not intended to work with huge indexes. Everything beyond several hundred megabytes will waste resources, because it uses an internal buffer size of 1024 bytes, producing millions of byte[1024] arrays.
- Recommended to materialize large indexes on disk and use [MMapDirectory](#),
 - high-performance directory implementation working directly on the file system cache of the operating system