Project Report

Project Title:

Evaluation of Standard Information retrieval system related to specific queries

Submitted by:

Sindhu Hosamane Thippeswamy

Information and Media Technologies

Matriculation number :46442

sindhu.hosamane@tu-harburg.de

ACKNOWLEDGEMENT

My first experience of project has been successfully, thanks to the support of University with gratitude. I wish to acknowledge all of them. However, I wish to make special mention of the following.

I take this opportunity to express my profound gratitude and deep regards to my guide Prof. Ralf Möller for his exemplary guidance, monitoring and constant encouragement throughout the course of this project.

I must make special mention of Sylvia Melzer our project supervisor for giving us her valuable time & attention & for providing us a systematic way for completing our project.

I am obliged to staff members of University, for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my project.

Sindhu Hosamane

Abstract

The aim of this project is about measuring the effectiveness of standard Information Retrieval systems. The standard approach to information retrieval system evaluation revolves around the notion of relevant and non-relevant documents. Basic measures for information retrieval effectiveness that are standardly used for document retrieval are Precision and Recall. So using standard information retrieval systems we define specific queries and then answer these specific queries. Measure precision and recall values with the standard information retrieval systems. For these experiments we use the Boemie Repository.

Contents

Figures

Tables

Chapter 1

INTRODUCTION

1.1 Motivation

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).[1]

There are many Information retrieval systems. Information retrieval has developed as a highly empirical discipline, requiring careful and thorough evaluation to demonstrate the superior performance of different novel techniques used by IR systems on representative document collections.

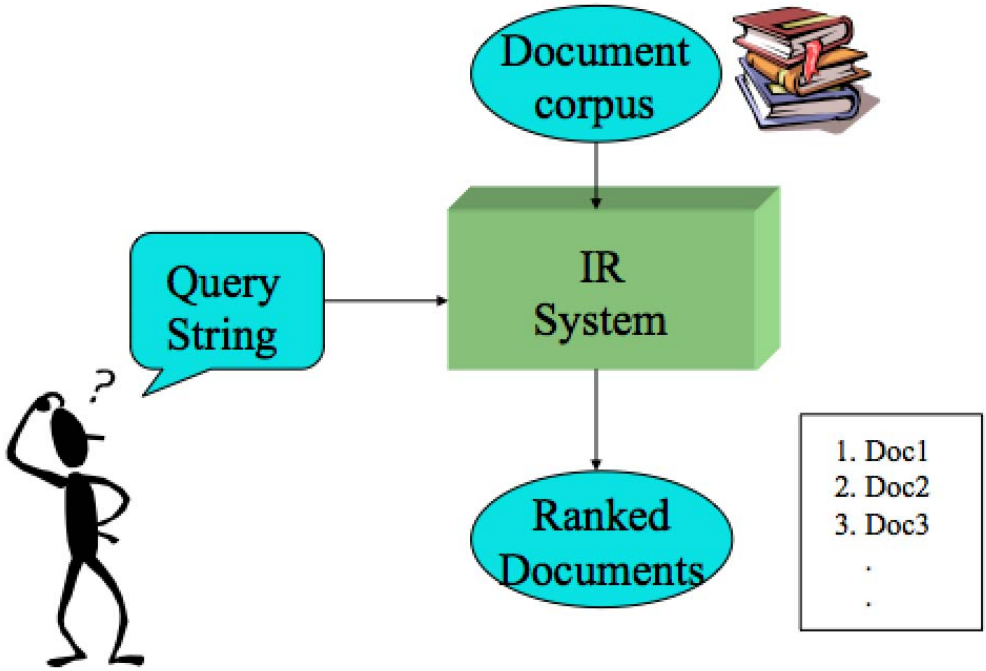


Fig 1: Typical Information Retrieval task

The group of documents over which information retrieval is performed is called as collection. It is also called as Document corpus. User submits query. Query is what the user conveys to the computer in an attempt to communicate the information need. Information need is the topic about which the user desires to know more. Documents retrieved by IR systems are arranged according to their relevance to a given search query. A document is relevant if it is one that the user perceives as containing information of value with respect to their personal information need.

There exist many IR systems like Google, Bing, MSN and many more. But the performance and quality are not known. The main aim of this work is to measure the effectiveness of search engines.

1.2 Requirements

The requirements of the project are:

a) Boemie Repository

b) Standard Information Retrieval Systems

c) Queries.

a) Boemie Repository:

Boemie repository is a multimedia repository. Multimedia documents can be text, images and video. This is public and available at http://repository.boemie.org/BoemieRepository

It contains 693 multimedia documents (web pages). These Web pages concern the collection of information about the domain of athletic events that includes concepts tournaments, meetings, training, athletes, persons, faces, etc. In order to make multimedia content like videos or images searchable the data must be meaningfully annotated. Humans commonly do this, but it is a hard and expensive task. Using sophisticated algorithms to extract semantics from multimedia content, BOEMIE annotates content with semantics automatically and provides valuable knowledge for both, content providers and content consumers.

Symbolic and holistic content descriptions represent multimedia documents such as texts, images, and videos. Symbolic content descriptions are symbolic representations that use formal languages, such as description logics (DLs). They consist of Aboxes, which contain

instances and atomic concepts. These Aboxes are the result of an analysis process and an interpretation process.

Boemie repository contains documents in their raw data format and their symbolic representation in RDF format. A symbolic content description of a multimedia object is a symbol. Hence symbolic content descriptions are characterized as atomic and static.

b) Standard information retrieval systems:

The great explosion of Internet and electronic data repositories has brought lots of data in our reach. Day by day data is increasing exponentially, not only on Internet but also on desktop computers. It is very impractical to look for a picture or an audio file out of many folders and subfolders on a desktop. Although user can classify data, crawling through many categories and subcategories of data, which is not an efficient way. There is a need for alternate and dynamic ways of finding information. That's where Information retrieval systems come into existence.

Information retrieval system finds material of an unstructured nature that satisfies an information need of user from within large collections.

c) Queries:

User defines queries to IR systems. Query is what the user conveys to the computer in an attempt to communicate the information need. Queries have an influence on matching result. Queries must match with content description of multimedia documents.

1.3 Outline of the project

This project is structured as follows.

Chapter 2 introduces Lucene and its features. It also explains indexing and searching features of Lucene.

Chapter 3 describes the index structure of Lucene.

Chapter 4 describes the experiment on Boemie repository with Lucene.

Chapter 5 Results are evaluated.

Chapter 6 Finally the project is concluded.

Chapter 2

LUCENE

2.1 About Lucene

Apache Lucene is a full featured, high performance, scalable text search engine library. It is written in java. Lucene has been recognized for its utility in implementation of Internet search engine. It provides full text indexing and searching. It was originally written by Doug Cutting. It belongs to Apache Jakarta family of projects.

Lucene is not a complete application and so it is not ready to use application like web search engine, rather it is code library that can be easily integrated into any application. It is popular because of its simplicity. User need not know how it does the indexing and searching, rather just have to learn few classes of Lucene library for implementation.

There are many applications, which uses Lucene. For example

- Eclipse Lucene: Eclipse IDE uses for searching its documentation
- Nutch: open source web search engine
- Twitter Trends: Twitter analyzing tool
- 7digital: Digital media delivery Company
- Linked In, Apple, IBM and many more [3]

Lucene can be seen as layer above which sits the application as shown in fig 2 .

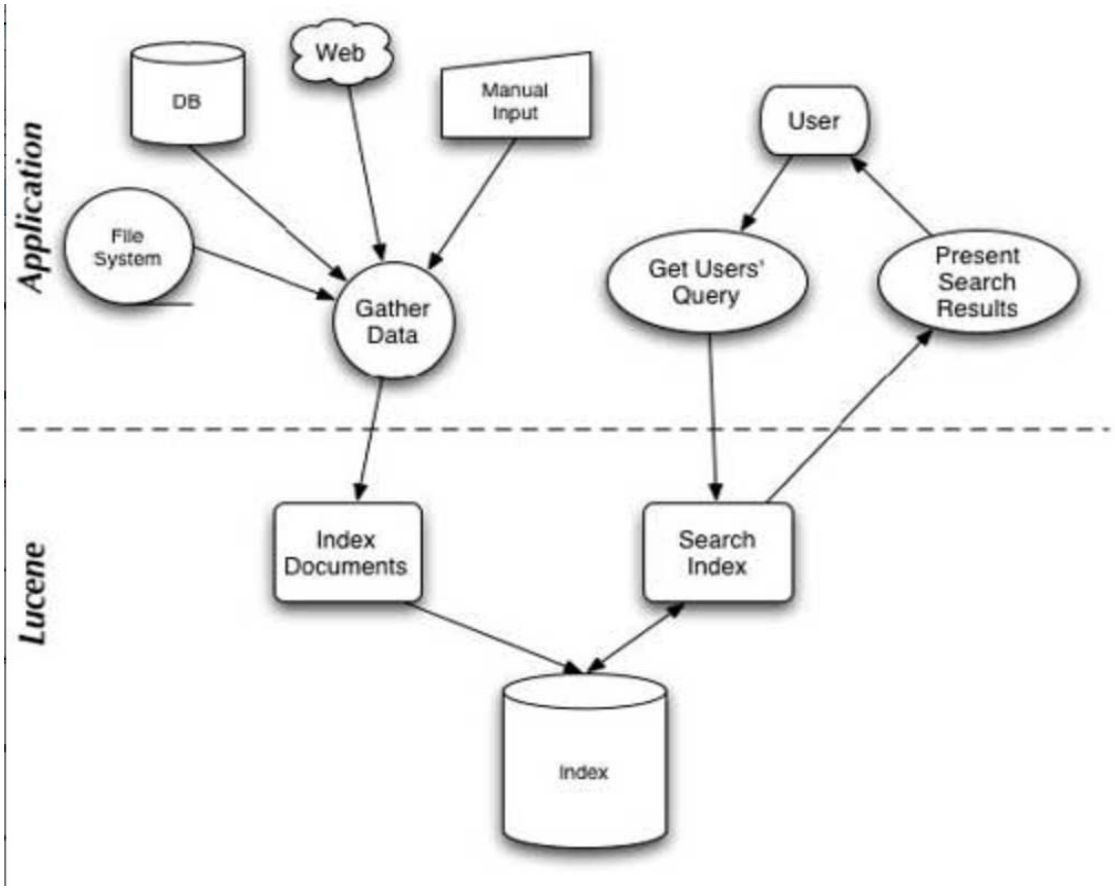


Fig 2: A typical application integration with Lucene [2]

As seen in above figure Lucene is not only used on web, but also on local File system, databases, manual input and many other sources of data. It can index and search any data that can be converted into text. It works on all data formats like html, pdf, txt, and Microsoft word. It gathers the data and indexes those documents. And later when user gives a query, it performs a search on index generated.

Lucene offers many features. It can provide ranked searching which means best results are returned first .It also provides fielded searching, i.e. user can search for content in title field, author or content field. It supports different types of queries like phrase queries, range queries, wildcard queries. Lucene also supports simultaneous searching and updating. It is highly flexible and scalable for any number of documents. It also does sorting, filtering, highlighting search results.

2.2 Lucene Indexing

Indexing is the center concept of any search engine. Indexing is the process of converting original data into an efficient lookup, which helps for rapid searching.
Suppose if we want to look for a file with a specific word, then we could have a program that sequentially scans for all the files and look for a file with a specific word. But this is quite impractical when the file set is large. Here comes the importance of indexing. In these cases, first the text must be indexed into a format, which helps for rapid searching and this process eliminates the slow scanning process and it is called indexing. And the output of it is called index. Index is a data structure, which facilitates rapid search for the words present in it.

Lucene converts any format of data to text and then indexes it. Lucene uses different parsers for different documents like HTML parser for html documents .HTML parser does some preprocessing by filtering html tags and so on. The parser outputs text content. Lucene Analyzer extracts the tokens and other related information and stores it in index files. Like html there are different parsers for pdf, Microsoft word and text files. Below figure shows the indexing architecture of Lucene.

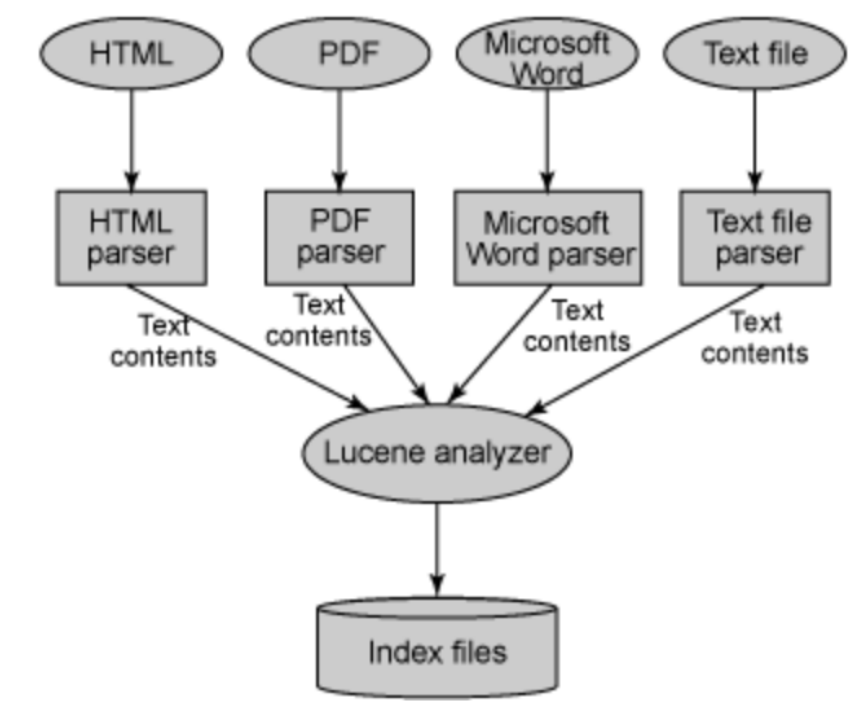


Fig 3: Indexing architecture of Lucene

2.2.1 Indexing example using Lucene:

```
public static void createIndex() throws
CorruptIndexException, LockObtainFailedException,
IOException {
          Analyzer analyzer = new StandardAnalyzer();
          boolean recreateIndexIfExists = true;
          IndexWriter indexWriter = new
IndexWriter(INDEX_DIRECTORY, analyzer,
recreateIndexIfExists);
          File dir = new File(FILES_TO_INDEX_DIRECTORY);
          File[] files = dir.listFiles();
          for (File file : files) {
               Document document = new Document();

               String path = file.getCanonicalPath();
               document.add(new Field(FIELD_PATH, path,
Field.Store.YES, Field.Index.UN_TOKENIZED));

               Reader reader = new FileReader(file);
               document.add(new Field(FIELD_CONTENTS,
reader));

               indexWriter.addDocument(document);
          }
          indexWriter.optimize();
          indexWriter.close();
     }
```

Indexer program needs two important command line arguments.

- A path to a directory where Lucene index is to be stored
- A path to a directory which contains files to be indexed

Running the above Indexer program will create a Lucene index. Indexer prints the names of files it indexes. It displays the total number of documents indexed and also time took in milliseconds. This includes time needed for directory traversal and time needed for indexing.

2.2.2 Indexing classes of Lucene:

Here are some indexing classes that are used for indexing process using lucene.

- IndexWriter
- Directory
- Analyzer
- Document
- Field

IndexWriter:

It is a central component of indexing process. It creates the index and also adds documents to the existing index.
It just gives the write access to the index, but does not allow reading or searching on index.

Directory:

Directory class represents the location of Lucene index. When indexing is to be done, then Directory class is given to the IndexWriter so that IndexWriter creates an index in a location specified by Directory class. There are different implementations of Directory like FSDirectory, RAMDirectory. Both have similar interfaces. FSDirectory stores the index on a disk. But RAMDirectory holds all its data in memory. It can be destroyed after application terminates. Thus, searching on index generated by RAMDirectory is faster than index generated by FSDirectory, because fetching from hard disk is bit more slow than from memory.

Analyzer:

IndexWriter specifies analyzer. And this analyzer is responsible for extracting tokens from the text. There are different implementations of analyzers. Few analyzers skips stop words like the, is, at. Few other analyzers index words with case insensitivity. Depending on the requirement of application, suitable analyzer can be used.

Below table shows different analyzers:

Analyzers	Description
`Standard Analyzer`	A sophisticated general-purpose analyzer.
`Whitespace Analyzer`	A very simple analyzer that just separates tokens using white space.
`Stop Analyzer`	Removes common English words that are not usually useful for indexing.
`Snowball Analyzer`	An interesting experimental analyzer that works on word roots (a search on *rain* should also return entries with *raining*, *rained*, and so on).

Table 1: Lucene Analyzers [4]

Other than the above-mentioned analyzers in table, there are also language specific analyzers for German, French, Russian and others.

Example:

IndexWriter IndexWriter = new IndexWriter ("index-directory", new StandardAnalyzer (), true);

This example uses all the 3 above-mentioned classes. First parameter for Index Writer is “index-directory”, this is the location where index has to be created. Second parameter tells which document analyzer should be used. In this case it uses StandardAnalyzer.

Document:

Document is a bundle of data or collection of fields like title, author, content and so on. For every file that is to be indexed, a Document class is created, populated with fields and added to the index. Document can be simple text file, webpage, email or a message.

Example:

```
Document doc = new Document ();
doc.add (new Field ("description", Field.Store.YES,
Field.Index.TOKENIZED));
```

Field:

Document consists of one or more fields. Upon these fields in index a search can be done. There are different types of fields.

- Keyword: These are not analyzed but indexed, so that the original value is preserved
- Unindexed: These fields are not analyzed or indexed, but simply stored, so that it can retrieved as the way they are during a search.
- Unstored: These are analyzed and indexed but not stored.
- Text: This is analyzed and indexed.

2.3 Lucene Searching

Searching is a process of looking words in an index in order to find out in which documents they appear in file set. The search term can be a single word, phrase query, wildcard query etc.
The quality of search is measured using Recall and Precision. Recall tells how efficiently are relevant documents retrieved and precision tells how efficiently are the irrelevant documents filtered.

2.3.1 Searching example using Lucene:

```
public static void searchIndex(String searchString)
throws IOException, ParseException {
          System.out.println("Searching for '" +
searchString + "'");
          Directory directory =
FSDirectory.getDirectory(INDEX_DIRECTORY);
          IndexReader indexReader =
IndexReader.open(directory);
          IndexSearcher indexSearcher = new
IndexSearcher(indexReader);

          Analyzer analyzer = new StandardAnalyzer();
          QueryParser queryParser = new
QueryParser(FIELD_CONTENTS, analyzer);
          Query query = queryParser.parse(searchString);
          Hits hits = indexSearcher.search(query);
          System.out.println("Number of hits: " +
hits.length());

          Iterator<Hit> it = hits.iterator();
          while (it.hasNext()) {
               Hit hit = it.next();
               Document document = hit.getDocument();
               String path = document.get(FIELD_PATH);
               System.out.println("Hit: " + path);
          }

     }
```

Searcher program above needs two important command line arguments.

- A path to the index created with Indexer
- A query to use to search the index

Searcher program returns the documents that match the query in the form of Hits. It also prints number of documents matched. For performance reasons not all the hits are returned. Only few are printed.

2.3.2 Searching classes of Lucene:

Here are few classes used for searching using lucene.

- IndexSearcher
- Term
- Query
- TermQuery
- Hits

IndexSearcher:

IndexSearcher is the main link to index. It opens the index in the read only mode. It contains many search methods. Few of those are implemented in its parent class Searcher. IndexSearcher takes query object as a parameter and returns Hits object.

Example:

```
IndexSearcher is= new IndexSearcher (FSDirectory.getDirectory
("/volumes/User/project/index", false));
```

Term:

Term is the basic unit of searching. It consists of pair of string elements. They are name of field and value of field. Term objects are together used with TermQuery while searching.
Example:

```
Query q=new TermQuery (new Term ("title", "Manning"));
Hits hits=is.search (q);
```

Query:

There are different Query classes like PhraseQuery, BooleanQuery and few others. Query class is the parent of all the above classes.

TermQuery:

TermQuery is Lucene's basic query type. It is used to search a field with a specific value. It is mostly used together with Term.

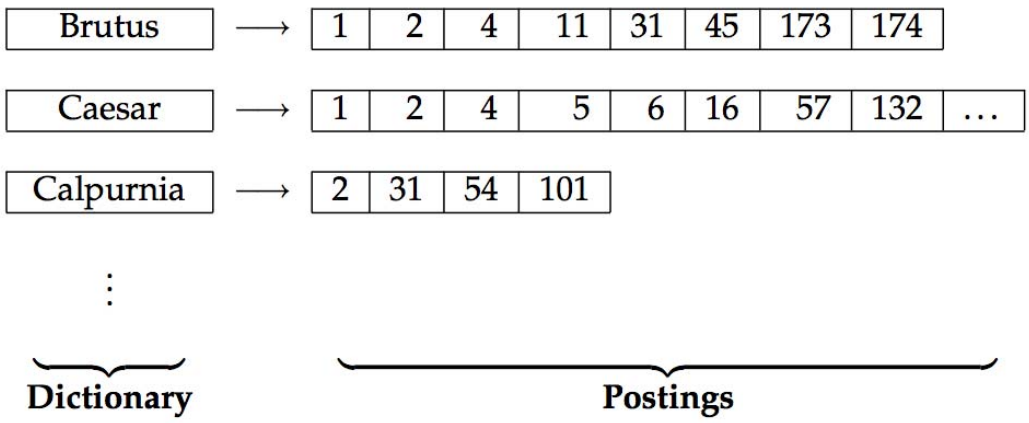
Hits:

Hits are a set of documents that match the query.

Chapter 3

LUCENE INDEX STRUCTURE

Lucene stores its data in the form of inverted index. An inverted index is an inside-out arrangement of documents in which terms take center stage. Each term points to a list of documents that contain it. It is an index data structure mapping terms and the documents that contain it. The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added.



Example:

Fig 4: Inverted index [1]

The inverted index as seen above contains two parts - Dictionary and Postings. Dictionary contains the terms and Postings contains the list of documents that contain the term.

Lucene index structure is shown in fig 5 .

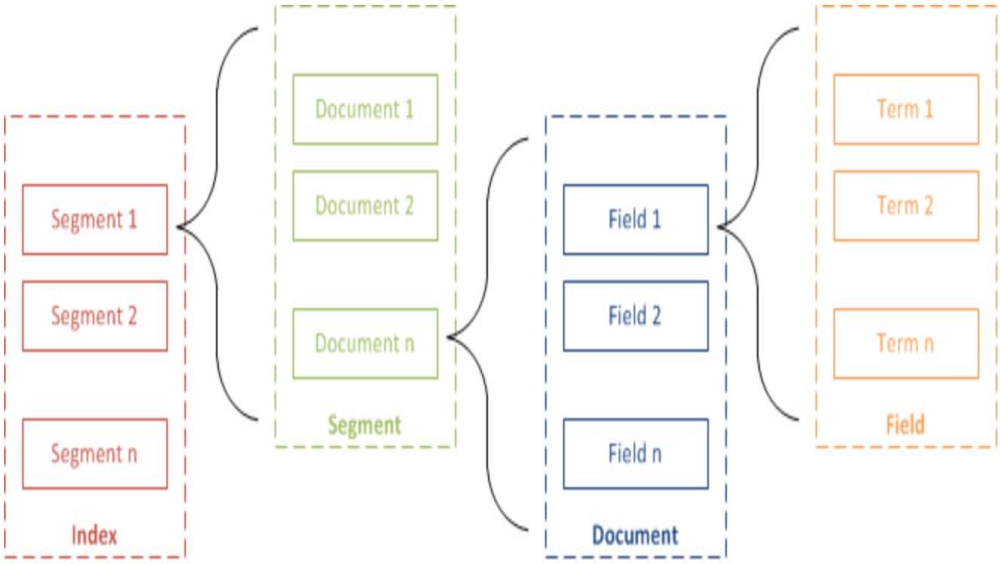


Fig 5: Lucene index structure [6]

Segments:

Lucene index consists of sub-indexes or segments. These are independent index, and these can be searched separately.
Index is created by:

- Creating segments for newly added documents
- Merging existing segments

Each segment contains the following:

- Field names: Contains set of field names used in the index.
- Stored Field values: Contains attribute-value pairs for each document. Attribute is the field name.
- Term dictionary: Contains all the terms in all of the indexed fields of all the documents.
- Term Frequency data: Contain for each term in the dictionary, number of all documents that contain the term, and also number of times it occurs in each document.
- Term Proximity data: For each term in the document, it contains the position of terms in the documents.
- Term Vectors: Contains the term text and term frequency.
- Deleted documents: Indicates which files are deleted. This is optional.

The number of documents to be indexed and the number of documents a segment can contain determine the number of segments. Below figure indicates this.

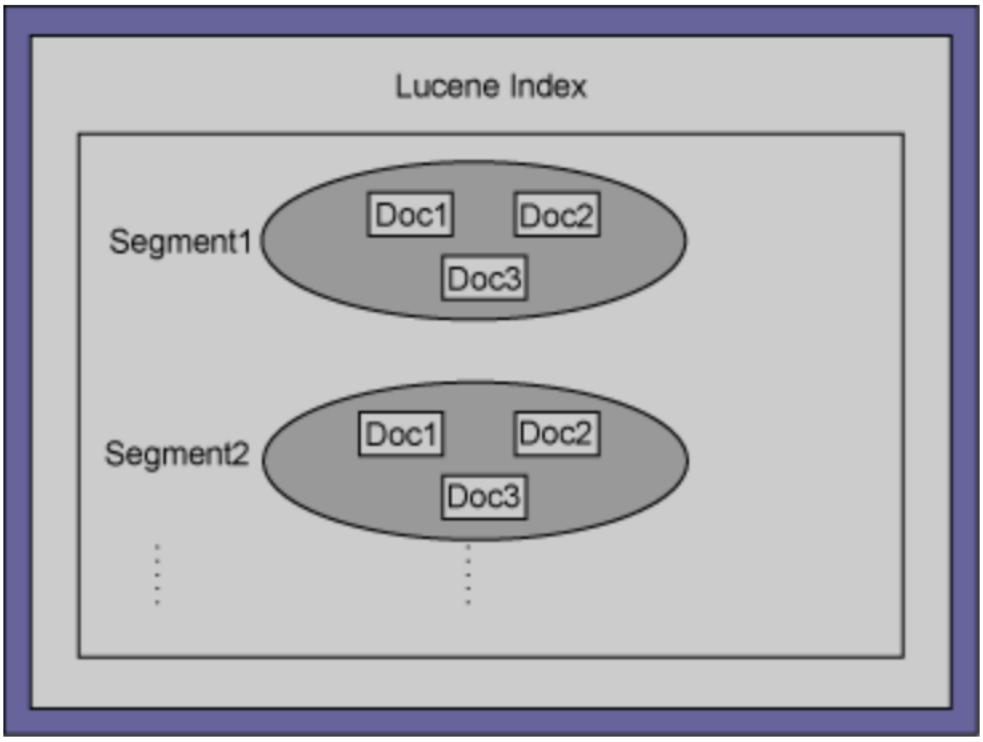


Fig 6: Logical view of index files [6]

Document Numbers

Internally Lucene refers to documents with an integer document number. The first document added to index is numbered zero and subsequent documents are numbered one more than previous.

File Naming

Index contains different files with different extensions with different information. Below table summarizes different files.

Name	Extension	Description
Segments File	segments.gen, segments_N	Stores information about segments
Lock File	write.lock	The Write lock prevents multiple IndexWriters from writing to the same file.
Compound File	.cfs	An optional "virtual" file consisting of all the other index files for systems that frequently run out of file handles.
Fields	.fnm	Stores information about the fields
Field Index	.fdx	Contains pointers to field data
Field Data	.fdt	The stored fields for documents
Term Infos	.tis	Part of the term dictionary, stores term info
Term Info Index	.tii	The index into the Term Infos file
Frequencies	.frq	Contains the list of docs which contain each term along with frequency
Positions	.prx	Stores position information about where a term occurs in the index
Norms	.nrm	Encodes length and boost factors for docs and fields
Term Vector Index	.tvx	Stores offset into the document data file
Term Vector Documents	.tvd	Contains information about each document that has term vectors

Term Vector Fields	.tvf	The field level info about term vectors
Deleted Documents	.del	Info about what files are deleted

Table 2: Names and extensions of the files in Lucene: [5]

Structure of few of the above files is shown below.

Fields information file

Column name	Data type	Description
FieldsCount	VInt	The number of fields.
FieldName	String	The name of one field.
FieldBits	Byte	Contains various flags. For example, if the lowest bit is 1, it means this is an indexed field; if 0, it's a nonindexed field.

Table 3. Structure of fields information file: [6]

Frequency file

Column name	Data type	Description
DocDelta	VInt	It determines both the document number and term frequency. If the value is odd, the term frequency is 1; otherwise, the Freq column determines the term frequency.
Freq	VInt	If the value of DocDelta is even, this column determines the term frequency.

Table 4. Structure of the frequency file: [6]

Position file

Column name	Data type	Description
PositionDelta	VInt	The position at which each term occurs within the documents.

Table 5. Structure of the position file: [6]

Chapter 4

Using Lucene with Boemie Repository

First step for using Lucene to index or search on Boemie repository requires, the necessary Lucene jar files to be added as referenced libraries to the project. Since Lucene is just a library and not a complete framework, java programming is done using this library to conduct the experiment.

As mentioned in the previous sections Boemie repository consists of 690 multimedia documents, which concerns about sports. The first step is to index these multimedia documents. These are .html files, which are present different levels of directory tree.

4.1 Algorithm for indexing Boemie

```
Main(){
      new Index_with_TermFreq();
}
Index_with_TermFreq(){
      createIndexWriter(){
            // Creates the index directory on hard disk or
memory, specifies the analyzer to be used, and configures the
index writer
      }
      checkFileValidity(){
            //checks if it is a file or directory .If it is a
directory recursively traverses to find files
      }
       indexTextFiles(){
            //creates a document for every file and adds
document to index.Also specifies the fields to be indexed
       }
       TotalDocumentsIndexed(){
            //prints the total number of documents indexed
       }
       closeIndexWriter(){
            //closes the index writer
       }
  }
```

Running the above program , gives the output like below:

```
INDEXED FILE
/Volumes/BoemieRepository/url1/www.iaaf.org/athletes/athlete=1
36012/BioPopUpIndoor.html
INDEXED FILE
/Volumes/BoemieRepository/url2/edition.cnn.com/2005/index.html
INDEXED FILE
/Volumes/BoemieRepository/url3/www.iaaf.org/GP02/news/Kind=2/n
ewsId=19344.html
       .
       .
       .
INDEXED FILE
/Volumes/BoemieRepository/url690/www.walkleamington2007.org/ne
ws.php@id=55.html
Total Document Indexed: 690
Total time 1
```

The above output shows which documents are indexed, total number of documents indexed and time taken to index. And also it creates a index folder.

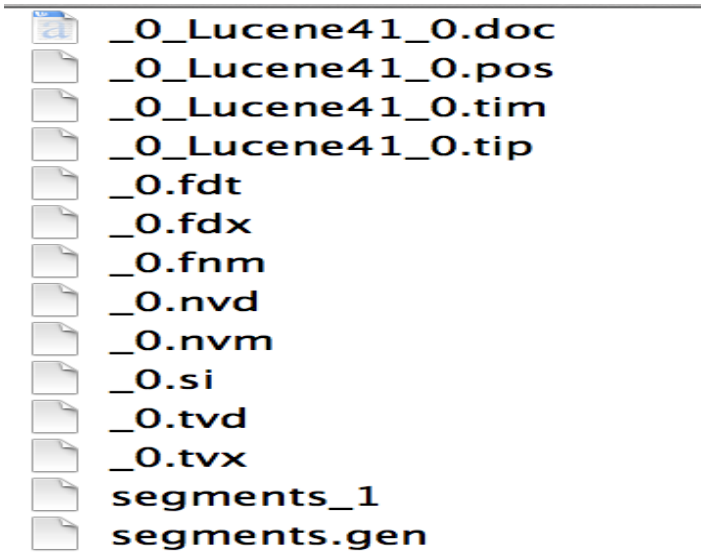


Fig 7: Boemie index structure

We see different files in Boemie index structure with different file extensions. All these file extensions are explained in the previous section.

4.2 Algorithm for searching on Boemie index

```
Main(){
      new.searchIndex("\"high jump\"");
}
searchIndex(){
    //opens index
   IndexReader reader = DirectoryReader.open();
   IndexSearcher searcher = new IndexSearcher(reader);
   //searches for specific query
   Query query = queryParser.parse(instring);
   //hits for the query
   TopDocs hits = searcher.search(query, 100);
   //prints the hits for the query
}
```

Running the above search Index program, gives the output like below:

```
Searching for ' "high jump" '
Total no of hits for content: 524
/Volumes/BoemieRepository/url57/www.iaaf.org/WCH05/news/Kind=2
/newsId=31773.html
/Volumes/BoemieRepository/url56/www.iaaf.org/news/newsId=31698
,printer.html
            .
            .
            .
/Volumes/SindhuHosamane/IR_project/BoemieRepository/url469/www
.iaaf.org/news/newsId=31109,printer.html
```

This searched “high jump” as a single string with a space in between. Phrase query as mentioned in the previous section could also be used. Many of the Lucene classes used in the algorithm are explained in the previous sections 2.2.2 and 2.3.2
Same program is run on the query “pole vault”. The result set returned by this algorithm is said to be R1.

4.3 Algorithm for retrieving documents based on relevance condition

```
Main(){

      Findfreq("high","jump");

}

Findfreq(String instring1,String instring2){

          //opens the index
          IndexReader reader = DirectoryReader.open();
          // Total number of documents in index
          int totalDocs = reader.maxDoc();
          for(int i = 0 ; i< totalDocs; i++)
          {
             //tells the term frequency of each term
              Terms terms = reader.getTermVector(i, "content");
              Document doc = reader.document(i);
              IndexableField path =doc.getField("fullpath");
             TermsEnum termsEnum = terms.iterator(null);

              while ((text = termsEnum.next()) != null)
              {
               if(text.utf8ToString().equals(instring1))
               {
                  term1 = text.utf8ToString();
                  freq1 = (int) termsEnum.totalTermFreq();
               }
               if(text.utf8ToString().equals(instring2))
               {
                    term2 = text.utf8ToString();
                    freq2 = (int) termsEnum.totalTermFreq();
                }
               }
               if (relevance condition){
                   // Print relevant documents with term and
frequency
                }
          }
    }
```

Running the above program, outputs the following:

```
stored,indexed,tokenized
<fullpath:/Volumes/BoemieRepository/url13/www.iaaf.org/GP03/ne
ws/Kind=2/newsId=22176.html>--high : 4--jump : 3
                .
                .
                .
stored,indexed,tokenized
<fullpath:/Volumes/BoemieRepository/url14/www.iaaf.org/GP03/ne
ws/Kind=2/newsId=22213.html>--high : 4--jump : 4
```

The output shows set of documents that are relevant for the query based on relevance condition specified. The same program is run on query “pole vault”. Since Lucene indexes the document word by word, it’s difficult to get the frequency of phrase queries like “high jump”. So using the individual term frequency of 2 words of phrase query, a relevance condition is made. Documents satisfying the relevance condition are printed.

Relevance condition is based on term frequency of words of phrase query. Based on range of term frequency of words of phrase query, a threshold frequency is decided. Relevance condition is based on threshold frequency. And consider only the documents whose frequency is greater than threshold frequency. The result set returned by this algorithm is said to be R2.

Next chapter measures uses these two result sets R1 and R2 and measures the effectiveness of Lucene library.

Chapter 5

EVALUATION

5.1 Steps of evaluation

Quality and performance of standard IR systems is to be measured. Evaluation steps are :

a) start with a corpus of documents
b) define a set of queries
c) set of relevance judgments.

Corpus of documents is Boemie repository. Second step is to define specific queries. Queries should be chosen such that tuning them should not maximize the effectiveness on that corpus. And the last step is to create gold standard.

The standard approach to information retrieval system evaluation revolves around the notion of relevant and nonrelevant documents. With respect to a user information need, a document in the test collection (corpus) is given a binary classification as either relevant or nonrelevant. This decision is referred to as the gold standard or ground truth judgment of relevance.

Experts usually do gold standard. Gold standard is prepared for all the multimedia documents in Boemie repository. We use as gold standard one of the result set returned by the algorithms in the previous chapter.

5.2 Basic measures of Evaluation

To assess the effectiveness of an IR system (i.e., the quality of its search results), a user will usually want to know two key statistics about the system's returned results for a query:

***Precision*:** What fractions of the returned results are relevant to the information need?

$$\text{Precision} = \frac{\#(\text{relevant items retrieved})}{\#(\text{retrieved items})} = P(\text{relevant}|\text{retrieved})$$

Recall: What fractions of the relevant documents in the collection were returned by the system?

$$\text{Recall} = \frac{\#(\text{relevant items retrieved})}{\#(\text{relevant items})} = P(\text{retrieved}|\text{relevant})$$

5.3 Results of the experiment

Considering the experiment for specific queries, like “high jump”,” pole vault”.

Query “high jump”

Considering the result set returned by algorithm in section 4.2 as relevant documents, and result set returned by algorithm in section 4.3 as retrieved, precision and recall are measured.

R1 ={…..} Set of relevant documents on the query “high jump”

R2 ={…..} Set of retrieved documents based on term frequency of “high” and “jump”

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

As mentioned in the previous section 5.2

Precision = |R1 ∩ R2| / |R2| =103/106 = 0.97

There are some documents as an exception because, such documents satisfy the relevance condition, where the frequency of words of phrase query reaches threshold frequency, but do not contain high jump as a phrase. It means the words of phrase query are in different positions and not next to each other. This is reflected looking at the fraction above where the intersection of relevant documents and retrieved documents is less than the number of retrieved documents.

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

Recall = |R1 ∩ R2| / |R1| = 103/269 = 0.39

Query “pole vault”

Considering the same sets as above, but for query pole vault

Precision = |R1 ∩ R2| / |R2| =53/54 = 0.98

Recall = |R1 ∩ R2| / |R1| = 53/271 = 0.20

As we see in the above results, we have high values of precision but not high values of recall. Trying to increase one reduces the other. It is hard to maintain high values of both precision and recall. Precision values of 0.97 and 0.98 indicate most of the returned results are relevant to the query.

Query “of the”

Lucene does not index stopwords like a, an, is, the, of and many other. So it is difficult to get the term frequency of these stopwords. So is the phrase query “of the”.

Chapter 5

Conclusion

The aim of the project was to measure the effectiveness of standard information retrieval systems using specific queries.
So using the Lucene, an information retrieval library, Boemie repository that is corpus of multimedia documents, and few specific queries, the experiment is conducted.

Precision and recall are the measures for measuring the effectiveness of any information retrieval systems. These two measures are based on relevant and retrieved set of documents. Looking at high values of precision it is clear that, most of the returned results are relevant to the query. Which means less number of false positives, proving Lucene is efficient in a way.

Precision can be seen as a measure of exactness or quality, whereas recall is a measure of completeness or quantity. In simple terms, high recall means that an algorithm returned most of the relevant results. But in our results the recall values are not high, that means not all relevant documents are returned. While high precision means that an algorithm returned substantially more relevant results than irrelevant.

Bibliography

[1] Christopher D. Manning , Prabhakar Raghavan, Hinrich Schütze- An introduction to information retrieval,Cambridge university,England

[2] Gospodnetic, Otis; Erik Hatcher, Michael McCandless (28 June 2009). Lucene in Action (2nd ed.). Manning Publications.

[3] http://wiki.apache.org/lucene-java/PoweredBy

[4] http://oak.cs.ucla.edu/cs144/projects/lucene/

[5] lucene.apache.org

[6] http://www.ibm.com