Universiteit Antwerpen

Assignment IR

Information Retrieval: Lucene

By
Lore Hilgert

2020-2021

Contents

1 Introduction

This project has as goal to discuss the information retrieval utilities of the full-text search library Lucene [1], which is an extensive library available in Java. It can also be used from different programming languages such as Object Pascal, Perl, C#, C++, Python, Ruby and PHP. In the first section, we highlight the operational structure and details of the library. Afterwards, we build a simple information retrieval system that can index, search and retrieve text documents. Therefore, it is important to first understand the indexing, search and document concepts of Lucene. The structure of the Lucene package can be found on figure 1.1

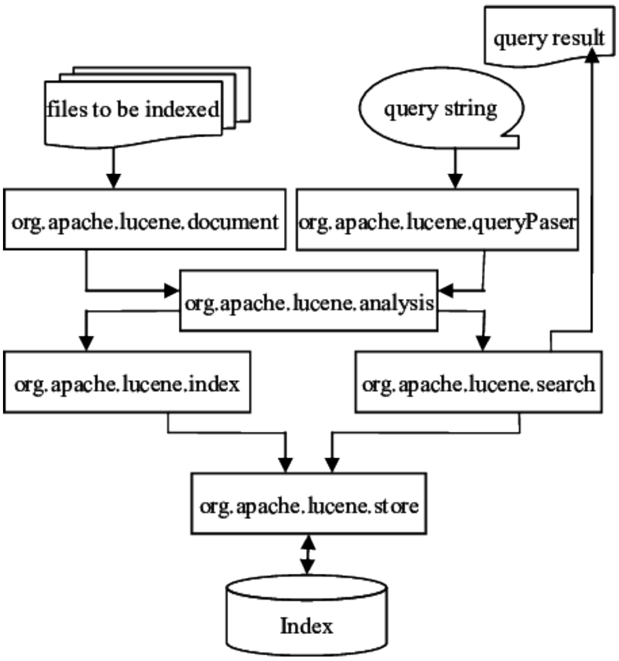


Figure 1.1: System Architecture of Lucene Package org.apache.lucene.store, from [3]

2 Core concepts

2.1 Indexing

Before creating the index, one should prepare the text documents. Lucene is able to index every data that can be converted into textual format. When using PDF or HTML files as search pool, we should first extract the textual information from the documents and then send the information to Lucene for indexing [2].

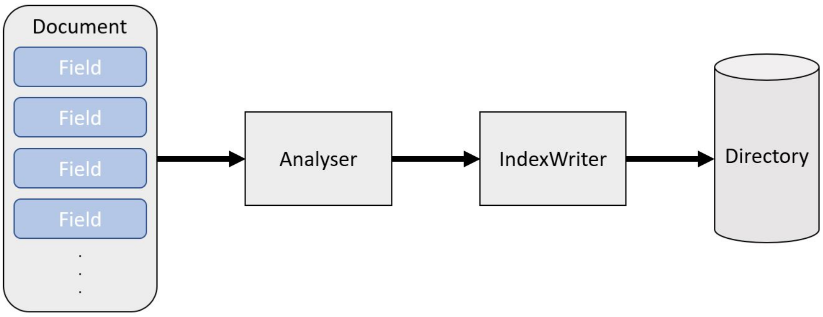


Figure 2.1: Indexing process

As seen as in figure 2.1, the main concepts in Lucene are [7]:

- **Index**: contains a sequence of documents
- **Document**: is a sequence of fields
- **Field**: is a sequence of terms
- **Term**: is a string

Assuming the text files got extracted into .txt files, the next step would be creating the index. This is one of the main aspects of an information retrieval system. The index API provides functionality for *reading from* and *writing to* in the form of two interfaces, `IndexReader` and `IndexWriter`. To make sure new indexes can be created, an instance of the `IndexWriter` class should be made. This instance is a key component in the indexing process. `IndexWriter` is thread-safe and is configured for expert use. It can create a new index or open an existing index and add documents to it. To configure the class, three parameters need to be filled out: the directory that stores the index files (local OS directories, other memory structures, such as RAM, ...), the analyzer and a Boolean variable, which is true if the class creates a new index. The index consists of different segments, containing subsets of indexed documents. The aforementioned segments can only be created or deleted, never changed. Therefore, one should merge and clean up the segments from time to time.

Now an instance of the `Document` class should be created. Because after all, an index is a collection of documents. Each document contains `Fields`, which will be discussed later on.

2.1.1 Improving performance

The bottleneck of the operation is writing the documents into the index files on the disk [2]. Therefore, Lucene provides three parameters to optimize the indexing performance. This is useful when a great amount of documents needs to be indexed. With these parameters, the size of the buffer pool and the merge policy can be controlled:

```
indexWriter.mergeFactor;
indexWriter.minMergeDocs;
indexWriter.maxMergeDocs;
```

2.1.2 Types of indexes

Every index document contains different `Fields`, which are used to store information. Each field has three parts: name, type and value. Some interesting fields are the `TextField` and the `StringField`. The first one is a field that is indexed and tokenized, without term vectors [4]. It would be used on a field that contains the bulk of a document's text. The latter is a field that is indexed but not tokenized, which means that the entire `String` value is indexed as a single token. It can be used for a 'country' field or an 'ID' field, thus identifier fields or any field that is used for sorting. Every field also has a `Store` value, it specifies whether and how a field should be stored. Storing the original field value is useful for short texts, such as a document's title, which should be displayed together with the results.

Lucene uses an *inverted index* of data, also referred to as a *postings file*. This means that there is a mapping of `Terms` (or tokens) to the `Documents` that include that particular `Term`, instead of mapping documents to keywords. It allows for faster search responses, as it searches through an index, instead of searching through text directly [6]. Consequently, a term-to-document lookup will be handled very efficiently. Opposite to the inefficiency to find all the terms a document consists of (the entire index has to be scanned). The solution to this problem lies with the stored `Fields`. These fields map the opposite of posting, 'monitoring' all `Terms` in a `Document`. In other words, the text of the fields is stored in a non-inverted manner.

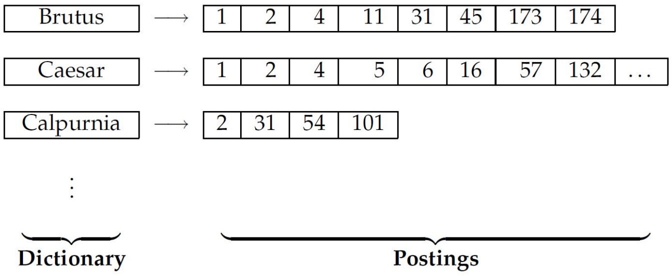

Brutus → 1 2 4 11 31 45 173 174
Caesar → 1 2 4 5 6 16 57 132 . . .
Calpurnia → 2 31 54 101
⋮
Dictionary
Postings

Figure 2.2: Inverted index example, from [9]

Besides postings and stored `Fields`, Lucene can use two other types to meet more specific searching requirements, namely `DocValues` and `PointValues`.

`DocValues` are a way of recording field values internally to be more efficient for some purposes, such as sorting and faceting [8]. The concept is the storing of a 'table' with Document ID's as rows and `Field` values as columns. Next to this we have `PointValues`. Points represent numeric values and are indexed differently than ordinary text. `PointValues` allow for efficient range and interval querying.

2.1.3 Analysing

An analyzer is responsible for extracting tokens out of text to be indexed. Lucene contains language analysers, class implementations for the removal of possessives and several stemming algorithms like the Kstem algorithm or Porter stemming. Next to these analysers, Lucene offers spelling correction and auto suggest classes. In this project, the `StandardAnalyzer` will be used because it is the most sophisticated analyzer. `StandardAnalyzer` can handle names, email addresses, and so on. It lowercases each token and removes common words and punctuations, if needed.

2.2 Searching

Once an index is built, we can search that index. Therefore we need a `Query` and an `IndexSearcher` [6]. The search result is typically a result set, containing the retrieved data. Note that an `IndexWriter`, as mentioned in section 2.1, creates the index that can then be searched by the `IndexSearcher`. Important is that the searching of a collection of documents proceeds in an efficient way. Using Lucene, we are provided with a search application programming interface and its relevant functionalities to allow searching a predefined index. The core of the searching still lies in the use of `Queries`. Lucene contains a lot of different `Query` types. Each type is suited for some specific searching requirements and needs. Once you are familiar with Lucene, one can also extend this functionality using your own custom `Query` classes.

After creating an acceptable `Query`, the index can be searched and the scoring process starts. The different scoring models will be explained in section 2.3.

Indexes are searched using the `IndexSearcher` interface, which gives us the functionality for retrieving documents with an appointed `Query`. We can instantiate the `IndexSearcher` class, adding the index repository name as an essential parameter. The analyser that we provide to the constructor should be of the same type as the one used in the indexing process. Here this should thus be the `StandardAnalyzer`. The `QueryParser` instance applies the analyzer to the query string, ensuring that the tokenization and other transformations applied to terms are consistent. We must also specify which fields will be queried by default for each query [10].

To do an asymmetric search, the `IndexSearcher` can be configured to give a weight to a specific `Query`. Another option is to give specific subsets of documents, for example the top k documents. Next to these, the used similarity measure can also be configured. The latter will be explained further on in the project.

As already mentioned, the query and its syntax is one of the main components of the search. There are different types of `Query` available in Lucene, we will discuss the most commonly used ones in the next paragraph.

2.2.1 Query

This section will discuss the query and its syntax. There are different types of queries. The core component `TermQuery` is a `Query` that can search an index for a given `Term`. A term is a value of a particular field. Most of the time, searching an index with one single term is not sufficient for the information retrieval. That is where the other types of `Query` will be needed. The `TermQuery` acts as a Boolean-like query and matches the documents that contain the `Term`. Additional to the `TermQuery`, there are the `TermRangeQuery` which is used when a range of textual terms is to be searched, `PrefixQuery` which is used to match documents whose index starts with a specified string and the `BooleanQuery`. The latter is a `Query` that can contain several clauses, each containing an instance of `TermQuery`. The maximal clauses that are permitted in the `Query` is by default limited to 1024, but can be adjusted by the user. These clauses can be combined by some keywords (AND, OR, NOT). Another functionality is to determine whether the clauses may or may not occur using some operators like:

- MUST: clauses must occur an contribute to the score
- MUST NOT: clause that may not occur
- SHOULD: clauses that may occur, but not necessarily
- FILTER: clauses that have to occur but do not contribute to the score

When one wants to find a phrase in an index, one of these `Query`types should be used: `PhraseQuery`, `MultiPhraseQuery` or `SpanNearQuery`. These queries can have multiple complexity levels. For the phrase queries, a slop can be determined. This slop is an edit-distance[1], where the units correspond to moving the terms in the query phrase out of position. The edit-distance is one of the techniques for addressing isolated-term correction. A second complexity factor would be to allow synonyms of the terms in the query.

To make a term of a `Query` more important than the others, one can boost a term. The default boost factor is 1, but can be modified using for example *term^ 3*, which sets the boost factor for that term to 3. Boosting can determine a document's relevance, by boosting a term that can be found in that document.

[1] Also known as Levenshtein Distance - fuzzy search

Two more important search methods are Wildcard search and Fuzzy search. The former is initiated using an instance of the `WildCardQuery`. This allows the use of '?' and '*', which stands for 1 or 0+ matching characters at the place of the wildcard. The position of the wildcard concludes whether the search is still efficient or not. For example, general wildcards (wildcard possibly in the middle of a term) tend for a permuterm index. The key from this concept is to rotate the term in the query such that the wildcard appears at the end of the term. Trailing and leading wildcard queries can be easily found using the B-tree [9].

Finally, we will discuss Fuzzy search. The first implementation of the information retrieval of this project will be an example of Fuzzy search. The `FuzzyQuery` is used to search documents using fuzzy implementation which is an approximate search based on the edit-distance algorithm (Levenshtein Distance). Adding '~' for the use of fuzzy search. It will only enable this type of search, but when specifying with a number; maximum that number of edits may be done for that particular query. The Levenshtein distance is calculated as follows:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Fuzzy search is indispensable when taking into account spelling variations.

2.3 Score models

`Similarity` defines the components of Lucene scoring. Selecting a good similarity measure is required for the retrieval of the desirable, relevant documents for a given query. The `Similarity` interface implements a lot of different models into the library, such as BM25, PerField, Boolean, Vector Space Model and so on.

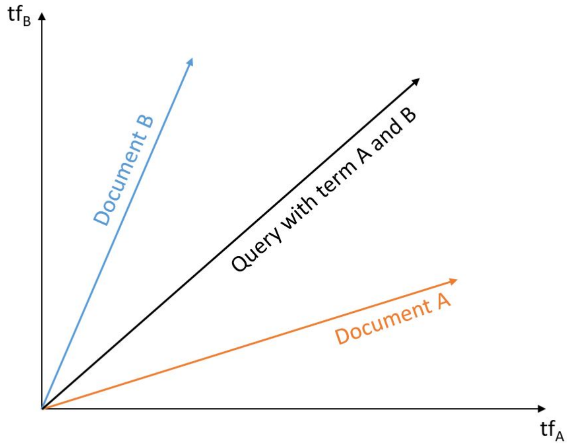


Figure 2.3: Example of vector space modelling

These models vary in complexity and efficiency. These models are used to score documents and rank them accordingly, to make sure that the returning matches are not solely based on the query result. A document gets a score which is calculated based on all the `Fields` of that document. The models also allow for boosting the score contribution of certain `Fields` by giving them more weight, and thus to correspond with the importance of the term in the query or document. This was already mentioned in section 2.2.1. Lucene also allows experts to create their own `Similarity` measures, allowing for advanced optimisations. Lucene combines the Boolean Model and Vector Space Model of Information Retrieval, which gives for (as assumption: one single field in the index):

$$score(q,d) = coord.factor(q,d) \cdot query.boost(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot doc.len.norm(d) \cdot doc.boost(d)$$

The above conceptual formula is a simplification in the sense that the terms and documents are fielded and the boosts are usually per query term rather than per query [5]. The document vector is not normalised because this would remove all document length information. The conceptual scoring formula relates to a practical scoring function, which will be dependent on the inverse document frequency of a term and the term frequency of that term in the document.

Now, the basic and core concepts of Lucene are clear, we start with the implementation of a document retrieval system based on Lucene.

3 Implementation document retrieval system

For this project, a fuzzy search will be implemented. This can be adapted easily by adapting the search query technique. The code of the implementation can be found on GitHub: https://github.com/LorHil/Project_Lucene. Below, the different steps to get to a complete implementation are briefly discussed.

3.1 Index

A first step is the building of an index. Therefore, we need three Java scripts from the code, namely `LuceneConstants.java`, `TextFileFilter.java` and `Indexer.java`. Indeed, the Indexer will be the main component for indexing the documents. The TextFileFilter just filters .txt files that need to be indexed. When PDF or HTML files have to be indexed, instead of textfiles, they should first be parsed. LuceneConstants contains some trivial constants that are used during the process. The constant `MAX_SEARCH` is the maximum top of documents that is retrieved. Here, for this example, we set the value to 10, to retrieve the top 10 documents. This can be adjusted at all times.

The most important script is `Indexer.java`. We will go through some basic, but crucial stages of the process. The first step is to create an index in a chosen directory. After creating and updating the index, you must close it.

```
public class Indexer {
 private IndexWriter writer;
 public Indexer(String indexDirectoryPath) throws IOException{
        Directory indexDirectory =
        FSDirectory.open(new File(indexDirectoryPath));

        writer = new IndexWriter(indexDirectory,
        new StandardAnalyzer(Version.LUCENE_36),true,
        IndexWriter.MaxFieldLength.UNLIMITED);
        }
        public void close() throws CorruptIndexException, IOException{
                try {
                     writer.optimize();
                     writer.close();
                     } catch (Exception e) {
                      System.out.println("Got an Exception: " +
                         e.getMessage());
                     }
                    }
```

Above, the defining and closing of the index can be found. Then, the documents that need to be indexed are found in the foreseen directory. If they are available in this directory and they meet the file-filter, they are indexed. Below, one can see that for every document, the given fields are created: `contentField`, `fileNameField` and `filePathField`. The document then gets indexed, and this can be printed optionally. The code in this report do not contain a lot of comments, they are added in the GitHub repository.

```
private Document getDoc(File file) throws IOException{
         Document document = new Document();
         Field contentField = new Field(LuceneConstants.CONTENTS,
         new FileReader(file));
         Field fileNameField = new Field(LuceneConstants.FILE_NAME,
         file.getName(),
         Field.Store.YES,Field.Index.NOT_ANALYZED);
         Field filePathField = new Field(LuceneConstants.FILE_PATH,
         file.getCanonicalPath(),
         Field.Store.YES,Field.Index.NOT_ANALYZED);
         document.add(contentField);
         document.add(fileNameField);
         document.add(filePathField);
         return document;
         }
         private void indexFile(File file) throws IOException{
         //System.out.println("Indexing "+file.getCanonicalPath());
         Document document = getDoc(file);
         writer.addDocument(document);
         }
         public int createIndex(String dataDirPath, FileFilter filter)
         throws IOException{
         File[] files = new File(dataDirPath).listFiles();
         for (File file : files) {
         if(!file.isDirectory()
         && !file.isHidden()
         && file.exists()
         && file.canRead()
         && filter.accept(file)
         ){
         indexFile(file);
         }
         }
         return writer.numDocs();
         }
        }
```

This also marks the end of the class Indexer. A next step will be to create the search functionality, which will search for the given query in the index. The following section will briefly discuss the Searcher class.

3.2 Searching the index

The next step is searching the index for the given query. The searcher that is created returns the found documents and their score. Mostly, not all retrieved documents are returned but only a best-top of them. The Searcher.java is also found in the repository GitHub. All code where I based this solution on, is found in the Source Branch of the repository.

Then finally, a global script (GlobalTester.java) should be made to test the document retrieval system. It will make us of all other, previously generated classes. An important first step is the creation of the index:

```
private void createIndex() throws IOException{
            indexer = new Indexer(indexDir);
            int numIndexed;
            long startTime = System.currentTimeMillis();
            numIndexed = indexer.createIndex(dataDir, new
               TextFileFilter());
            long endTime = System.currentTimeMillis();
            indexer.close();
            System.out.println(numIndexed+" File(s) indexed, time
               taken: "
            +(endTime-startTime)+" ms");
            }
```

For queries with multiple words, the words are individually searched in the index, when taking into account this fuzzy search. As we know, documents consist of Fields, and it is possible to do a search based on a specific field. *Do I want to search in the titles of the documents? Do I want to search in the content of the indexed documents?* These are some questions the user can ask him- or herself. Another option is to add more weight to the score of a document if the Term of the query is found in the title of that document instead of only in the content, because this should imply that that term is relevant for that document.

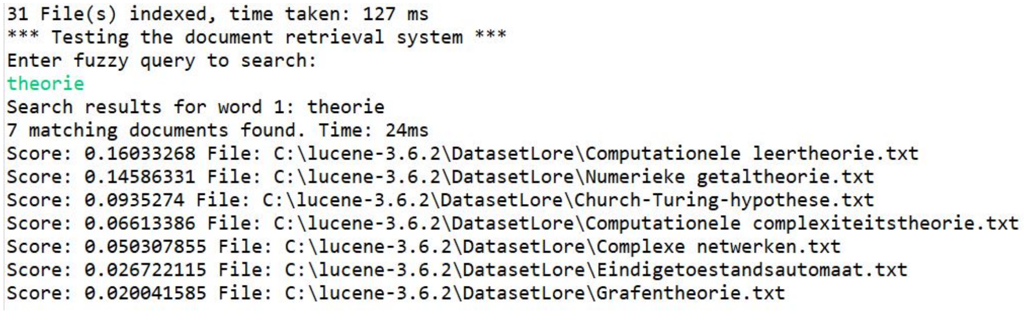
The following section will give some information on the used dataset and some examples that clarify the working of the document retrieval system.

3.3 Dataset

The dataset that is used for this project is a small[2] dataset composed of textfiles and is to be found on the following link: https://github.com/LorHil/Project_Lucene/tree/Dataset. When testing the code, do not forget to create the data and index directory folders locally in C://Lucene. The dataset contains the articles of Wikipedia who fall under the category *"Theoretische informatica"* on 10/11/2020. The dataset is generated for test-purpose.

[2] I choose a smaller database as test pool for this individual project

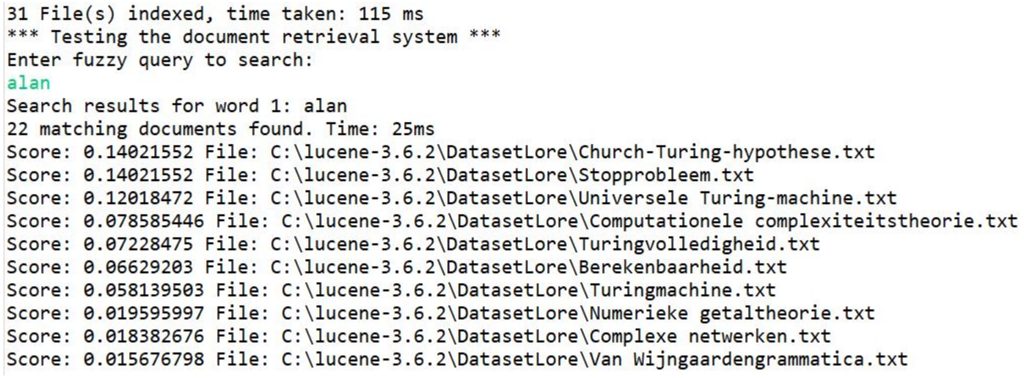
As an example, we can search for **theorie**:



```
31 File(s) indexed, time taken: 127 ms
*** Testing the document retrieval system ***
Enter fuzzy query to search:
theorie
Search results for word 1: theorie
7 matching documents found. Time: 24ms
Score: 0.16033268 File: C:\lucene-3.6.2\DatasetLore\Computationele leertheorie.txt
Score: 0.14586331 File: C:\lucene-3.6.2\DatasetLore\Numerieke getaltheorie.txt
Score: 0.0935274 File: C:\lucene-3.6.2\DatasetLore\Church-Turing-hypothese.txt
Score: 0.06613386 File: C:\lucene-3.6.2\DatasetLore\Computationele complexiteitstheorie.txt
Score: 0.050307855 File: C:\lucene-3.6.2\DatasetLore\Complexe netwerken.txt
Score: 0.026722115 File: C:\lucene-3.6.2\DatasetLore\Eindigetoestandsautomaat.txt
Score: 0.020041585 File: C:\lucene-3.6.2\DatasetLore\Grafentheorie.txt
```

Figure 3.1

Another example would be to search for **alan** from Alan Turing:



```
31 File(s) indexed, time taken: 115 ms
*** Testing the document retrieval system ***
Enter fuzzy query to search:
alan
Search results for word 1: alan
22 matching documents found. Time: 25ms
Score: 0.14021552 File: C:\lucene-3.6.2\DatasetLore\Church-Turing-hypothese.txt
Score: 0.14021552 File: C:\lucene-3.6.2\DatasetLore\Stopprobleem.txt
Score: 0.12018472 File: C:\lucene-3.6.2\DatasetLore\Universele Turing-machine.txt
Score: 0.078585446 File: C:\lucene-3.6.2\DatasetLore\Computationele complexiteitstheorie.txt
Score: 0.07228475 File: C:\lucene-3.6.2\DatasetLore\Turingvolledigheid.txt
Score: 0.06629203 File: C:\lucene-3.6.2\DatasetLore\Berekenbaarheid.txt
Score: 0.058139503 File: C:\lucene-3.6.2\DatasetLore\Turingmachine.txt
Score: 0.019595997 File: C:\lucene-3.6.2\DatasetLore\Numerieke getaltheorie.txt
Score: 0.018382676 File: C:\lucene-3.6.2\DatasetLore\Complexe netwerken.txt
Score: 0.015676798 File: C:\lucene-3.6.2\DatasetLore\Van Wijngaardengrammatica.txt
```

Figure 3.2

Here, one notices that there are more than 10 documents found, but only the top 10 documents are displayed. This is because we set the maximum of retrievals at 10, which can always be modified by the user (in the `LuceneConstants` script).

3.4 Conclusion

Now all crucial parts of a document retrieval system are clear. The figure below summarizes as a conclusion the components of such a system and their use:

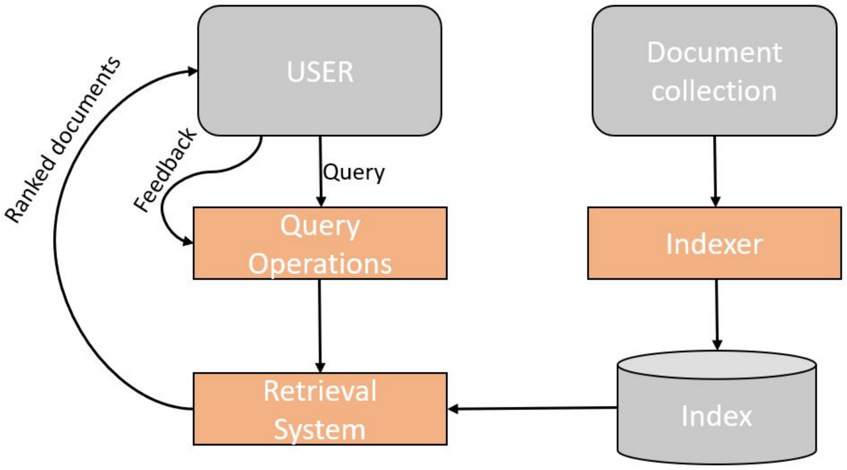


Figure 3.3: Document retrieval system

References

[1] Kelvin Tan. Lucene Tutorial: Basic concepts. Retrieved from *http://www.lucenetutorial.com/basic-concepts/* at 01/11/2020

[2] Deng Peng Zhou. Delve inside the Lucene indexing mechanism. Retrieved from *https://web.archive.org/web/20130904073403/http://www.ibm.com/developerworks/library/wa-lucene/* at 01/11/2020

[3] Y.H. Gu. System Architecture of Lucene Package org.apache.lucene.store. Retrieved from *https://www.researchgate.net/figure/System-Architecture-of-Lucene-Package-orgapachelucenestore-provides-binary-I-O-APIs_fig1_251981509* at 01/11/2020

[4] Apache Software Foundation. Class TextField. Retrieved from *https://lucene.apache.org/core/7_3_1/core/org/apache/lucene/document/TextField.html* at 01/11/2020

[5] Apache Software Foundation. Class imilarity. Retrieved from *https://lucene.apache.org/core/3_5_0/api/core/org/apache/lucene/search/Similarity.html* at 03/11/2020

[6] Baeldung. Introduction to Apache Lucene. Retrieved from *https://www.baeldung.com/lucene* at 20/10/2020

[7] Apache Software Foundation. Apache Lucene - Index File Formats. Retrieved from *https://lucene.apache.org/core/3_0_3/fileformats.html#Inverted%20Indexing* at 23/10/2020

[8] Apache Software Foundation. DocValues. Retrieved from *https://lucene.apache.org/solr/guide/6_6/docvalues.html* at 24/10/2020

[9] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. *An Introduction to Information Retrieval.* Cambridge University Press, Cambridge. 2009.

[10] Nicolas Travers. Putting into Practice: Full-Text Indexing with LUCENE. Retrieved from *http://webdam.inria.fr/Jorge/html/wdmch18.html#x24-37200017.3.2* at 02/11/2020