VRIJE
UNIVERSITEIT
BRUSSEL

PROJECT REPORT

Group 3

Arno De Witte (0500504)
Douglas Horemans (500239)
Wout Van Riel (500229)

January 8, 2017

1 Overview Aspects

Below of all the assigned aspects for our group. All aspects were implemented.

Number	Letter	Aspect
1	a	Implement full boolean querying supporting AND, OR and NOT.
2	b	Include the use of soundex to deal with misspellings and improve recall. Demonstrate the improvement
3	b	Build and use B-tree over the dictionary to deal with * wild-cards.
4	a	Implement cosine ranking, using a heap to get the top k ranked documents.
5	a	Implement cosine ranking with high-idf query terms only for optimization of top-k ranking. Demonstrate the improvement.
6	c	Implement and demonstrate the vector space model for XML information retrieval.

2 Technologies

To solve almost all of the aspects the Apache Lucene[1] was used. Lucene is a Java library which provides a lot of information retrieval functionalities out of the box. However because of its large API, finding the right way to do certain tasks becomes more complicated as none of the team members were familiar with the library.
For the soundex aspect, the Apache Commons codec[2] libary was used as it provides a soundex codec.
To parse the HTML documents to text JTidy[3] was used.
The project is written as a java application. The project can be compiled by adding the *lib* folder to the classpath. To use the system to search, first an index should be created as follows:

```
java -cp "classpath/:lib/*" Main index ./articles ./indexTarget
```

To search see the listing 3.6.

3 Aspects

3.1 Boolean querying

Lucene provides support for boolean queries. To implement this an index should be created and then be searched. The default implementation of lucene already supports these boolean queries. The syntax[4] for these queries is rather trivial. By default leaving spaces between query terms implies an OR-relation between these terms. Using the term *AND* implies an AND-relation between terms and using the term *NOT* implies a NOT-relation with the following term.
To query the index you can use the *search* command on the application as follows:

```
java -cp "classpath/:lib/*" Main search `car AND NOT bike' ./index
```

To test this implementation we used the small wikipedia dataset[5]. The implementation was tested by searching with different query combinations and manually checking the resulting documents. While testing with the *auto* query term, the system showed results that had *auto* in their HTML syntax. Therefor a HTML parser was implemented to prevent this.

[1] http://lucene.apache.org/core/
[2] https://commons.apache.org/proper/commons-codec/
[3] http://jtidy.sourceforge.net/
[4] http://lucene.apache.org/core/6_3_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package.description
[5] http://www.search-engines-book.com/collections//

3.2 Soundex

To implement a soundex, the default analyzer, which converts text into terms, had to be changed. An analyzer was implemented that filters a standard tokenstream to a soundex tokenstream. This is done in the java method *getSoundexAnalyzer*. This analyzer is then used by lucene's *IndexWriter* and *IndexReader* classes to convert the documents and queries to soundex alphabet. Because we need a new index with soundex terms the index command should be ran again, but a *true* should be added as argument. The same has to be done when searching. It is recommended to also change the directory of the index to prevent the other index to be overwritten.
To illustrate the implementation of the soundex we used one of the example of the slides. We searched *Herman* and also *Herman**n***. Both queries resulted in the same output (the one listed below). You can see that the first result is about *Harman* which is phonetically close to *Herman*.

Listing 1: High threshold

```
1. 0.43999153 -> /home/arno/workspace/informationRetrieval/project/articles/h/a/r/Harman's_Cross_railway_station_0dae.html
 Title: Harman's Cross railway station - Wikipedia, the free encyclopedia
2. 0.40165547 -> /home/arno/workspace/informationRetrieval/project/articles/o/c/_/OC_Times_3f4f.html
 Title: OC Times - Wikipedia, the free encyclopedia
3. 0.39354038 -> /home/arno/workspace/informationRetrieval/project/articles/d/e/s/Deshengmen.html
 Title: Deshengmen - Wikipedia, the free encyclopedia
4. 0.34081593 -> /home/arno/workspace/informationRetrieval/project/articles/j/o/h/Johann_Hermann_Bauer_4adc.html
 Title: Johann Hermann Bauer - Wikipedia, the free encyclopedia
5. 0.3279503 -> /home/arno/workspace/informationRetrieval/project/articles/o/n/_/On_the_Road_Again_(Canned_Heat_song)_ed5e.html
 Title: On the Road Again (Canned Heat song) - Wikipedia, the free encyclopedia
```

3.3 B-tree wildcard

We had to build a B-tree over the dictionary to deal with * wild-cards. To do so we first searched for an implementation of B-tree in Java. The implementation we based our code on is provided by the textbook "Algorithms, 4th Edition" written by two professors of the Department of Computer Science at the Princeton university, Robert Sedgewick and Kevin Wayne. The implementation has been modified in multiple ways to be used as described in the slides.
First, to be able to work with wildcards, the 'biggest' attribute has been added to every node in the three. This attribute represents the biggest key in the subtree of the given node. The smallest value we already knew because it is the value of the first child of the given node, due to the structure of a B-tree. With the smallest and biggest value of the subtree of every node available, we can easily know which children it is worth to investigate.
Secondly, to be sure the 'biggest' attribute always stays up to date, some functions have been modified. As an example the split function has been modified to update the 'biggest' attribute of a node when it is split in two.
Thirdly, we had to change the BTree values to Lucene PostingsEnum. Why is explained further.

For the first part of the assignment we had to implement a full Boolean querying mechanism. For this, all the articles were indexed and written to a file. This means we already have a file containing all the terms of the documents (thus already normalized etc) and for every term its postingslist. For this part of the assignment the goal is to make it possible to query with wild-cards by making a BTree dictionary.
The dictionary has to store every term and a pointer to the associated postingslist list of that term. Because we are using Java the terms are stored in an efficient way because a Java string is an array of characters of dynamic length. Of course this could be even better with compression as described in the course. But this is difficult to do in Java because we do not manage the memory, the JVM does this. For the postingslist we decided to use a Lucene data-structure, PostingsEnum. PostingsEnum represents an enumeration of postings were we able to iterate over. Because we are using Java it stores a reference to that PostingsEnum, but again we can't

manage or optimise the storage of this reference because the JVM does it. But what we know is that one of the advantages of Lucene is that it promises a "small RAM requirement – only 1MB heap". Thanks to this, we think that this is an efficient way to store our key-value pairs. Also this approach does not make the heap explode with our test scenario of 221023 terms as other approaches does.
The biggest memory usage comes from the fact that when using BTrees for wildcard queries two different BTrees should be stored as described further.

We have created a dictionary class that relies on two BTrees to be able to process wildcards. A dictionary can do two things, add and get.
The add function will add the given Key-Value pair, which is a term-PostingList pair, to the dictionary.
The get function takes a wildcard String as input, and processes it. Three wildcard types are applicable to our dictionary, wildcard at the beginning ("*a"), wildcard at the end ("a*") and wildcard in the middle ("a*b")

To be able to handle the wildcard queries with the wildcard at the beginning or in the middle, a dictionary contains two BTrees as already mentioned. The first dictionary contains the normal terms. And the second contains the reversed terms. This way we can use the first BTree for the queries of type "a*" and the second for queries of type "*a". For the queries of the last type we just need to intersect the results of the query "a*" and "*b".

3.4 Cosine ranking

Lucene's default ranking system is already based on a cosine ranking. It implements a *TFIDFSimilarity* abstract class which provides the possibility to do cosine ranking (in lucene ranking is called scoring). When indexing lucene will calculate the inverse document frequency for each term by default.
To implement this aspect we thus based ourselves on the default ranking of lucene. We modified it because it has some additional features which are not relevant for our system. For example we removed the overlap discount. Implementation details can be found in the *Cosine* class and an explanation of each method can be found in the lucene documentation[6].

3.5 High idf query optimization

This optimization for the queries was rather difficult to implement. First because of the way the queries are parsed in lucene. Lucene creates immutable *BooleanQuery* objects when using multiple terms in a query. Therefor each object has to recreated when going over each term to check it has an high idf. The Cosine class previously created was used to check the idf value of each term, terms below a threshold were filtered.
When testing this implemenation a problem was found. The QueryParser of lucene already filters stopwords[7] such as *and*, *this*, *the*... These are the typical low idf value words we want to filter. To test the results we increased the threshold to 4 (which is already quite high) and compared

[6] http://lucene.apache.org/core/6_3_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

[7] http://lucene.apache.org/core/6_3_0/core/org/apache/lucene/analysis/package-summary.html#package.description

the results:

Listing 2: High threshold

```
1. 0.9303905 -> articles/c/h/a/Charlie_Kelly_c049.html
 Title: Charlie Kelly - Wikipedia, the free encyclopedia
2. 0.63305056 -> articles/d/u/c/Ducati_Multistrada_1162.html
 Title: Ducati Multistrada - Wikipedia, the free encyclopedia
3. 0.41350687 -> articles/g/u/i/Guido_Leoni_5682.html
 Title: Guido Leoni - Wikipedia, the free encyclopedia
4. 0.36549187 -> articles/l/a/k/Lake_Harriet_cf88.html
 Title: Lake Harriet - Wikipedia, the free encyclopedia
5. 0.3618185 -> articles/d/a/r/Top_Gear~_Dare_Devil_fe12.html
 Title: Top Gear: Dare Devil - Wikipedia, the free encyclopedia
```

Listing 3: Low threshold

```
1. 0.624157 -> articles/d/u/c/Ducati_Multistrada_1162.html
 Title: Ducati Multistrada - Wikipedia, the free encyclopedia
2. 0.3456331 -> articles/m/i/c/Michel_Pollentier_0f6f.html
 Title: Michel Pollentier - Wikipedia, the free encyclopedia
```

These results were gathered using the query *bike AND because*. The word *because* has a low idf value and is this filtered when we set the threshold to 4. This will generate more results as we only use the term *bike* for searching.

3.6 XML retrieval

For the sixth aspect we had to implement the vector space model for XML information retrieval. Because XML is a standard for encoding structured documents, the indexing of the files and searching through the files had to be adapted for these documents. To work with the data, we represent the path from the root to the leaf in a XML-tree as the pair $< c, t >$ where c is the XML-context and t is the vocabulary term. If a leaf consists of text, every word in the text is seen as a vocabulary term and will become a pair with the XML-context. The context is not split in smaller pieces or shortened before it is indexed to keep it simple and not lose any information. This however will have implications on the scoring because deeper nested vocabulary terms will be scored lower by the context resemblance. To make it work with Lucene can be seen as a challenge because Lucene is intended to be used with unstructured data. We will represent this pairs as fields where the vocabulary term is the fields name and the context is its value. For the retrieval part, a custom scorer *SimNoMerge* was created that could work with the XML-context. The scoring function is $\sum_{c_d \in d, c_q \in q} \frac{weight(d,t,c_d) * Cr(c_q,c_d)}{normalizer}$ where the weight function is the inverse document frequence (idf) times the term frequency weighting (wf). Cr is the context resemblance function and checks if c_q can be transformed into c_d with only additions. If this is the case, the function will return $\frac{\|c_q\|+1}{\|c_d\|+1}$. The scoring function will go over every document that is returned by the query, in our case the documents that contain fields with a name that equals the vocabulary term, and return a score for every document. Because Lucene scores all the documents one by one, it is not possible to implement the normalization function that is suggested in the book. The value used for the normalization in our implementation, is the weighted term frequency of all terms that have given vocabulary term. The weight function for queries is not implemented because our implementation can only process queries that contain one path with following syntax: *xmlcontext#vocabularyterm*. The following commands can be used to work with the XML retrieval:

```
#for indexing
java -cp "classpath/:lib/*" Main XML ./pages/to/index ./index

#for searching
java -cp "classpath/:lib/*" Main searchXML `context#vocterm' ./index
```

The data used to test the XML retrieval was a part of the INEX 2009 collection[8]. As an example we index directory 000 of the INEX 2009 collection. We then searched for the vocabulary term *#belgian* only. Nine XML documents are then returned that contain the word *belgian* as vocabulary term.

Listing 4: Search results for #belgian

```
1. 2.2662091 -> pages/000/1000.xml
2. 0.70147693 -> pages/000/8805000.xml
3. 0.4171691 -> pages/000/11538000.xml
4. 0.36837515 -> pages/000/17423000.xml
5. 0.20538285 -> pages/000/16049000.xml
6. 0.09226496 -> pages/000/368000.xml
7. 0.061509967 -> pages/000/7000.xml
8. 0.055358972 -> pages/000/16197000.xml
9. 0.05032634 -> pages/000/994000.xml
```

As you can see the first result has a value that is bigger than one. This means that our scoring function is not a true cosine. Then if we add a small path to it, for example *nationality#belgian*, only the files that contain a node *nationality* and the vocabulary term *belgian* are returned. In our case:

Listing 5: Search results for nationality#belgian

```
1. 0.11225033 -> pages/000/1000.xml
```

Only one file of the nine remains because none of the other files have a node *nationality*. The resulting score is also much lower as previous because the vocabulary term alone is more present in the file than *nationality#belgian*.

4 Workload

Below an overview of how the workload was distributed for this project. Note that due to the available API's of Lucene some tasks were easier than others. The XML retrieval and BTree aspects were more complicated because a lack of support in lucene. We think that the workload was evenly distributed.

Task	**Member**
Aspect 1	Arno De Witte
Aspect 2	Arno De Witte
Aspect 3	Douglas Horemans
Aspect 4	Arno De Witte
Aspect 5	Arno De Witte
Aspect 6	Wout Van Riel
Report	Group effort

t

[8] http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/software/inex/