# Report Lucene

Nick Wils, Gauthier Le Compte

November 8, 2021

Github Project

## 1 What is Lucene

Lucene is a text search engine software library that provides a search and indexing platform based on Java. Lucene is considered in high regard because it can return search responses very quickly. This is made possible due to the fact that it searches an index which has been created in relation to that content, instead of searching text or content directly. This method is also known as an inverted index.

## 2 What are the main functionalities in Lucene?

### 2.1 Which types of indices are available? How does Lucene store the index?

As mentioned above, lucene uses an inverted index which means that it searches an index which has been created in relation to that content.

Lucene holds an index over a dynamic collection of documents and returns very rapid updates to the index whilst documents are added to the collection and also whilst they are deleted from the collection. An index may store a varying set of documents. Lucene searches over terms. A term combines a field name with a token. The terms created from the non-text fields in the document are pairs consisting of the field name and the field value. The terms created from text fields are pairs of field name and token.

The Lucene index provides a mapping from terms to documents. This is called an inverted index because it reverses the usual mapping of a document to the terms it contains. The inverted index provides the mechanism for scoring search results: if a number of search terms all map to the same document, then that document is likely to be relevant.

### 2.2 Various types of queries

- **TermQuery:** This query matches documents which contain a term.
- **BooleanQuery:** This query matches documents matching boolean combinations of other queries
- **WildcardQuery:** This query implements the wildcard search query.The wildcard * matches any character sequence, and the wildcard ? matches any single character.
- **PhraseQuery:** This is a query that matches documents containing a particual sequence of terms. A PhraseQuery is built by the QueryParser for inputs like "San Francisco, New York,..."

- **MultiPhraseQuery:** This query is a generalized version of PhraseQuery, with the extra probability of adding more then one term at the same position that are treated as "OR".
- **PrefixQuery:** This query matches documents containg terms with a certain prefix.
- **FuzzyQuery:** This query implements the fuzzy search query. This query is based on the Damerau-Levenshtein algorithm, but you can also choose the Levenshtein algoritm by adjusting the parameters.
- **RegexpQuery:** This is a fast regular experssion query
- **TermRangeQuery:** This is a query that matches documents within a range of terms
- **PointRangeQuery:** This query is for subclasses and works on the underlying binary encoding
- **ConstantScoreQuery:** This query maps another query and returns a constant score to equal to 1 for every document that matches the query
- **DisjunctionMaxQuery:** This query generates the union of documents produced by its subqueries, and that scores each document with the maximum score for that document as produced by any subquery, and also a tie breaking increment for any additional matching subqueries.
- **MatchAllDocsQuery:** This is a query that matches all documents

## 2.3 How is search and scoring done in Lucene?

The searching in Lucene is done as follows: Lucene take a search query and returns a set of documents ranked by relevancy, where documents who are most similar to the query get the highest score. Lucene provides a highly configurable hybrid form of search that combines exact boolean searches with softer, more relevance-ranking-oriented vector-space search methods. All searches are field specific because Lucene indexes terms and a term is composed of a field name and a token.

Lucene scoring uses a combination of Vector Space Model (VSM) of Information Retrieval and the Boolean model. Using this combination, it determines how relevant a given document is to a user's query. The idea behind the Vector Space Model is the more times a query term appears in a document in regards to the number of times the term appears in all documents in the collection, the more relevant that document is to the query. First, it uses the Boolean Model to slim down the documents that need to be scored, such that irrelevant documents are not processed.

- **BM25 Similarity:** This query matches documents which contain a term.
- **Boolean Similarity:** This query matches documents matching boolean combinations of other queries
- **Multi Similarity:** This query implements the wildcard search query.The wildcard * matches any character sequence, and the wildcard ? matches any single character.
- **Axiomatic Similarity:** There are a family of models. All of them are based on BM25. In Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '05). ACM, New York, NY, USA, 480-487.
- **DFI Similarity:** This simimarity implements the Divergence from Independence (DFI) model based on Chi-square statistics. DFI is both parameter-free and non-parametric. With parameter-free we mean that it does not require any parameter tuning or training and non-parametric means that it does not make any assumptions about word frequency distributions on document collections. It is highly recommended not to remove stopwords if you plan on using this similarity.
- **DFR Similarity:** This simalirity implements the divergence from randomness (DFR) framework introduced in Gianni Amati and Cornelis Joost Van Rijsbergen. 2002. Probabilistic models of information retrieval based on measuring the divergence from randomness. ACM Trans. Inf. Syst. 20, 4 (October 2002), 357-389.
- **IB Similarity:** Provides a framework for the family of information-based models, as described in Stéphane Clinchant and Eric Gaussier. 2010. Information-based models for ad hoc IR. In Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval (SIGIR '10). ACM, New York, NY, USA, 234-241.
- **LM Similarity:**
- **TFIDF Similarity:** This similarity implements the Vector Space Model for the scoring API. It defines the components of Lucene scoring. Overriding computation of these components is a convenient way to alter Lucene scoring. In VSM, documents and queries are represented as weighted vectors in a multi-dimensional space, where each distinct index term is a dimension, and weights are Tf-idf values.

  Lucene refines VSM score for both search quality and usability:

  - Normalizing V(d) to the unit vector can sometimes be problematic because it removes all document length information. For some documents removing this info is probably fine, but for a document which contains no duplicated paragraphs, this might be wrong. This can be avoided by using a different document length normalization factor, which normalizes to a vector equal to or larger than the unit vector: doc-len-norm(d).

- Users can specify at indexing that certain documents are more valuable or important than others. This can be done by assigning a document boost. For this, the score of each document is also multiplied by its boost value doc-boost(d).
- Users can at searchtime specify boosts to each query, sub-query, and each query term, hence the contribution of a query term to the score of a document is multiplied by the boost of that query term query-boost(q).
- A document may match a multi term query without containing all the terms of that query (this is correct for some of the queries).

## 3 Details of Implementation

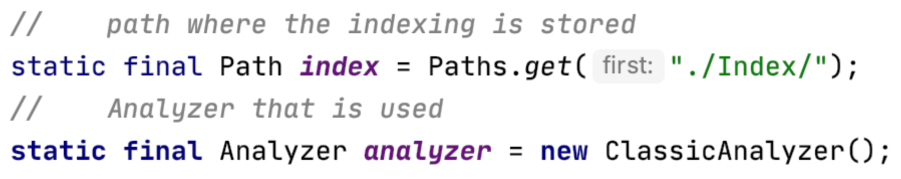
### 3.1 Structure

Our code consists of 2 classes: Parser and SearchEngine. The Parser is used to perform read and write functions, while the SearchEngine contains the main part of our code.

Of course we were not tasked with reinventing the wheel, so we based our code on reference [5]. With this code as a base we made loads of adjustments and implemented our own things. With some trial and error we got to the final result we have now.

### 3.2 Search Engine

Broadly speaking, the SearchEngine will index the database, prepare the query, search for the query and give the best results back. In this class we have two class variables: Path (relative path to the index folder) and Analyzer (the analyzer used in indexing).



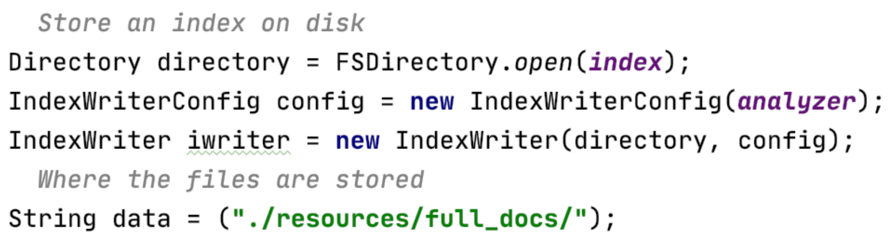
```
//    path where the indexing is stored
static final Path index = Paths.get( first: "./Index/");
//    Analyzer that is used
static final Analyzer analyzer = new ClassicAnalyzer();
```

Our code starts in the main function where we will call the necessary functions to search a query. We start by indexing the database and end by deleting the indexed files, to clean up for the next use. In between we have three options:

- Search a single query
- Search all the main queries and write them to a csv file
- Test our engine by searching development queries and comparing them with the given results.

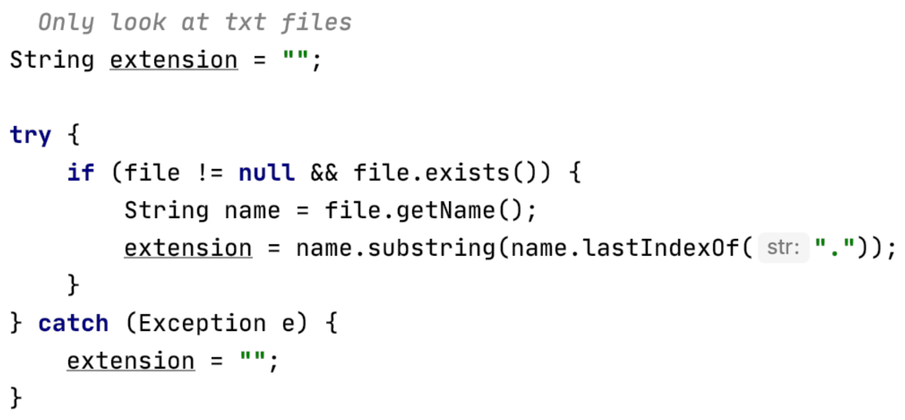
### 3.2.1 Indexing

This will be done in the index() function. We first open the directory where we will store the indexed files and then configure the index writer with this directory and an analyzer. Troughout the whole project we made use of the Classical analyzer.



```
   Store an index on disk
 Directory directory = FSDirectory.open(index);
 IndexWriterConfig config = new IndexWriterConfig(analyzer);
 IndexWriter iwriter = new IndexWriter(directory, config);
   Where the files are stored
 String data = ("./resources/full_docs/");
```

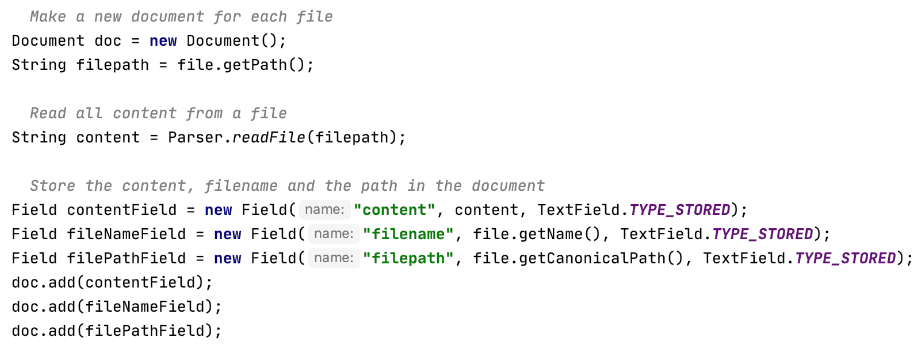
After this we will loop over all the files in our database, but only look at the textfiles. We check this by looking at the extension (if it exists) and skip the file if it is not a text file.



```
   Only look at txt files
 String extension = "";

 try {
     if (file != null && file.exists()) {
         String name = file.getName();
         extension = name.substring(name.lastIndexOf( str: "."));
     }
 } catch (Exception e) {
     extension = "";
 }
```

When we have a usable file we can start making a new document. In this document we will store the content that the Parser class will give us: the filename and the absolute path. After we added all the needed information we can add this document to the index writer. We do this for every file, until the full database is indexed.



```
  Make a new document for each file
Document doc = new Document();
String filepath = file.getPath();

  Read all content from a file
String content = Parser.readFile(filepath);

  Store the content, filename and the path in the document
Field contentField = new Field( name: "content", content, TextField.TYPE_STORED);
Field fileNameField = new Field( name: "filename", file.getName(), TextField.TYPE_STORED);
Field filePathField = new Field( name: "filepath", file.getCanonicalPath(), TextField.TYPE_STORED);
doc.add(contentField);
doc.add(fileNameField);
doc.add(filePathField);
```

### 3.2.2 Searching

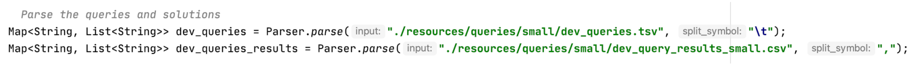
In this function we search for a query through the indexed files. To do this we first need to parse our query using a queryparser. This will make use of the analyzer and the field where we need to search a document. The queries we use are related to the content of the document, therefore the field will be "content". Before we could use this parser we needed to escape the special symbols in the query.

After we parsed our query, we opened an index reader, which reads the indexed files. Using this reader we made an index searcher. In this searcher we set that we would use the BM25Similarty, as this was the one that yielded the best results.

Finally we searched the indexed files using our query and found the top 10 files. Of these top 10 files we returned, in order, their filename.

### 3.2.3 Testing

To test our implementation we used our parser to make dictionaries of the development queries and their best scoring database files.



```
  Parse the queries and solutions
Map<String, List<String>> dev_queries = Parser.parse( input: "./resources/queries/small/dev_queries.tsv",  split_symbol: "\t");
Map<String, List<String>> dev_queries_results = Parser.parse( input: "./resources/queries/small/dev_query_results_small.csv",  split_symbol: ",");
```

With these dictionaries we can loop over all the development queries and search our top scoring database files. The search() returns us the name of the top database files, from which we can extract the file number. We could now search the dictionary to see if our file numbers where the top scoring files given the query number.

## 3.3 Parser

Writing a parser was not the goal of this assignment. So instead of writing one ourselves, we used one from javatpoint[6]. Which we of course altered to satisfy our needs. In the parser you will find following main functions:

- readFile(): Used to read the database files.
- parse(): Used to read a csv or tsv file and return a dictionary.
- write(): Used to write our top scoring database files to a csv.

### 3.4 Our Findings

Troughout this project we tried a lot of combinations with different types of queries, scoring systems and analyzers. To name a few:

- BM25 Similarity
- Boolean Similarity
- PhraseQuery
- FuzzyQuery
- ...

When we switched between analyzers, we found that there wasn't too much of a difference in the results. The scoring systems and query types did make a big difference. We expected the PhraseQuery to provide the best result as it was designed to retrieve phrases. Contrary to our expectations, this produced a poor result. We think a phraseQuery doesn't account for typos and haven't found a way to fix this.
For us, a combination of the ClassicAnalyzer, the Queryparser and the BM25Similarity worked best. When testing with the large dataset, we made the following findings: 63.5 of our top 10 found file numbers per query matched the ones in the statement. When we expanded our search to the top 20, we found that almost every filenumber given was at least in our top 20.

## References

[1] https://techmonitor.ai/techonology/hardware/apache-lucene

[2] https://dzone.com/articles/apache-lucene-a-high-performance-and-full-featured

[3] https://lucene.apache.org/core/7_3_1/core/org/apache/lucene/search/Query.html

[4] https://lucene.apache.org/core/3_5_0/scoring.html

[5] https://lucene.apache.org/core/7_3_1/core/overview-summary.html

[6] https://www.javatpoint.com/how-to-read-file-line-by-line-in-java

[7] https://lucene.apache.org/core/8_0_0/core/org/apache/lucene/search/similarities/Similarity.html