Information retrieval Project

-Report-

Topic: Search Engine Lucene

To:

Prof.Dr. Toon Calders

Student:

Name: Ali Doku

ID: S0163474

Email: ali.doku@student.uantwerpen.be

1 Introduction

Nowadays, Web Search Engine is one of the most powerful software system for retrieving specific information based on what people are looking for in a middle of billions of documents. As we know the amount of information is growing exponentially, and for users it is very important to find the information they need faster. So the aim of this project is about building a retrieval system, based only in Lucene API and using Java programming language and java server pages to build a normal search engine. Lucene is an open-source projects, java based, which combines the Information retrieval techniques in order to serve as better as to the people. It deals with indexing the files, parsing queries, scoring top documents based on query given and many more features included. In the open-source market exists many search engines to be implemented like Solr and Elastic Search, both built on Lucene API, using the schema of JSON and XML documents. The reason why I choose to work with Lucene is that other systems like Solr and Elastic Search are very optimized and needs only some little configuration, but with Lucene you can build your own retrieval system and optimize it on your need. Moreover, you understand the idea and work done in behind those systems. The following sections are organized in a sense to meet the requirements proposed in the project proposal. Chapter 2 provides the information how lucene indexing works, starting from analyzers, html document parsers and how they are saved in the space memory. Chapter 3 is about Searching, how the search is done in the index, parsing the query, returning the most relevant document based on the query and suggestions. Chapter 4 evaluates the result and testing, and chapter 5 is the conclusion.

2 Lucene File Indexing

The creation of a search engine starts from file indexing, which is the first and most important step in this system. Indexing is the process of parsing and storing of data in such way to eliminate the sequential scanning and make searching faster.

In order to better understanding Indexing let’s take a simple example. Suppose a person want to find some information and searches giving a query that can be a word or a phrase. The simplest

method is to scan sequentially every document word by word and then applying scoring formula to return the most relevant documents. But scanning the whole collection of documents is not suitable, so the best approach to solve this problem is by indexing. First the files are scanned word by word, where a word is treated as a term, and saving the terms in a data structure of such a format that will allows a person to search faster, not considering sequential scan. The following sections will explain in detail the steps of indexing.

2.1 Lucene Index

Starting indexing files, first we have to decide where to store the index files that will be created. There are 2 opportunities, to use RAMDirectory or File System Directory(FSDirectory). In my implementation I have used FSdirectory because the index files are save to the disk memory and It is not necessary to execute indexing every time you want to start the program. FSDirectory is slower than RAMDirectory, because normally Ram is faster because is closer to the CPU then disk, but the disadvantage of RAMDirectory is that the files exists as long as the program is running. Once the program is stopped, the indexing has to be created from the beginning. Moreover, RAMDirectory uses more memory while searching than index storing in the disk, because RAMDirectory loads all the index inside memory, while disk pays only I/O buffer.

Generally, the core elements of Lucene are terms, fields, documents, and segments. The Lucene index has a straight-forward structure of saving indexes, where all the core elements are related to each other. A Lucene index is built by one or more segments, and each segment consists one or more documents. Likewise, each document has one or more fields, where each field consists by one or more terms, and each term is a pair of a field name and values. Segments refers to inverted index blocks which are merged to obtain the final index.

The collection of documents used in this project are html documents, thus those files have to be parsed in plain text, extracting the title and contents which will be the fields used in indexing. So to parse those file It is used an external library, i.e. Tika library. After Parsing each file, it will be created a Lucene document, where documents refer by Document ID and consists with 2 fields

title and contents. However, there is an important step on term-processing. The term-processing into tokens, case normalization into lowercase, stemming and removal of stop-words are done by analyzer, which is an abstract class in Lucene API and is discussed in more detail in the section 2.3. Moreover, for each term it is created a posting list of terms in which documents it appears, including positions of the term in the document, to make the term-based search more efficient and faster. In information Retrieval this feature is called inverted index.

Index Writer class takes as arguments Index Write Config, which it is configured to deal with all the steps abovementioned, and start storing the indexes to memory. The following sections all those steps will explained in details.

2.2 Index Writer

Index Writer is the Lucene class which is used to write indexing into the disk. Index writer takes as arguments the index directory and Index Writer Config(IWC). IWC configures the properties of the Index Writer, taking the analyzer as the argument and defining the mode operation. The mode operation has to choices, Create and Create_or_Append, which instructs the Index Writer to create new segments for indexing or to append to current segments. Initially, Index Writer store a fix number of documents to the memory before creating the segments to store in disk. The default value in 10 documents, but to optimize the performance of writing speed, forceMerge function can be used. This function takes as argument an integer to specify how many documents will be stored in memory before creating the segment. Once the number of the documents is reached in memory, segment is built and written to index. Lucene index need to merge all the segments to obtain the final index. So the strategy followed by Lucene is that every 10 segments are added to index they will be merged to a single large segment. So at the end there will not be more than 9 segments of power 10 index size.

2.3 Analyzers

As I stated above Analyzer is the abstract class in Lucene which is used for tokenizing, stemming, stop-word removal, token filters and case normalization. So each token represents a

word in the text. Moreover, here you can apply stemming to derive roots of words, or singular and plural forms of a word. Lucene offers many features to use in the user needs. There are 4 useful implementations, which differs from each other in how they brake the characters' stream into tokens. An example how those analyzer works will be given below while explaining them. The sentence for analysis example is "The email address in the university of Antwerp is ali.doku@student.uantwerpen.be". The 4 analyzers are:

1. Standard Analyzers

   This Analyzer is the most powerful and common used comparing with other analyzers. It can handle addresses, email addresses, names phone numbers etc. It implements stop words removal, punctuations and lowercase tokens.

   Example: [email] [address] [university] [antwerp] [is] [ali.doku@student.uantwerpen.be]

2. White Space Analyzer

   White space analyzer it splits the text document into tokens based on whitespace. This is not efficient, because it includes the stop-word removal which effect in the scoring result. But still has advantage that does not split words which includes non-letter characters like email address.

   Example: [The] [email] [address] [in] [the] [university] [of] [Antwerp] [is] [ali.doku@student.uantwerpen.be]

3. Stop Analyzer

   Stop analyzer as the name itself it deals with removal of common words in English language. The advantage is that removing stop-words in the text increase the performance, but this is not a good choice if the text includes email addresses or any other that include non-letter characters.

   Example: [email] [address] [university] [antwerp] [is] [ali] [doku] [student] [uantwerpen][be]

4. Simple Analyzer

Simple Analyzer works in the same way like Stop Analyzer, splitting the text base on non-letter characters, lowercasing the tokens.

Example: [The] [email] [address] [in] [the] [university] [of] [antwerp] [is] [ali] [doku] [student] [uantwerpen] [be]

2.3 Tika Library

Tika is an open-source library used for search engine indexing to parse different binary files types such as PDF, XML, html. As in this project are being used html files, Tika API comes in our help. Tika uses html tags such as <p>, <title>, <author> etc to extracts metadata and text from html files. In addition, Tika read and extract data in memory, using a minimal amount of RAM memory. So after all documents are proceed, it is able to output the information in the fields which are stated above to build documents and then creating the index.

3 Lucene File Searching

As Indexing is the most important step in building such a system, searching process is the core functionality of this system. This section will explain how Lucene searching works, how the query given is worked out, and how the scoring is done retrieving the most scored documents to the user. Lucene searching offers a lot of variety of building a query like Boolean queries (AND, NOT, OR), searching in all the fields built in documents called multi field queries (title: university content: university, title: antwerp contents: antwerp), wildcard(universi*), fuzzy and range queries. All those query types influence in the scoring documents, based on conditions that each query applies which will be explain later.

Lucene to start searching, first it should be defined the query parser. Parsing means to convert a user query built by some text, to a Query object which tells Lucene what to look for at the index. But also here it analyzers are present again. The text query given should first be proceed by an analyzer for tokenizing, normalization and stop-word removal steps. The choice of an analyzer influences a lot searching and scoring, because the query is broken into terms in different ways

by different analyzers as it is explained in section 2.3. then it starts the parsing process, defining the fields, what to search for. For example, if you search (University + Antwerp), this is a Boolean query searching for two terms that must appear in the document. The default value of Lucene is OR Boolean operator. As in our case we have only 2 fields, if field title is selected, there it will be search only the documents which has those 2 words in the title. To perform searching there are two classes Index Reader and Searcher, once the query is parsed it reads the index in the directory given and search for the terms. However, all those processes, starting from building queries, reading and searching in the index, scoring documents and checking giving suggestion for a query are explained in the following sections.

3.1 Building an Index Reader and Searcher

The Index Reader and Index Searcher are 2 classes in Lucene which perform the searching in the system. Firstly, we initialize an Index Searcher, which takes as the arguments the index directory and an Index Reader. To perform the searching, Index searcher takes the query and lets the Index reader first to access the index and reads them, then it is performed the searching. As we stated the data are stored as an inverted index, so the search is performed by looking in the inverted index of every term, applying scoring formula of Lucene for every document and returning the top scored documents corresponding the query given.

3.2 Lucene query Parser

As we know the queries given by the user are in human-readable textual form so the system cannot understand how to deal with the query. So Query parser class is the key of transforming the textual query in such a query in order the system understand it and satisfy the user expectations. Query parser also have some features such as allowing some query type. Some query types in information retrieval are:

- Boolean query

  Boolean queries are built using the Boolean operators like NOT, AND, OR. So using those operators to construct such queries affects the scoring. For example, let’s consider the query

"University+Antwerp" and perform searching in the field title. The documents which are retrieved are only document which include both words in the title, while using OR (-) will be retrieved the documents which include one of them in the title.

- Fuzzy Query

Fuzzy queries are queries using a tilde (~), represent a float number, which is the minimum similarity to a term in the index dictionary. The Fuzzy query use Levenshtein distance to calculate the similarity of a term to another term. So for example, if the user searches for "compter", the most likely the user is looking for query "computer". So Fuzzy queries gives a suggestion of the nearest term calculation by levenshtein distance.

- Wildcard query

Wildcard queries is the technique of using a wildcard to complete a query. The sign '*' tells lucene to search for a wildcard. For example, if the user is searching for the query "universi*" the wildcard query in this case will return all the documents which include in its fields the words starting with universi.

- Range query

Range queries are the queries to search in numeric range of numbers or dates. For example, if you are searching for a car and the production year varies from 2003 to 2006, the query will look like [2003, 2006].

However, in my system the types of queries implemented are Boolean query, wildcard query and fuzzy query. The range query it is not implemented because the target documents are html documents, which don't include a field like date of publication in order to search according a date range. The Wildcard queries are called automatically if the documents number scored is 0. This feature is implemented at least to return to user some documents, which maybe are the right ones he is looking for. After explaining the query type let's move to Query Parser. There are many types of query parser in Lucene, but I have implemented only 2 of them, which are:

- Multi Field Query Parser

Multi Field Query Parser parse a query in such way to perform the search in all the fields of a document. So This parser divides the whole textual query into single words, and parse query to every field. For example, if the query is “University of Antwerp”, the parse result of this query will be “(title: university contents: university) (title: antwerp contents: antwerp).

- Query Parser

Query Parser runs differently from Multi Field Query Parser. Query parser perform the parsing for only one field once. Similarly, Query Parser divides the textual query into single words as Multi Field Query Parser do, but parse those single words for only one field. For example, if the query given is “University of Antwerp”, the parse result of this query will be “title: university title: antwerp”. So Generally, it will be search in every documents which include in the title field one of this words.

3.3 Scoring Documents

After all the above mentioned steps have finished, it is time to return scoring documents to the user. But how the scoring process is done?

Lucene combines the Boolean Model and Vector Spaces Model of IR. As we learnt in this course documents and queries are represented as weighted vectors, where weights are tf-idf values and index term is a dimension. Vector Space Model calculate the tf-idf, where tf states for term frequency, which means the number of occurrences of a term in a document, and idf states for inverse document frequency which is the number of documents containing term t. Moreover, for Normalization, the cosine similarity of a query and a document it is used.

Lucene Scoring formula is:

Score (q, d) = coord(q, d) * queryNorm(q) * ∑(tf(t in d) * idf(t)^2 * t.getBoost() * norm(t,d))

Coord(q, d) is the number of query terms are found in the specified document.

queryNorm(q) is a normalizing factor to scores between queries. But this means that all document will have the same factor as it is the same query, so this will not affect the scoring result.

t.getBoost boost factor value assigned during indexing.

norm(t,d) = doc.getboost() * lengthNorm * f.getBoost()

doc.getboost() - the float value assign to the document before indexing.

LengthNorm – document is inserted to the index with number of tokens of the field in the document. As shorter will be fields gets a higher boost and a higher scoring.

f.getBoost() – the float value assign to the field before indexing.

Boosting is the process of assigning a bost factor value to a field or document, when added in the index. This can bring some benefits to the application like storage efficiency, but the system can work well without doing this step.

3.4 Spell checker and suggestion

Spell checker and giving suggestion is another feature implemented in this application. Spell checker class builds a Lucene dictionary, which is based on the term indexed. So whenever a user gives a query the spellchecker class will compare every single word of the query to see if this term appears in the dictionary. If this term doesn't appear, it will give a suggestion of a term, calculating the Levenshtein distance and taking the term with the minimum distance for the term given.

4 Evaluation of Results

This section is about evaluation and testing of the application in the following ways:

4.1 Memory Indexing usage

This test evaluates the computational time and memory used for building the inverted index, saving the term, merging the segments and the total memory used for index files. Firstly, it is

important to note that Lucene performance depends to computer architecture, the efficiency of the hardware of a machine. We have exactly 118376 documents in our index. According Lucene documentation, Lucene use nearly 20 – 30 % of the total size of documents indexed. So in our application the total size is 2.77 GB size and the size used to index is nearly 500 MB (18 %), but this depends from analyzers as table 1 shows. However, the merge factor explained in section 2.1 is the key for improving the performance of indexing. The default value used by Lucene is 10 documents for a segment. As the table shows incrementing the value of merge factor to 50, reduces the time of indexing, which is better for performance of the system. This comes due to segments are merging rarely and I/O operations are less. Moreover, we see that Stop Analyzer and Standard Analyzer are using less memory comparing with others due to stop-word removal. As I stated in the sections above that Lucene uses Ram memory to read and index documents till it equals the merge factor value, I tried to increase the Ram size used by Lucene from 16 MB to 48MB, which is the maximum buffer that Lucene can use. This test showed that 16MB Ram size was more efficient in time indexing than using 48MB Ram. The time taken to index all the files with 16MB and 48MB is respectively, 2604 seconds and 2414 seconds. So having more space in Ram directory means more efficiency.

	Memory used	*Time indexing*	*Terms count*	*Merge Factor*	*RAM Size*
Standard	495Mb	2604414ms	1731977	10	16MB
Analyzer		1622492ms		50	16MB
Stop Analyzer	479 MB	2319397ms	1352241	10	16MB
		1844907ms		50	16MB
White Space	541 MB	3713268ms	3721145	10	16MB
Analyzer		2985226ms		50	16MB
Simple	503 MB	2894277ms	1352306	10	16MB
Analyzer		2163356ms		50	16MB

Table 1. the Evaluation values for file indexing

4.2 Result of documents scoring

This subsection evaluates the result of documents scoring. Generally, we gave the formula and factors involved in this formula which effects the scoring in section 3.3. As the Lucene is using Vector space model, the scoring is done through term frequency and inverse document frequency.

However, there are other internal factors involved in the performance of result, which are the query type, query parsers, what kind of analyzer is used for indexing and searching, Improvements in Index Searcher and Spell checker and suggestion. The table 2 show the difference using different query type and analyzer. As we can see from the table the White Space Analyzer is less efficient, and Simple and Stop Analyzer are the winners.

However, Spellchecker and suggestion plays a big role in scoring the documents. if a user enters a query, which doesn't score any document, the wildcard query will be activated automatically and a suggestion is given. The reason why a wildcard query is implemented in this step, is because to be returned some documents, which are nearly similar to what user is looking for. While suggestions are given from Lucene dictionary. Lucene Dictionary is built by term indexed. This suggestion tool calculates the similarity using Levenshtein string distance, and return the word with minimum distance. For example, when a user search for a document title "belgiu", the number of documents returned is 0, so the query will be transformed in a wildcard query "belgiu*" and will give suggestion the word "belgium". But, it is important to note that using suggestion tool, affect the performance, making the system slower than usual because, first Index Reader have to read the index in order to build the dictionary, then calculate the levenshtein distance between query term and dictionary. So there is a lot of background processes due to the fact there more than 1 million terms. In order to test the result of scoring of my applications, I will use Luke library, which is a ready Tool for searching the documents based on indexing.

Query	Standard Analyzer		Simple Analyzer		Stop Analyzer		White Space Analyzer	
	Time to process	Number of results	*Time to process*	Number of results	*Time to process*	Number of results	*Time to process*	Number of results
Class OR computer	2.899	15533	2.667	15562	2.571	15562	5.071	6595
Class AND computer	2.923	609	0.27	613	1.71	613	4.177	253
Class* computer*	4.701	23507	4.897	23518	4.331	23518	5.409	17763
class* AND computer*	3.336	1430	2.259	1430	2.067	1430	4.76	878
Class Not computer	2.777	11147	2.368	11157	2.256	11157	4.511	3737

Table 2. Illustration of query example using different analyzers.

4.3 Evaluation of searching speed

Besides result, in a search engine system the speed is also very important. First of all, a big player of having a good performance in retrieving the result faster are the hardware and a faster I/O system. Because the index files are stored in disk, there will be a lot I/O request to retrieve information. However, there are many features to implement, which indicate in speed performance. The first one is using the Index Reader in readOnly mode because this help the multi-threading to cooperate better with each other as they are sharing the same reader. Moreover, the mergeFactor is again a key to increase performance. Using a mergeFactor as smaller as possible, will brings faster result as there will be fewer segments, which Index reader have to read. Another trick is to do not iterate through all the results. Retrieving all the result in one shoot, the search engine will be very slow. So that's the benefit of pagination. And Lastly Caching result. To perform quickly, Lucene use the cache of the operating system in order to speed up the I/O operations. Moreover, when an Index Reader is created the term indexes are loaded into RAM. So this will minimize the I/O to the disk for every term in the query and will

speed up the searching process. For example, in first attempt when searching for query “hello world”, it returns the result in 11.595 seconds, while in second attempt 3.767.

5 Conclusion

After building and analyzing the system I end up with the following conclusions:

- Building a perfect search engine is very challenging due to lot of techniques to be crosslinked cooperation to each other to increase the performance.
- Building a search engine system in JAVA programming language is not suitable, due the fact that JAVA works based on objects and spends a lot of memory. Moreover, JAVA language runs in JMV (Java Virtual Machine), which also requires a few time to process things.
- Using Standard Analyzer help with the efficiency and performance in indexing and searching result, due to stop-words removal, and is more useful in cases of searching email addresses as it doesn’t separate it according non-letter characters as other analyzers do.
- The indexing process is very time consuming due to pre-processing of strings, building indexes for fields and segments, merging the segments with each other. Therefore, it is preferable to use disk to store the indexes, not RAM. Because RAM will require execution every time the system will start and this is not suitable.
- To optimize the searching quality, different type of queries should be implemented and also defining the correct query parser, in order to perform searching based on the field you are looking for. So for example, using Multi Parser Query to look in some different field in order to have a more result. Or using Query parser if user is interested for title, using field title, which is stored in index.
- To help the user find the information looking for, word suggestion should be implemented, but it is very time consuming due to more than 1 million terms stored in index.

- The merge factor value 10 has proven to be more effective in searching speed, but not in indexing performance. But searching is more important. Indexing is being run once and saved in disk, while searching will be used by user anytime.
- Using operating system cache, Lucene searching becomes more efficient and faster, because indexes retrieved from disk, are temporarily stored in cache and Ram directory, which decrease the I/O to the disk.

6. References

6.1. Indexing and Searching

- Sridevi Addagada.(2007). Indexing and Searching Document Collection using Lucene. University of New Orleans Theses and Dissertations.
- Michael McCandless, Erik Hatcher, Otis Gospodnetic (2010). Lucene in action, second edition. ISBN-13: 978-1932394283
- Lucene tutorial https://www.tutorialspoint.com/lucene/
- Lucene Similarity and Scoring https://lucene.apache.org/core/3_5_0/api/core/apache/lucene/search/Similarity/

6.1. Library API Documentation

- Apacheorg (2016). Apache Lucene API. https://lucene.apache.org/core/
- Apache Software foundation. Apache Tika. http://tika.apache.org/index.html
- Luke – Lucene Index Toolbox https://github.com/DmitryKey/luke/releases/