Master's Thesis

Architecture and Implementation of Apache Lucene

By

Josiane Gamgo

Submitted to the department of Mathematics, Natural Sciences and
Computer Sciences at the Fachhochschule Gießen-Friedberg
University of Applied Sciences

In Partial Fulfillment of the Requirements of the Degree of

Master of Science in Computer Science.

Gießen, November 2010

Supervisor: Prof. Dr. Burkhardt Renz

Second Reviewer: Prof. Dr. Achim H. Kaufmann

Declaration

This Thesis is the result of my own independent work, except where otherwise stated. Other sources are acknowledge explicit reference.

This work has not been previously accepted in substance for any degree and is not being currently submitted in candidature for any degree.

Gießen, November 2010 Josiane, Gamgo

Abstract

The purpose of this thesis was to study the architecture and implementation of Apache Lucene. At the beginning of this project, the D.k.d , an internet company in Frankfurt(Germany), which develop web pages based on Typo3 had the idea to replace the typo3 indexed search extension. Thus, they found Solr, which is a Lucene based web search application. Then , the idea of studying Lucene architecture came from my supervisor, Prof. Dr. Burkhardt Renz

A first implementation of a small search engine based on Lucene, named SeboL, helps to go in-depth in to the Lucene components. Research was first done on the indexing functionalities of Lucene, then on the index search. At the end of the research a study of Solr and its usage by the D.k.d. Company has been done. Books and web pages discussing about Information retrieval and Lucene helps to achieve a comparative study of the Lucene algorithms and those used in Information retrieval. The core Lucene classes has been analyzed for each important class involved in indexing and search processes. Testing and Debugging the SeboL search application helps to shape the Lucene components architectures.

Delving into the Lucene indexing mechanism, was a quite difficult task, because of the complexity of the library. On the long run, it was possible to illustrate the internal architecture of the following components : Lucene field, Lucene document, Lucene analysis, Lucene index, and much more. The index Writing mechanism, the decorator pattern used by Lucene analyzer, as well as the tree structures of some Lucene Queries was illustrated. The implementation steps of a document handler for Pdf files using the Lucene document handler interface was discussed. An in-depth illustration of Lucene document data structure as well as the Lucene index file formats definition was given. In fact, classes and methods involved in indexing and search process have been thoroughly detailed. A CD containing the source code of SeboL is attached to this thesis

The author recommends that the D.k.d Solr team introduce more Lucene optimization capabilities to their Solr search engine. It is recommended, that further research is carried out to extend this architecture .It is recommended, that the Lucene Document handler is extended for other common file format like Pdf and Xml file formats. It is also recommended to study the possibilities to add Solr-like features directly to Lucene, for more user friendliness.

Dedication

To Stephanie and Merlin

Acknowledgments

I am heartily thankful to my supervisor, Prof. Dr. Burkhardt Renz, whose patience, encouragement, guidance and support, from the initial to the final level enabled me to develop an understanding of the subject. Also,hanks to my second reviewer, Prof. Dr. Achim Kaufmann for advising me during my Master's studies. My gratitude also goes to Olivier Dobberkau , for giving me the opportunity to participate to the Typo3 Solr project.

Thanks to Lilian, Laura, Sissih, Tina and Vilna, a group of nice hard working women. Thank you for correcting my work. Also thanks to my ant. Tata Beatrice, thanks for your assistance and for reviewing my work.

Table of contents

List of figures

Index of tables

chapter 1 Introduction

1.1 Background

It all begins with a natural human need of Information to be satisfied. Yves-François Le Coadic [1] said that, the need of information reflects the state of knowledge in which a user is, when he is confronted to the requirement of a missing information. In order to fulfil this need, studies and research have been done: first of all to examine the nature and the extent of needs, next to identify the search strategies to the satisfaction of those needs, then research have been done on the information sources and the access to those sources.

The arrival of the first computers, gives the idea to automate information sources for search purposes. This is how information retrieval [4] arises in the field of computer science. This new science born between 1945 and 1950 has several applications including digital libraries, information filtering, media search and search engines. The last one is the most commonly used today. Search engines also have different applications such as desktop search engine, Enterprise search engine, web search engine and so on.

This thesis is based on some particular aspects of search engines namely indexing and searching process. As define in Wikipedia[1] , “a web search engine uses a web crawler or a spider to retrieve web pages and store information about those pages”. Afterwards,These are analysed to determine how it should be indexed. Data about a web pages are therefore stored in an index database, which is later to be used in the search of a user giving query. An index can be seen as a traditional back-of-the-book index which contains a list of words, names or phrases which pointed to materials relating to that heading.

Indexing and searching features of web search engines could be provided by some specific applications called search engine libraries. One of those application namely Apache Lucene, is the main subject of this thesis. This thesis studies the internal functionalities of Apache Lucene. On the basis of Lucene core API(Application Programming Interface), its structure is put out in this thesis. Furthermore the interaction of its most important components would be analysed . A case study of the adaptation of Lucene in a particular web search application called Solr is discussed. In the whole thesis, Apache Lucene will simply be called “Lucene”.

1.2 Motivation

In the field of Information retrieval in particularly text retrieval, when searching for a large quantity of documents, two processes are mainly used: indexing and searching. Those processes improve the retrieval of information and are essential features of the Lucene search engine library . Certain facts about Lucene must be underline before: Lucene is one of the most popular search engine libraries used by various search applications of famous companies such as *AOL, Apple, Disney, Eclipse, IBM, CNET Reviews*, *e.on, Monster, Wikipedia, Sourceforge.net* and much more. Despite its popularity, there is a lack of literature reports on the structural constitution of Lucene as well as on the interaction between its components and the applied algorithms. These fact led to the achievement of this thesis. Some of those information gaps about Lucene are to be closed by this work.

1 Wikipedia: web search engines, URL: http://en.wikipedia.org/wiki/Web_search_engine [accessed june 2009]

1.3 Research objectives

The first Objective of this Work is to define common terms used in the Lucene and in the Information retrieval world. The next step is to illustrate, describe and analyse the structure of Lucene components, their implementation and their algorithms. Then give more detail on the interactions between the main components involved in indexing and search; for this purpose, the source code of some important Java classes of Lucene will be analysed.

In conclusion, a case study of a Lucene based web application called Solr is made in order to show an example of the use of Lucene components.

In order to achieve the previous objectives the main questions to be answered are:

- What is information retrieval and how has it been growing up over years.
- Which methods have been used before to retrieve information and which one are used in these days in the field of search.
- Why Lucene? Are there any other search engines libraries? How good are they compared to Lucene?
- How is the structure of Lucene source code, especially its components structure: design patterns and used infrastructures.
- What is the relationship between components used during the indexing, searching and analysis processes.
- Which concepts and algorithms are used in Lucene, and which data structure does it deals with.
- How is the index used for search, and how big is an index for huge data. Which operations could be done on the index?
- How does Solr adjust Lucene. In other words which components of Lucene are used in Solr, and how are they used?

1.1 Structure of this work

This work is divided into the following topics:

- Chapter 2 tells the story of information retrieval. It gives a brief introduction in Apache Lucene: Definitions, uses, target groups, comparison with other search engine libraries. Moreover the basic concepts are described including indexing, Lucene Document, analysis and index search. An overview of the designed architecture of Lucene closes this chapter.
- Chapter3 contains illustrations of the architecture of Lucene code. The essential components are depicted, the role of each other is explained and the interaction between components are demonstrated. Examples are given for better understanding of indexing and search implementations. For the purpose of this thesis, a Java search application based on Lucene have been implemented to index and search within .pdf and .txt files. This application has been called *SeboL*(Search engine based on Lucene)
- Chapter 4 discusses essentially about the so called Lucene Document. Following themes are debate:
  - Creating a Lucene Document using the Lucene document handler interface.

  - components and data structure of a Lucene Document
  - functionality of a document parser and its components. Interaction with each component.
  - An example of Document handler implementation with PDFBOX Library for creating Lucene Document from Pdf files.

- Chapter 5 is about the Index building mechanism and the strategy of the Index generation. As it will be discussed in this part, the index is the core of Lucene and the basics of any index search. The first part of the chapter reports about the Index components and interaction, its data structure and files formats. The second part shows how to construct an index and which algorithms are behind the indexing process. Some indexing algorithms in information retrieval are compared to Lucene indexing algorithm.

- Chapter 6 talks about the index search process in Lucene especially: how the index is accessed for search . Which algorithm is used for search, and which components are involved? How does the Lucene query looks like. More details about the Lucene query language will be given in this chapter. At the end of the chapter the Lucene scoring algorithm would be study including the scoring models and formula used .

- Chapter 7 gives a use case of Lucene particularly, the adaptation of Lucene in Solr, and the usage of Solr int the D.k.d. web search application.

chapter 2 Basics and Concepts

2.1 Basics

This section deals with the history of searching and the basics of Apache Lucene. A comparison of Lucene with other search engines libraries gives an overview of the strength and the weakness of Lucene.

2.1.4 The search History

From the beginning, searching is an instinctive and natural process in each one's life. Most of the time we seek answers to precise questions or solutions to a given problem. The search we are talking about is the finding of Information . An Information is define as an ordered sequence of symbols, this sequence has a particular meaning for the person who reads it. In order to automate this natural process of searching, a new science is born called I*nformation Retrieva*l(IR) .Information retrieval is finding material(usually documents) of an unstructured nature(usually text) that satisfies an information need from within large collections(usually stored on computers) [4].The search history in the field of computer sciences is therefore the information retrieval's history. The figure1 illustrated in a paper by Fielden [6], shows how information retrieval changed over the time. Let us see how the search for information has evolved:

Historical records goes back to about **9000 BC**: According to Arnold and Libby [31], who discover the radiocarbon dating, there is no record of mankind prior to 3000 BC. However the first Egyptian papyrus were discover between **3032-2853 BC**. So writing and information recording is situated about **2500 -2853 BC**. At this period, information were written and stored on *papyrus. A Papyrus* is "a material on which to write, prepared from thin strips of the pith of this plant laid together, soaked, pressed, and dried, used by the ancient Egyptians, Greeks, and Romans"[2]. As soon as human being discover writing, Information retrieval becomes a personal investment. People starts looking for writings of scientists and philosophers to acquire knowledge. Afterwards, between 390 and 340 BC , *Plato* and *Aristotle* contributes to the general human knowledge by retrieving informations in the field of philosophy . These are their theories :

- **390 BC**: Plato's define *the theory of Forms*[3], as an abstract property and quality according to him, the forms have properties :
  - They are not located in space and time,
  - they exemplify one property,
  - they are perfect example of the property that they exemplify,
  - they are not material objects,
  - they are the causes of all things
  - Forms are systematically interconnected.

Example of Forms are redness, roundness, they are all properties of object, a ball that has a red colour is an object. Redness and roundness are pure Forms there are the causes.

2 Dictionay.com: Papyrus[2010]. URL: http://dictionary.reference.com/browse/Papyrus

3 David Banch: Plato's theory of forms[2006].URL: http://www.anselm.edu/homepage/dbanach/platform.htm

Another concept is given by Aristotle

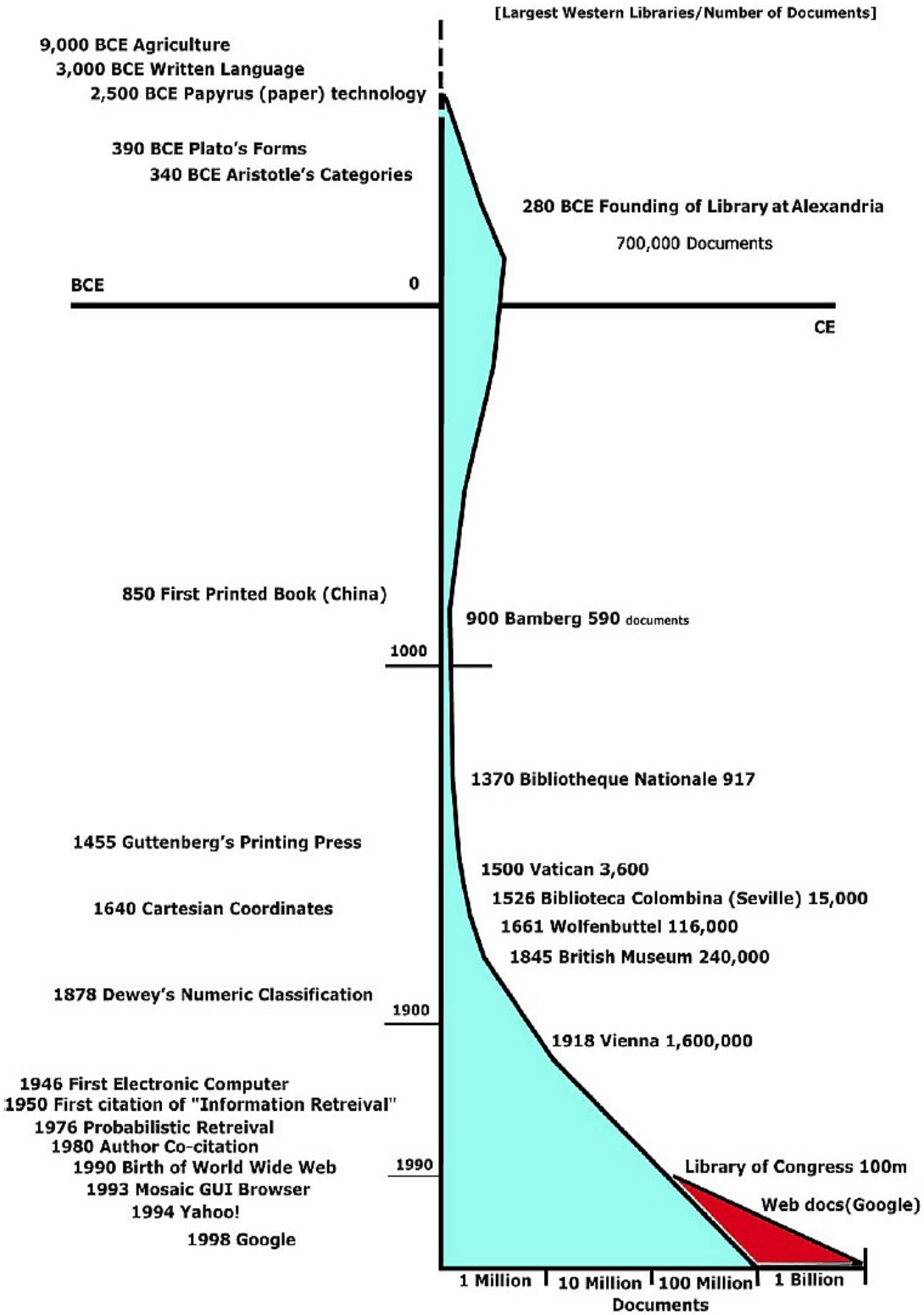


Figure 1: Evolution of information retrieval of the time(source: N. Fielden: Timeline of Information and Retrieval systems, May 2002. URL: http://userwww.sfsu.edu/~fielden/hist.htm)

340 BC: Aristotle divided his so called subjects in 10 categories: substances, quantity, quality, relation, places, time, position or posture, state, action and affection. The schema given by Aristotle, has a wide-reaching influence in people's mind. At that time, there were less than 1 million documents on various fields.

280 BC: In the library of Alexandria , the most known data pool of the ancient world, something like 700 000 documents could be found . The library was considered as the world's first major seat learning[4]. After the fourth destruction of the library of Alexandria by the Arabs in 642 CE . The next evolution of information retrieval was made in china

around **850 CE**: The Chinese printed there first book in 862, it was a Buddist sutra[5]

Between **1367-1368**: Charles V of France made a collection of 910 books that constitute a library called "***la librairie royale***". Books concerning methods and samples for good government.[6]

After the discovering of the printing press **1455**, writing becomes easier and the amount of documents starts growing up:For instance, the *vatican*'s library gather a total of 3498 manuscripts in year **1481**[7] other facts and numbers are giving in the diagram below. At the end of this period, there were about 2 millions documents collected. With the arrival of the first electronic computer **ENIAC**[8] in may **1945** the storage of informations changes from paper form to virtual form. From that time on, the search of information becomes electronic and mechanized.

1945 *Vannevar Bush* had the idea of sensitizing scientists to work together in order to build a body of knowledge for all mankind[9]. Publication records on different interesting topics were not as useful as they expected it to be. According to Bush, a scientific record has to be extended, stored and above all it must be consulted by all. The human mind should be presented as it naturally works. Therefore, *Bush* created a system he called *memex* (Memory extender) Memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. A Memex animation is available at http://sloan.standford.eu/MouseSIte/Secondary.html

1947 Hans *Peter Luhn* developed a punched cards based system to search for chemical compounds recorded in code form. Note that a punched card is like a piece of paper that contains digital information represented by the presence or absence of holes in predefined position. Luhn also initiate the basic concept of automatic indexing in the 1950s. It was called *KWIC* (Keyword in Context) it was an index based on permutations of significant words in titles, abstracts or full text[10]

1948-1950 The term ***"information retrieval"*** has been coined by Calvin N. Mooers [37] during his master's thesis at the Massachusetts Institute of Technology. Mooers define

4 Michael Lamanas. The Bibliotheke oer library of Alexandria. Http://www.mlahanas.de/Greeks/Library.htm [accessed: July 2010]

5 Society for Anglo-Chinese Understanding. Available at Http://www.sacu.org/greatinventions.html [accessed July 2010]

6 bibliotheque nationale de France,dossier pédagogique, Available at Http://calsses.bnf.fr/dossitsm/biblroya.htm [accessed July 2010]

7 [Vatican Library, available at http://www.vaticanlibrary.va/home.php?pag=storia&ling=en&BC=11 [accessed July 2010]

8 Till Zoppke: The ENIAC(Electronic Numeral Integrator) simulation. 2003/2004, available at http://www.zib.de/zuse/Inhalt/Programme/eniac/about.html [accessed July 2010]

9 Vannevar Bush: As we may think [1969]. URL : http://www.theatlantic.com/magazine/archive/1969/12/as-we-may-think/3881/

10 S. Soy, University of texasClass Lecture Notes: H.P. Luhn and Automatic indexing. URL:http://www.ischool.utexas.edu/~ssoy/organizing/l391d2c.htm [accessed July 2010]

Information retrieval as the finding of information whose location or very existence is a priori unknown.

Early 1960 one of the Information retrieval models was born: the probabilistic model . It is an indexing model defined by Maron and Kuhns [59] where the indexer(a human being) should run through the various index terms T assigned to a Document D, and assign a probability P(T|D) to a giving term in a document. The Baye's rule[11] is used to compute the results of a user query formulated by using [38]. In this thesis we would discussed about two other information Retrieval's models namely the Boolean and the vector space models. We can observe according to the graph of the time line in figure1, that the amount of documents stored by data pools exceed the millions of documents from 1945.

1960- 1990s Gerard Salton [39] considered as the leader in the field of information retrieval , developed with his group at the Cornell University, the SMART Information retrieval system [60]. Some information retrieval concepts like ***"the vector space model"*** were included in the research works on SMART.

With the birth of the world wide web **1990s**, the collection of the library of congress in Washington nears 100 millions of documents. According to James H. Billington, It is the largest library of the world [12]

After the creation of the hypertext concept 1970 by ***Theodor Nelson*** [13]and the birth of the world wide , the first search engine called Archie was coined and implemented in 1990 by ***Alan Emtage***[14] . Archie [15]is a tool for indexing FTP archives. Archie combine script based data gatherer with a regular expression matcher for file names retrieval. After Archie, the next generation of search engine were more powerful and could retrieve billions of documents.

From the **20th Century** till now, on-line search engines has populated the web, the most known are ***Yahoo***(since February1994 [41]) and ***Google*** [42](since 1998). The last one is the most powerful one. According to the Google patents, it is a web search engine designed for **IR**(Information retrieval) and especially to the methods used to generate search results. Google's first search technique is indexing documents. In this thesis we are also dealing with indexing and index search but this time using Lucene. The next part of this chapter talks about Lucene. What is it? What are the concepts behind and which relationship does Lucene has with IR?

11 K. Murphy. Abrief introduction to Baye's Rule. URL: http://www.cs.ubc.ca/~murphyk/Bayes/bayesrule.html [accessed October 2010]
12 Library of congress. [2010]. URL: Http://www.loc.gov/about/
13 T. H. Nelson. URL: http://ted.hyperland.com/ [accessed October 2010]
14 A.Emtage. URL: http://www..alanemtage.com [accessed October 2010]
15 Archie Query Form. URL: http://archie.icm.edu.pl/archie-adv_eng.html [accessed October 2010]

2.1.5 What is “Apache Lucene”?

Lucene was developed **1998** by Doug Cutting and published on *Sourceforge*[16] as Open source project. „Lucene“ is not an abbreviation but the second name of Doug Cutting's second name's wife. Since 2001 up till date, Lucene is part of the Apache foundation and is called „**Apache Lucene**“. According to the founder „Lucene is a Software library for full-text search . It's not an application but rather a technology that can be incorporated in applications.”***Doug Cutting*** [43]. Lucene is a scalable search library for full text search. It is a solid basis, on which a search application can be developed. Lucene alone doesn't constitute an application. It can analyse and index textual contents, it can search inside the created index and display hits for a given query. Lucene integrate no graphical user interface .

the compositional structure of an application based on Lucene may have the following components:

- A data pool which holds all kinds of documents, for instance PDF,HTML pages, XML documents, plain text document, Word documents or others . They may be files in the file system or contained in a database or generated by an application crawling the web . Lucene expects that data to be indexed are provided as Lucene Documents.
- Lucene Documents: The Lucene library does not contains the functionality to convert the original files in the data pool into Lucene documents. An application using Lucene should implement a document handler based on the provided Lucene document handler interface, in order to transform content into Lucene Documents. Nevertheless, for text extraction, the application needs a so called document parser. Chapter4 deals with the Lucene Document and the implementation of Lucene document handler.
- An Index:Lucene Documents are analysed and processed for indexing by the IndexWriter, that uses an analyzer and one or more filters, to generate index entries. Chapter 5 gives more details about the structure of a Lucene index and the process of creating an index.
- An index search implementation: An Application can search the index by providing Lucene with search requests . A user request is analysed by the Lucene QueryParser and formatted in the Lucene query language. The Query parser builds a Lucene query data structure, which is a tree of clauses respecting the kind of user request. The Lucene query is passed to the Index searcher that retrieves the hits in the index. The result of the searching can then be displayed by the user application. Chapter 3 gives an overview of the structure of some Queries architectures, and chapter 6 study how the query are parsed with the query language and how it used to get the search results.

2.1.6 Comparison with other search engine libraries

Lucene is used by Wikipedia for search but it is not the only search engine library. Table 1 Shows a comparison of Lucene, Egothor and Xapian. These are search engines libraries that can also be used to create a search application.

Xapian indexer processes documents and stores it into a data pool or a “Xapian data base. “

A Boolean search for term1 AND tem2 is a conjunction of documents that are indexed by term1 and term2. Compare to Lucene, Xapian is written in C++.

16 Geeknet. Sourceforge find and develop open source software.URL: http://sourceforge.net/

Search results in Xapian are filtered by Value or by generated key. On the contrary, Lucene uses a field selector,or payload, or boost factor to filter results by relevance, or by defined field parameters

Both Xapian and Lucene have query parser , but Xapian has no Document handler. Lucene provides a document handler interface, that any application can implement to produce Lucene Document. A complete File extension document handler for .txt and .html files is available in Lucene core . The final document obtain after parsing is usually a set of fields. Apart from that, Lucene and Xapian have the Stemming algorithm in common. An example of the stemming algorithm is given in chapter5.

Egothor is both a standalone search engine application and a search engine library that can be integrated in any search application. Here goes the comparison between Egothor, Xapian and Lucene.

	Lucene	**Egothor**	**Xapian**
Open source, free	Yes	Yes	Yes
Search Application	No	Yes	No
Additional components	Webcraller (Nutch)	Webcraller (Capek)	Website search's application(Omega)
Language	Java	Java	C++
Ports / Implementations	Java, C, .NET, Python, Ruby, Perl	Java	C++,Perl,Python,Java, Tcl,C#, Ruby
Stemming of Terms	Yes	Yes	Yes
Boolean model	Yes	Yes	
Vector model	Yes	Yes	
QueryParser	Yes	Yes	Yes
Document Handler	Yes	Yes (RTF, PDF, Word - Parser)	No

Table 1: Comparison between Lucene and other information retrieval search libraries

All things considered, Lucene gives more flexibility to the application programmer. It has many ports and both boolean and vector model are used for effective retrieval of queries. Its Library (API) provides a lot of classes, that can be used for text analysis in particular text stemming. A basis interface called DocumentHandler provide any Parser which needs to produce Lucene Document with a method to achieve it. Like Lucene, Egothor has a ready-to use document handler, but it is only available for Java programmers. Xapian is in many points similar to Lucene but it has no retrieval model for search. In fact, the choice of a search engine library depends on the user, whether the library fit its needs. We choose to study Lucene for its flexibility and its ability to be extensible. The next part present the ideas behind Lucene.

2.2 Concepts

This part deals with the concepts of Apache Lucene. What does it mean to index a document with Lucene? The idea behind the analysis of textual contents and the meaning of the Lucene document. Furthermore searching in an index follows a certain logic which will be outlined here.

2.2.4 The Lucene document

In essence a Lucene document is the Lucene logical representation of the human readable contents extracted from common documents files. The extraction of textual content is not done by Lucene, rather, by a document parser or by a document handler provided by the application. Lucene support this process by providing a useful DocumentHandler interface to transform textual contents into Lucene Document. A Lucene document is an instance of the Lucene.document.Document class , it is define as a set of fields . A field is a name-value pair, where the name is the field's name commonly defined as a constant of type String, the value is the field value which is hold the content of the field. A Lucene field's value can be a String or a word, a Reader or a *TokenStream* containing tokens extracted from this field. The functionalities of creating new fields and adding them to a Lucene document is provided by the Lucene.document.Document class. Another functionality of Lucene Document class is to supply the user application with fields' options , which are specifications, that determines how the Lucene Documents should be handle during indexing and search.

Let's consider the following example. A Lucene document is to be created from the PDF file "***broder_websearch.pdf***" f: In the Lucene core library, there is a document handler interface provided, it can be used for the creation of personalized document handlers together with a common file document parser for text extraction. In our case he PDFBOX library from Apache is used to create a Pdf document handler that extracts text from PDF files and fill Lucene document fields with them. The content of the PDF file may be given as the value of the defined field "contents" of the Lucene document. And the author of the PDF file is stored as the value of the field named "author". Those fields will be stored in the index according to their options, which are define by the user, or the application, some of those options are: "*Stored"*, *"indexed"*, *"termvector*". Usage of those options, are developed in chapter4. The next important concept is the analysis concept

2.2.5 The analyzer

The analysis of a given Lucene Document stored in occurs after their fields have been populated with contents supplies by the parser or the document handler. Analysis belongs to the indexing process, and is done during the index writing process. The idea behind Lucene analysis is depicted in the FMC[17] Block diagram below.

17 FMC: Fundamental modeling concepts. URL:http://www.fmc-modeling.org/

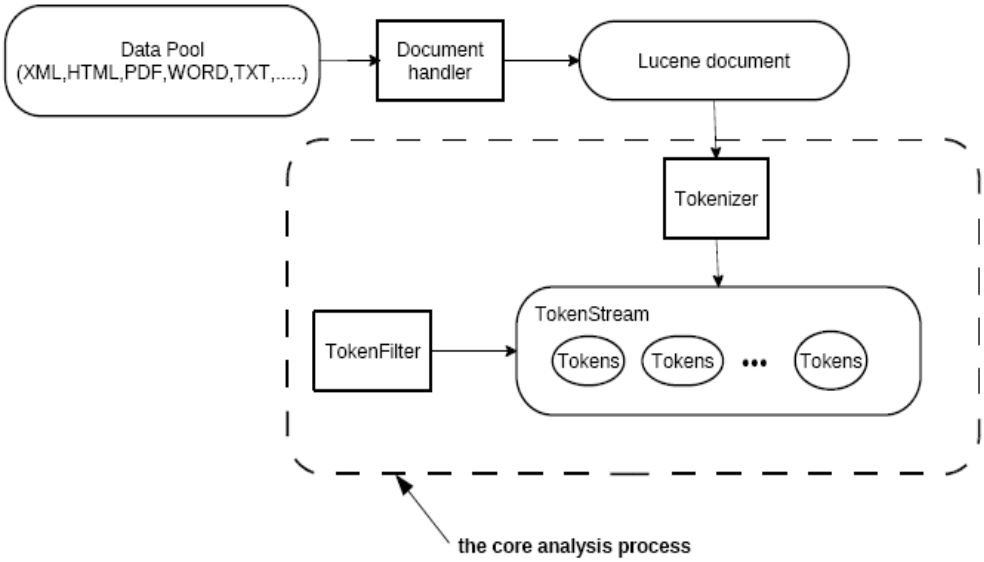


Figure 2: The Analysis process(concept)

The core Lucene analysis process consist of the following steps:

Step1: Lucene Document is broken into small indexing elements called tokens, this process is therefore called tokenization and is performed by a Tokenizer. Once Tokens are produced and gather in a tokens stream,

Step2: the TokenFilter can be used to consume that tokens stream and filter useless tokens out of it. The way a document is analyzed depends on the Lucene's user application parametrization. For instance white spaces can be removed from the token stream using a WhitespaceAnalyzer which tokenized with a WhitespaceTokenizer but uses no Token Filter. On the contrary English stop words can be removed by a StopAnalyzer,which uses a LowerCaseFilter and a StopFilter to filter LetterTokenizer's output Stream.

Chapter 3.3.2 provides a detailed study of the Lucene analysis strategy, for instance: which steps are encounter during tokenization? Which design pattern is used for the construction of a Tokens Stream ? and more.

2.2.6 Indexing

Indexing in the field of information retrieval and in Lucene in particular is collecting , analyzing , storing data from different textual contents to allow fast retrieval of information inside those data. The index is not an assemblage of documents(pdf,Html,Xml,...) in which one can search , rather a list of terms, mostly words , being in these documents and which can be retrieved . Those terms are stored as lists of called **posting list** . A posting is a list of Lucene document's identification numbers for a giving index term in the terms dictionary. The terms dictionary is a list of terms that have been extracted out of the original document. After parsing, those terms were contained into field's values for a given field's name. Once those fields are analyzed ,the IndexWriter write for each term of a field the corresponding Lucene document identification number(docID). The same term can occur in different Lucene document, therefore the document's numbers in that term postings are merged together in a list that constitute the postings list of that single term. Here goes a representation of the indexing concept in Lucene.

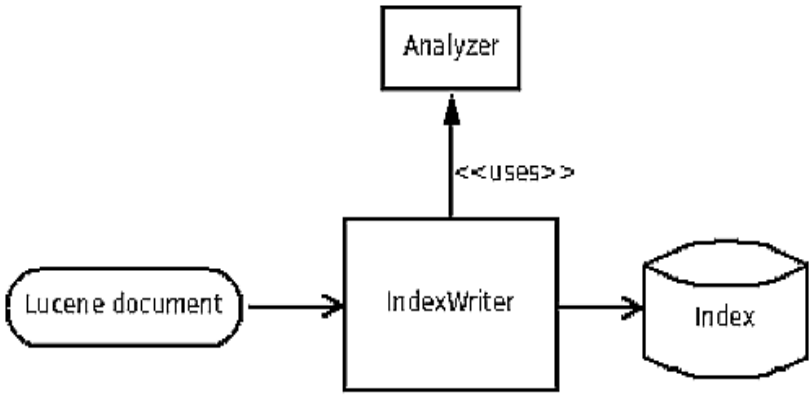


Figure 3: Indexing with Lucene

A user application must create an IndexWriter, that consume Lucene Documents. The IndexWriter first calls the analyzer, to break and filter Lucene fields. Once the analyzed fields are returned to the indexWriter,this one can write them into the index. How precisely the IndexWriter adds fields to the index is explain in chapter5, were the indexing process is study. Additional Information can also be stored inside the index together with fields, these are field's respectively Lucene document's attributes like the frequency of a term in a Lucene Document, the maximum number of Lucene documents in an index and much more.

2.2.7 Index search

Generally Lucene supplies components to search inside the index and to obtain hits on the searched query. *QueryParser* and *indexSearcher* are the main components involved in most Lucene based search engines. After the Index have been constructed with postings lists , the search application will retrieve the user query in the index. It first analyzes the user query using the same analyzer as in the indexing process, then transform the user query in to a Query object with respect to the Lucene query language .In short here is the concept behind it:

The QueryParser: dissects a user query Strings and passed it to the IndexSearcher .The user query may be a single word, a sentence, a conjunction, or a disjunction of terms it can also have another form. To put it differently, Lucene defined a grammar according to which a query is a clause or a set of clauses connected with each other through a condition, or symbols . A clause is a set of terms with symbols like wild cards (*,?) or like tilde; the term is the textual part of the query, that is a word or a number. A condition between two query terms can be a “SHOULD” to specify that the term can be found in the index or not. MUST means the term must necessary be found in the index. The QueryParser translates an AND operator in a query as a MUST and an OR operator as a SHOULD. Once the user gives a query to the search application, the QueryParser.java class process the query with the help of the Lucene query grammatic define in the **“QueryParser.jj”**class. QueryParser.jj is the part of the Lucene search library, that defines semantic elements that a query should have, and the interpretation of those element. For instance AND, OR, +, NOT are semantic elements that can be used in a boolean-like user query, their meaning are provided by QueryParser.jj. Additionally, this class also gives the possible syntax of a clause in a query. With all those specification the Query Parser can translate user queries according to the

grammar define in QueryParser.jj which is generated by the Java compiler compiler (Javacc [26]). The query syntax will be developed in chapter 6.

The IndexSearcher: is created using an analyzer, an IndexReader, a Scorer and a results collector. QueryParser displays the top hits for a query. Because the index is made of postings lists(terms and docIDs), the IndexSearcher can access them trough an IndexReader reading the index. The content of the original document, which references are stored in the posting lists are display. Moreover, the IndexSearcher can display Fields values, that have been previously stored in the index during indexing process. The figure below depict the search implementation process of a Lucene based application.

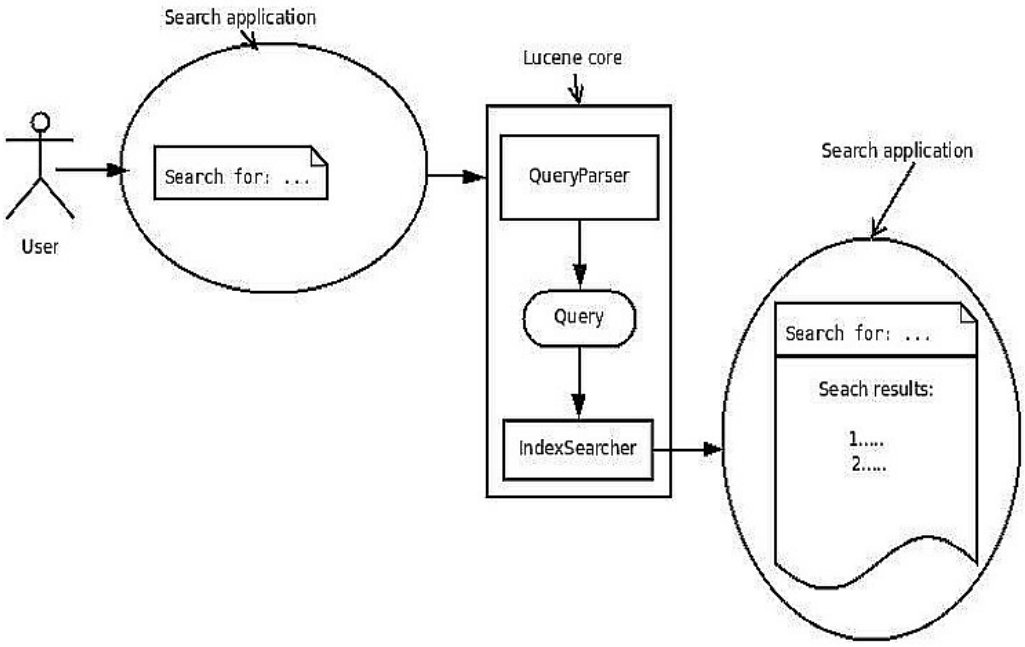


Figure 4: Lucene index search(concept)

With these concepts in mind, let's have an overview of Lucene components structure

2.3. Overview of the compositional structure of Lucene

An overview of Apache Lucene and its compositional structure is illustrates in figure5. Any application using Lucene must first of all transform its original data, into Lucene Documents.

For this purpose a Document Handler interface is needed. Recall from the previous section that, the Document Handler interface allows the extraction of information like textual contents, numbers and meta data from original documents and provide them as Lucene Documents. These are used for further processing during indexing and search.

Each common document type like Html, Pdf, Xml and so on needs a specific document parser to extract its contents . Document parsers are not part of Lucene core, a lot of them are available as open source file and usually called document parser. For instance: Jtidy a parser for Html documents, Pdfbox for Pdf documents and SAX for Xml. There is a

DocumentHandler interface provided in the Lucene contribution Library. Each class implementing this interface returns Lucene Documents. So a custom document handler may extract contents with open source document parser, and use the DocumentHandler interface to transform their contents into Lucene Documents. In Chapter 4 we will see an example of implementation of a document handler.

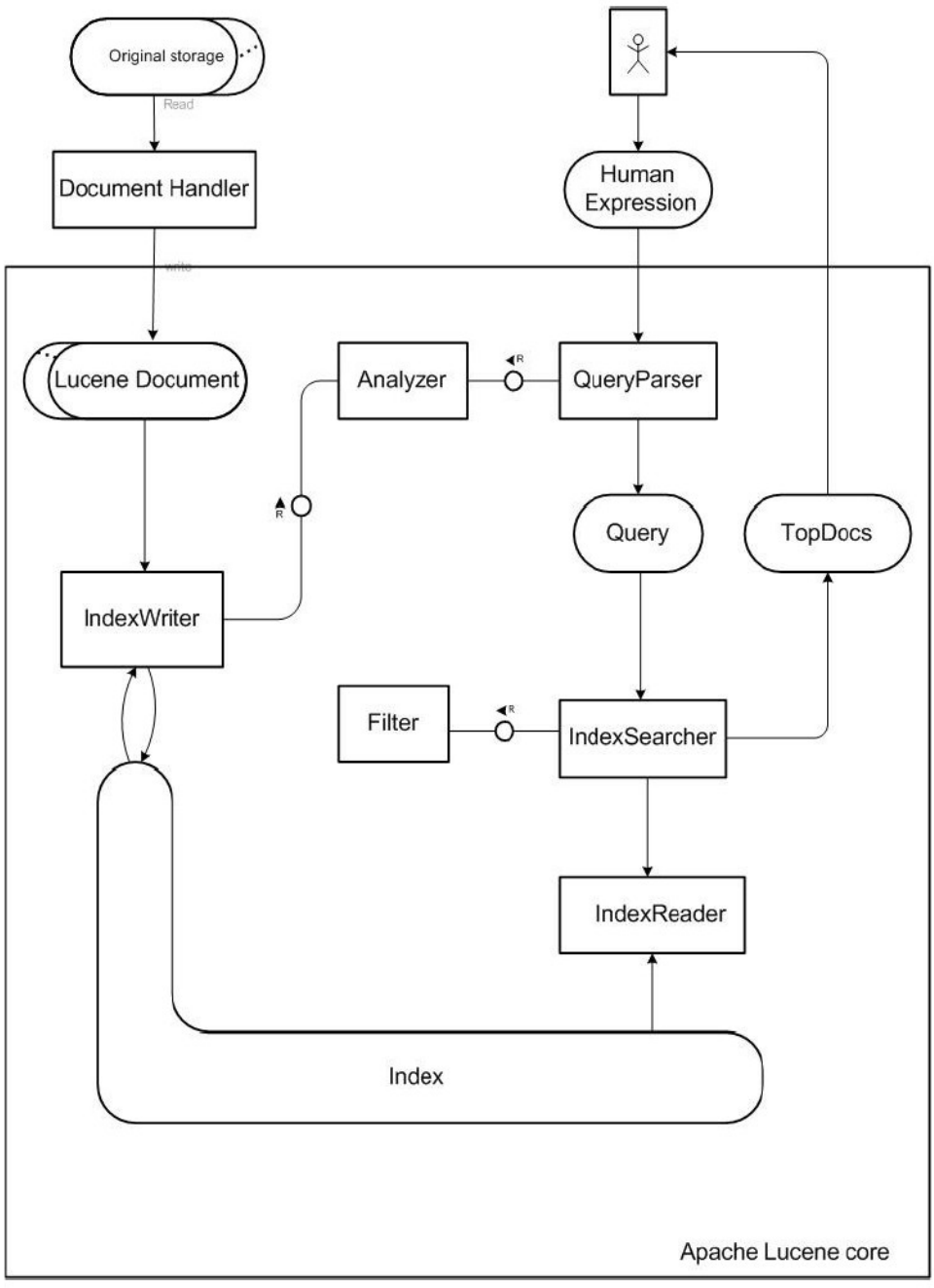


Figure 5: Lucene architecture overview

Let's assume that we have a Lucene document at hand. A Lucene document is organised as a sequence of Fields. A Field is a pair of name and value for example

<filename,"broder_websearch.pdf"> or *<author, "Andrei Broder">* or *<"contents," central tenet of classical information retrieval is that the user is driven by an information need.">*. The name of a Field is a string like *filename*, *author* and *contents*. The name of the Field is application specific and determined by a document handler. The values of the Field are sequences of terms, which are atomic value tokens of the Lucene document. For example in the field (filename, "broder_websearch.pdf"): filename is the name of the Field and "broder_websearch.pdf " is the value of this Field. Fields have different attributes which determine how they are used in the indexing process, for example Fields may be stored in the index or not, they may be indexed in a certain way, they may be tokenised in terms or not. We will talk about the construction of a Lucene document and its components. In chapter 4.

The IndexWriter analyses and stores Lucene Documents in the index according to the fields' attributes . There are essentially two forms of store: either in the File system using a File System Directory class to implement this directory or in the main memory using the RAM Directory class . Note that a new Lucene document can be added incrementally to the index, and each time this is done, the analyser is used by the IndexWriter to produce tokens from the Lucene Document. The IndexWriter design pattern can be define as : IndexWriter using the provided Analyser as a Strategy for index writing. The way to break the Lucene Document into terms depends on the chosen Analyser.

As said in the previous section, an Index consist of postings lists; it is stored as flat files within a directory. We will get into the building of the index in the fifth chapter.

In brief, we have just provided an outline of the indexing process, by browsing the important components and data structures involved. But, the existence of the index is the precondition to the searching process. A User has to enter a Human-readable Expression called query String, into the Lucene based search Application. The user can use a syntax to write its query expression. This syntax is defined in the so called Query language . For example :"An?rei" or "And*i" search for a single character to replace the "?", or for multiple character to replace the "*" this kind of query is called WildcardQuery. So to search for "Andrei" the user can enter one of the two wildcard queries. This Query syntax is interpreted by the QueryParser .

In fact, the Query Parser transforms the user query String to an object of type Query. There are different types of Query like WildcardQuery , fuzzyQuery, booleanQuery and so on. We will get in detail on the Query and QueryParser in chapter 6. After the Query is Analysed, it is assign to the IndexSearcher. This one retrieve the postings lists in the index for the matching terms according to the Query. It uses the IndexReader to access the index. A Filter can also be needed to specify restrictions on the search results. The Filter provides a mechanism to permit or prohibit a document in the search results. For Instance a spanFilter mark the occurrence of the word "Andrei" in all documents. The IndexReader is used to open and read the index in order to get terms matching with the search Query. The IndexSearcher returns the results as TopDocs to the user .

The TopDocs is a List of Document number, which can be used to get the stored fields from the Index. We will see in a further chapter, that the documents (Lucene Documents) are stored in the index in the form of a Dictionary of terms and postings lists . The Lucene Document id or Document number(docID) are the one returns to the user or the application, there are pointers on(references to) the original files.

The stored fields can also contains path, or URL to the original document. Those are used by the user or the application to access to the contents of the original files matching the search.

Based on that Schematic, it is clear that Lucene doesn't do everything in a search

application. This is what Lucene can do :

- providing an Index from an amount of data from different types. This process is called indexing
- parsing user query
- searching for occurrences of query terms within the index.
- computing statistics over indexed documents .These contains different information including the documents where the query expressions is mostly found, the number of times the expression occurs in one document and the hits on this query .

Lucene offers several functions and modules. Briefly here is what Lucene can not do:

- Process Management : The application programmer has to select suitable components available in Lucene, those who satisfies its needs. For instance using a GermanAnalyzer for the indexing process does not produce the same index as contents analysed with a FrenchAnalyzer. Indexing and search are managed by the application.
- Files selection: The application, respectively the programmer decides which Files are to be indexed either Pdf , Xml or Html Files.
- Files parsing: Till this date, Document Parsing is not a function of Lucene. It is can be done by combining the document handler interface with any common Document Parser . We would see in chapter 4.3 how a Pdf document is transform into Lucene Document.
- Display Search query: This is done by the user application

chapter 3 Overview of the architecture of Apache Lucene

When building a home it is important to first know the global plan of this house including the number of floors and rooms and their disposition before getting down to the layout of each floor and each room. Comparatively, before studying in detail the internal structure of the various components of Lucene, it is necessary to get an overview of its basic organization and its hierarchical structure. Only the most important components involved in the process of indexing and search will be the subject of our study in this chapter. The actual Lucene Version used here is version 3.0.1.

3.1 Overview of the code architecture of Lucene

We begin with a concise description of the packages of Lucene involved in indexing and search.

Package org.apache.lucene.document

The abstraction of a Lucene Document represents the documents provided as a list of fields. Figure 6 shows a class diagram of the class Document and its components. A Lucene Document contains one or several Fieldables. The Fieldable interface defines the operations that can be done with a Field to manage the contents of a Lucene Document. The abstract class AbstractField implements Fieldable methods, that are common to Field and NumericField.

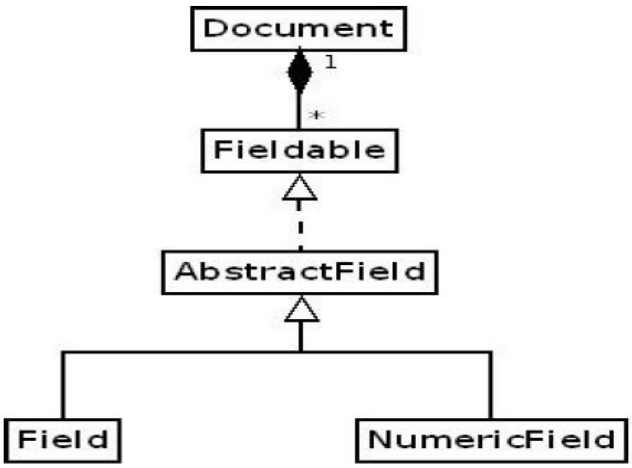


Figure 6: A Lucene Document: class diagramm

A Field represents a name - value pair; the Field's name is mostly a String or a word like "t*itle*", its value is either a String or some content provided by a reader. A figure of the structure of the class Field is given in *Figure 7* .

According to the Lucene 3.0.1 Api, the following operations from the basis interface *Fieldable* are used on Fields to personalize indexing and search.

- **Display the Field's value**: The value of a Field can be displayed during search as String or provided via a reader; for example "*Grant Ingersoll*" is the String value displayed when searching for "*grant*". The value of the tokens in the field can also be displayed as TokenStream value.
- **Change field's value**: A field's value can be changed during indexing for Instance *f.setvalue("webportal")* set the value of the Field named *f* to "*webportal*".

- **Setting indexing parameters for a Field**: A Field is either set as indexed, stored or tokenized; it can have stored term vector or a boost factor . More details about Lucene field and its parameters are discussed in the next chapter.

Some of these operations are also applicable to NumericField. The only difference between Field and NumericField is that the first one deals with textual content whereas the second one deals with numerical content.

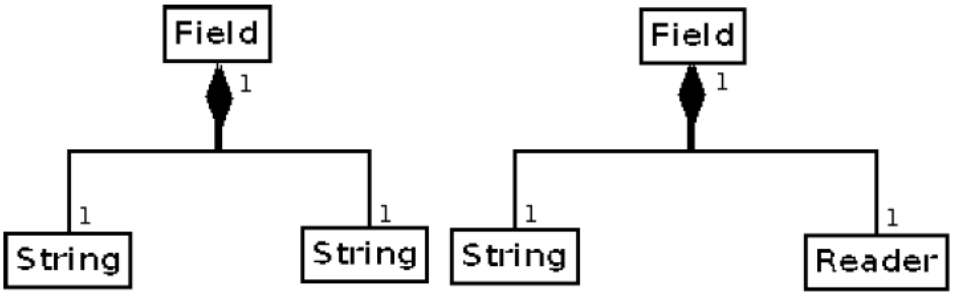


Figure 7: A Field is a name-value pair, implemented either as a (String,String) pair or a (String,Reader) pair

As illustrated below, a Numeric Field contains a String and a Number. The String is the name of the Field like "*price*" and the Number is a number of type *int*, *long*, or *double*, the Number can also be the *long* value of a Date obtain by calling the *Date.getTime()* method. A numeric Field is mostly not stored but indexed. More Information about *NumericField* is found in the next chapter.

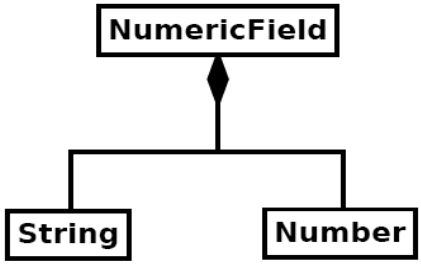


Figure 8: The sturcture of class Numeric Field

Generally, a user application will start the indexing process by parsing the raw documents and constructing a Lucene Document by extracting the contents from the given document and organize it into Lucene Document. Once fields are populated with name and values, they are they are prepared for the indexing process. During indexing Lucene Documents, they are analyzed during the analysis process. In the following section, the code structure of the package responsible for this analysis process is discussed.

Package org.apache.lucene.analysis

The Lucene analysis performs the *tokenization* and filtering of Lucene documents intended for *indexing*. During searching Lucene Documents, the analysis of the user query is also done by an analyzer, it is recommended to use the same analyzer that was applied for indexing. Actually, there are about six different analyzers in the Lucene core namely: *KeywordAnalyzer, PerfieldAnalyzer, SimpleAnalyzer, StandardAnalyzer, StopAnalyzer* and *WhitespaceAnalyzer*. Additional analyzers are available in contribution packages to Lucene core; most of them are analyzers for specific languages there are for example analyzer for

the German language, French language, analyzer for Arabic and so on . Let's start by the central component, the Analyzer.

The Analyzer provides methods to create streams of tokens from the Lucene Document . Analyzing a Lucene document is generally done in two steps:

First the analyzer creates a T*okenizer* which reads Lucene Documents provided and splits its fields into *tokens producing* a tokens stream. Then this stream of tokens is passed through the T*okenFilter* which cleanses them from useless or unwanted items. The overall structure of an analyzer is illustrated in the figure9 .

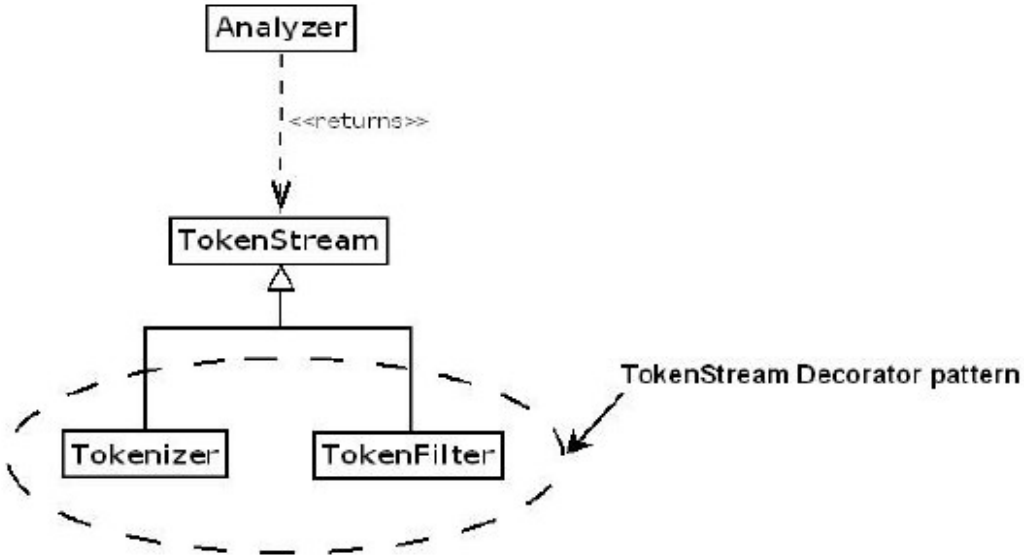


Figure 9: The structure of class Analyzer

The design pattern used by TokenStream is the decorator pattern [61] .The abstract class Tokenizer is the base for Tokenizers and TokenFilters that can decorate the TokenStreams.

The Tokenizer is a TokenStream that breaks the name or the contents of a Lucene Document's field into tokens with the corresponding attributes , that have been selected during indexing. There is a variety of Tokenizers available in the Lucene core. Some of them are: C*harTokenizer*, S*tandardTokenizer*, L*etterTokenizer* in section 3.2 we would see some examples of Tokenizers.

The TokenFilter receives a Token Stream from the Tokenizer, filters it and returns a refined TokenStream . This is the mechanism of decorating the result of a Tokenizer by a TokenFilter, of course,other TokenFilters are decorated . The usage of the decorator design pattern in the structure of TokenStream is depicted in Figure 10.

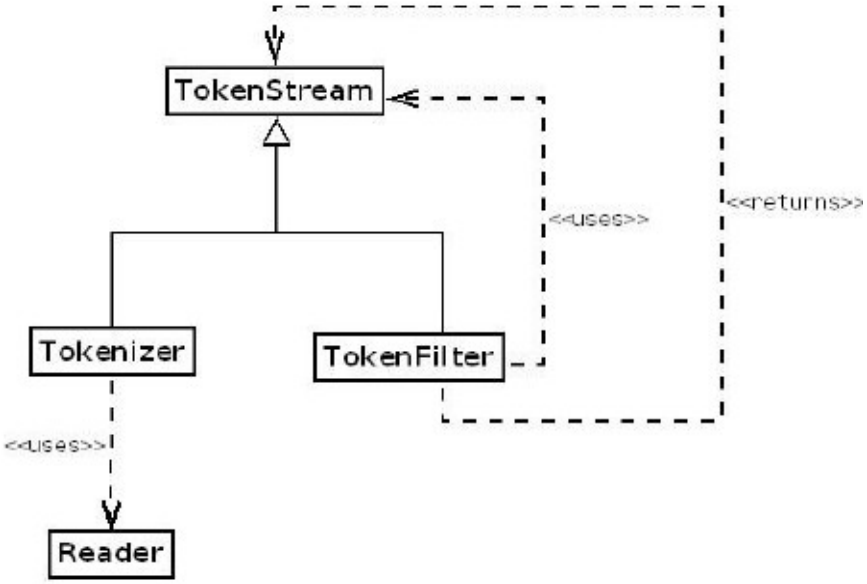


Figure 10: TokenStream decorator pattern

To explain this design pattern more concrete , we will focus on the Tokenizer and the Tokenfilter, then we will talk about token and token attributes.

Tokenizer

We took a text snippet of 2 words from our example file “broder_websearch.pdf”. Let's tokenize the following text using the StandardTokenizer and the following TokenAttributes: TermAttribute,TypeAttribute, OffsetAttribute. The output of the Tokenizer would conceptional be like in the table below, the table shows the first 13 characters.

Offset Attribute	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Term Attribute	S	e	a	r	c	h		e	n	g	i	n	e	s
Type Attribute	Word													

Table 2: Tokenizer breaking input into stream of tokens

The Tokenizer use its incrementToken() method to add the tokens to the resulting TokenStream one token after another until there is no more token attributes to add for the last token. In the example of table2 the *TermAttribute* of the Token is the text of the token, its offsets are 0 and 13 which are the start and end character offset .It's lexical type is Word. The *CharTokenizer* and the *KeywordTokenizer* are subclasses of the Tokenizer as well (see Table2 above). They are used in combination with one or more TokenFilters. On the disk, tokens are store in an array.

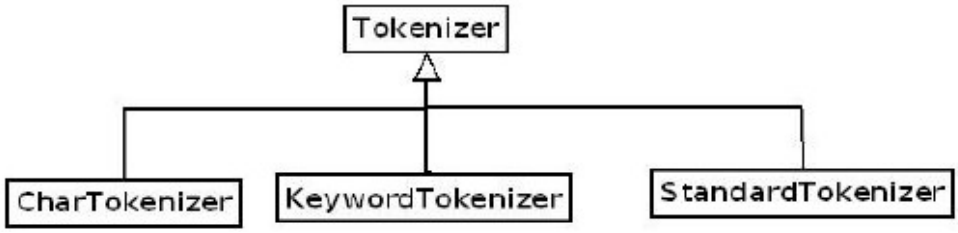


Figure 11: The Tokenizer

TokenFilter

The TokenFilter get its input from the Tokenizer. It filters the given tokens stream and returns them after processing. A variety of filters used for analysis are available in the Lucene core Api. Figure12 gives a list of some of the available Lucene TokenFilters.

To have an idea of how tokens filtering works, let's continue the example. The input “Search engines” is analyzed by a StandardAnalyzerwhich uses a StandardTokenizer with a LowerCaseFilter, a StandardFilter and a StopFilter. The Tokenizer has already divide the text into tokens .Now the tokens are being passed through differents filtering processes:

The *StandardFilter*: this filter normalize tokens, for instance it splits words at punctuations, remove hyphens from words. In our example we obtain:

S,e,a,r,c,h, ,e,n,g,i,n,e,s, ,u,s,e,s, “,s,p,i,d,e,r,”, w,h,i,c,h, ,s,e,a,r,c,h, ,(,o,r, ,s,p,i,d,e,r,), ,t,h,e, ,w,e,b, ,f,o,r, ,i,n,f,o,r,m,a,t,i,o,n

The LowerCaseFilter: this filter transform all letters to lower case.

s,e,a,r,c,h, ,e,n,g,i,n,e,s, ,u,s,e,s, “,s,p,i,d,e,r,”, w,h,i,c,h, ,s,e,a,r,c,h, ,(,o,r, ,s,p,i,d,e,r,), ,t,h,e, ,w,e,b, ,f,o,r, ,i,n,f,o,r,m,a,t,i,o,n

And finally the StopFilter removes so called stop words like: for, the, or, which.

So the terms attribute of the tokens becomes:

s,e,a,r,c,h, ,e,n,g,i,n,e,s, ,u,s,e,s, “,s,p,i,d,e,r,”, ,s,e,a,r,c,h, ,(, ,s,p,i,d,e,r,), ,w,e,b, , ,i,n,f,o,r,m,a,t,i,o,n

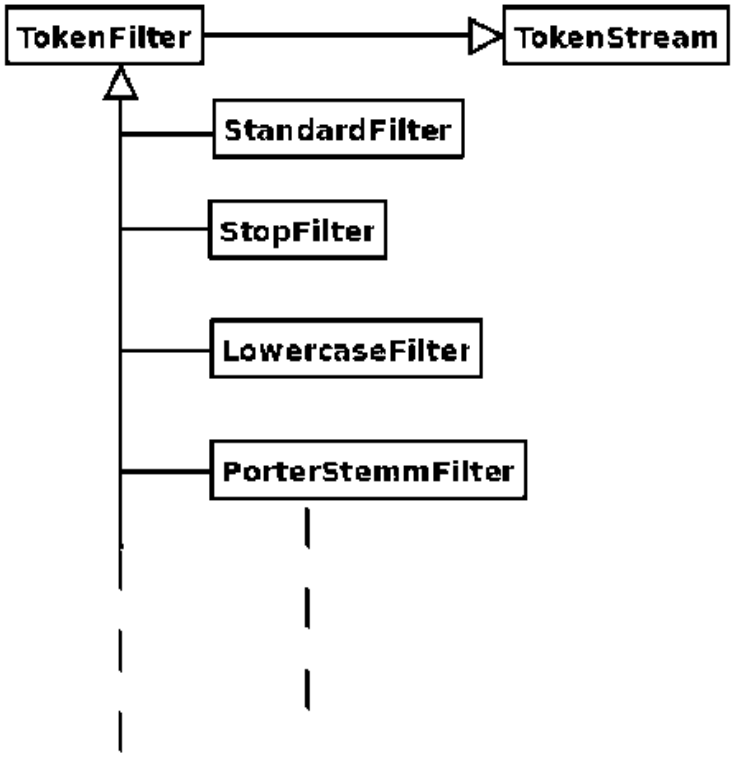


Figure 12: The TokenFilter structure overview

More about the different types of Tokenizers and TokenFilter will be discussed in section 3.2. We have been talking about tokens been created and filtered. But what is a token precisely? Here goes the explanation.

Token: The unit of indexing and search

Token						
Start offset	Term	Type	Payload(optional)	Position increment	Flags	End offset

Tokens are the elementary, atomic indexing and search particles. A token is often improperly called a term, in fact it is a collection of attributes representing an occurrence of a term from a (Lucene) field. Each token contains a **Position increment**, **Offsets** (a start and an end Offset), a **term** which is the text itself and a **type.** These values are also called **TokenAttributes**. The table above shows what a token can contain. In the current version of Lucene (3.0.1), there is more flexibility in the creation of tokens, they can be customized to fit an application's needs. Thus said, a Token is adaptable by attributes. These attributes are necessary for the storage of Tokens' information in the index. The 3.0.* version of Lucene are based on the Attributes implementation and not directly on Tokens. There is more flexibility in the way to add Tokens to a *TokenStream* when using attributes. In previous version of Lucene, attributes of a Token were set up using getters and setters method, now classes are provided to define each part of a Token, such as: *OffsetAttribute* is the class that determines the position of a Token relative to the previous token. This is the start offset or the end offset of the token. Let us go deep into the structure of a token and the possibility for

Lucene based applications to adjust the fields through tokens attributes. Here is the code structure of the token attributes.

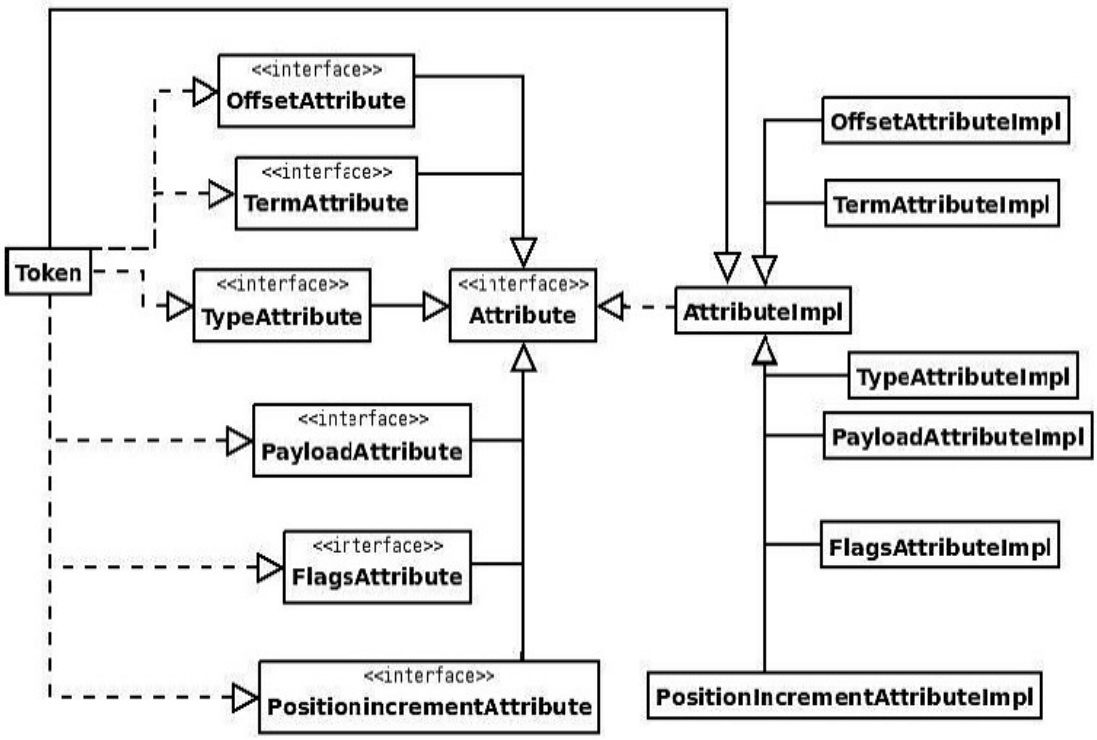


Figure 13: Token attributes factory (architecture)

We know from the discussion of the TokenStream decorator design pattern that an Analyzer first calls a Tokenizer to build streams of tokens, then a TokenFilter to purge those tokens. According to the API, this is how the Tokenizer should be called:

Tokenizer(AttributeFactory factory, Reader input)

This call causes the creation of a token stream by processing the provided input using the given factory. Before processing the field's value, all Attributes are reset to the initial value by the method *AttributeSource.clearAttributes()*. Afterwards a token attributes factory is used. This starts the mechanism illustrated in the figure above. These are the steps involved in the construction of tokens :

- The Tokenizer (this is the same for all TokenStream) creates an instance of the factory. This is an object of type **AttributeSource.AttributeFactory** let's call it the **factory;**
- The factory creates objects of type**AttributeImpl**: To do this it takes the class name of the attribute supplied by the Attribute interface, then append Impl to it. For instance, to produce *PayloadAttributeImpl* attribute, the factory reads the interface name *PayloadAttribute* and adds Impl at the end of it. It is not just the name that are carried over, but the methods specified in each Attribute interface are implemented in the corresponding AttributeImpl class. For example the methods getPayload() and SetPayload() are declared in the PayloadAttribute interface and implemented in the PayloadAttributeImpl class. .

- Later on, when attributes are created, Token implements their interfaces, and extends each *AttributeImpl* class. Given this mechanisms, each method in an AttributeImpl can be rewritten by the user application. To put it differently, the new Lucene API gives the possibility to use only those attributes of tokens that are needed by the user application. For compatibility reasons it is also possible to use Tokens, which provide access to all attributes. It is the choice of the user application to use the Attributes API or the Tokens without user-defined adjustment of attributes.

Once attributes are constructed, the method *TokenStream.incrementToken()* is called by the IndexWriter.DocConsumer for each token. This method advances the token stream to the next token after consuming attributes of the previous token. The illustration below shows the workflow of the new Lucene TokenStream as described in the API[18]

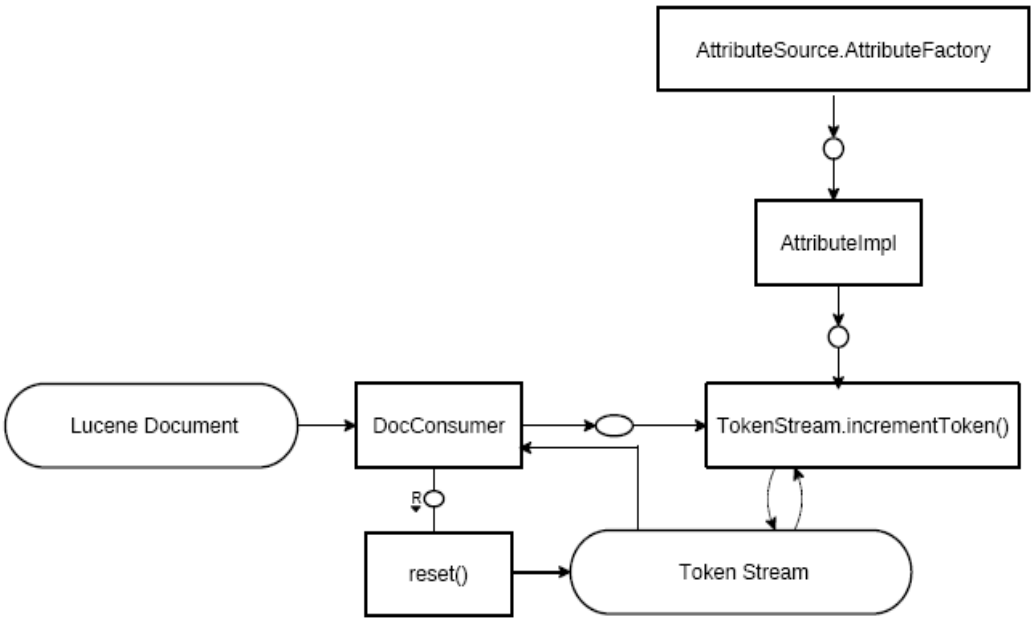


Figure 14: The tokenization process(FMC Notation)

The Tokenizer uses the TokenStream's *increment()* method to browse and advance each token in the TokenStream storage (the storage element used is a reader) until there is no more attributes for that token. This mechanism of the Tokenizer is exploited by the DocConsumer. It consumes each token that it receives from the TokenStream storage. The Tokenizer also provides the consumer with attributes for each token added. Attributes are created by the AttributeImpl class as explained before.

On the whole, we observe that the analysis consists in breaking down Lucene document's field into tokens and filter them.

In the new Lucene API (library) a token is constituted by Attributes, provided by the *AttributeFactory*. In the actual 3.0.* version a Token has six Attributes which are its components: Term attribute, Offset attribute, Type attribute, Position increment attribute and meta data like Payload attribute and Flag attribute.

18 http://lucene.apache.org/java/3_0_1/api/core/org/apache/lucene/analysis/TokenStream.html#incrementToken

Another point worth mentioning is that indexing depends on the analysis. In fact, we will see in the next section, that the IndexWriter uses the DocConsumer class of the index package to consume Lucene document before writing them into the index. This means analysis provide the strategy of adding analyzed tokens to the index through the DocConsumer. Now let's talk about the index.

Package org.apache.lucene.index

The key of indexing Files in the creation and storage of Lucene documents' fields extracted from those Files. Conceptionally, a Lucene Index is made up of files and/or segments and document identifications(document Ids). Files in the index contains information about fields that have been indexed; this can be the field name, the field terms frequency in a Lucene documents. More about the index files format is given in chapter 5. Segments are sets of indexes, in other words indexes are merged into one segment according to a policy called merge policy. The objective of this part is to show which components of the Lucene index package are used in indexing, index search and index storage.

To begin with indexing Figures 15 ,16 and 17 depict Index internal dependencies. They illustrate the main classes of Lucene index package involved in the indexing-search process.

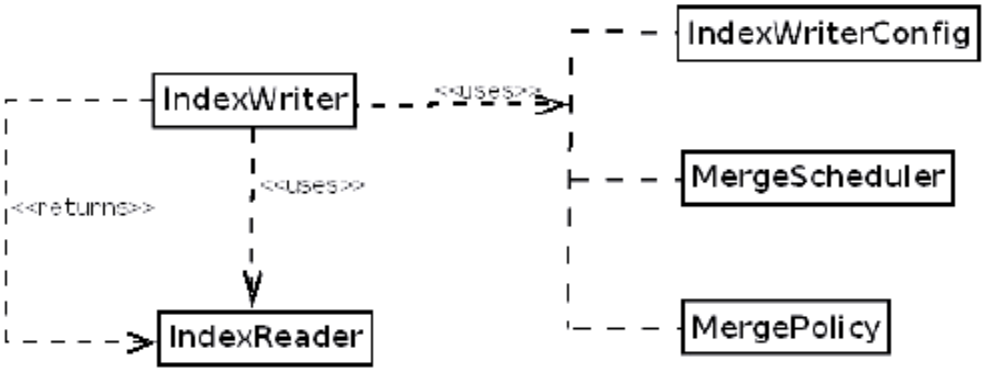


Figure 15: IndexWriter internal dependencies

According to the Lucene core API, an *IndexWriter* creates and maintains the index.

An IndexWriter can maintain an already existing indexes or create a new one.

We first consider an *IndexWriter* that maintains an existing index. The *IndexWriter* opens the index using an *IndexReader*. If there is a new or modified segment to merge in the index, the *mergePolicy* is invoked by *the IndexWriter* to specify how the segments should be handled during merging. The MergeScheduler manages and merges segments, so that there is no concurrency between two merge operations. By default 10 Indexes can be merged together in a single segment. This value can be modified to improve indexing or search. The Merge algorithm is discussed in chapter 5.3.1. It is also important to note that this algorithm is executed each time the IndexWriter's addIndexes() method is called to merge various indexes into one Index.

Next we discuss an IndexWriter,that creates a new index. *It* needs an IndexReader to access the index directory. In the 3.0.1 API , all configurations for an *IndexWriter* are stored in an *IndexWriterConfig* object. The purpose of those configurations are to enable the developer to set up how the index should be accessed, how terms are stored in the index

and much more. Illustration16 describes the IndexWriter Configuration and its strategy for setting users parameters during IndexWriter construction.

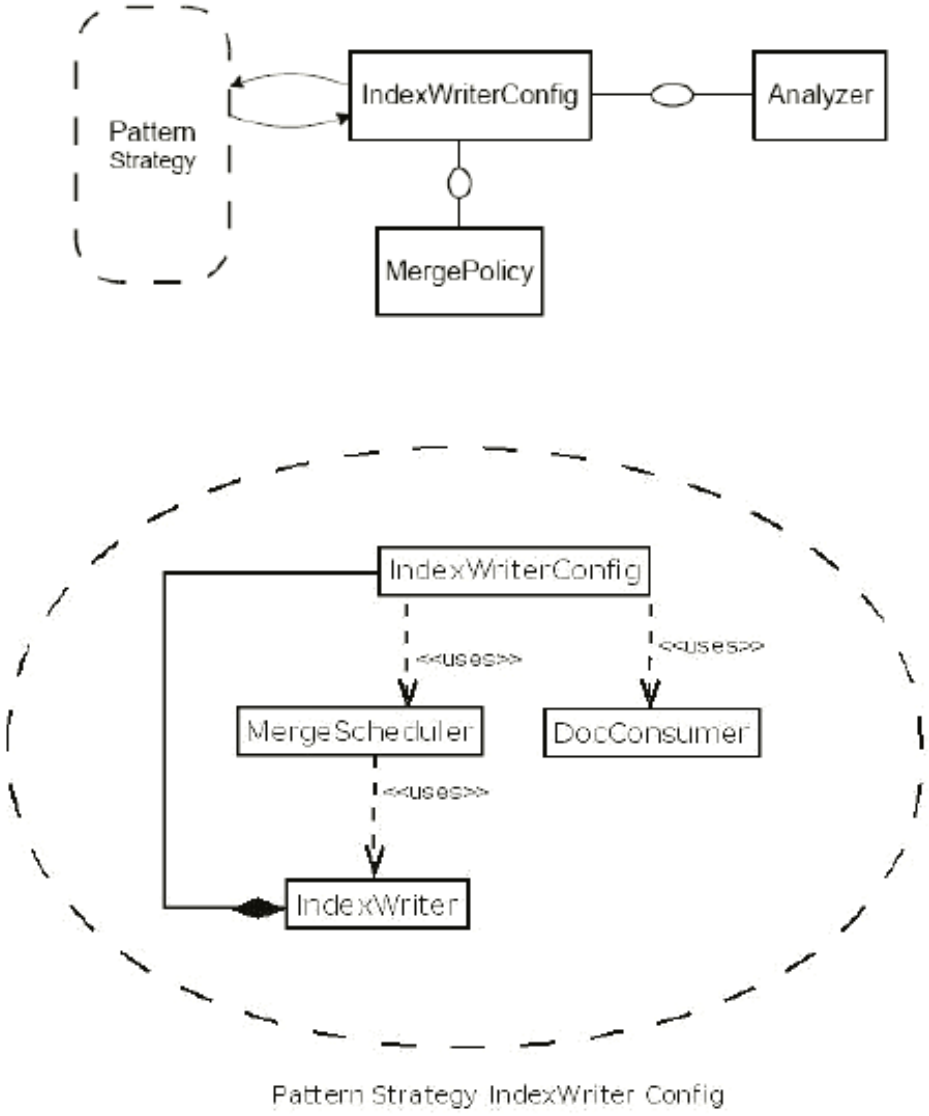


Figure 16: IndexWriter configuration for indexing

The *IndexWriterConfig* strategy is divided into two stages. The first stage concerns Lucene IndexWriter processing and the second one is for merging specifications and rules.

While processing index writing, *IndexWriterConfig* allow the application/the developer to set certain parameters like:

- the open mode .This shows how an index is opened
- The maximum number of terms inside the index.
- The commit point, this is the point from which the index should be committed in the directory in file system. It can be said that a commit is the action of uploading any modification on an index inside the directory containing it.

- The interval between two terms in the index.
- The maximum number of deleted terms that can be buffered in the memory, before they are definitively dropped from the file system
- The amount of memory that my be used for buffering added documents,document deletion , before those are flushed to the directory in the file system.
- The minimal number of documents required before the buffered documents are flushed in the directory as a new segment.

The IndexWriter itself consume field's values input using the ***DocConsumer***. The *DocConsumer* uses the *incrementToken*() method of the TokenStream class to advance the stream of tokens to the next token. Before doing that, it retrieves attributes from the stream of tokens, and store references to those he wants to access. The usage of the Attributes is explain in the previous part concerning the package org.apache.lucene.analysis. DocConsumer is actually one of the important class of the index. It is a bridge between the analysis mechanism and the index writing. The IndexWriter uses it to access Lucene document field's. The DocConsumer goes inside the tokenization machine, takes the Lucene fields first collect available attributes through the TokenStream.incrementToken() method. Each Attributes matches a part of a Lucene Field's value. We show in the previous section, the set of Attributes building a Token .Collecting these Attributes means filling the right value into the right attributes, the value of an Attribute is modify by the user application:

- The **TermAttribute** contains the text of the token, this is a word inside the field's value.
- The **OffsetAttribute** contains the start and end character offset of the token,
- The **PositionIncrementAttribute** contains a number, which is the position of this token relative to the previous one
- The **TypeAttribute** is the lexical type of the token. The type of a String is “word” for example.
- The **PayloadAttribute** is a meta data that can be store with the token. It is optional.
- The **FlagsAttribute** is a number that can be set-up as flag . This can be used for example from one Tokenizer to another

After gathering attributes for each token the DocConsumer returns tokens to the IndexWriter which store them into the index. Let's talk about merging.

Concerning the merging specifications, those can also be set up before the Mergescheduler merges indexes .Some specifications are:

- The *IndexDeletionPolicy*, this is to determines when a commit should be deleted.
- The mergePolicy determines which merges should be done when any modification occurs on the index.

Apart from those two steps, one can set-up how long an IndexWriter Process should wait for another one to write inside an index. This is called a *write Lock*. That is to say, the Lock ensures mutual exclusion while writing into the Index, which is the shared resource.

In brief, all those parameters give the search application more control on the Index Writer process. This is an outstanding example for Lucene flexibility.

While IndexWriter creates a new index with Lucene documents or modify an index by

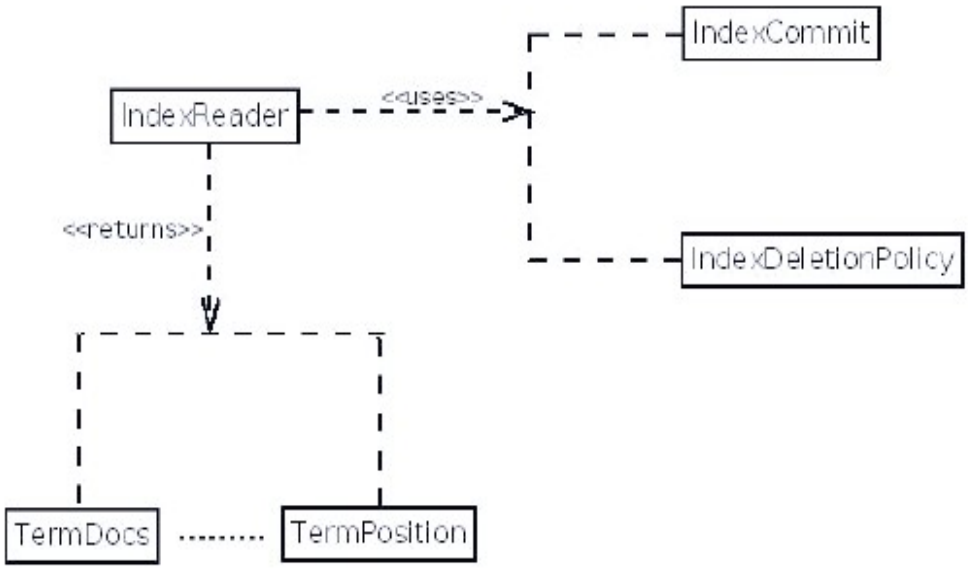


Figure 17: IndexReader internal dependencies

deleting fields or merging indexes, the IndexReader reads the index contents and returns most of the time information regarding terms.

Generally, an IndexReader provides methods to read the Index. The architecture depicted in Figure 17 shows the internal dependencies of an IndexReader.

The classes *IndexCommit* and *IndexDeletionPolicy* can both be used to open the index.

The *IndexCommit* performs any change made in an index generally by the *IndexWriter*, while adding or deleting a segment. A commit is done, when its segment's files are stored into the directory containing the index. As a result of a commit, the directory is returned to the IndexReader open() method for read only access.

The *IndexDeletionPolicy* ensures the deletion of old index commits. Old changes completed by the IndexWriter are stored in a list and passed to the Policy which deletes them. The deletion can occur when a new IndexWriter is created. This is called *onInit-deletion*, or it can occur each time an IndexWriter commits new segments into the index; this is called *onCommit-deletion*.However the most often used way to open the index is to use a Directory. In this situation, Figure 18 shows the IndexReader's external dependencies. An IndexReader requests from a directory object to access the index .This directory reads the index and returns different values stored in the index. As seen before, each time an Index is opened for reading or writing an IndexCommit is necessary to manage changes made by the IndexWriter. In addition, the policies to use during a commit can also be modify through the IndexDeletionPolicy.

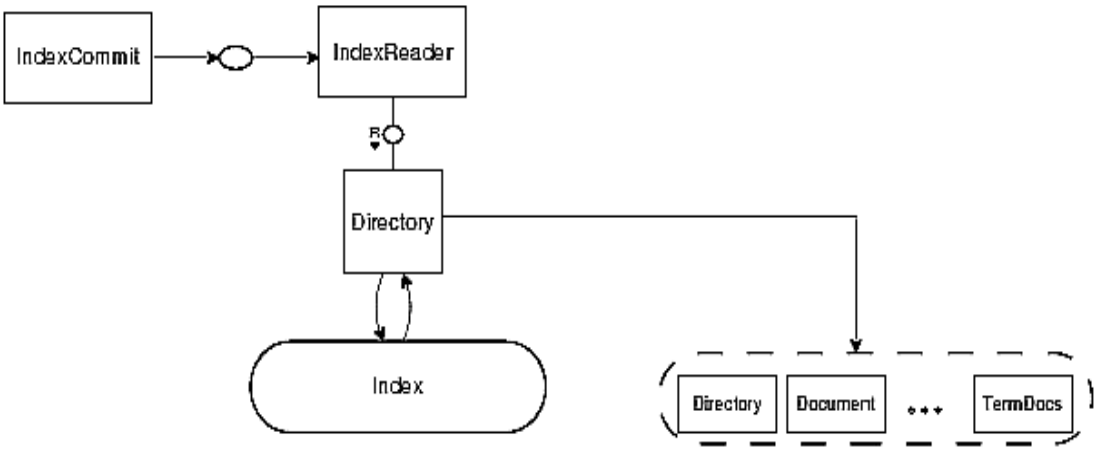


Figure 18: Accessing the index using IndexReader

As result values, IndexReader's Open() method can output different Objects .Briefly, the following objects can be taken out of an index:

- document(i): the stored fields of a non deleted Lucene document with Id = i in the index. By the way, the index contains posting lists which are made up of Lucene document identification (document ids), terms, terms frequency and more. The document() method reads the document id from the posting lists and at the same time the stored terms respectively the stored fields corresponding to this id.
- Document(i, fieldSelector): A non deleted document at a given i position containing a fieldSelector's specified fields. This is to say, when all fields should not be loaded for search, a fieldSelector helps the developer to choose those that should be loaded in a Lucene document. The field selector is used by the IndexReader for search purpose, and not during the creation of a Lucene Document. More details about the FieldSelector is given in chapter4
- Directory(): The directory in the file system where the index is stored.
- A Term Frequency Vector or a list of term frequency vectors: Term frequency is the frequency of appearance of a term in a single Lucene document.
- TermDocs: enumerates all Lucene Documents containing the given term
- numDocs: is the number of Lucene Documents in the index as a numeric value
- numDeletedDocs : the number of deleted documents as a numeric value
- TermEnums : The list of Terms stored in the Index

Package org.apache.lucene.store

As has been said, indexing consists in the creation and storage of Lucene documents' fields extracted from files like Pdf, Word, Xml Files. While the creation of the index is provided by the IndexWriter class of Lucene.index package, its storage is ensured by the classes in the package Lucene.store

The abstract class *Directory* is the base of the storage mechanism of Lucene. In fact, the directory class is an abstraction of the common directory in an operating system. It is a list of files of different types and structure. According to its recording mode, a directory can be a FSDirectory (read File System Directory) which is saved in a file system on the hard drive, or it can be a RAMDirectory (read Random Access Memory Directory), which is used for an in-Memory index. A FSDirectory contains Files while RAMDirectory contains RAMFiles. To read and write in a directory, Lucene uses two abstractions for reading files respectively for writing files in a directory (FSDirectory or RAMDirectory). They are called IndexInput and IndexOutput.

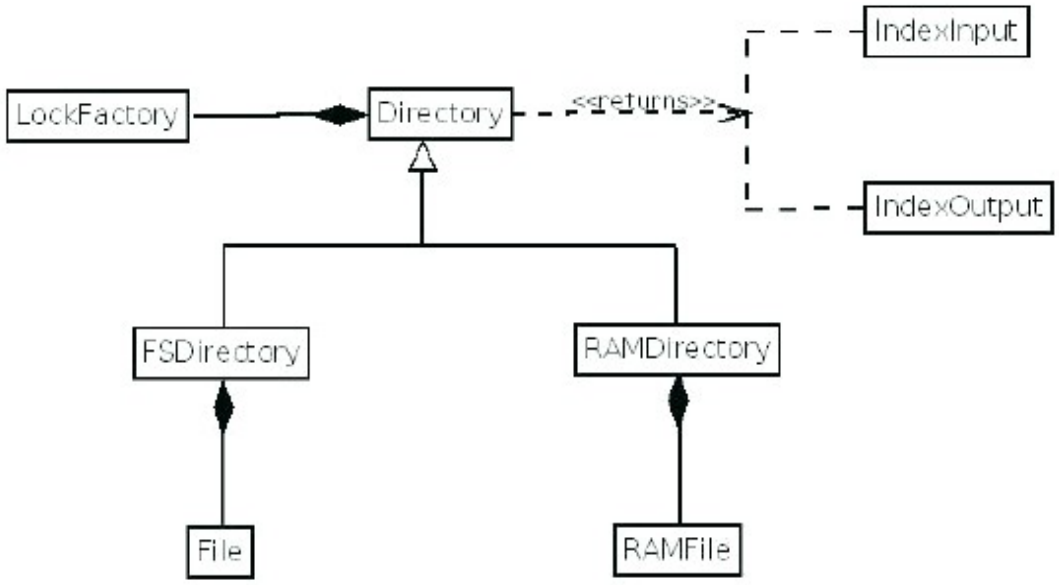


Figure 19: Lucene storage: conceptional structure

IndexInput reads Bytes, integer or numbers, Strings or Words from a File and returns it to an Index reading operation like IndexReader see Illustration 8 above. Specifically, when an index is to be read, an IndexReader request from a directory the list of terms stored in the index. This directory Object uses an IndexInput to read Strings from the file which contains the index . As a result, Strings are returned to the directory which returns them as response to the IndexReader. Thus the IndexReader gets the list of terms called TermEnums (illustration18).

Likewise, the IndexOutput writes bytes, numbers, strings into a file and returns it to any Index writing operation like the IndexWriter. The same way like the IndexReader, the IndexWriter can request from a directory the storage of a Lucene document into the file system. This time, the directory will request an IndexOutput's *WriteString*(*string*) method to write this document as a String in the file system or in the memory.

Additionally, a directory implements locking for its file. This is to prevent mutual exclusion. In

fact, both reading and writing processes can access the directory at the same time. This is why a directory object must set a lock on its file during a single processing. A lock can be constructed and destroyed after usage. All locks are provided by a Lock factory instance.

This discussion closes the first part of our study of the code architecture concerning indexing with Lucene.

The next two packages deal with searching a Lucene index. The important classes for searching are implemented in the packages QueryParser and Search.

Package org.apache.lucene.QueryParser

Let's assume an index has been created and now a search for different words and expressions in the index is to be done. The first thing to do is to analyze the requested search words entered by the user.

A user query string is parsed by a QueryParser. It transforms the query string into query objects using a specific grammar for the Lucene query language. During this transformation, the query terms are analyzed by a user specified analyzer, normally, as mentioned before, the analyzer that has been used during the construction of the index. The illustration below depict the process of query parsing.

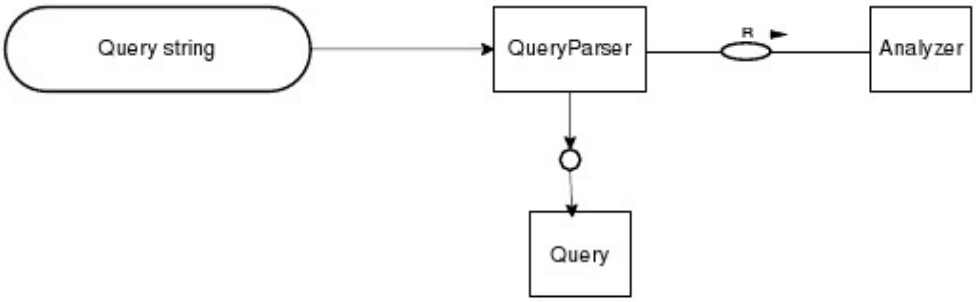


Figure 20: Lucene QueryParser

Lucene's QueryParser is generated by Javacc[19], the *Java compiler compiler*. A QueryParser uses an Analyzer which extracts terms from the user query text.

Lucene has a query syntax that each user query should follow. The syntax is given by a Javacc grammar from which the implementation of the QueryParser and its supporting classes is generated. At run time the QueryParser can take a query string from the user and recognize matches to the grammar. All matches are stored in form of a Query.

The query grammar is a list of specifications of what a user query should look like before they are consumed by the Lucene search engine. Especially the method parse() define in the Queryparser.jj file is the one responsible of parsing user queries into a Lucene query. A Lucene query is defined as a set of *clauses*. A *clause* is made up of one or more terms and can also contain another Lucene query.

Table 3 shows the representation of a query in BNF[20](*Backus-Naur Form*)

19[Treutwein,1979]

20[Javacc[tm] grammar files,2010]

```
Query  ::= ( Clause )*
Clause ::= ["+", "-"] [<TERM> ":"] ( <TERM> | "(" Query ")" )
```

Table 3: Lucene query grammar: BNF production

The second line in table 3 is the representation of a clause. A clause contains terms ,with plus and minus sign,or a query enclosed in parentheses which is again a set of clauses. A term can be a word written in human readable language like, English, German, French, Chinese... It can be a natural number,decimal number, it can also be a date. The kind of clauses in query defines the kind of a query. Those are discussed in a further chapter of this thesis.

In fact a Term is grammatically defined as shown in table 4 below:

```
1. <TERM:   <_TERM_START_CHAR> (<_TERM_CHAR>)*  >

2. <#_TERM_START_CHAR: ( ~[ " ", "\t", "\n", "\r", "\u3000", "+", "-", "!", "(", ")", ":", "^",
                         "[", "]", "\"", "{", "}", "~", "*", "?", "\\" ]
                       | <_ESCAPED_CHAR> ) >

3. <#_TERM_CHAR: ( <_TERM_START_CHAR> | <_ESCAPED_CHAR> | "-" | "+" ) >

4. <#_ESCAPED_CHAR: "\\" ~[] >
```

Table 4: specification of a term in a Lucene query

A term is essentially composed of one or more characters this is represented by (<_TERM_CHAR>)* . A term can start with any other character than those included in the square brackets at line 2

A _TERM_CHAR is any character including blank, which is not listed in the square brackets of line2.

Particularly, "*giessen + official-homepage*" is a valid user query it contains a clause which is composed of terms: *"*giessen*",*"+*","*official*", "-","*homepage*".* Searching for this query means to search in the index all documents containing the terms "*giessen*" and "*official*" but not "*homepage"*.

Another example is the query: *author: (grant + inger*)* where the clause is compose of term and a query. The term is author, the query is *"(grant +inger*)".* Lucene will therefore search for all field named *"author"* and which value contains the terms *grant a*nd term starting with *inger* and having one or more other term behind it. This query will match index entries like *author: grant ingersoll, author: grant inger .*

The meaning of different kinds of query expressions will be given in chapter 6.1 and 6.2, together with more details about the way a QueryParser interprets the query language.

Note that all parsing methods define in QuerParser.jj are implemented by QueryParser.java, which is responsible for interpreting the query grammar. More details about those methods are given in chapter 6.1.

Package org.apache.lucene.Search

To perform a search it is necessary to analyze and verify the validity of the user query. This work is done by the QueryParser together with the search application. The previous paragraph discussed about the architecture of a Lucene QueryParser and the structure of a Query, but more details about them are provided in chapter6 . As a result of the query parsing process , a Lucene Query is produced. Now this Query is to be used by the search package for retrieving its terms in the index.

First of all, a sketch of the search code architecture is given below for a better understanding of the internal work done during a search with Lucene.

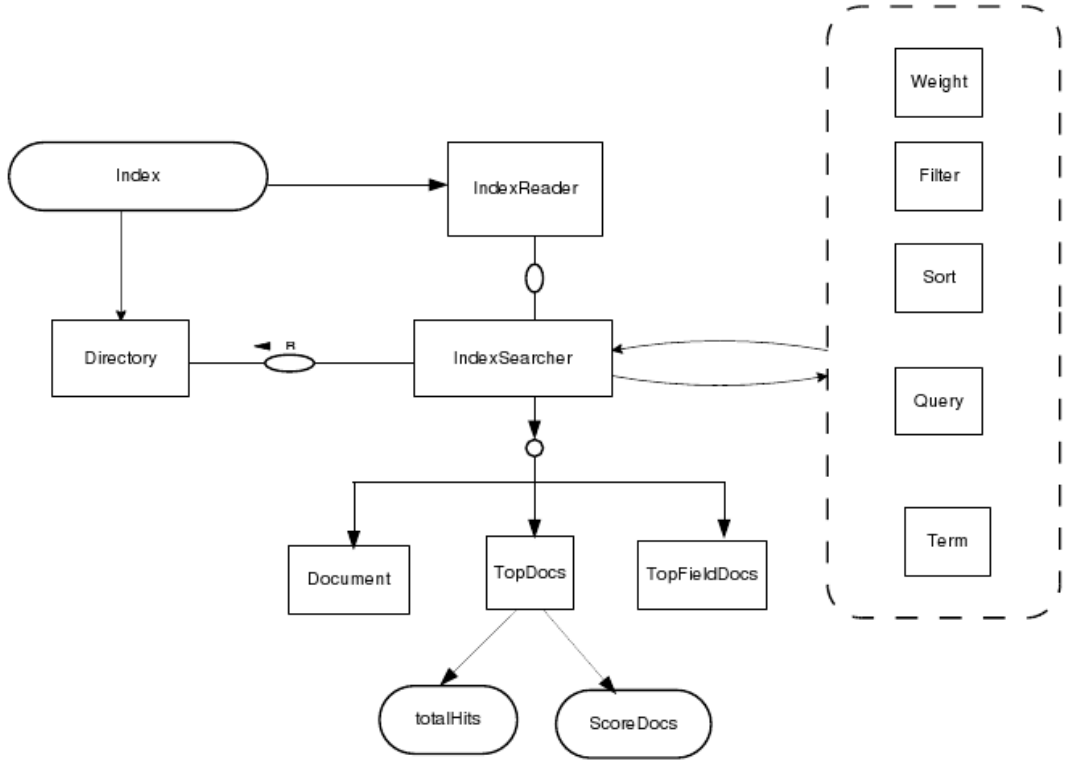


Figure 21: IndexSearch: architecture overview

The central component here is the IndexSearcher. It uses a directory or an IndexReader to access the index and collect information out of it. As seen previously, a Directory is the representation of an Index as a File in the file system, this means an IndexSearcher using an instance of a Directory works with the path containing the index. Likewise, the IndexReader can access the directory read-only with its open method as seen in the section concerning the index package.

Now that an IndexSearcher is created, it should implement all methods defined in the Lucene searcher[21] abstract class. The core search functionality is discussed in chapter 6

21[foundation, 2000-2010]

with attention to the process of retrieving Information out of the index using an IndexReader. After accessing the index, the IndexSearcher use one of the four implementations of the Searcher.search() method to search for the Lucene Query. The search possibilities are represented by the structure variant on the right of the above illustration. This structure contains Weight, Filter, Sort, Query and Term. They are all used to display results .

First of all, a weight returns the weight of a query and scores documents for a query. A weight can be created while creating a kind of Query, this is done with the *createWeight*() method of a Lucene query. Next, a Filter is used to select which documents is to be displayed during searching. Obviously a filter uses an *IndexReader* to read the document Ids set that should be allowed as search results. Sort is a class used for displaying a search result according to some sorting criteria. The results can be sorted by numbers or by strings. Finally, a Term can be taken out of the query and used to retrieve all documents containing it. Additionally, an integer can be used for the number of Lucene documents found in the index. A Query is also contain in this structure variant because it can be rewritten.

As result of the search process, various Information are returned to the user:

- the stored fields of a document
- the top documents with the given weight.
- top hits on a document.
- the number of documents containing a given term t.

This part of the thesis has given the concepts and theoreis behind the Lucene core regarding indexing, index storage and search.

The following part of this chapter gives a practical approach to indexing and search with Lucene. For the purpose of this thesis , a small indexing and search application have been developed. It is called **SeboL** meaning "**S**earch **e**ngine **b**ased **o**n **L**ucene". SeboL indexes Pdf, Xml and Txt Files and provides the possibilities to search the created index. The next part is about code examples in SeboL which implements indexing and search.

3.2 The Indexing process in example

How are each Lucene packages and components seen in 3.2 used for indexing of a given pdf file? This is the question to be answered in this section.

Let's take the "broder_websearch.pdf" file and examine how it is indexed. Let ***source*** be the name of the file system directory containing files to be index; yet it contains only the single pdf file "***broder_websearch.pdf***" let ***indexdir*** be the name of another file system directory that will contain the created index. Here is the implementation of the index class for indexing:

```
public class indexer {
   public static void main(String[] args) {
        File source = new File("/home/josiane/Documents/indextest/");
        File indexdir = new File("/home/josiane/Documents/index/");
        File[] filelist = source.listFiles();
        try {
                Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_30);
                IndexWriter w = new IndexWriter(FSDirectory.open(indexdir),analyzer,
                IndexWriter.MaxFieldLength.UNLIMITED);
                w.setUseCompoundFile(false);

                for (File f : filelist) {
                        System.out.println("filename: " + f.getName()+ "\n----------\n");
                        PdfHandler handler = new PdfHandler();
```

```
                    Document doc = new Document();
                    doc = handler.getDocument(f);
                    w.addDocument(doc, analyzer);
                }
                w.optimize();
                w.close();
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

The indexing process in the implementation above is divided into three main steps which are:

- Parsing the given pdf file with the implemented Pdf parser. The parser creates Lucene documents and passes it to the StandardAnalyzer for further processing
- Analyzing Lucene documents with the StandardAnalyzer.
- Creating an index. This includes the creation of an IndexWriter. This one is responsible for
    - opening the ***indexdir*** directory,
    - calling the StandardAnalyzer to process Lucene documents
    - storing Lucene documents into *indexdir* including all their parameters
    - updating the indexWriter
- Addition of Fields in a Lucene document instance
- Addition of Lucene documents to the index in "*indexdir*"using IndexWriter
- optimization and closing of the IndexWriter

First of all let's see how our pdf file "broder_websearch.pdf" is processed by our Pdf parser. Then we will inspect the dissection of the created Lucene documents into tokens, afterward the creation and storage of the index will be examined.

3.2.1 Example of creating a Lucene document

The purpose in this part is to extract textual contents from our pdf file. Let's examine what we have, and what we need. We have an 8 pages Pdf file. This file is available for free at the acm.org portal [8]. The original file name is a "A taxonomy of web search" the author is Andrei Broder. We renamed the document to "broder_websearch". In our example application, we use PdfBox for text extraction. The class PdfHandler created for the circumstance includes two Methods:

The pdfLuceneDocument(FileInputstream) method takes the Pdf files as an inputstream and extract the textual content to a String. We call the string "contents" and we store the extracted text as value of that field. The strategy used by PDFBox to extract text would be developed in chapter 5. The second method getDocumentInformation() collects meta data of the Pdf file and store it as PDDocumentInformation. The meta data in a pdf file usually contains information like the author name, the date of creation, the title and so on. In our case we store the date of creation in a field with the same name. The path and the filename are provided by the methods getAbsolutePath() and getName() of the Java class File.

All things considered the PdfParser class feed fields with string values. The field we use are

"contents" and fields for the pdf metadata.Additional fields can be usefull for search, like the filename and its path. Now let's see how those fields are stored on the disk.

As we have seen in short a Lucene Document is a set of Fields and a Field is a name-value pair. To illustrate its structure , we took a piece of text in the "broder_websearch.pdf" file and created a new pdf file we called (1).pdf. The illustration 16 below shows how a Lucene Document is internally represented. It owns a boost factor *boost* which is 1 in default case, when nothing is set. More details about the boost factor is given in chapter 4. It also contains a set of fields. The field is an ArrayList, holding the extracted text as string in an Array called FieldsData.

In the SeboL indexer we define four fields : *date of creation stored in fields[0], contents in fields[1], path in fields[2] and filename in fields[3]*.

To create those fields and add them to the Lucene Document ,one should use the Document.add() Method of the org.apache.lucene.document package.

Figure 22: A Lucene Document with four fields: [0]date of creation, [1]contents,[2] path, [3] filename

To create those fields we use the following Field signature:

doc.add(new Field(field name,field value,Field.Store.,Field.Index.*));*

where field name is the name of the field. In this case: date of creation, contents, path and filename field value is the value of the field hier: *cal.toString , pdfcontent . f.getAbsolutePath()* and *f.getName().*

Figure 23: The field internal structure.Field [1] is the field "contents"

A field has several parameters, which determines the manner with which it is stored in the index. A field can have a boost which is the same as set for the Lucene document. Here, boost = 1.0. fieldsData contains the value of the field. Illustartion18 display the content of that value for field content. Interesting parameters of a filed are *isIndexed, isStored, isTokenized* and *tokenStream*, they define how a field is to be stored or indexed or break . The meaning of those are discussed in chapter 4. Now that the Lucene document are created, let's see how they are analyzed.

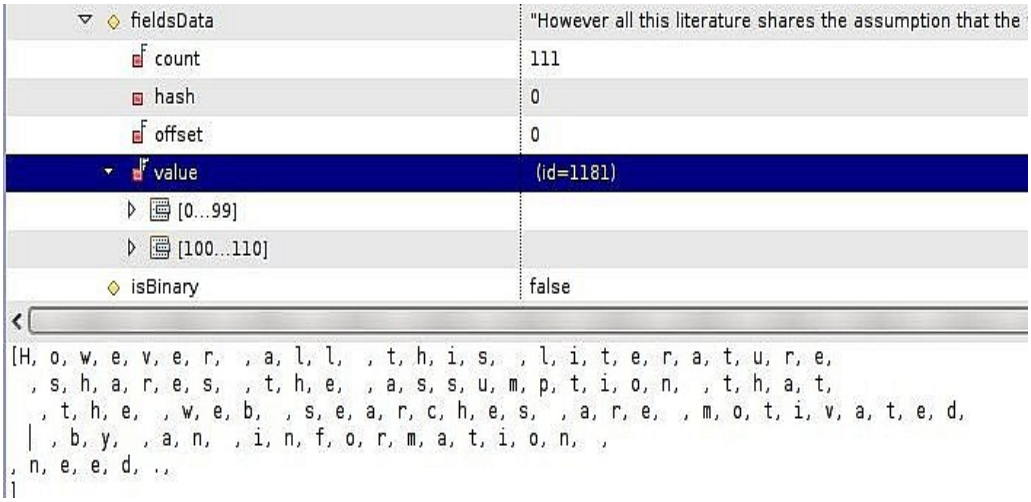


Figure 24: fieldsData: The value of a field named contents (<contents:value>)

3.2.2 Analyzing a Lucene Document with the StandardAnalyzer

We know from the previous part, that an analyzer builds a stream of tokens from the field's values of a Lucene document. As seen in the code architecture in section 3.1, an Analyzer must to be chosen. In our case, we are using the StandardAnalyzer, the most often used Lucene analyzer for the English language. The piece of code below shows how the StandardAnalyzer is used for indexing.

```
IndexWriter w = new IndexWriter(FSDirectory.open(index),
                                new StandardAnalyzer(Version.LUCENE_30),true,
                                IndexWriter.MaxFieldLength.UNLIMITED
                                );
```

Before delving into the analysis of the given pdf file, an overview of the concept behind the StandardAnalyzer is in order.

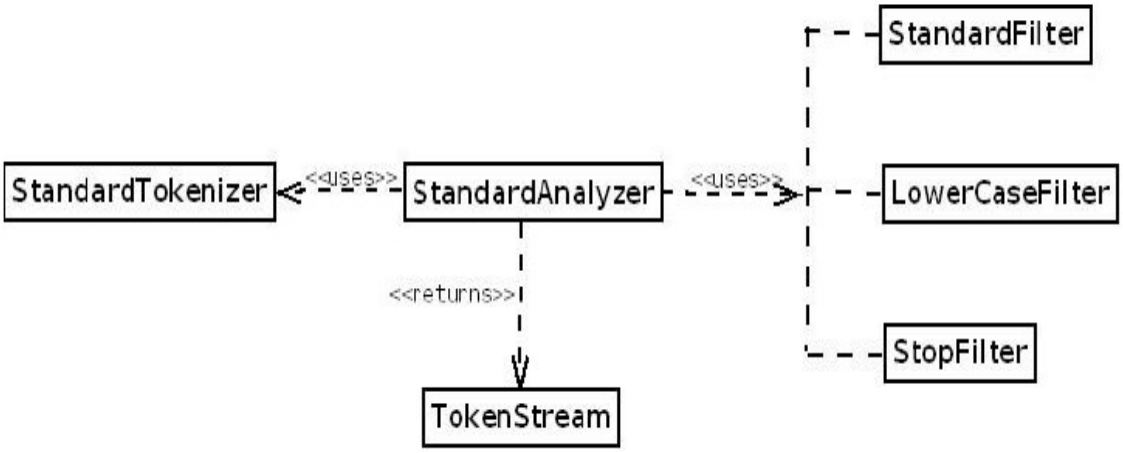


Figure 25: StandardAnalyzer conceptual structure

The processing method of the StandardAnalyzer comprises the following steps according to the architecture of Lucene analysis and the TokenStream decorator pattern described in 3.1:

- The StandardTokenizer breaks the field's values into tokens, where tokens are separated by white space. It removes punctuations characters like (“, ”,”. ”,”; “,”: “) a punctuation should be followed by a whitespace. The standardTokenizer also removes hyphens in token which has no numbers e.g. *web-specific needs* , it also consider an email adress and a host name as a token e.g. muster@mustermail.com and http://www.this.com
- The StandardFilter normalizes tokens by removing “'s” from the end of words or dots from acronyms . e.g.: mike's car and O.N.U.
- The LowercaseFilter change uppercased characters to lower case
- The StopFilter removes all English stop words from the token. Stop words is a list of words that are not significant for indexing and searching like ***but, be, with, such, then, for, no, will, not, are, and, their, if, this, on, into, a, or,...***

In the code snippet above, we can see that the StandardAnalyzer is used , according to the strategy pattern, by the IndexWriter in constructing the Lucene index (see illustration 15 in 3.1). The only parameter to be used is the Lucene Version, in our case, Version 3.0 .

With the help of the Eclipse debugging we can see that:The StandardAnalyser is made of a **StopSet** and **TokenStreams**.

Stopset is a two dimensional array of English stop words

TokenStreams is a stream of tokens extracted from the “broder_websearch.pdf” file . It has a hashMap structure, this is a data structure which map keys with their associated values . Values are of two different types: a **filteredTokenStream** and a **tokenStream** Each of them holds tokens. .Each token holds a group of attributes represented as **(term=,startOffset=20,endOffset=20,positionIncrement=1,type=word)**

The figure below is a screen shot of the TokenStreams values build during indexing of our pdf file. This was taken, while debugging the index class. It appears that a TokenStream comprises different elements which are:

- **“AttributeImpls”** This is a <key,value> pair, where the key is an integer representing the position of an attribute it matches a value which is the attribute value, in our case the value contains : **TypeAttributeImpl=type=word**, **offsetAttributeImpl**=startOffset=20,endOffset=20, **PositionIncrementAttributeImpl**=1,**TypeAttributeImpl**=type=word.

We can observe, that these attributes are the same as the token attribute described in 3.1

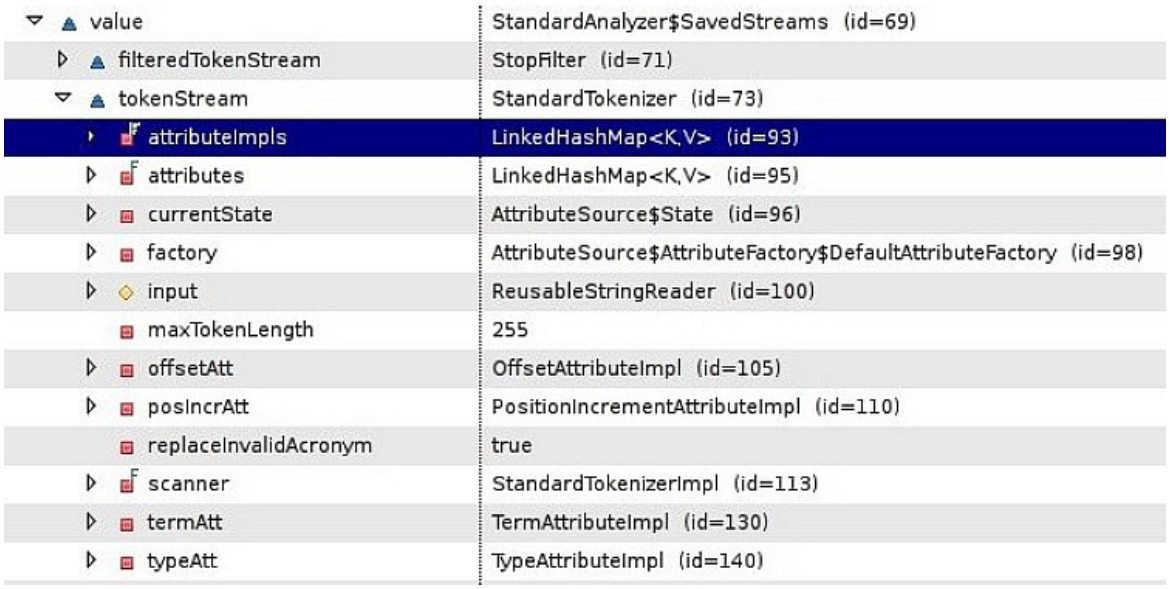


Figure 26: TokenStream produced by StandardTokenizer

- **“attributes”** are token attributes as defined in the index class. In our implementation, there was no particular choice of token attributes, So the Default attributes of a token will be considered, this means they are the same as defined in attributeImpls. However Lucene supplies methods in it's core API to set up each token attribute. For example: with the “setType(String type) method”, a token lexical type can be modified from “word” to another type, like “keyword”; calling setType(“keyword”). For more

details about the token attribute, read the previous paragraph(3.1).

- **“currentState”** is actually the list of 42 terms from type word, extracted from “broder_websearch.pdf” the first terms are [w,e,b,s,e,a,r,c,h,.,p,d,f]
- **“factory”** is an attributes factory. It produces attributes for a token, by using the class name of the attributes and adding *Impl* to it. In our case factory is the value of the AttributeSource.AttributeFactory.DEFAULT_ATRIBUTE_FACTORY.It is only use to creates attributes.This is done when the factory method *createAttributeInstance*(Class<? extends Attribute> attClass) is called. These are attributes made by the factory in a general case:

Class name	**Manufactured attributes**
FlagsAttribute	FlagsAttributeImpl
OffsetAttribute	OffsetAttributeImpl
TypeAttribute	TypeAttributeImpl
TermAttribute	TermAttributeImpl
PositionIncrementAttribute	PositionIncrementAttributeImpl
PayloadAttribute	PayloadAttributeImpl

- To create FlagsAttributeImpl, the factory calls createAttributeInstance(FlagsAttribute).
- **“input”-** This is where Fields values of “Path” and “fieldname” are stored . Looking inside it, we can notice that,input is an array of length 2 which values are a reusable string which is the filename with path, and a Field which value is

[/, h, o, m, e, /, j, o, s, i, a, n, e, /, D, o, c, u, m, e, n, t, s, /, i, n, d, e, x, t, e, s, t, /, b, r, o, d, e, r, _, w, e, b, s, e, a, r, c, h, ., p, d, f]

and name = “name” this is the given field name in the main method.

- **“scanner”** The content field is depicted in it;it is made of zz-and YY-values where zzBuffer is an array of length 16384 in this example, with terms extracted from the pdf content. The Figure below is an output of the scanner value.

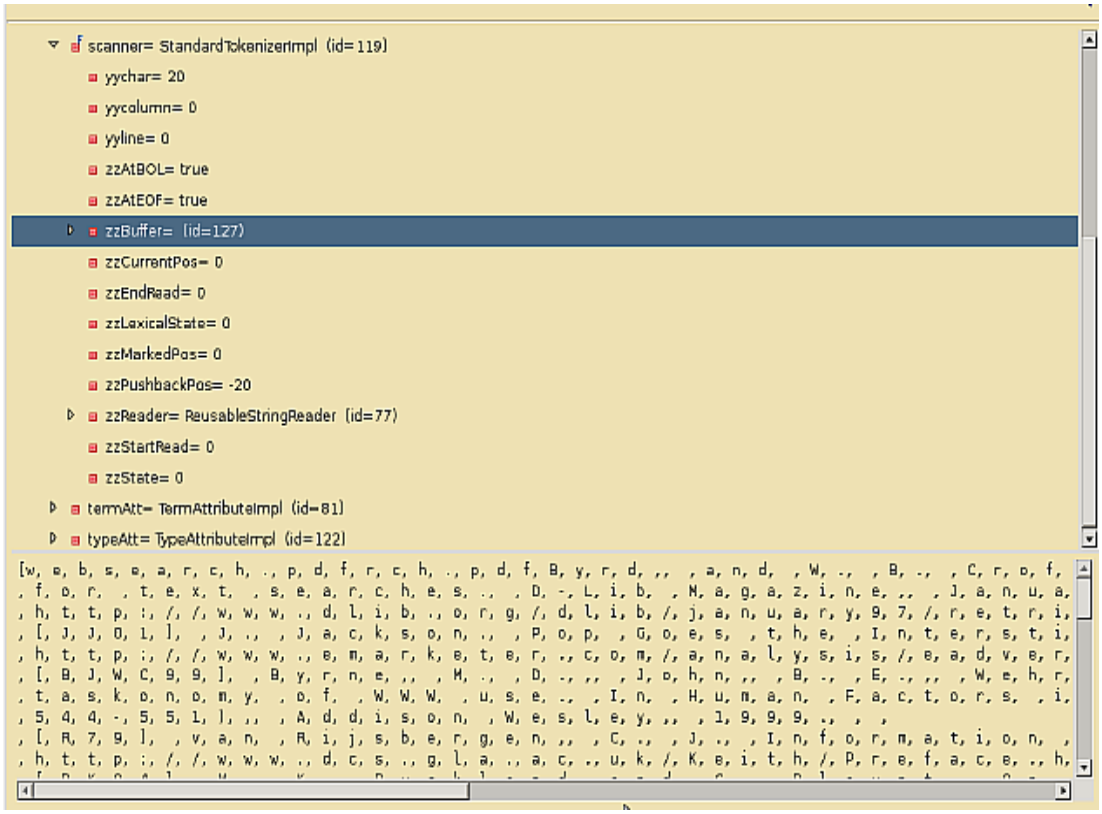


Figure 27: StandardTokenizer applied to "broder_websearch.pdf"

The scanner value holds in ZZBuffer tokens broken out of the Lucene document fields of our pdf document. Notice that the tokens contains stop words and are all lowercased. Let's highlight the stop words and reconstitute the tokens processing by the StandardFilter and stopFilter.

Here is a piece of tokens stream produced. Those, which are highlighted are Stop words.

[w, e, b, s, e, a, r, c, h, ., p, d, f, r, c, h, ., p, d, f, B, y, r, d, ,, **a, n, d**, , W, ., , B, ., , C, r, o, f, t, ., ,
C, l, a, r, i, f, y, i, n, g, , s, e, a, r, c, h, :, , **A**, , u, s, e, r, -, i, n, t, e, r, f, a, c, e, , f, r, a, m, e, w, o, r, k, ,
, **f, o, r**, , t, e, x, t, , s, e, a, r, c, h, e, s, ., , D, -, L, i, b, , M, a, g, a, z, i, n, e, ,, , J, a, n, u, a, r, y, , 1, 9, 9, 7, ., , A, v, a, i, l, a, b, l, e, , **a, t**, ,
, h, t, t, p, :, /, /, w, w, w, ., d, l, i, b, ., o, r, g, /, d, l, i, b, /, j, a, n, u, a, r, y, 9, 7, /, r, e, t, r, i, e, v, a, l, /, 0, 1, s, h, n, e, i, d, e, r, m, a, n, ., h, t, m, l,), .,....]

Table 5: StandardTokenizer and LowerCaseFilter output

On the whole, the TokenStream which is a StandardTokenizer reads the Lucene documents fields and breaks them into tokens. Table 6 shows a part of the tokens' list. Tokens are characterize by various attributes, which can be modified. Those Tokens are reused as input of the StandardFilter. The StandardFilter normalizes all tokens and passed them to the LowerCaseFilter. Finally the list of lowercased tokens are passed to the StopFilter. It removes all the 33 English stop words (see table5) contained in the stream of tokens, therefore this list of tokens is a **filteredTokenStream.**

While a **filteredTokenStream** is the result of the StopFilter, removing stop words out of a set

of tokens, a **tokenStream** is the result of the fragmentation of fields values into tokens .

3.2.3 Writing Lucene Fields in an Index

Creating an index involves text extraction, analyzing and index writing. This part is about the index writing and classes taking part at it. The *addDocument*() method of IndexWriter is the central class for building an index. By calling that method, analysis and index writing are activated. First comes the analysis, then the indexWriter. Internally a couple of classes of this packages step in the writing process. With help of our example class indexer, let's see what is happening . From the methods in the index package, our indexer needs a few of them to achieve index writing. Those are the IndexFileDeleter and the DocumentsWriter .

The IndexFileDeleter

According to its API, the IndexFileDeleter count the number of Indexcommits that reference each file in the directory. An Indexcommit is a committed segment Information, this one corresponds to a segment in the directory. When all commits referencing a file have been deleted, the number of reference to this file is 0. In our example, the number of references is *refCounts=11,* and on the disk we have 11 files corresponding to each Indexcommits. Another aspect is that it keeps the last committed Segments, all previously committed segments are deleted.

The DocumentsWriter and the DocConsumer

They are responsible for processing Fields after the analyzer. The DocumentWriter takes Lucene Documents and write them into a single segment file. The indexing algorithm describe in chapter 5 explains how documents are written into segment. Each document is passed to a *DocConsumer*, which launches the indexing sequence. Different consumer are chosen during the sequence:

- When fields are set as stored(isStore) the documentsWriter chooses a *StoredFieldsWriter*, which writes the fields to the index directory on the disk as .fdx and .fdt files.The meaning of these Index file format and much more is given in chapter 5.
- When fields are first buffered in the memory, then flushed into the disk by creating a new segment, the DocumentsWriter uses a FreqProxTermsWriter, or a NormsWriter. Those fields are stored as .frq or .nrm files.Other Consumers will be describe in chapter5

Coming back to our example, the consumer is a *DocFieldProcessor*. This one collects all fields under the same name and process them with a *DocFieldConsumer* which is a type of *DocConsumer*. Each time the *DocumentsWriter* decides to create a new segment, the *DocFieldProcessor* is called to wash out fields from the memory(RAM) into the disk. It releases the RAM when it is full. It closes the document storage at the end of all operations. But we can't talk about writing Fields in the index without mentioning the storage of the indexing. Storage occurs after writing and while updating or deleting a field. The Lucene store package implements the storage activities in Lucene.

3.2.4 Storing an index in a directory

In essence, the *org.apache.lucene.store* package provides the IndexWriter with methods for the storage of the index in the memory or in the file system. Also the segment Information and the *writelock* are part of it. After calling I*ndexWriter.addDocument(doc)*, the indexWriter set a *writelock* on the index directory in the file system. This prevents another *IndexWriter* to access the same directory at the same time. In the Indexer example, the writeLock is a *NativeFSLock* , this is provided by a *NativeFSLockFactory* von java.nio. The lock directory is

set to the write.lock file on the file system because it is used with a FSDirectory in /Documents/index . Apart from locking the Index directory this package supplies the IndexWriter with Input streams, reading or writing into the index. The figure below shows the index files as created by indexer.java:

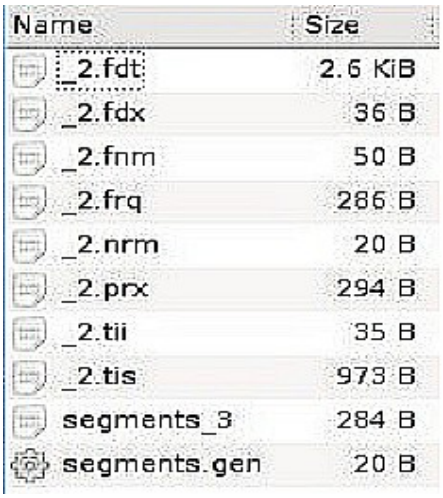


Figure 28: example of Index Files created out of (1).pdf and (2).pdf files

After creating the index, the next step is to search into. Using the SeboL searcher.java class, we would describe how search occurs for a given user query. That is the purpose of the next part of this chapter.

3.3 The search process in example

For the search purpose, we prepared two pdf files called (1).pdf and (2).pdf they contain two piece of text taken out of "broder_websearch.pdf". These are their substances:

(1).pdf: "However all this literature shares the assumption that the web searches are motivated by an information need. "

(2).pdf: "Search engines uses “spider” which search (or spider) the web for information."

With the help of these we will see how the queryparser interpret some queries send to searcher.java, and how their results are retrieved.

This is the search implementation in searcher.java in *SeboL*, this code snippet will help understanding the search process. We first talk about parsing a user query then retrieving the matching documents containing that query.

```
public class searcher {
        public static void main(String[] args) {
                System.out.println("this is search");
                Analyzer a = new StandardAnalyzer(Version.LUCENE_30);
                QueryParser p = new QueryParser(Version.LUCENE_30,"contents",a);
                try {
```

```
                    //TermQuery
                    Query q = p.parse("web");
                    //how this query looks like
                    System.out.println("query= " +q.toString());
                    File index = new File("/home/josiane/Documents/index");
                    IndexReader r = IndexReader.open(FSDirectory.open(index),true);
                    IndexSearcher searcher = new IndexSearcher(r);
                    TopDocs topdoc = null;
                    topdoc = searcher.search(q, 5);
                    System.out.println("found: "+topdoc.scoreDocs.length );
                    for(ScoreDoc scoredoc:topdoc.scoreDocs){
                            Document hits = searcher.doc(scoredoc.doc);
                            System.out.println("["+(scoredoc.doc+1)+"] " +
                            hits.get("filename")+"("+scoredoc.score+")");
                            System.out.println(hits.get("path")+"\n");

                    }
            } catch (Exception e) {   e.printStackTrace();}
        }
}
```

3.3.4 Parsing a user query

Talking about the QueryParser in our example is the same as examining how queries are parsed using the Lucene query language. To create a Query parser we need to create an instance of QueryParser with a couple of parameters like the Lucene version to be used, the field in which search will occur and the analyzer. More details about the query parser and Language are given in chapter 6. Let's see how query parsing work in our case. Beginning with the most simplest one the TermQuery.

TermQuery

searching for **"web"**:

The QueryParser translates this request as **contents:web** where contents is the field name, and web the field value. So the search will look for the entire term in the index.

A Term, as defined in Lucene core, is a word from text. It comprise the field's name of the field containing the word, and the word itself. In our case the term is made up of "*contents*" as field name and "*web*" as text . Let us search documents (1).pdf and (2).pdf for the Term “web”. A TermQuery as well as each other query type we will study contains a Term defined as previously, and a boost factor. The boost factor is a number from type float, set by the user or the application. Documents matching the term will have their score multiply by the boost.. When the boost factor is not given, it is set to be 1.0. Here goes the illustration of the TermQuery q=contents:web.

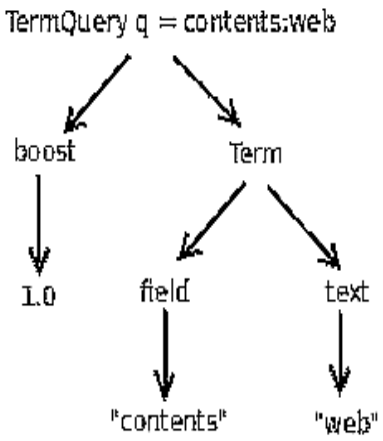


Figure 29: structure of a TermQuery

BooleanQuery

searching for **"web AND search"**:

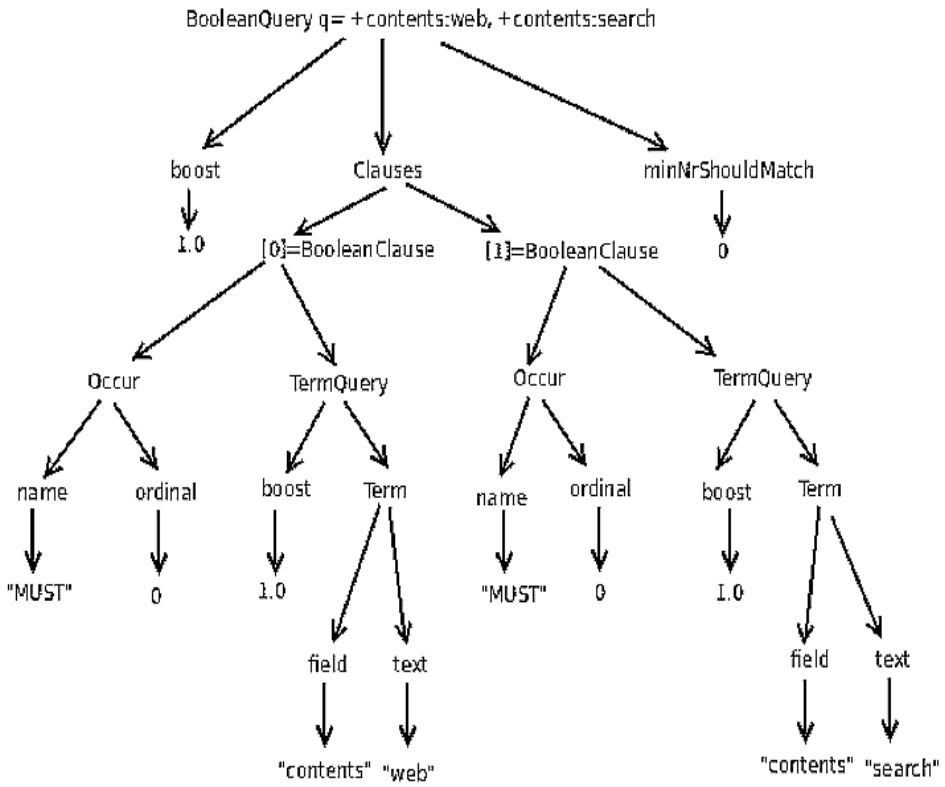


Figure 30: structure of a BooleanQuery q= +contents:web +contents:search

this is a disjunction of two Term queries. The result is an AND between query web and query search. Therefore it is define by the QueryParser as **+contents:web +contents:search .**This means :'*search all documents where the field is contents and the value contains both words web and search*'. The result is (2).pdf. This is how the Boolean query is parsed:

This BooleanQuery is a combination of two TermQueries: contents:web and contents:search. Each termQuery is include in a BooleanClause. This last has a value called Occur, which uses an operator to specify how the TermQuery should occur in the document. The operator in both clauses is "MUST" this means the clause must appear in the matching documents. In fact, the AND operator is translated as MUST by the query parser our query becomes MUST contents:web MUST contents:search.

If the Boolean operator is changed to OR it is interpreted as "SHOULD" and the resulting query ***contents:web contents:search*** would be parsed as:

SHOULD contents:web SHOULD contents:search.

For a NOT combination we have : SHOULD web MUST_NOT search corresponding to the query ***contents:web -contents:search*** is

SHOULD contents:web MUST_NOT contents:search

The *minNrShouldMatch* is the minimum of optional clauses that must match the BoleanQuery. 0 is the default value. When a number is given, let's say 3, then 3 clauses are required for search.

WildcardQuery

seaching for "s*e?":

A wildcard character is a character used to replace one or more characters. In our case, we use the wildcars * and ?.

***** means more than one characters and **?** Means exactly one character. So,the result expected here is the list of Files where there is a word starting with s followed by one or more characters, having an e inside and a single character at its end. When we examine our pdf files (1) and (2) we note that the following words match our search:

s**pid**e**r*,s**har**e**s* and ***s**earch**e**s.* Here is the structure of this WildcardQuery

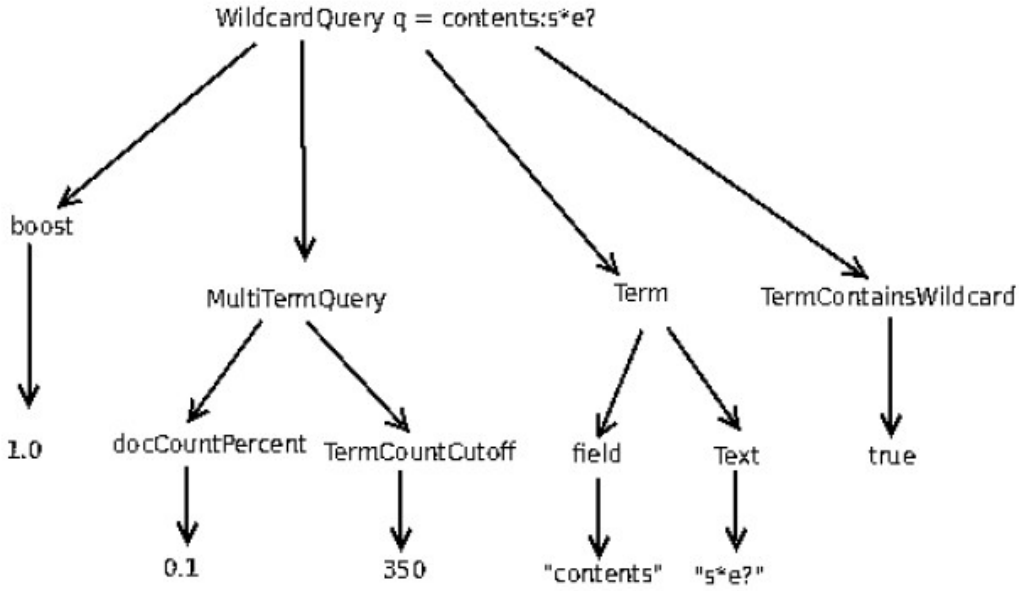


*Figure 31: Structure of a WildcardQuery q= contents:s*e?*

The *Wildcardquery* is a *MultitermQuery*. This is a query type that matches a collection of terms. In this case documents containing terms ***s**pid**e**r*,***s**har**e**s* and ***s**earch**e**s.*will build the results set of this search. A MultitermQuery has two constants which bounds documents and query in a certain way:

The *docCountPercent* is the number of documents to be visited in the postings lists (posting list are explain in chapter5) in percentage of the maximum number of documents for the index(maxDoc()). This percentage is between -0.1 and 100.0. If it exceeds 100.0, then a constant value is used to create a private filter. The constant is called CONSTANT_SCORE_FILTER_REWRITE. This filter seeks each query term in sequence, and marks all matching documents containing that term. Those documents are allocated a constant score equal to the query's boost. This avoids a *ToomanyClauseException*.[22] which occur when the query is too long.

The *TermCountCutoff* is the maximum number of terms in a query. In our case it is 350 terms. If this number of terms is exceed, then CONSTANT_SCORE_FILTER_REWRITE is called as previously.

FuzzQuery

searching for **"literature~"**:

This will search for queries similar to literature, regarding the pronounciation of the word.The query parser parses it as **contents:literature~0.5**. The tild is part of the fuzzy syntax , and 0.5 is the default value for the minimum Similarity of that term.

In fact the *fuzzyQuery* use the **Levenshtein distance**[23] for computing the distance between

22 MultiTermQuery:http://lucene.apache.org/java/3_0_1/api/core/index.html?org/apache/lucene/search/TopDocs.html

23 Michael Gilleland. Levenshtein distance, in three Flavors. URL:

"literature" and each other string. The 0.5 means terms matching literature should have a similarity between 0.5 and 1. The figure below shows the tree structure for that query

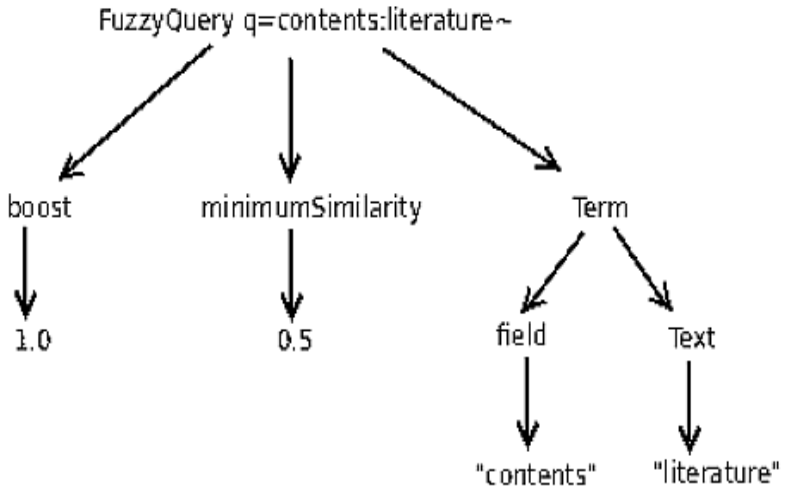


Figure 32: Structure of a FuzzyQuery q= contents:literature~

The FuzzyQuery is also a MultitermQuery. It holds a termQuery and measure the similarity of that term to every other terms in the postings list. The default similarity here the minimumSimilarity is 0.5.

More in-depth study of query language and those query types is made in chapter 6. For now let us see how search occurs in our application.

3.3.5 Experience query search

Assume that we are searching for the TermQuery "**web**" in our index. The first thing to do is to open the index, this is done by creating an IndexReader reading the file system directory containing it:

IndexReader r = IndexReader.open(FSDirectory.open(index),true);

Then an IndexSearcher is initiated using the created IndexReader:

IndexSearcher searcher = new IndexSearcher(r);

Once an IndexSearcher is created, a query is given to him for computing top scores. The top matches are collected in an object called ***topDocs***, it uses a ***scoreDoc*** array to store hits and can display it's length, which is total number of documents matching the query:

TopDocs topdoc = null;

topdoc = searcher.search(q, 5);

System.out.println("found: "+topdoc.scoreDocs.length);

Calling *searcher.search(q, 5)* collects the top 5 results for our search. For the given

http://www.merriampark.com/ld.htm

TermQuery, the number of Lucene documents found is 2. The word web is in both pdf files.The scoreDocs Api provide Information on any matching Lucene document like its DocId(doc) and its score(score).The code snippet below shows how to browse those Information:

```
for(ScoreDoc scoredoc:topdoc.scoreDocs){

        Document hits = searcher.doc(scoredoc.doc);
        System.out.println("["+(scoredoc.doc+1)+"] " +hits.get("filename")+"("+scoredoc.score+")");
        System.out.println(hits.get("path")+"\n");

                        }
```

Knowing the document's id doc, one can access to its fields and extract its value by using the *get(“fieldname”)* method of Lucene document.

hits.get("filename") display the filename of the matching document; scoredoc.doc returns the Lucene document Id, as store in the posting lists in the index. Each Id is a reference to the stored fields of its corresponding document . Calling hits.get("filename") will look for DocId =0 respectively 1 in the posting list, then retrieve all terms where field name = "filename" and display the value of that field. The Illustration of the TermQuery Structure show the composition of a term as <fieldname,text >pair. The same thing is done for the field name "path".these are the results display by searcher.java:

[1] (1).pdf(0.18579213)
/home/josiane/Documents/indextest/(1).pdf

[2] (2).pdf(0.18579213)
/home/josiane/Documents/indextest/(2).pdf

chapter 4 Lucene Document

One thing to know about indexing in Lucene is that each File has to be prepared before it is indexed. The reason why those files have to be prepared is simply because they are not Lucene suitable, they may contain useless words for search. For example an html file contains tags like <body>,<p>, etc... these are mostly not retrieved. Only the textual content bound by tags are useful for search. Since Lucene doesn't possess functionalities to extract the textual content of a file, this is carried out by an external application called *document parser*. Nevertheless, Lucene offers a document handler for the processing of .txt and .html files into Lucene documents. For other file types like *Word, PDF, Xml, Excel* and so forth, it is the work of the user application to create Lucene Documents. Commonly, applications use a document parser to extract content a create Lucene documents .How a parser process the original document into Lucene Documents is explained in the third part of this chapter. Before dealing with the document parser, we will see the structure and the construction of a Lucene Document and we will study each of its components.

4.1 Overview of the Lucene document structure

As elaborated in chapter 3, Lucene can index and search different data type irrespective of its format, language or source. Just the textual content needs to be extracted from those Data and transferred to Lucene. This Textual content must be an object of a defined structure called a **Lucene Document** or a **Lucene Document object**.

A Lucene document is a set of Fields. A field comprises a name and one or more values. The name is usually a word(String type) describing the field like **content** ,***path***,***date of creation*** are examples of field's name. The value is the text of that field. An example of a Field value is "***web***" for the field content, "***/document/indextes*t**" for the field path or "***2010-09-22T00:00:00.000Z***" for the field date of creation. A Lucene Document is used in three cases:

- As a logical representation of the original documents(t*xt,pdf,html*,...) provided by the document handler or the document parser

- As part of the index which hold stored (Lucene)fields. The following components of the Lucene Document are usually stored in the index:

  - Terms of each field's value are stored in a so called terms dictionary. The building of a terms dictionary is detail in chapter 5.

  - Documents' identification numbers(DocIds) are also stored in the index. These are numbers that are automatically increment when a new document is added, A DocId is a number used as a pointer on the document it represent. It is also use during search to retrieve documents. Each term has a list of DocId, from documents where it occurs.

- As representation of the result of a query. Field's values matching the query are gather and displayed by the search application. This is done with the help of each DocId. A document's field can only be display if it is set as *stored*. A document can also act as a filter through the so called *FieldSelector*. This one determines which fields of the set of stored fields are loaded in the resulting document.

Here goes an illustration of the Lucene document structure:

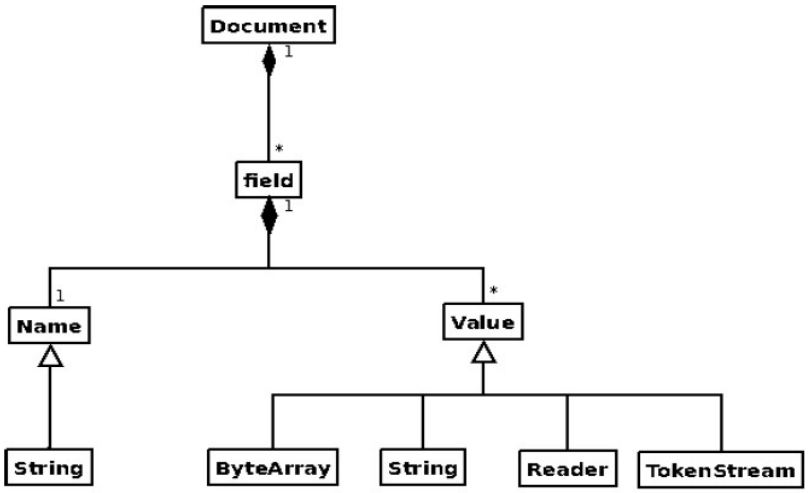


Figure 33: Lucene Document: Structure overview

A field has a single name which is of type **String** but it has several values such as the field "*path*" which has two values stored as String :" */home/josiane/Documents/indextest/(1).pdf*"and "*/home/josiane/Documents/indextest/(2).pdf*". The type of field's value can be of different kind: *ByteArray,Reader* or *TokenStream*.

A ***ByteArray*** value is an array of bytes . A field with ByteArray values is generally compressed and doesn't need further processing.

A **Reader** value is a character stream reader, reading characters out of a file, a standard input or a data pool.

A **TokenStream** value is a stream of terms also called tokens created by an analyzer, this kind of field is usually indexed and tokenized and its terms are stored in a termvector. So far we have given an overview of the conceptual structure of a Lucene document. The next paragraph deals with each previously mentioned components.

4.2 Components and data structure

4.2.4 Components of a Lucene Document

From the definition of a Lucene Document we know that its only components are its fields. So, talking about the component of a Document is the same as talking about its fields. The fields of Lucene document have a lot of properties that specify, how the Document or Field will be processed during indexing. A Field is mostly set as indexed. Some can be stored or have terms vectors and others can have binary value(value of type ByteArray) or reader values. A Field with binary values is optionally compressed. Other components of Lucene Document are the **FieldSelector**, **compression tools** and **boost factor**. These are also options to be used with FIelds. First of all let's go deep into the field's and tokens' architecture, then we will talk about other settings like the boost factor, the field selector and

the compression tool.

4.2.4.a) Fields

A Lucene field can be a result of analyzing the original documents, for example the text content of a pdf file or a html file, or it can be the representation of the search results computed for a given query. Let's start with the first aspect of the Lucene Field:

Fields as representation of the original documents.

After parsing an original file, the Lucene documents are created and passed through Lucene for indexing. Lucene first analysis the document according to the specifications the user allocates to its fields. The following is the list of field' specifications, that can be set up during the construction of the index:

Field specifications		**Field's value type**			
		ByteArray	String	Reader	Token Stream
STORE	NO			X	X
	YES	X	X		
INDEX	NO				
	ANALYZED(= TOKENIZED)		X	X	X
	NOT_ANALYZED				
	ANALYZED_NO_NORMS				
	NOT_ANALYZED_NO_NORMS				
TERMVECTOR	NO		X		
	YES			X	X
	WITH_OFFSETS				
	WITH_POSITIONS				
	WITH_POSITIONS_OFFSETS				

Table 6: Field specifications

STORE

A stored field is a field whose values are kept inside the index; either in a directory in the filesystem (*FSDirectory*) oder in the memory. Field with ByteArray values and String values are stored in the index. This is the case of the fields *path*, *date of creation* and *filename* as we can see in the code snippet below. Their values are short text which can be entirely registered into the index as a String.

```
doc.add(
```

```
new Field("path",f.getAbsolutePath(),Field.Store.YES,Field.Index.ANALYZED));
doc.add(new Field("filename",f.getName(),Field.Store.YES,Field.Index.ANALYZED));
```

On the contrary, the values of the field "*content"* should not be directly stored in the index because of disk space they require, therefore the field option to assign to that field is **Field.Store.NO**. Commonly user applications get around this problem in the following manner: the value of the field *content* is used to generate the posting list, but is not as a whole stored in the index, just terms of that field's value are taken out for the postings list generation. This option is only useful if the index contains information about retrieving the original document, like in the field "path" that contains the path to the original document. In this case th field Path has the option **Field.Store.YES.** This is the same for the file name, see the code snippet above.. Fields with value of type *Reader* and *TokenStream* are containers for several character streams, they should not be stored because of their big capacity.

Let's go back to the field with ByteArray value, and see how it is stored. We said that this field is generally stored in the original form like the string value. Moreover, the Byte-array value of a field can be processed with a utility tool called *compressionTools*. Lucene provides this tool to reduce the disk space used by binary fields when storing them to the index. The field's values can be compressed and decompressed when needed, for these functionalities, Lucene uses the compression levels supply by the *java.util.zip.Deflater* class[24]. The compression level must be specified, otherwise the default level is set to the constant value BEST_COMPRESSION. Other levels are available in the *Deflater* class API.

In fact, specifying a field as stored or not, depends on the length of the text in field's value. It is recommended to store value with short text for the purpose of quick indexing. Storing a field with the original content is not suitable for all fields; they are fields which need to be analyzed before keeping them into the index, other fields like *contents* do not need to be store completely but terms can be taken out of them and stored in a terms vector. This is what we are going to see in the following.

INDEX

An indexed field is processed by an analyzer, before its stored inside the index. A Field can be index in different ways, to set a field as indexed or not, the parameter **Field.index.parameter name** is added to the Field , the parameter name determines how the field is processed, these are the values it may have:

NO: The field will not be indexed. This is the case of a Field with *ByteArray* value . It doesn't need to be tokenized or filtered. In fact, *tokenization* and T*okenfilter* are applicable for String values.

ANALYZED: The field will be analyzed with the provided analyzer. Analyzing includes tokenization and or token filtering. Apart from the ByteArray value Fields, this parameter can be set for all Field with textual values . The Field's value has to be broken into tokens, and then it has to be filtered. Detail of the Analysis are given in the next chapter. The analyzed fields are stored in the index, when their value's text are short. Norms are also stored.

A norm is a file containing the boost factors and the field lengths. These are regularly modified and stored in the index as *.nrm* file. More about the index file format and

24 Java™ Platform Standard Edition 6.Class Deflater .URL: Http://download-llnw.oracle.com/javase/6/docs/api/java/util/zip/Deflater.html

their data structure is given in chapter 5.

NOT_ANALYZED: the field will not be analyze. This is applicable for field, that can be searched as a whole term. For instance, the file extension can be stored without been analyzed.

ANALYZED_NO_NORMS: The field will be Analyzed, but the norms will not be allowed. It is better to disable norms during indexing, to prevent unnecessary use of the memory. Each field's norm for each document cost 1 byte. This mean for our 4 fields, if we have 10 documents in our index, we will need something like 80 bytes free in the memory to store the norms of each field for each document. For a larger index much more memory would be lost. In addition, a norm file is created for each modified segment's norm. To disable or enable norm on the date of creation, we used the *setOmitNorms*(boolean) method like this:

```
Field fcal = null;

...

fcal = new Field("date of
creation",cal.toString(),Field.Store.YES,Field.Index.ANALYZED);

fcal.setOmitNorms(true);
```

NOT_ANALYZED_NO_NORMS : The field will not be analyzed and the norms will not be stored.

Now let's see how term vectors can be extracted from the original documents.

TERMVECTOR

A termvector is a list of Lucene document's terms and their number of appearance in that document.The termvector is commonly called postings list. The construction of this list is described in the next chapter. When a field has Terms vectors, an additional option is add during the creation of this field namely, **Fied.Termvector.Parametername**, where **Parametername** is one of the fourth options we listed in the table above for a TERMVECTOR specification. This parameter instructs the indexing application about the way term vectors should be conserved. The vector can carry additional information depending on the parameter's name. These information are relatives to the token of the term, like its offset and its position

NO : this parameter means the field should not have term vectors stored: no document's number and no term occurrence. Just the term is stored when Field.Store.YES is set.

YES : the indexer will store the term with document's number where the term occurs, and with the number of term occurrences in the document.

WITH_OFFSETS: The term vector will be stored with the token offset information. The offset is a part of a token.

WITH_POSITIONS : The term vector is stored with the position information of the token. This is the position of the token relative to the previous token.

WITH_POSITION_OFFSETS: store both position and offset information with the term vector.

In short,we have just seen the Lucene field as part of the index, let's see the second

aspect of fields.

Fields as representation of the search results:

The result of a search is the list of values of the fields found for the given query. Only Fields that have been stored in the index can be display. A Field set as STORE.NO will never appear in the search result but it can be analyzed and its terms can be stored using the option *Field.Termvector.YES*. As an illustration, we use our SeboL *searcher.java* program to retrieve the word **web** in two pdf files:(1).pdf and (2).pdf.
The result display looks like this:

```
[1] (1).pdf
/home/josiane/Documents/indextest/(1).pdf

[2] (2).pdf
/home/josiane/Documents/indextest/(2).pdf
```

The first field displayed is the file name respectively *(1).pdf* and *(2).pdf*, then the path to this file */home/josiane/Documents/indextest/(1).pdf* respectively */home/josiane/Documents/indextest/(2).pdf.*
In much cases the user doesn't need to see the field's name *path* and *filename* their values are more important issues.
Another aspect to outline, is that the options that was established for a field during analyzing and indexing, affects the search. In the book of 'Lucene in action' [2] the issue of a queryParser encountering an unanalyzed field is discussed. In fact, the chapter 4.7.3 of the book shows that,searching for terms contained in fields which were analyzed with the option *Field.Index.NOT_ANALYZED_NO_NORMS* will display no result. This is because, if the queryParser analyzed a query containing a number like *web2.0,* then the number's part is cut. Assume the following field is added to the index :

```
doc.add(newField("title","web2.0",Field.Store.YES,Field.Index.NOT_ANALYZED_NO_NORMS));
```

searching for the ***Termquery*** "**web2.0"** will display no result, because the queryParser removed the numbered part of web2.0 . No results are displayed although the term is contained in the field. Changing the last field's option to Field.Index.ANALYZED will resolve the problem. Because the analyzer used for indexing is the same used for search. This shows that the Lucene document is the central component that links indexing and search. The way an indexing process works on the original documents depends largely on the user application. This one can orient indexing and search with field's options, but it can also filter fields that should be display as result; Lucene make this possible, by providing the user with a field selector, and the possibility to set a boost factor for a given field.

4.2.4.b)Field selector

A field selector is a filter supplied by the *FieldSelector* class; it's purpose is to provide the application with selection options that specifies which fields of a Lucene document should be loaded or not inside the index or as search result. The select acts as a filter and allows the application to choose fields, that doesn't have a big size so that the indexing and search will be faster [28]. The *FieldSelector* class has a single method *accept(String)* which takes a field with the given name as parameter and returns an instance of *FieldSelectorResult*. Options for the selection of a field are available in the *FieldSelectorResult* Api these are:

LAZY_LOAD: loads the field but the field will contain its values and name either when it is invoked by the search application to display the field values(in this case *document.getFieldable(String)* is the method used to display the value of that field) or by the

indexing application to add the field in the index, in this case *document.add(Fieldable)* will be the method to be used

LOAD: loads the field each time the document is loaded

NO_LOAD: don't load the selected field

LOAD_AND_BREAK: load the field and immediately leaves the document after that field is loaded

SIZE :load the field's size,

SIZE_AND_BREAK load the field's size and immediately leaves the document, don't load the size of another field.

Let's move to another characterization of fields, that may also affect the search performance.

4.2.4.c) boost factor

The boost factor is a number of type *float*. This number can be initialized or modified during indexing or during search either for a field, for a whole Lucene document, or for a query. In any case it gives more importance to the field or to the query.

When the boost is used during indexing, the *setBoost(float)* method of the Document class is called for a single field before the field is added to the document, and before that document is added to the index. e.g.(*Fieldable.setBoost(0.65)*, the boost factor is set to be 0.65). If it is called for a document, the boost is added like this *document.setBoost(0.87)*, then the boost is set before the document is added to the index.

When the boost should be set during search, then it is added to the query with *Query.setBoost(0.50)*.Therefore, the documents matching this query will have their score multiplied by 0.50.If nothing is set. The computation of Lucene document score will be explain in chapter 6. As default, each field, document and query has a boost factor of 1.0.

We know from the previous chapter, that Lucene documents can also contain numeric fields. Those fields are representation of numbers, including dates. Any original documents containing numbers or date can be represented as fields with value of type *int, long, float* or *double* . Date are represented as *long* value . A numeric field can be indexed, store and have term vectors as other string-named field.

According to the Lucene core Api [12], an extension to Lucene has been developed that stores the numerical values in a special string-encoded format with variable precision all numerical values like doubles,longs, and *ints* are converted to lexicographic sortable representations and stored with different precisions. The NumericUtils class gives the list of constant values that a Precision can have these are:

BUF_SIZE_LONG = 6 is the maximum term length for long values

BUF_SIZE_INT =11 is the maximum term length for int values

PRECISIONS_STEP_DEFAULT = 4 the default value

SHIFT_START_INT=96 is set for int value .

SHIFT_START_LONG =32 is set for long value

On balance, we can say that Lucene offers a lot of options to the user application, for a better processing of its fields. Let's see which kind of data structure a Lucene document has.

4.2.5 Data structure

While debugging the *indexer.java* of *SeboL*, we observe that the single Lucene document stored has the following form:

```
[<
stored,indexed,tokenized,omitNorms
<date of creation :
java.util.GregorianCalendar[time=?,areFieldsSet=false,areAllFieldsSet=true,lenient=true,zone=java.
util.SimpleTimeZone[id=Unknown,offset=7200000,dstSavings=3600000,useDaylight=false,startYear
=0,startMode=0,startMonth=0,startDay=0,startDayOfWeek=0,startTime=0,startTimeMode=0,endMo
de=0,endMonth=0,endDay=0,endDayOfWeek=0,endTime=0,endTimeMode=0],firstDayOfWeek=1,m
inimalDaysInFirstWeek=1,ERA=1,YEAR=2010,MONTH=8,WEEK_OF_YEAR=40,WEEK_OF_MON
TH=5,DAY_OF_MONTH=14,DAY_OF_YEAR=273,DAY_OF_WEEK=5,DAY_OF_WEEK_IN_MONT
H=5,AM_PM=1,HOUR=2,HOUR_OF_DAY=15,MINUTE=43,SECOND=8,MILLISECOND=0,ZONE_
OFFSET=7200000,DST_OFFSET=0]
>
indexed,tokenized
<contents:
However all this literature shares the assumption that the web searches are motivated by an
information need.
>
stored,indexed,tokenized
<path:/home/josiane/Documents/indextest/(1).pdf>
stored,indexed,tokenized
<filename:(1).pdf>]
```

Table 7: A stored field's data structure

We can see from the structure above, that all fields of this Lucene document are stored in an **ArrayList** of elements and can be presented like this:

doc[0]=[<fieldoptionsA<fieldnameA:fieldvalueA>fieldoptionsB<fieldnameB:fieldvaluesB>...]

Where **doc[0]** is the first Lucene document in the index , the **fieldoptionsX** are constant values which match the options set by the user application for each field and **fieldvalueX** represent one or more values of that field. More precisely, this is how fields are stored in the index:

First of all, the application have to call appropriate methods to create fields with options and add them to the index; these methods are highlighted in the code snipped below:

```
fcal = new Field(
"date of creation",cal.toString(),Field.Store.YES,Field.Index.ANALYZED);
fcal.setOmitNorms(true);
doc.add(fcal);
doc.add(new Field("contents",pdfcontent,Field.Store.NO,Field.Index.ANALYZED));
doc.add(new Field("path",f.getAbsolutePath(),Field.Store.YES,Field.Index.ANALYZED));
doc.add(new Field("filename",f.getName(),Field.Store.YES,Field.Index.ANALYZED));
w.addDocument(doc,analyzer);
```

The add(Field) method adds a new Field to a Lucene Document including storage options. The addDocument(Document, Analyzer) method add a new Lucene Document to the

index .The strategy used to populate the index with Lucene documents is described in chapter5.

The next step is the storage of Fields' options and Field data in the index.

Field's options

Each highlighted option is transform to a constant value and kept in the index with its corresponding field. An option can be a boolean or a number, let's start with boolean options:

Field.Store.*YES* turns into isStored=true

Field.Index.*ANALYZED* turns into isTokenized=true and isIndexed=true

fcal.setOmitNorms(true) turns into omitNorms = true

other options that have not be modify by the user are automatically set to have a value equal *false*, like: *storeTermVector= false*: in the application no terms vector are stored,

l*azy=false*: means no field is loaded with the option LAZY_LOAD .

isbinary = false : there is no binary field in our example.

Once the options are transform to booleans values, the final information hold by the index would be a single: **stored** for isStored=true, **tokenized** for isTokenized=true, **indexed** for *isIndexed*=true and **omitNorms** for omitNorms=true. Other options are stored as numeric value, these are :

- *binaryLength* which is the length of the field when this one has a binary value. The Default value is 0.

- *binaryOffset* is the Offset for the Field with binary value. Default value is also 0. boost is the boost factor , if it is not set, then its value is 1.0 .

Options are simply append at the beginning of the Lucene Document collection like here:

indexed,tokenized<*contents:However....*>

Table 8: Field's options append before the field's name and value

Let's see in which form field's name and field's values are stored.

Field's name and Field's values

Starting from the example above, we see that each field is stored in the form

<name:fieldsData> .field's values are kept in a variable called *fieldsData* which is an array. And the field's name is stored in a variable name also as an array.

As an illustration, this is the field *path* stored in the disk:

stored,indexed,tokenized<*path:/home/josiane/Documents/indextest/(1).pdf*>

Table 9: Field's name and Field's data append in the collection

To sum up, we can say that a Lucene Document is a collection of type *ArrayList*[25]. It holds options, name and values of fields. Field's options are constant values, either of boolean or numeric type. A Field has a single name which values are stored in a one-dimensional array variable called *name*. The field's value are also stored in an array variable called *fiedsData.*

25 http://download.oracle.com/javase/7/docs/api/java/util/ArrayList.html

With this in mind, we can look at the construction of a Lucene Document.

4.3 Building a Lucene Document

The process of Building a Lucene suitable Document starts from the extraction of textual contents out of the Data source and ends with the storage of those textual Information into Fields, which build a Lucene Document. The extraction of textual contents is ensured by an extraction module. We use this expression to call each application that can extract text from common data format. In the following parts we would see that the construction of a Lucene document is a quite flexible process.

4.3.4 Using the Document handler interface

A D*ocumentHandler* is an interface that specifies a method to be used to create a Lucene Document. It is provided by the contribution package to Lucene (*ant)*. Any class which implements this interface act as a Lucene Document Handler and can creates Lucene Fields . The figure below is a class diagram showing the internal dependencies of the *DocumentHandler* interface. It specifies the *getDocument(File)* method, which provide the functionality of building a Lucene Document from a given originalFile. A user application can also choose to use the FileExtensionDocumentHandler.This class is an implementation of the *DocumentHandler* interface, it overwrites the *getDocument(File)* methodand focus its achievement on original files that have the format .txt or .html.. If it is a ".txt" file, then the corresponding extraction module is called, namely the TextDocument class, that will create Lucene Document from that file.

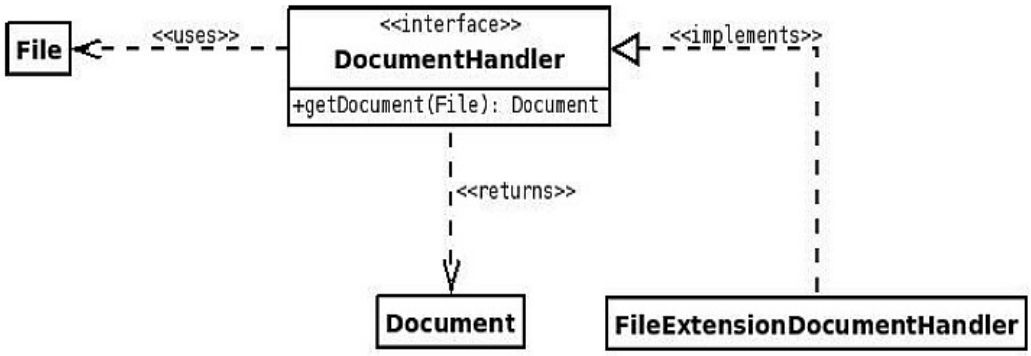


Figure 34: Processing a common file format with a DocumentHandler

In the current Lucene version(3.0) only .txt and .html Files can be proceed by the FileExtension DocumentHandler.

It is advisable for any user application, that wants to creates Lucene document, to use the DocumentHandler interface and overwrite the *getDocument*(File) method the same way like the FileExtenDocumentHandler does. The syntax of such an application look like this:

```
public class FileFormatHandler implements DocumentHandler{
        private Document doc; // creating an instance of Lucene Document
        public Document getDocument(File){
        //overwrite the getDocument() method
```

```
        Iplement DocumentHandler
        ….
        //start the parsing process
        // populates Lucene Documents with Fields
        doc.add(Field.FieldName(Fieldvalues,Field options))
        ….
        return doc; // returns the Lucene Document to the user application
        }
}
```

The parsing process in the syntax above, is done by a document parser. The next section explain how Lucene documents are created from a Pdf file, using the interface DocumentHandler and a PDF Parser.

4.3.5 Implementing a Document handler for Pdf files

In order to create Lucene documents from Pdf files, a user application muss first extract the textual contents from that file; this is accomplish by a document parser for Pdf files. A Document Parser is an application, that provides the mechanism for reading and extracting data from some kind of file format document . There are several free document Parsers available on the Internet such as,

- the SAX [26] Parser for XML document parsing,
- PDFBOX[27] for Pdf text extraction ,
- the Apache POI [28] parser for Microsoft documents extraction. This extract text from the following Microsoft files:Excel (.xls),word (.doc) powerpoint(.ppt),Publisher(.pub) and visio(.vsd)

The next step is to overwrite the getDocument(File) method of the DocumentHandler interface The figure below illustrates those steps

26 Simple API for XML (SAX) parser. project home page: http://www.saxproject.org/
27 PDFBox User guide is available at http://pdfbox.apache.org/userguide/index.html
28 The Apache POI parser is available at http://poi.apache.org/

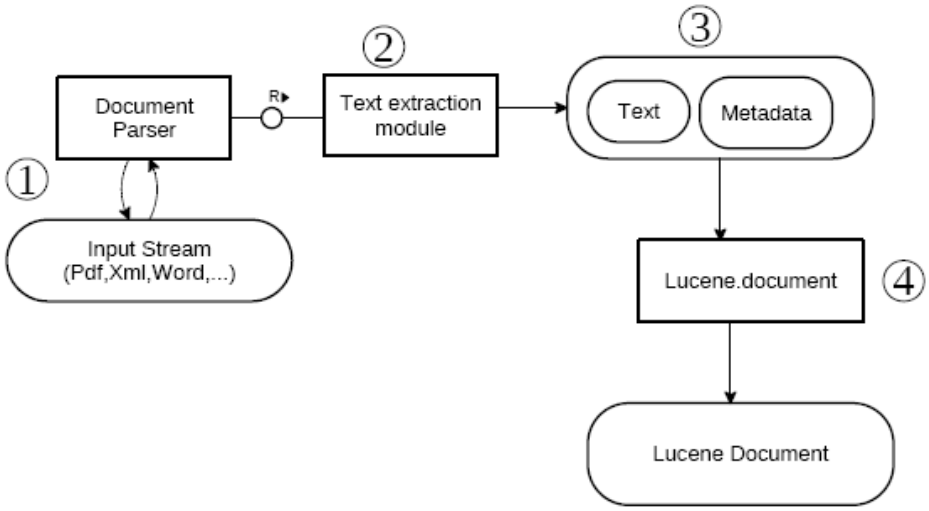


Figure 35: creating a Lucene Document with a document parser

While reading the original file, the parser encounter different semantic elements, and generate for each of them an event that it handles. A semantic element can be a Pdf object like COSString =ESCAPE which means the escape character in the Pdf document.

A parser handles an event by executing a method for example:

The getString() method of the PDFBOX parser, will be the event generated when the parser encounter a COSString element, this would return the string that the element contains. methods like getString(), are provided by the Parser Library(API). Semantic elements are values define in the Parser File specifications. There are methods to recognize the beginning and the end of the text in the original file, and methods to get the textual content. So this is the way to get to the text content. Afterwards, the application would gather the text and meta data to create Lucene document by using the getDocument(File) method of DocumentHandler interface.

In the SeboL programm, a PdfHandler was implemented for the purpose of this thesis, it extracts text out of Pdf files, and use the getDocument Handler to create a Lucene document. .here is the code snippet of PdfHandler.java:

```
public class PdfHandler implements DocumentHandler{
        public String author,title;
        public PDDocument pdfdoc =null;

        public Document getDocument(File originalFile) throws DocumentHandlerException {

        String PdfContents;  //  text content of the Pdf file
        Document doc = new Document(); // create an empty Lucene document
```

```
try {

        pdfdoc = PDDocument.load(originalFile); //load original document from the file
        if(pdfdoc.isEncrypted()){

                try {
                        pdfdoc.decrypt(""); //decrypt Pdf document if encrypted
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
        PDFTextStripper stripper = new PDFTextStripper();
        StringWriter writer = new StringWriter();
        stripper.writeText(pdfdoc, writer);
        PdfContents =writer.getBuffer().toString();
        PDDocumentInformation metadata = pdfdoc.getDocumentInformation(); // meta data

        /**Populates Lucene Document with Fields.
         *Each Field value contains the text or meta data extracted from the pdf file
        *Fields added: author, date of creation, contents,path,filename
         */
        String author = metadata.getAuthor();
        if(author!=null)
                doc.add( new Field("author",author,
                                Field.Store.YES,Field.Index.ANALYZED,Field.TermVector.YES));
        else author="no author name";
        Field fcal = null;
        Calendar cal = metadata.getCreationDate();
        if(cal==null) cal=Calendar.getInstance();
        else
                fcal = new Field("date of creation",cal.toString(),
                                Field.Store.YES,Field.Index.ANALYZED);
        fcal.setOmitNorms(true); // no norms stored for this field
        doc.add(fcal);
        doc.add(new Field("contents",PdfContents,
                                Field.Store.NO,Field.Index.ANALYZED,Field.TermVector.YES));
        doc.add(new Field("path",originalFile.getAbsolutePath(),
                                Field.Store.YES,Field.Index.ANALYZED));
```

```
                doc.add(new Field("filename",originalFile.getName(),
                                Field.Store.YES,Field.Index.ANALYZED));


        } catch (EOFException e) {
                e.printStackTrace();
        }catch(IOException ioe){
                ioe.printStackTrace();
        }
        finally{
                if(pdfdoc !=null){
                        try {
                                pdfdoc.close(); // close the COSDocument Object

                                } catch (Exception e) {   e.printStackTrace();}
                        }

                }
                return doc; // returns a Lucene document


        }


}
```

The PDFHandler class realize the 4 steps represented by the figure above.

The first 3 steps are achieve by the PDFBOX parser:

step 1: The PDDocument.load(is) method is used to open and fill-in the file in an input stream. This creates a *PDDocument* instance, which represents a Pdf file in the memory. The *PDDocument* stores Pdf File as so called Cos Objects[29] which is the base object that all Pdf document have. In other words COSObjects represents aspects of the Pdf file; a description of the COSObjects and the PDF file format is supply by the on-line Pdfguide[50].Once the Pdf File are represented according to the PDFBOX specifications, there are passed to the extraction module.

Step2: The basis of the extraction module is made up of a single class called PDFTextStripper. This class is responsible of event handling for each new semantic element.In our case the first semantic element is the element that shows, that the Pdf File is encrypted. When this element is encountered, we call the **decrypt**(“ ”) function with no parameter to decode the document.

Another event handling is done by the **writeText(PDDocument,StringWriter)** method which write the text of the Pdf file into the String Writer provided. The last event handling we use, is when the parser gets to the end of the document, then we call close() to close the Pdf file we

29 Http://pdfbox.apache.org/apidocs/org/apache/pdfbox/pdmodel/common/COSObjectable.html

opened before. Let's get to the third step.

Step3. Now that our StringWriter contains the text, we can transform this into a String using the Java methods getBuffer() and toString() . We also decide to extract meta data like the author name, the date of creation of the Pdf file. Therefore we use the class *PDDocumentInformation* which contains the Pdf meta data. The code below shows how meta data are extracted:

PDDocumentInformation supplies a list of *getXXX()* methods that returns a value if it exists in the Pdf file. Thus, we have *getAuthor()which* gives the Author name of the Pdf document, *getCreationDate() ,* the creation date . Other get methods are available in the Pdfbox Library . Now that they are all transform into string , a Lucene Document can be created.

Step4: As said before, the mechanism to create Lucene Document is supplied by the getDocument(File) method of the DocumentHandler, which was overwritten. This help the PDFHandler class to return Lucene document. Lucene Fields are previously populated with values extracted from the Pdf file, and add to the Lucene document applying the doc.add(Field) method of the Lucene document class.

In the final analysis, we can state that the creation of Lucene Document is application dependent. An application can choose to implement the *DocumentHandler* interface provided by Lucene, to transform the extracted content into Lucene Document. Or it can decide to use a common document parser and handles the extraction according to each parser Library (Api). However there are often an issue of incompatibility between the current version of the Parser and that of Lucene.

chapter 5 Structure and creation of the lucene index

5.1 Creation of a Lucene index

As seen before, it is the responsibility of the application to convert original data into Lucene Document, using an appropriate document parser. Once the data is prepared for indexing and Lucene documents are created, IndexWriter's *addDocument(Document)* method can be used to gather Lucene Document's fields value into the file index.

According to Lucene API, an IndexWriter creates and maintains an index. Each of IndexWriter's available constructors needs to specify a directory in the file system for index storage and an analyzer which will split Lucene document's fields into tokens and filter them. The FMC diagram below represents the process of creating a new index. This is the same for the opening of an existing index or the merging of different indexes. For a better understanding, let us create an Indexwriter and use it to add Lucene Documents to the index .The code below shows the syntax to use to create an IndexWriter choosing the must simplest construct: IndexWriter(Directory,Analyzer, IndexWriter.MaxFieldLength):

```
IndexWriter w = new IndexWriter(FSDirectory.open(index),new
StandardAnalyzer(Version.LUCENE_30),IndexWriter.MaxFieldLength.UNLIMITED);
```

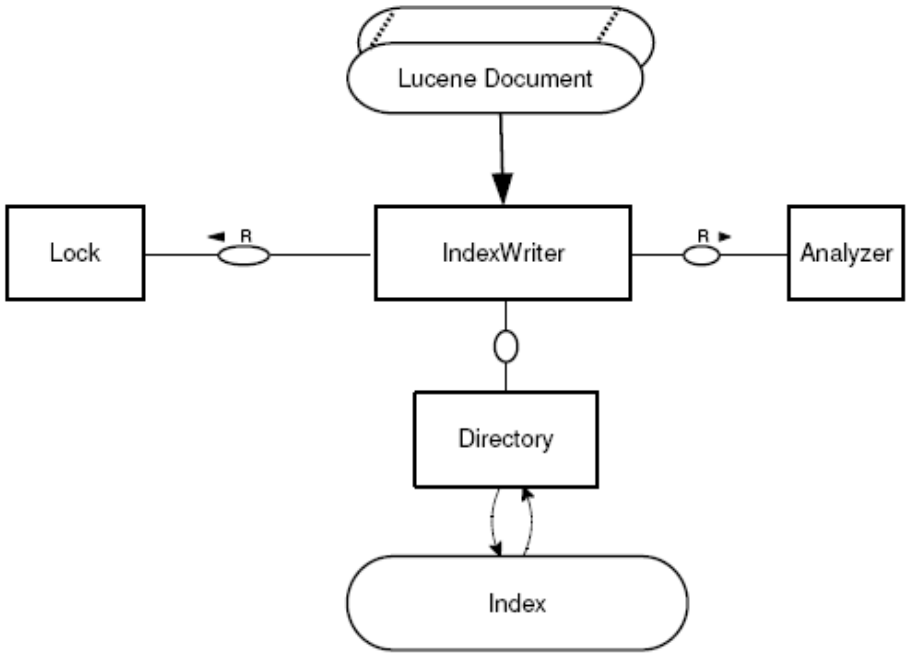


Figure 36: Creating, opening, and merging an index

First,suppose that there is no index creted , an empty one must be created in the file system. This is done by the *FSDirectory.open(index)* method."*index*" is the path to the common file system directory that will contains index files. This method returns a kind of Directory representation, which is NIOFSDirectory for a non-Windows environment and a

SimpleFSDirectory for Windows operating system. Both are Directory implementations that store the index files in the file system. This Directory is the index directory. An IndexWriter can only be assigned to one directory in the fiel system, this is why the IndexWriter automatically creates a lock on the index directory it currently uses. This functionality is provided by *IndexWriter.open(Directory)* method. In case an IndexWriter tries to open the same index directory that another IndexWriter has open, an error occurs and a LockObtainFailedException exception is thrown.

Next, the analyzer,which is a standardAnalyzer in our example, is applied on the Lucene Documents to extract index terms from them. The Analyzer is therefore a strategy for the indexWriter for the insertion of terms into the index. This strategy would be discussed in the next part .Because changes happens regularly on the Lucene core, the analyzer requires a Lucene version, with which it is compatible. In our example it is Lucene version 3.0.

Finally, the third parameter sets the maximum length a field should have. This is the number of terms a field should contain. In our case we set this value to UNLIMITED, therefore, all the terms of the field should be considered.

In the following paragraph, we will see how the IndexWriter strategically employed the analyzer to create a Lucene index,then we would depict the building process of an inverted index. Afterwards the indexing algorithm hiding behind that process will be studied.

5.1.4 The Analyzer as a strategy for the IndexWriter

According to the book “Lucene in Action”[2] ,” When you call IndexWriter's addDocument(Document) method, Lucene first analyse the data to make it more suitable for indexing ...”. Therefore, Lucene Analysis provides the mechanism to extract terms from String value of a Lucene Field. Moreover, calling the *addDocument(Document)* method comes back to calling the *addDocument(Document, Analyzer)* method as we can see in the IndexWriter code snippet below:

```
public void addDocument(Document doc) throws CorruptIndexException, IOException {
  addDocument(doc, analyzer);
}
```

This method is used by the IndexWriter to add a Lucene document to the index using the provided analyzer. The reason why the IndexWriter can't directly add some Field's terms into the index, is because they contain useless words for search. Such as stop words, hyphens, apostrophes, punctuations, and so on.

The strategy of an analyzer consists on two steps:

- Building a Tokenizer to split Strings from Lucene Document's field into tokens
- Applying one or more TokenFilter(s) to those tokens

Assume an initial text is parsed, indexed and stored as a field <content,String value,..> where String value contains the value of content is the field name chosen by the application. The analysis process starts with a Tokenizer, then one or more Tokenfilters. In each process an example is provided for better understanding.

Building a Tokenizer

Most Analyzers first builds a stream of tokens from a Lucene Document,. This Stream is a tokenstream which opens a reader, reading the content of the name-value pair of each field, and returning a them as a set of tokens . That set is generally devoid of punctuations, apostrophes, white spaces and more, depending on the type of tokenizer applied to it. There are different types of Tokenizers according to the type of analyzer chosen during indexing. The table below gives an overview of some Tokenizers with their use.

Lucene core Tokenizers	What does it do?
StandardTokenizer	• Remove punctuations followed by whitespace, • Remove hyphens, unless there is a number inside • Recognise email addresses and hostnames
KeyWordTokenizer	• Consider the entire input Stream(text) as a single token
CharTokenizer	• Splits text at character
LetterTokenizer (this is a CharTokenizer)	• Devide text at letters.
WhiteSpaceTokenizer (this is a CharTokenizer)	• Devide text at whitespace

Example: **Tokenizing a content field's value with a StandardTokenizer**

Let's consider the following text taken out of “broder_websearch.pdf” Pdf file:

(On the web "the need behind the query" might be

. Informational

• Navigational

• Transactional) [5]

Assume the text is parsed and stored as a field's value content, this is what we obtain when applying the StandardTokenizer: The punctuations and white spaces are removed and the value of the field(String value) has been broken into tokens.

Original text(String value)	Tokens Stream output(Term text)
On the web "the need behind the query" might be **. Informational** **• Navigational** **• Transactional**	On
	the
	web
	the
	need
	behind
	the
	query
	might
	be
	Informational

	Navigational
	Transactional

Filtering tokens:

Whereas the Tokenizer extracts tokens out of Lucene Fields, the TokenFilter pass them through its filtering mechanism. The mechanism of Token filtering consist of:

- removing meaningless tokens from them such as stop words,
- reducing tokens to their roots : this is called stemming

Here is a list of some filters and their respective functionalities:

Lucene core Tokenfilters	What does it do?
StandardFilter	• Normalizes token extracted by standardTokenizer; this means: ◦ removes **'s** from the end of words ◦ removes dots from acronyms
LowercaseFilter	• Lowercased token stream
StopFilter	• Removes stop words: A stopwords is an englisch word which can be contained in this list **{but, be, with, such, then, for, no, will, not, are, and, their, if, this, on, into, a, or, there, in, that, they, was, is, it, an, the, as, at, these, by, to, of}.** Notice that, this list is not exhaustive
PorterStemFilter	• Uses the porter Stemming algorithm to stripp tokens

Example: **Filtering a Content Value with the PorterStemFilter**

Considering our previous example the token stream obtained has to be filtered according to the Stemming algorithm, here is how the PorterStemFilter acts on those tokens :

Tokens Stream input	Stemmed tokens output
on	on
the	the
web	web
the	the
need	need
behind	behind
the	the
quer**y**	queri
might	might
be	be
inform**ational**	informat
navig**ational**	navig
transact**ional**	transact

“The Porter stemming algorithm (or ‘Porter stemmer’) is a process for removing the commoner morphological and inflectional endings from words in English. Its main use is as part of a term normalisation process that is usually done when setting up Information Retrieval systems.”[26]
In our case the endings or suffix **y**, **ational** and **tional** are removed from the token's term. The algorithm stipulates that, “A *consonant* in a word is a letter other than A, E, I, O or U, and other than Y preceded by a consonant.”[27] .So the **y** preceded by the **r** in the term **query** is not considered as a consonant and therefore it is called a **vowel**.

According to Martin Porter [18], a Word may be written as $[C](VC)^m[V]$ where C is a consonant, V is a vowel. And $(VC)^m$ means vowel followed by consonant repeated m times. The integer m represents the measure of a word or a word part represented in that form

m=0 is the null word like Y, BE, THE
m=1 is a single word like WEB,MIGHT, TREES
m=2 is a word f length 2 like PRIVATE .

To remove suffixes, Porter uses this rule : **(condition) S1 -> S2** where S1 and S2 are suffixes under one of the following conditions (condition) :

*S - the stem ends with S (and similarly for the other letters).
v - the stem contains a vowel. *d - the stem ends with a double consonant (e.g. -TT, -SS).
*o - the stem ends with cvc, where the second c is not W, X or Y (e.g. -WIL, -HOP).[27]

If a word ends with S1 and the stem before S1 satisfies the given condition, S1 is replaced by S2.
To illustrate the stemming process, let's take our example tokens stream particularly the terms query, informational, navigational and transactional:
The 'Porter stemmer' says:

- a word in form of (*v*)Y becomes →i ,so **query → queri**
- a word in form of (m>0)ational becomes → ate : this is to say **informational → informate** and **navigational → navigate**
- a word in form of (m>1)ATE → , the suffixe is to be removed **informate → inform** and **navigate → navig**
- a word in form (m>0)TIONAL becomes → TION. **transactional → transaction**
- a word in form ((m>1) and (*S or *T))ION → lost it suffix. **transaction → transact**

5.1.5 Building an inverted index

Now that the analyzer has split and filter terms inside the tokens, the next step is to build the inverted index and store it. For this purpose we would store another Pdf file in the source directory **indextest** and index them.We now have two pdf files: ***"broder_websearch.pdf"*** , and ***"Search engine history.pdf"*** in the indextest source directory.

First of all ,each the example Pdf file is automatically assigned to an id, as was seen in chapter 4 , those Pdf Files are parsed using an appropriate Pdf Parser and transformed into Lucene Documents. Afterward, the Lucene Document fields are analysed similar to the description in 5.1.1.Then, to achieve the whole process the resulting stream of tokens are inserted in the inverted index . This is done while building the inverted index:

- A list of (term, DocId) is created, where term is a token's term and docId is the document Id number, where the term occurs.
- The list is then sorted in an alphabetical order of terms.
- Multiple instances of a term may occur, these are then grouped in a single term and the list of ids of the document where the term occurred are stored in an array.
- The list of terms are stored in a so called term dictionary, and the DocIds constitute the posting Lists. Additionally, the frequency of a term in all documents it occurs in is also stored in the term dictionary, this reduces the memory requirement. Instead of accessing the memory n times to retrieve the n occurrences of a term, it is better to collect them when collecting the term itself.

As an illustration to these operations let's consider these two pdf files content:

(1) “***However all this literature shares the assumption that web searches are motivated by an information need”***

(2) “***Search engines uses "spiders" which search(or spider) the web for information.”***

the document id numbers are respectively (1) for ***"broder_websearch.pdf"*** ,(2) for***"Search engine history.pdf"*** .After parsing them, they are Analyzed with the StandardAnalyzer. The StandardAnalyzer uses a StandardTokenizer,then applies to the resulting token stream a StandardFilter, LowercaseFilter and stopFilter, referring to the description of their functions in 5.1.1 we obtain the following the following tokens stream :

This is the list of terms with document Ids where they occur:

Extracted and anylzed terms	DocIds
all	1
literature	1
shares	1
assumption	1
web	1
searches	1
motivated	1
information	1
need	1

search	2
engines	2
uses	2
spider	2
search	2
spider	2
web	2
information	2

The next steps is to sort the list alphabetically, The following result is obtained:

terms(sorted)	DocIds
all	1
assumption	1
engines	2
information	1
information	2
literature	1
motivated	1
need	1
search	2
search	2
searches	1
shares	1
spider	2
spiders	2
uses	2
web	1
web	2

The last two steps leads with the removal of the duplicated terms and the addition of the frequency (tfq) of an appearance of a term in both documents. By the way, according to the Information retrieval book[4] , complexes queries like ***web AND search OR literature*** can be processed in order of increasing term frequencies.

The resulting list constitute the terms dictionary and the posting lists for those two files.

The terms dictionary contains the list of terms encounter in both Pdf files to each term stored in the index, its term frequency is associated to its postings list.

Terms Dictionary (term,tfq)		Posting lists
all	1	1
assumption	1	1
engines	1	2
information	**2**	1,2
literature	1	1
motivated	1	1
need	1	1
search	**2**	2
searches	1	1
shares	1	1
spider	1	2
spiders	1	2
uses	1	2
web	**2**	1,2

Suppose the porterStemFilter is also applied to the standardanalyzer. Stemming terms to their roots will lead to better search because the disk access is reduced, therefore making it easier to retrieve the terms that are similar to the terms in the user query.
Following the Porter's rule [27] for removing suffixes: (condition)S1->S2, leads us to the following results:

Terms Dictionary (stemmed terms,tfq)		Posting lists
all	1	1
assumpt	1	1
engin	1	2
inform	2	1,2
literatur	1	1
motiv	1	1
need	1	1
search	3	1,2
share	1	1
spider	2	2
us	1	2
web	2	1,2

That is to say *searches* and *search* have the same roots. As well as *spiders* and *spider*. Calculating the terms frequency makes the search application more efficient for each search. Postings lists are stored in the hard drive while the terms dictionary is store in the memory. Terms are frequently updated, after some changes the updated terms are merged in the

hard drive. This causes modifications in the posting lists. The actualization of terms includes their removal and their modification. Before going in to details about the operation on the index terms, let us see which algorithm hides behind the building of an index.

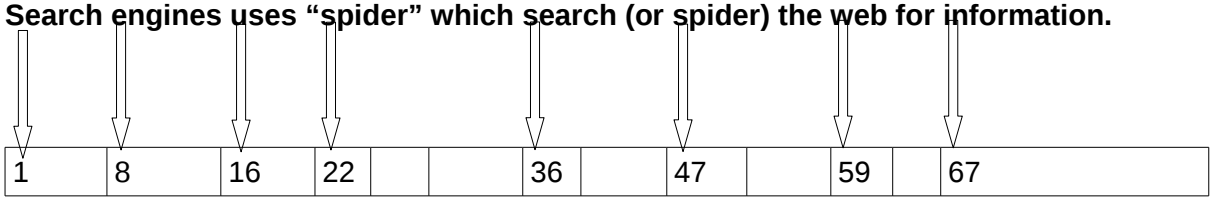
5.1.6 indexing Algorithms in information retrieval

Since the arrival of the world wide web, information retrieval has developed algorithms for efficient search results. There are retrieval algorithms, filtering algorithms and indexing algorithms. This Part focuses on the last algorithm. In the implementation of Information Retrieval systems, various indexing algorithms are used to determine how an index is constructed. According to[45] , those algorithms : *signature files*, *inverted files* and *patricia arrays* (Suffix-arrays). For each algorithm, an approximation of the index storage capacity needed for large indexes is given. Later in chapter 6, a comparison of search methods will be done between these three algorithms.

5.1.6.a)Inverted Files

An inverted file is a word-oriented mechanism for indexing a text collection in order to speed up the searching task [45]; it is composed of : the *vocabulary* and the *occurrences*. The vocabulary is the list of words, that the text contains, and the occurrences is the list of positions in the text, where each word in the text occurs. To exemplify this process, let's take the example of the following text taken out from (2).pdf:

Search engines uses “spider” which search (or spider) the web for information.



The inverted file for this text will look like this:

Vocabulary	**occurrences(positions of words in the text)**
search	1, 36
engines	8
uses	16
spider	22, 47
web	59
information	67

Building and maintaining such an index does not cost too much. According to Yates and Ribeiro-Neto[45],if a text has n characters its index can be built in $O(n)$ time, and with many samples, it was observed that both space requirement and amount of text traversed are near to $O(n^{0.85})$. This means that, for a large text, an inverted file has a sub linear search time. This is why inverted files are better than other indexing methods.

Two methods can be used to construct an inverted file, the most popular one which is considered as good practice, consist on the following steps:

split the index into two files, where

the first file contains the list of occurrences also called *"**posting list*** ", and is stored contiguously in the memory

the second file contains the vocabulary stored in an alphabetical order in the memory at search time. This file includes for each word in the vocabulary, as well as a pointer to its posting list in the first file. This is why the number of appearances of a word can be known from the vocabulary.

The second approach is to store the inverted file in a data base, this method is described by Böhm [47]as been transaction safe. The steps encountered during the building are:

- Build a text table , that will contains ids and meta data
- build a term table, that will contain the vocabulary with weight for each term
- build a doc-term table with document-term pair and occurrences of terms in the text

If text is permanently added to the inverted file, it would soon became very large. Avoiding memory overload or slow searches is a necessity. Brown, Callan and Croft[48] proposed an alternative solution to this problem. This consist of using a data management facility of a persistent object store to design a more sophisticated inverted file index based on a fast incremental update. In other words, while the size of the text collections grows, postings lists are updated.

For large texts, which require large index and more memory space, there is an alternative algorithm described by Baeza-Yates and Ribeiro-Neto [45]. In fact, when the memory is full, a partial index $\mathbf{I_i}$ obtained up till now is created and written to disk then, later on it is removed from the memory. Afterwards the indexing of the rest of the text continues. Each time a new partial index is created, it is merged together with the old one, in a hierarchical fashion. For instance, $\mathbf{I_1}$ and $\mathbf{I_2}$ are merged together to obtain $\mathbf{I_{1..2}}$; $\mathbf{I_3}$ and $\mathbf{I_4}$ are merge to $\mathbf{I_{3..4}}$ then $I_{1..2}$ and $\mathbf{I_{3..4}}$ are merged to obtain $\mathbf{I_{1...4}}$ and so on.

5.1.6.b)Signature Files

Indexes based on signature files are word-oriented indexes based on hashing. Each text in the file is braked in blocks of x words each. For each text block of size x, a bit mask called a block signature is assigned to it. The block signature is the OR of the signatures of all the words it contains. Therefore,it is easier to add text into such a structure because the signature of new words are just added to the old one. According to Faloutos and Christodoulakis[49] , while documents are sequentially stored in a "text file", their abstractions are stored sequentially in the "signature file". When query arrives, the relevant documents are checked or returned to the user as they are. More about searching signature files is given in chapter 6. An example of signature files is given in[49] .

5.1.6.c)Suffix Arrays / Patricia Arrays

A suffix array, is an implementation of suffix trees[30], but it requires less space. A text is considered as a long string, in which each text part from position i to position n is called suffix where i is the position number where the text suffix begins and n is the end of the text. This indexing method can be used for word, like the inverted files, likewise for text characters, this is why suffix arrays are suitable for biological applications[31] like: genome

30 L.Allisons. Suffix trees. [2010].URL: http://www.allisons.org/ll/AlgDS/Tree/Suffix/

31 Suffix trees in computational biology , available at : http://homepages.usask.ca/~ctl271/857/suffix_tree.shtml

alignment and signature selection.

To construct a suffix array, a tree is built, the edge are words contains in suffixes, while the leaves at the end of this tree are pointers on each suffixes. When reading the tree from top to bottom all the suffixes of the text are retrieved. In fact, the suffix Array is an array containing all the positions of text suffixes, when these are ordered in lexicographical order. The suffix array was originally described as “Patricia” trees by Gonnet [46], in his paper “**Unstructured data bases or very efficient text searching**”.

In the figure below describes an example of building a suffix array for the file text in (2).pdf

example text: ***engines***

This is the list of suffixes it contains:

List of suffixes(start positions)	**Lexicographical order of suffixes**
engines(1)	engines(1)
ngines(2)	es(6)
gines(3)	gines(3)
ines(4)	ines(4)
nes(5)	nes(5)
es(6)	ngines(2)
s(7)	s(7)

In order to build a tree we would start by drawing the leaves which represents common prefixes . Common prefixes are substrings of suffixes. In our case, these are 'e' which is the common prefix to suffixes 'engines' and 'es'. 'n' is a common prefix to 'nes' and 'ngines'; while removing the prefixes from those four suffixes, we obtain these 3 branches nodes

first branching node: is the root node

second branching node: 'ngines' and 's' are edge of the leave 'e'

third branching node: 'es' and 'gines' are edges of the leave 'n' then we have

leaves with other suffixes in alphabetical orders: 'gines','ines' and 's'

Suffixes are accessed by each Index points which is the position where the suffix started. This is what we obtain:

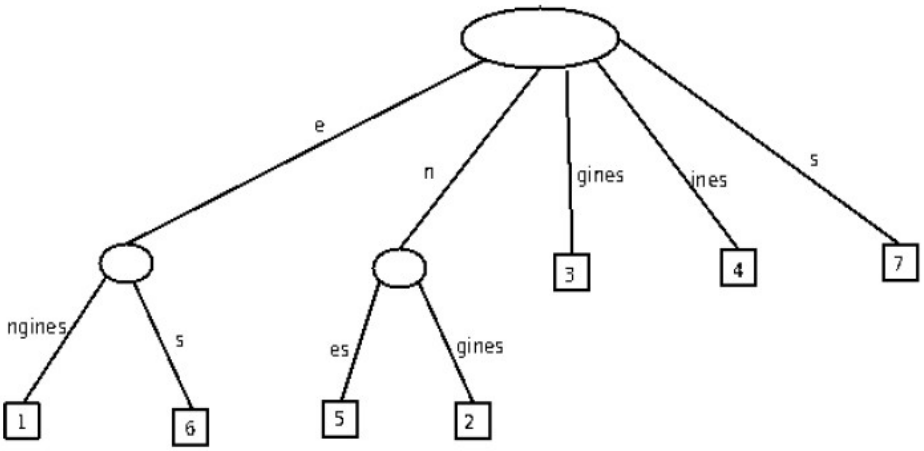


Figure 37: Example of suffix_array for the text "engines"

The tree is reading from the bottom to the top, in order to retrieve all suffixes.

According to Baeza-Yates and Ribeiro-Neto[45], each node of the tree takes 12 to 24bytes space, this is 120% to 240% more text size. If n is the size of the text collection, then the suffix array will have a space complexity of O(n), which is more than the inverted files .Besides that, suffixes needs to be sorted in lexicographic order. This is not the case in the inverted files method since text are sorted by text position. Concerning the construction of suffix Arrays for large index Baeza-Yates and Ribeiro-Neto shows that a special algorithm is needed. The original text should be split into blocks sorted in the main memory. A suffix array is then built for each block, each new suffix is merged with the previous one, until there is no more blocks. The number of suffixes is computed and stored in counters. This construction process takes 5 to 10 times more than the construction of an inverted index [45].

Now that we have given an overview of the indexing methods in information retrieval, we can state, that Lucene indexing method is a way similar to the inverted files. First of all let's see the algorithm behind Lucene indexing.

5.1.7 Lucene indexing algorithm

According to Lucene's founder Doug cutting[], the creation of an index is incrementally possible.

In his Lecture at Pisa[17], Cutting describe the indexing algorithm in two steps:

(a) A basic algorithm which consist on:
- building an index for each document.
- Merging a set of (primitive) indexes

(b) An incremental algorithm.

The Lucene indexing algorithm has been depicted by Cutting in his lecture at the Pisa university [17]. Here goes the algorithm:

- push new indexes onto the stack
- let b=10 be the merge factor; M=∞

```
for (size = 1; size < M; size *= b) {
  if (there are b indexes with size docs on top of the stack) {
    pop them off the stack;
    merge them into a single index;
    push the merged index onto the stack;
  } else {
    break;
  }
}
```

Figure 38: Lucene Indexing algorithm (source:http://lucene.sourceforge.net/talks/pisa)

(a) Basic algorithm

The first step of this process has been described in part 5.1.2, where an index was constructed for two Pdf documents (1).pdf and (2).pdf. The index is made up of terms dictionary and posting lists. Comparing the building of Lucene index for those two documents, with the previous indexing methods, a similarity can be observed with the inverted files method. The so called *vocabulary* is represented in Lucene by the terms dictionary. This also contains the list of terms that occurs in the text in documents (1) and (2). Moreover, the terms dictionary of Lucene contains term frequency, for effective search. In inverted files, weight can be added to the vocabulary for the purpose of search. Another affinity is between the posting list. They are called the same in both methods, and also accomplish the same task, this is to keep the references to each document. In information retrieval, postings are occurrences of a word in a single document, Lucene rather stores this occurrences in the terms dictionary, and uses the posting to keep the document ID of all documents where the term occurred. More details about the building of an index was given in 5.1.2. In that part, the Lucene index has been built for 2 Pdf files.

Thereby, we anticipate the next step of the basis algorithm,which consist on merging multiple primitive indexes into one . The word *primitive index* is used because it is the first index resulting from the indexing process, applied to a given file, a primitive index is an index. For a given number of indexes , let's say n, as soon as the n+1 segments is created, the first n one are merged into a single index .a segment is a *sub-index, i*t is an independent index, that can be searched.

According to cutting, a stack should be created to hold new incoming segments and indexes. In the algorithm of the figure above,the variable b is the merge factor. It was previously discussed in chapter 3.1 , while talking about the indexing package, it was said, that a merge policy and a merge factor is used, to control merging during index writing, the merge factor is 10 by default this is why *b=10*. The *mergePolicy* is the index component that determines the

sequence of merge operations to be used during merging and optimization.

It has a subclass called *logMergePolicy*, that merges indexes(and segments) into ***levels*** of exponentially increasing sizes: it is the **logarithmic merge**. Two indexes of the same level are either primitive index(level 0) or have been merged at least once . Büttcher and Clarke [50] describe a level as a ***generation***; he uses the concept of index generation to decide when to merge sub-indexes. Therefore, an index is of generation 0 when, it was created directly from in-memory postings and stored in to the disk. If the merge factor is 10, each level should have fewer than 10 indexes(1 to 10). Once a new index is pushed onto the stack, if a level is already full with the 10 indexes, no more index is added to it, and the indexes at that level are merged into one single index. Primitive merges are done on the first level, each time the number of indexes reaches 10. On the second level, the size grows up exponentially to size*b segment of indexes. From the second level on, the incremental algorithm is applied.

(b) Incremental algorithm (logarithmic merge)

After the first merge,the stack contains 10 (1*10) segments, for a merge factor b=10. The next incoming segments on the first level,are gathered in the memory until the 10th segments ,when the 11th segment is created, other 10 are merged into the stack as a single index, now the stack contains Index1*Index2 which is 100 indexes. The same process describe in (a) is done to create the next set of indexes, by the third set, the stack will have 10*10*10=1000 indexes . By the 11th index the stack would already contain 10 segments of indexes of size 10 each : Segment1, Segment2,.....,Segment10. Therefore those 10 Segments would all be merged in a single big segment of size 10 000 000 000 indexes(10 Milliard indexes). According to Cutting, the maximum size of the stack is infinite. This is to say , the algorithm is applicable for very large indexes. The illustration below shows an example of the merging process for M=14 documents indexed, and b=3 for the merge factor.

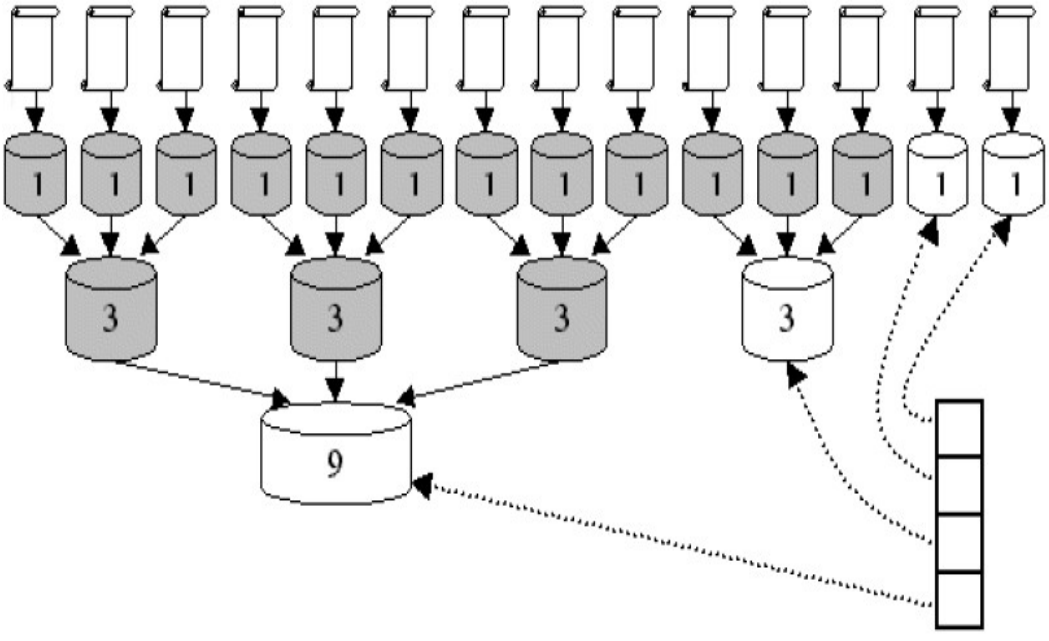


Figure 39: Indexing diagram(M=14, b=3)
source:http://lucene.sourceforge.net/talks/pisa

We notice that each single document from any common file format(Pdf,Html,Xml,tyt,...) is indexed in one single index. Therefore, we have 14 indexes for the 14 documents.

This is what happens Step-by-step:

Assume the documents are D_i (D1 , ..., D14) . and levels are L_0,L_1,L_2. The figure below shows the incremental algorithm.

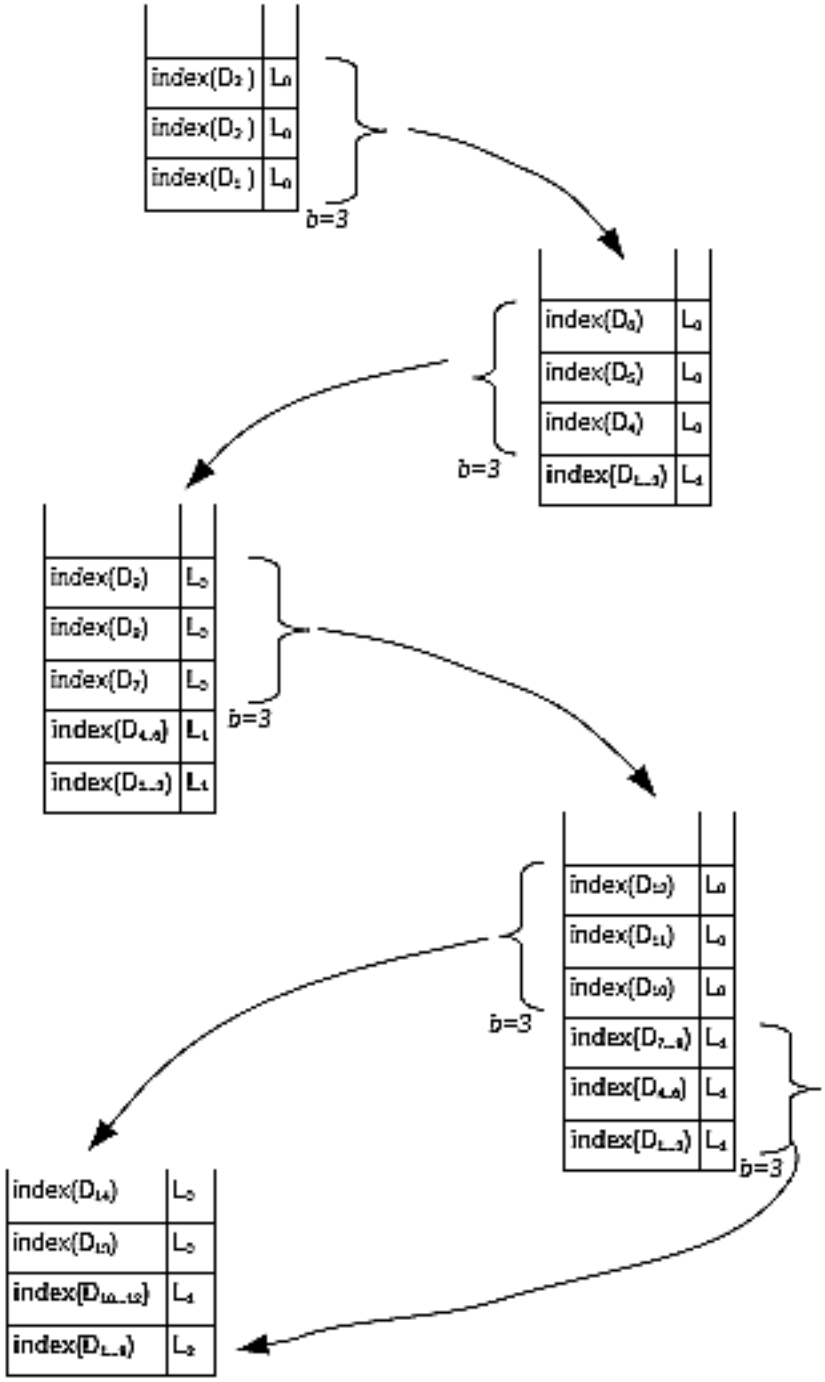


Figure 40: Logarithmic merge for 14 indexed documents(boost factor=3)

- A stack is created to maintain indexes
- the first three indexes are merged into one segment: index ($D_{1...3}$). Stack size= 1*3
- the next three are added incrementally into the stack, then merged into the second segment. Stack size = 2*3
- the last three are added one after another, then merged into the third segment. Stack size=3*3
- Two new indexes are created, but a third one is missing, so no merging under the define merge factor(b=3).
- On the first level,we have 3 segments, they can be merge into another segment
- two other segments have been created, they can't be merged, a third segment is required.
- In the stack we have the last merged segment on level2, one segment of level 3 and 2 indexes. Together, this is 4 files on the disk, corresponding to 14 indexes.

From the description of the inverted files indexing method in 5.1.3.a, we can state, that Lucene uses the inverted files as indexing method. Lucene index is made of a list with term dictionary and posting list. During indexing, partial indexes are created, and merge together into so called segments. The merging of indexing in Lucene is incremental. It is called in information retrieval: logarithmic merge. Büttcher and Clarke[50] calculate the total number of disk operations necessary to index a text of size n is $O(n.log(n))$.

The next thing to do is to study the structure of Lucene index on the disk.

5.2 Structure of the Lucene index

From the knowledge of the indexing concepts and algorithms given in chapter 3 and in the first part of this chapter, a study of the components and their data structure will help understand the actions behind indexing, and the on-disk file structure of a Lucene index.

5.2.4 Components and their relationship to each other

As a reminder, here is a class diagram showing the key components of Lucene, and their relationship to each other.

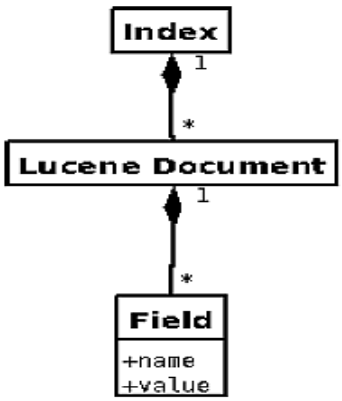


Figure 41: components of a Lucene index

A Lucene index may be a set of segments. Each segment is a single index , that has the same composition as the Lucene index. A Lucene index has one to several Lucene documents a Lucene Document comprises one or more Fields; both of them are nested and dependent on each other. Before index writing, a Lucene document comprise all the fields created by the user application parser including meta data, it is the representation of the extracted text with respect to the methods of the Lucene library for the representation of a Lucene document. After analysis and index writing, the document is made up of stored fields, these have been handled in Chapter4.2.1.a. From the previous studies in chapters(3 and 4) we can say that a Lucene field cannot exist by itself, it must belong to a Lucene document. Nevertheless, it contains all data intended for search. These data are inside the field's name and the field's values ,but much more inside the field's values ; these are references to the documents in which they belong as well as the user search. results They are words, numbers and URL. In short, the results display by the search application are Lucene Field's values. Since Lucene stores various informations with fields, the structure of the index on the disk does not look like a list of words or numbers. It is stored in special files with special data formats.

5.2.5 Data types and formats

Each index or segment has the same structure. This varies with the manner with which the fields have been indexed. At the beginning of this thesis I start indexing and searching into the pdf file *broder_websearch.pdf* [8]. After running the *indexer.java* class, this is how the on-disk index looks like:

Name	Size	Permissions	Owner	Type
_0.fdt	703 B	-rw-r--r--	josiane	unknown
_0.fdx	12 B	-rw-r--r--	josiane	unknown
_0.fnm	58 B	-rw-r--r--	josiane	unknown
_0.frq	1.3 KiB	-rw-r--r--	josiane	unknown
_0.nrm	8 B	-rw-r--r--	josiane	unknown
_0.prx	3.5 KiB	-rw-r--r--	josiane	unknown
_0.tii	160 B	-rw-r--r--	josiane	unknown
_0.tis	9.3 KiB	-rw-r--r--	josiane	unknown
_0.tvd	6 B	-rw-r--r--	josiane	unknown
_0.tvf	13 B	-rw-r--r--	josiane	unknown
_0.tvx	20 B	-rw-r--r--	josiane	unknown
segments_2	240 B	-rw-r--r--	josiane	unknown
segments.gen	20 B	-rw-r--r--	josiane	Genesis ROM
write.lock	0 B	-rw-r--r--	josiane	empty document

Figure 42: on-disk stored index files, after indexing "broder_websearch.pdf" Pdf file

This illustration shows all files a Lucene index can hold First of all, we will see the data types, that Lucene uses to store its data on the disk, then we will see which data it stores, and in which format there are kept on a disk

a) Data types

According to the Lucene documentation[32], there are two kind of data: The Primitive data type and compound data type.

- **Primitive data type**

There are six of them:

Byte: this is an eight-bit byte, Files are read and written as byte by the IndexInput and IndexOutput class of a Lucene.Store package (see chapter3.1) that provides these mechanism.

Uint32 : This is a 32-bit unsigned integer, written as four bytes whose value is between 0 and 4.294.967.295. Uint32 is available on Windows platforms.

Uint64 : is a 64-bit unsigned integer, written as 8 bits. Their values are ranged between 0 and 18.446.744.073.709.551.615.

Vint: This is a variable-length format for positive integers. It is used when there are more bytes in an integer value. Each integer between 0 and 127 will be represented by one byte, Between 128 and 16,383 it will be represented by two bytes and so on. Each byte is 8-bit in length

Chars This is the representation of a single character in UTF-8 encoded bytes.

String Lucene represents words as string. A single string is made up of its length written in Vint and characters written as chars.

- **Compound data type**

The single compound type is Map<String,String>, it is used to store couples of values like field names and values.

Each index file is stored in one of these seven data types. The figure above, shows that all index files that have been created during indexing occurs once in that index. These files give information about which field options were chosen during indexing: either STORE.YES, TERMVECTOR.YES, and so on. Lucene index Data are divided in two categories: Per-segment files and per-index files.

b) Lucene index Data and formate

- **Per segment files**
  - **Field names**

    This is the set of field names; these are stored in a field info file with an extension *.fnm* .Field names are kept in the form FNMVersion,FieldsCount,<FieldName,FieldBits>^{FieldsCount}

    let's take the example of the third field “contents” the field names should be:FNMVersion is allways -2, FieldsCount is a Vint , in our case the field contents has count=8, converting it in Vint, we obtain FieldsCount=00001000 . FieldName is “contents”. Field Bits is a 6-bits byte, where each byte is either 1 or zero for the first two bits. The other four are lowest-order bit set respectively to 0X04 when the field has a term position, 0X08 when the term offsets are stored

32 http://lucene.apache.org/java/3_0_2/fileformats.html

with the term vectors, 0X10 if the norms are omitted, 0X20 if payload are stored with Field in our case. We choose to store term vectors of the field “contents”, to index its content but not to keep it into the index. The Field names may contain this record:

FNMVersion	FieldsCount	IsIndexed	isTermvector	Termposition stored	Termoffset stored?	Norms omitted?	Payload stored?
-2	1000	1	1	0	0	0	0

Table 10: An entry in the Field names(.fnm) file for the field"contents"

- **Stored fields** contains a list of (name,value) pairs for each Lucene document. Where name is the field name and value is the field value . The stored fields are identified by field index, and the content of stored fields are kept in field data:
  - Field index are *.fdx* files , they contain references(or pointers) to each document's field data in Uint64 data format. It is used to find a field in the field data.
  - The Field's data are stored in another file with an extension .fdt. in the form <FieldNum,Bits,Value>^{FieldCount} where FieldNum and FieldCount are Vint, Bits is the Byte value of the field, if the field has binary values, Value is either a String or a binary value. stored fields are returned for each hit during search,
- **Term dictionary**

  A dictionary is the vocabulary of terms used in all indexed fields. It holds terms(Term) sorted in a lexicographic order,the DocFreq which is the number of documents containing a term t and pointers to: term frequency,called FreqDelta, proximity data (ProxDelta) and position data(SkipDelta). The term dictionary is stored in two files:
  - A *.tis* file which contains the Term informations in the form <TermInfo>^{TermCount}. TermCount is the number of term informations available. A single term information (TermInfo) is the list of <Term,DocFreq,FreyDelta,ProxDelta,SkipDelta> in this list, a term comprises of a VInt integer, a String called suffix, and another *Vint* number in the form <PrefixLength,Suffix,FieldNum> The second file representing the dictionary of terms is the ***.****tii* file. Note that each pointer XXDelta is the difference between this pointer and the previous one.
  - A *.tii* file contains index to term Information (.tis) file, .tii file give access to .tis file, It has the same structure as the tis file but except for the *indexDelta* which is a number of type *Vlong* that determines the position of a term's *termInfo* in the *.tis* file.

- **Term frequency data**

  This is for each term in the term dictionary, the number of documents containing that term, and the frequency of that term in the document If *omitTf* is false. OmitTf=false means the term frequency and position are not omitted. Remember while Building the inverted index in 5.1.2, we had entries like this one:

search	3	1,2

Where *3* is the frequency of the term "*search* "in the document and *1,2* are the numbers of all documents which contain the term "search".Both frequencies are stored in the frequency data in a *.frq* file.

The .frq file contains term frequencies(TermFreqs) ordered by increasing number of documents. In our example 1,2 .Its data formate is <TermFreqs>^{DocFreq} .Where TermFreq is a set o Vint numbers, and DocFreq is the same like in the Term dictionary

- **Positions**

  For each term in a Lucene document, the position(s) at which the term appears, are stored in the positions file which is a *.prx* file. If the variable OmitTf=true, no positions are stored. The term positions are ordered by term, . This file holds the position as a list of Vint numbers in the form <TermPositions>^{TermCount} .

- **Normalization factors**

  For each indexed field in each document, norms are stored in bytes with each byte per document. Each byte encode a value that would be multiplied by the score for hits on a field. Norms are stored in a *.nrm* file. An explanation of norms was given in Section 4.2.2. A norm is stored in the following format <Byte>^{SegSize} where SegSize is the size of the segment containing index files.

- **Term vectors**

  If the field option Field.Index.TermVector.YES have been given to a field during indexing, which is the case for the field "contents", then the term vector is added to that field. A Term vector contains: the term text and the term frequency. It is stored in 3 different files: *.tvx* file, *.tvd* file and *.tvf* file

  - *.tvx* file is the document index. Like other index files it stores references, like the offset into .tvd and .tvf files. It stores data in the form:

    TVXVersion<DocumentPosition,FieldPosition>^{NumDocs}

    TVXVersion is from type Int, DocumentPosition and FieldPosition are Uint64. NumDocs is an integer number

  - *.tvd* file is the document file, it holds: the number of fields in the document, a list of fields with term vector information, a list of pointers to the .tvf file. These are respectively stored as Vint(NumFields),<VInt>^{Vint},<VLong>^{VInt} in the format

    TVDVersion<NumFields,FieldNums,FieldPositions>^{NumDocs}

  - *.tvf* file is created when at least one field has term vector. It holds for each of those fields a list of terms, term frequencies for each term, and can also contain position and offset information. It has the same data types as .tvd and .tvx files. In our index, we have term vector stored in _0.tvd, _0.tvf and _0.tvx.

- **Deleted Documents**

  This file does not appear in our index, it is an optional file with the extension .**del**. It is created when a segment has deleted documents. It is a set of numbers of type Uint32, Bytes and VInt

The files we have seen are stored for each segment(partial index) in the index file. The next files are part of each index

- **Per Index-files**
  - **Segments file**

    This file stores active segments and segment information in a file called *segment_N* where N is the largest generation [50] or level. This file contains details about the norms(*NormGen*) and the file deletions (*DelGen*), it also holds the size of each segment(*SegSize*), the segment name(*SegName*) and a segment's name generator(*NameCounter*).
  - **compound files**

    A compound file is a files' holder stored in a *.cfx* file. It holds all per-segment files except the file deleted. If the value of *setUseCompoundFile(boolean)* is true,all stored fields values(.fnm,.fdx,.fdt,...) and term vectors(*tvx*,*tvd*,*tvf*) are stored in a single .cfx file
  - **Lock file**

    This file ensure that only one indexWriter is using the index directory at a giving time. The lock file is generally stored in the index directory in a file called *write.lock.*

After having seen the types of files contained in the index, it would be interesting to know which operations can be performed on the index and what changes it can bring on its structure.

5.3 Operations on the index

Starting this part with an illustration gives a better overview of the components involved in the index operations discussed in this section.

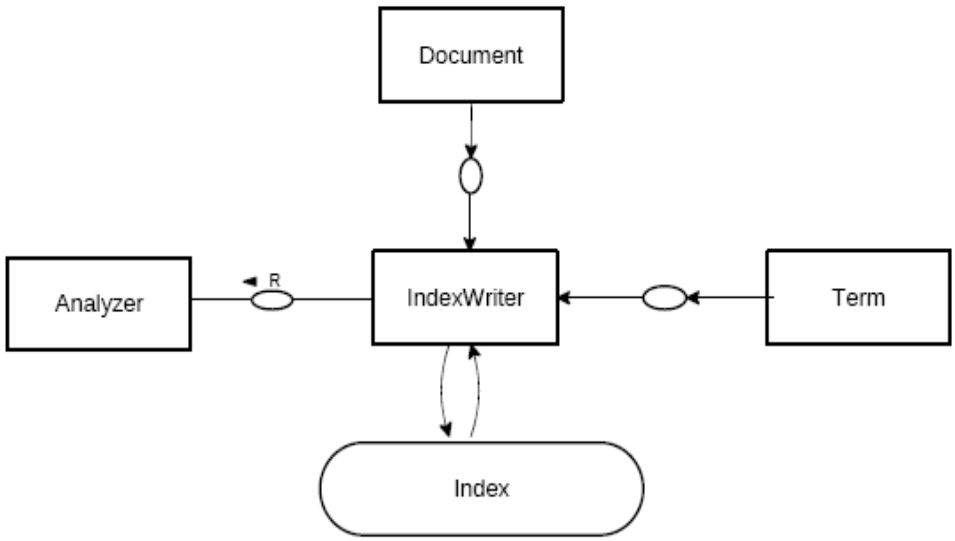


Figure 43: Add,delete and update operations

We focus on three main operations: Adding a new Lucene document to an index, deleting a document and updating the index when changes occurs.

5.3.4 Optimization

Generally, an optimization in the field of computer science is often associated to time, disk space or quality. The purpose of optimization is to save disk space and money, to keep more stuff in the memory, to increase the speed of data transfer from disk to memory, to speed the reading of large data. These causes are quite the same in the case of an inverted index.

The index optimization in information retrieval is the process of minimizing the time taken to read the index at search time. According to Manning, Raghavan and Schütze [4], optimization is done by merging multiple indexes together in a single index in order to reduce the number of index files on the disk. For instance, the merging process in section 5.1.3 shows, that the initial index files have been reduced from 14 to 4. This is to say, the inverted index algorithm carries an optimization process. Before investigating the optimization possibilities Lucene offers, let us see the reason and the circumstances that lead to an index optimization. An optimization is necessary because of the following reasons:

- make the index small for the memory and the disk: The terms dictionary should not be too large
- give postings lists a reduced disk space
- Reduce the time needed to access the disk to read
- Keep more significant postings lists in the memory.

With these in mind, here are the optimization possibilities that Lucene offers

(a) Controlling the Lucene buffer: mergefactor, minMergeDocs, maxMergeDocs.

The IndexWriter class supplies methods used to control the buffer necessary in the storing of a Lucene index. These methods are:

- *IndexWriter.SetMergeFactor(int) :* This method defines a number called *MergeFactor*. The *MergeFactor* defines the number of Lucene documents stored in the index. It also determines how many segments should be merged together into a single segment. By default the *MergeFactor* is 10. If the *MergeFactor* is smaller than 10, and if the index is not optimized, then search is fast and indexing is slow. On the contrary, when the MergeFactor is bigger than 10, if the index is optimized, then search is slow although indexing is fast. In fact, the higher the *MergeFactor* is, the more segments are produced. So, how big the *MergeFactor* should be depends on the user application .
- The *minMergeDocs* is the minimum number of merged documents that is buffer in the memory before the IndexWriter writes them on a disk. Its default value is 10, this should be enlarged to keep more documents in the memory for a single merge.
- Lucene defines a constant variable called *MaxMergeDocs*:Any segment, must not have more than *MaxMergeDocs* documents. If MaxMergeDocs = 2147483647 , then no more than 2147483647 documents can be kept in a segment. This number can also be increased, together with the *MergeFactor*. Reference of index optimization is given in the book “Lucene in action book”[2].

(b) First store the index in the memory

Instead of storing the index directly into the disk with FSDirectory, first store it in the RAM(memory). In the book “Lucene in action book”[2] it is recommended to follow these steps:

step1: create an FSDirectory-based index

step2: create a RAMDirectory-based index

step3: Flush everything buffered in RAM into the File system directory(FSDirectory)

(c) Work with multiple indexes in multi-threads

A user application can choose to use multiples RAMDirectory-based indexes parallely. The application(user) would call the *IndexWriter.addIndexes(readers)* method to merge multiple indexes into a single index . Readers are the seat of the IndexReaders reading those indexes. During this process, no document should be added or delete from the currently used index, otherwise the running thread will be interrupted.

(d) Use the IndexWriter.optmize() method

Lucene provide the *IndexWriter.optimize()* method to help reduce the resource usage of the index. Applying that method causes the merge of all segments in a new segment. As said before in section 5.2.2, this segment has the actual generation and contains the information of all the other segments. It is recommended to use this method at the end of the indexing process, in order to avoid a slowdown in the process.

The user can also contribute to the optimization of its index. This can be done by:

(e) Increasing the maximum number of open files on the computer

According to [2], the maximum number of open files on the computer can be increased. For example in unix, this can be done by setting the *ulimit*: *%ulimit -n* .This avoids the “two many open files” error, and assigns more space to the index.

(f) using multiple disks or multiple computers in parallely

This works like multiple indexes. The final index is stored in one disk or in one central computer. This one holds all the information about segments in other disks or computers.

(g) Limiting the number of fields to be indexed

The application programmer writing the code for indexing, can set the maximum number of field, that should be indexed. For instance, while creating the IndexWriter, the *maxFieldLength* should be defined like this:

int mfl = 50; then create the IndexWriter and set *IndexWriter.MaxFieldLength=mfl;*This means the number of terms in any field should not exceed 50 terms. If a field has more than 50 terms, it is truncated and stored in the index. The field is not truncated in the original document(Pdf,Html,Xml,...). Let us see how Lucene document can be deleted from a Lucene index.

5.3.5 Deletion of Index entries

As Recalled from chapter 5.2.2, while talking about index data, we saw that deleted files can be stored in a *.del* file in the index directory. Furthermore, for each indexed document, the deletion file carry a variable called Bits which is a one-Bit data. When Bits is set, the corresponding document is marked as deleted.

Lucene Document are deleted with the help of the *IndexReader deleteDocument(docNum)* method, where *docNum* is the document number. Once this method is executed, the document is marked as deleted. A deletion is effective when the IndexReader is closed. Closing an IndexReader causes a *commit* of changes into the index; this means, all modifications done on the index are stored.

Note that the removal can be done in different level. Calling *deleteDocument(docNum)* deletes a single document in the index. But the method deleteDocuments(Term) deletes all

documents holding the Term t. Nevertheless, all document deletions can be undone by calling the IndexReader *undeleteAll()* method. Another commonly-used operations is the update of an Index.

5.3.6 Index update

Once an Index is create, it can be modified. But an index update in Lucene is a deleted operation followed by an added operation[2]. We saw the deleted operation in 5.3.2, let us explain how a Lucene document is added to the index.

Adding a document to the index is the same as creating new Fields . As we saw in chapter 4, the Document.add(Fieldable) method of the Lucene Document class is used for that purpose. With this in mind we can see how to update an index.

Suppose we want to update the “author” field of our index and change it from “mark” to “Andrei Broder”. These are the steps to follow:

- open an Index Reader:

  IndexReader reader = IndexReader.open(FSDirectory.open(indexdir));

- delete all documents containing the term

  Term t = new Term("mark");

  reader.deleteDocuments(t);

- close the IndexReader

  reader.close();

- open IndexWriter and add all documents needed:

  Document doc = new Document();

  String newContent= "Andrei Brother" ;

  doc.add(newField("author",newContent,Field.Store.YES,Field.Index.ANALYZED,Field.TermVector.YES));

  w.addDocument(doc);

- close IndexWriter

  w.optimize();

  w.close();

To summarize, this chapter have given an understanding of what a Lucene index is. The demonstration of the stages of the index construction, highlighted the logic behind it. From the description of the information retrieval's indexing algorithms, one can state that the inverted index file algorithm is the same as the Lucene indexing algorithm, it is the fastest of all the three. A Lucene index is also stored normally like all common file, some of its file formate and data types are commonly used like char, integer,Byte and Long, others are special data formate, that are made up of byte or characters. These are Vint,UInt32 and Uint64. The knowledge about index optimization, update, and deletion is useful for user applications, who want to have efficient and effective indexing and search. With that known, we can see how Lucene searches into the index.

chapter 6 Searching in Lucene index

Consider a given user query *q* and a Lucene index stored in the directory as *index*. The purpose of this chapter, is to demonstrate how *q* is found in index. In the introduction, an overview of the search components is given, with focus on their structures. Afterwards, the Lucene query Language is study for some query types. Then, the way a query is found in an index is demonstrate d. Finally, the computation of result scoring is explained with a focus on the retrieval methods used by Lucene.

6.1 Components structure

In a Lucene based search application, the major components involved are the Query, the QueryParser, IndexSearcher, Filter and Analyzer. In section 3.1, sketches of the QueryParser 's and *IndexSearcher's* architecture have been made; combining those two, we obtain the FMC Block diagram below, that represents components involved in the Lucene index search process.

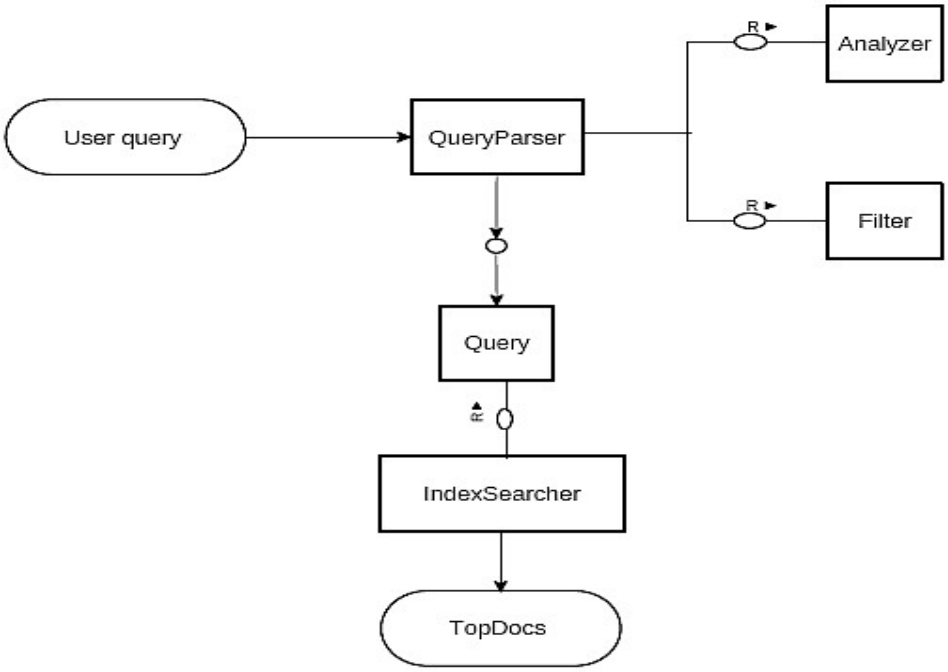


Figure 44: Searching a user query

6.1.4 QueryParser and Query

Recall from sections 2.2.1 and 4.3.2, that a Document parser is responsible for the extraction of text from original files and for the creation of Lucene document with the help of the *DocumentHandler* interface. Likewise, The Lucene QueryParser is responsible for transforming a given user query into a Query object. Unlike the Document parser which is supplied by the user application, the QueryParser is a Class in the Lucene core,which generated by the *Java compiler compiler*(*Javacc*). In section 3.1, we saw the architecture of the QueryParser. In section 6.2, we are going deep into the parsing methods used by the QueryParser.

A (Lucene)Query is a set of one or more terms , symbols and operators. A term is a word like “web”, “spider”, and an operator is one of the boolean operators AND, OR, NOT . A symbol is a character like * or ~ for a wildcardQuery, it can also be a double quotes enclosing the query like “web search ”. Queries are results of the query parsing process. Lucene 3.0.1 Library supplies the user with eleven types of queries: TermQuery, MultitermQuery, BooleanQuery, WildcardQuery, PhraseQuery, PrefixQuery, MultiphraseQuery, FuzzyQuery, TermRangeQuery, NumericRangeQuery and SpanQuery. Four of those Queries were presented in section 3.3.1 ; an example query for each of them was used to sketch the architecture. We notice that each of those query types has a tree structure and that they all hold a TermQuery. Therefore, the TermQuery is the basis for other query type like the BooleanQuery, PhraseQuery,WildcardQuery and FuzzyQuery. More precisely, each query type contains a Term and a boost, which builds a TermQuery . A Term holds the field name and the text of the field value, and boost is the boost factor described in section 4.2.1.c.

Once a Query object is created, it is passed to the IndexSearcher.

6.1.5 IndexSearcher

The retrieval of documents matching the query object is triggered by the IndexSearcher. By means of its architecture described in 3.1, we found that the IndexSearcher works together with the QueryParser and the IndexReader. The first one provides the IndexSearcher with valid Lucene query object, this is a query that respects the Lucene query language specifications. The second one provides a reader, which accesses the index in a read-only way. Once the index is opened, another Lucene component reads the postings lists out of it and returns it back to the IndexSearcher.

Once the IndexSearcher has retrieved postings matching the query, it returns the top hits for that query called *TopDocs*, this is a list of document numbers and/or score for each document. Generally, a user application gathers the resulting document numbers in an array and call the *doc(int i)* method of the IndexSearcher class to return the stored fields of document i.

In fact the IndexSearcher stands at the beginning and at the end of the retrieval of terms in the index corresponding to the query terms. Later in section 6.3 we will enter deeply into the search mechanisms used by IndexSearcher. With attention to the following points:

- How are the posting lists supplied to the IndexSearcher, which components are responsible for that?
- How does the IndexSearcher know that a term in the index is identical or similar to a query term, in other words, how is the similarity between a query term and an index term calculated? And who did it?
- How are terms displayed to the user?

But before answering to those questions, let's see the last components of the search process.

6.1.6 Filter and Analyzer

Filter and Analyzer are components used to purge useless contents, or to sort results, or to set restrictions. It belongs to the user application to decide what is considered as useless, and to select results to display to the user. Lucene helps the user by supplying classes and methods for filtering results and analyzing user query .

6.1.6.a)Filtering a Query object

The **Filter** in the search context is not the same type of filter used during indexing. This filter is used to select which documents should be allowed in search results. The Lucene library supply six Filters, that can be used to restrict the search results, these are: The CachingWrapperFilter, the *FieldCacheRangeFilter*, the *FieldCacheTermFilter*, the *MultiTermQueryWrapperFilter*, the *QuerryWrapperFilter* and the *SpanFilter*.

Each of those Filters extends the basis Filter class and each of them, except the SpanFilter, overwrite the *getDocIdSet*(*IndexReader*) to match their own filtering purposes. As example, here are filtering capabilities of some of those filters:

CachingWrapperFilter: This class allows other filters to realize only the filtering job and then uses methods given by this class to cache results of their filtering. So the user application just needs to provide this class with a filter which results should be cached. The CachingWrapperFilter holds a so called *DeletesMode* to specify how deletions should be handle when reopening the index. An application can choose to ignore deletions(*DeletesMode*=IGNORE), in this case, all documents Ids are handled the same way. This setting leads to an invalid query because, deleted documents can be returned as search results in case the Query does not exclude them from the cached filter. Thus, it is recommended to set *DeletesMode*=RECACHE or *DeletesMode*=DYNAMIC to automatically update the cached filter and remove deletions.

QueryWrapperFilter: This filter is used to restrain results to those which matches a given Query. As example we try to filter documents containing the query term “spider”, within the search results for the query “web,search”. four Pdf files are concerned by the search: *(1).pdf, (2).pdf, broder_websearch.pdf* and *Search Engine history.pdf*. The Pdf files have been previously parsed with PdfBox and indexed.

without filtering we just need to implement search as done in 3.3, these are the results display:

```
query= contents:web contents:search
found: 4
[2] (2).pdf(0.49793825)
/home/josiane/Documents/indextest/(2).pdf

[3] Search Engine History.pdf(0.18120934)
/home/josiane/Documents/indextest/Search Engine History.pdf

[1] broder_websearch.pdf(0.15097556)
/home/josiane/Documents/indextest/broder_websearch.pdf

[0] (1).pdf(0.074467406)
/home/josiane/Documents/indextest/(1).pdf
```

To add a QueryWrapperFilter to the search, first create an instance of that filter:

```
Query q2 = p.parse("spider");
QueryWrapperFilter filter = new QueryWrapperFilter(q2);
```

Then, call the IndexSearcher's search method with the created filter:

```
IndexReader r = IndexReader.open(FSDirectory.open(index),true);
IndexSearcher searcher = new IndexSearcher(r);
topdoc = searcher.search(q,filter,5);
```

Although the query retrieved is the same,the resulting documents returned are quite different from the previous one:

```
query= contents:web contents:search
found: 2
[2] (2).pdf(0.49793825)
/home/josiane/Documents/indextest/(2).pdf

[3] Search Engine History.pdf(0.18120934)
/home/josiane/Documents/indextest/Search Engine History.pdf
```

In fact this type of filter is useful to sort documents by modification date for example, or by theme. It is commonly used with a TermRangeQuery, which is a query matching a range of terms.

6.1.6.b)Query analysis

Analysis provides the mechanism to break down Lucene documents into tokens, providing the IndexWriter with tokens' terms. Similarly, the QueryParser uses the analyzer to find terms in the query text. It is recommended to use the same analyzer for indexing and search, but this choice depends on the user application. There are cases, where different analyzers would be necessary. An example of such a case is given in the 'Lucene in action' book[2] (Pages 129-134). In their example, a SynonymAnalyzer is wrapped with a Filter and used on the text *"The quick Brown fox jumps over the lazy dogs"*, the extracted tokens are indexed and buffered into a stack. When using the same SynonymAnalyzer to also analyze the user query q="fox jumps" , They notice that the query fails to match the document. But, when they use the *StandardAnalyzer* on the query, they could find the expected match.

In fact, Query Analysis returns token to the IndexSearcher. Note that there is a difference between the token return by the Query Parser, and the Token return by the Documents Analyzer:

- The *QueryParser* Token describes the user input query for search. With the help of the Token specifications in the Lucene library, the table below depicts what a (*QueryParser*)Token contains:

BeginColumn	Column number of the first character of this token
BeginLine	Line number of the first character of this token
EndColumn	Column number of the last character of this token
EndLine	Line number of the last character of this token
image	String image of this token. The String Image is defined in the *QueryParserConstants* class as the literal value of a token.
kind	An integer describing the kind of this token

next	A reference to the next regular token from the user input query
SpecialToken	This is used to access special tokens that occurs before this token

Later in section 6.2.2, examples of query token help understanding the internal query structure.

- The Analysis Tokens are produced after the tokenizing and Token filtering process. Recall from section 3.1, that a Token is a set of following attributes: Offset Attribute, Term Attribute, Type Attribute, PositionIncrement Attribute and Payload Attribute

With the knowledge of the architecture and the representation of some Query objects given in section 3.3.1, for instance the Boolean query: “web AND search” is represented by the QueryParser as “*+contents:web+contents:search”*, the next section will supply more information about the mechanism used by the QueryParser to recognize the type of a user query, and to transform it into a Query object.

6.2 Query Language

We start this part with an overview of the query languages available in information retrieval. Afterward, the mechanism used in Lucene Query language to transform query will be brought out.

6.2.4 Query Languages in information retrieval

In the book 'Modern information retrieval',[33] query languages determine which query can be formulated. Thus they distinguish keyword-based query languages from query language based on pattern matching.

Keyword-based query language:

is the simplest form of query representation. Naturally, a user query comprise one or more keywords, these are basis queries of the Keyword-based query language:

Single-word queries:the simplest and intuitive query, is made of a single word. A word is simply a set of letters of the alphabet with separators like hyphen

context queries:In Addition to the single-word query, the context query includes the possibility to search the word in a context. This query is either a phrase , or a proximity . A phrase is a sequence of words like 'web search'. A Proximity query is a kind of phrase query with a maximum allowed distance between the sequence of words or phrases.

Boolean queries: is a combination of keywords query with boolean operators like (Web AND search)

natural language queries: Boolean queries are simplified concept of natural language queries, moreover in natural language, documents and queries are considered as vectors of 'term weights'. Searching for a query means retrieving all documents vectors close to the query vector. In this language model, a document can also be used as query, since it is also considered as vector. This technique is used for relevance feedback.

The second query language is the one based on pattern matching

Query language based on Pattern matching:

“A pattern is a set of syntactic features that occur in a text segment”[45].In this query language, the query is a pattern or a set of patterns, that all matching document should contain. Some of those patterns are:

33 R.Baeza-Yates,B.Ribeiro-Neto,1999 [45],chapter 4

- **Words** like '*spider*', this is a word in the document
- **Prefixes** for example '*inform*' , '*eng*'. All words starting with those prefixes would be retrieved.
- **Suffixes** : This is a pattern to match termination of words . Example: 'ider' 'rieval'
- **Substrings** : This is a pattern to be found within a word. Example: 'web search'
- **Ranges**: this is a pattern which holds a pair of strings. It should match any word lying between the two strings in lexicographic order.

  For instance: considering the document (2).pdf containing the text: **"Search engines uses "spider" which search (or spider) the web for information."** The range between '*engines'* and '*information*' will retrieve: '*information'*, '*spider'* and '*search'.*
- **Allowing errors** :this pattern matches a word with a certain tolerance. In fact, each word similar to this pattern are retrieved. This pattern include the Levenshtein distance[34], which calculates the minimum number of character insertion, deletion and replacement necessary to make the word in the document equal to the query. For example the Levenshtein distance between '*search*' and '*searches*' is 2. The query pattern muss specify how big this distance must be.
- **Regular expression**: this pattern matches all word that respect the regular expression given by the query. For instance (exp1* exp2) matches a 0 to more repetitions of the expression exp1 and all occurrences of exp2.
- **Extended pattern**: These are kinds of user friendly regular pattern.

With these concepts of the query language in information retrieval, let's see which type of query language(s) Lucene uses for its query parsing.

6.2.5 Lucene Query Language

6.2.5.a) The grammatic

The Lucene Query language is based on a particular grammatic. This one is defined in a BNF in a Javacc generated class called QueryParser.jj. Recall from chapter 3.1 that, a user query is a set of clauses and a clause contains one or more terms prefixed by a plus(+) or a minus(-) sign, it can also contain another query. The query syntax differs according to the kind of query, but each query type use the same grammatic.

The following, is the definition of a query in BNF according to the Lucene query grammar:

```
 Query  ::= ( Clause )*
Clause ::= ["+", "-"] [<TERM> ":"] ( <TERM> | "(" Query ")" )
```

The meaning of a clause is defined by Lucene QueryParser as follow:

- a clause can start with "+" or "-": A plus(+) means the terms after the "+" is required in the document. A minus(-) means terms after it must not appear in the document.
- A clause can also start by a term ,which matches the regular expression: <TERM>, followed by a colon (":"). This term describes the field to be searched for. This is to say, support field-based search. When no field is specified, then the default field is used for every user query. In our example application , the default field is "contents"
- A clause can contain a single term, matching the regular expression <TERM> or it

34 The Levenshtein distance explained and demonstrate here http://www.merriampark.com/ld.htm

can be a nested query enclose with parentheses, this is represented by the regular expression “(“Query”)”. The Query itself being a set of one or more clauses.

From the previous section, we know, that a query(clauses) is made up of Tokens build during Query parsing. A specification of the Token grammatic is also given in the QueryParser.jj class this is shown in the table below:

```
TOKEN : {
  <AND:       ("AND" | "&&") >
| <OR:        ("OR" | "||") >
| <NOT:       ("NOT" | "!") >
| <PLUS:      "+" >
| <MINUS:     "-" >
| <LPAREN:    "(" >
| <RPAREN:    ")" >
| <COLON:     ":" >
| <STAR:      "*" >
| <CARAT:     "^" > : Boost
| <QUOTED:     "\"" (<_QUOTED_CHAR>)* "\"">
| <TERM:      <_TERM_START_CHAR> (<_TERM_CHAR>)*  >
| <FUZZY_SLOP:     "~" ( (<_NUM_CHAR>)+ ( "." (<_NUM_CHAR>)+ )? )? >
| <PREFIXTERM:  ("*") | ( <_TERM_START_CHAR> (<_TERM_CHAR>)* "*" ) >
| <WILDTERM:  (<_TERM_START_CHAR> | [ "*", "?" ]) (<_TERM_CHAR> | ( [ "*", "?" ] ))* >
| <REGEXPTERM: "/" (~[ "/" ] | "\\/" )* "/" >
| <RANGEIN_START: "[" > : Range
| <RANGEEX_START: "{" > : Range
}
```

Table 11: Syntax of the Lucene Query token

Lucene recognizes a token in a user query, through this definition. The Left part of each Token components is the name of this components and the right part represent the possible values it can have. For instance <AND: (“AND”|”&&”)> means the AND operator can be written as *AND* or as &&. Another example is <CARAT. “^”:Boost> this means each time the QueryParser encounter the sign ^ then it takes the following term usually a number, as the boost factor for that query term. A Term in a query Token is defined as

<TERM: <_TERM_START_CHAR> (<_TERM_CHAR>)* >. The term is prefixed by a character(_TERM_STAR_CHAR) and a term text which is a set of characters. There are some restrictions on the start character of a term, namely: A term should not start with an escape character, or with any type of brackets , or with a Unicode[35] character. No punctuation is allowed in front of a term, as well as special characters like ~,^,\ or \\. In fact the term should generally start with a character from the alphabet.

With this in mind, let's see which mechanisms the QueryParser uses to transform the user query into a Lucene Query, with respect to the Query Grammatic.

35 Unicode standard version 6.0[Unicode Inc. ; 1991-2010] available at http://www.unicode.org/versions/Unicode6.0.0/

6.2.5.b)The query parsing mechanism

The most important QueryParser method is parse(String), which launches the parsing mechanism. Once *QueryParser.parse(String uq)* is called for a given user query *uq*, the following implementation is done:

- QueryParser first creates an empty array list, where the clauses will be pushed into:

  List<BooleanClause> clauses = new ArrayList<BooleanClause>();
  A Boolean clause is a clause that contains a query and its occurrence. The query is represented as a field, and the occurrence is a constant value that tells the query parser how the query' clauses must appear in the document. More precisely the occurrence has three values :

  MUST: means that clauses must appear in the document, therefore the BooleanClause class returns a "+ " operator

  SHOULD: means that clauses can appear or not. The returned value is empty: " "

  MUST_NOT: means the clauses must not be in the document. The returned operator is "-". To display the value of a Lucene Query, use the Query.toString() method of the Lucene Query library.

- The user query is then represented as a field(name-value pair). If no field name is given, then the default field name is considered, and the given string will be the value. With this representation of user query as field, the QueryParser can create clauses from it.

- To do this, the QueryParser uses the *Clause(Field)* method like this:

  `Query q = Clause(field);`. This one consumes each term of the field's query using Javacc jj_consume() method,and transform it according to patterns define by regular expression in the QueryParserConstants class .

- A clause can be a set of terms this is the case for a term query or a boolean query. But it can also hold a star(*) or a Wild term defined as

```
<WILDTERM: (<_TERM_START_CHAR> | [ "*", "?" ]) (<_TERM_CHAR> | ( [ "*", "?" ] ))* >
```

Table 12: A wildcard term

This regular expression represent a set of characters or a star(*), or a question mark(?) preceding another set of character with wild cards * or ?. An example of wild car query is given in the next section. Once the query match this regular expression, the wildcard query search is set up. In this case, the *getWildcardQuery(qfield, termStr)*method is called with the token's term image and the query field. This method causes the implementation of a wildcard search query. The termStr is a String that represents a the text in the query.

- A Clause can also contain a so called *FuzzySlop* which is a A tilde followed by a floating point number like **~0.5**. A *FuzzySlop* is define in the query grammar as:

```
<FUZZY_SLOP:"~" ( (<_NUM_CHAR>)+ ( "." (<_NUM_CHAR>)+ )? )? >
<#_NUM_CHAR:["0"-"9"] >
```

Table 13: A Fuzzy Slop

In the example, 0 and 5 are the NUM_CHAR and ~ is the prefix term for each fuzzy query. The *getFuzzyQuery(field,termStr,minSimilarity)* method is called to create such a Fuzzy Query. Field is the query field, termStr is the term of the token, used to build the query, and minSimilarity is a number between 0 and 1, which is used to calculate the similarity between query term and terms in the documents. The getFuzzyQuery methods returns a query. Other query processing methods are available in the Lucene library.

- As result of the Clause(field) method call, clauses are created and use to fill in the Lucene Query .To add clauses to the Query the QueryParser calls

  addClause(clauses, CONJ_NONE, mods, q);

  the variable clauses is the empty array list of Boolean clauses or it is a Fuzzy clause, or a Wildcard clause. CONJ_NONE is one of the conjunctions AND or OR; mods is a modifier like PLUS, MINUS, OR NOT. All these were define in the section before, as part of the query Token . The *addClause(...)* method makes a term required if it is followed by the conjunction AND, by doing this, it assign the Boolean clause (clauses) the occurrence value MUST, likewise a term preceding the OR is made optional and its occurrence value is SHOULD unless the term is preceded by a NOT operator, in this case it is made prohibited and the corresponding occurrence is MUST_NOT. This processing is repeated until there is no more term in the clauses. After this method, the QueryParser returns the created Lucene query to the IndexSearcher. Before going to the search process, let's give some examples of query parsing for different kind of query.

6.2.6 TermQuery

A single word or term forms a TermQuery. An example was given in chapter3.1. While parsing the TermQuery q="web" we observe, that the query parser transform it into :

contents:web, following the query grammar, this query is written as:

q=[<TERM> ":"] <TERM> where *[<TERM>:] =[contents:].* Since no field's name is specified, contents is the default field name, the second term is the value of that field : *<TERM&gt n= web*. The resulting Lucene query is therefore **q=*contents:web***. Other query types are based on this structure.

6.2.7 BooleanQuery

The boolean query includes at least two TermQueries with the example query in section 3.1 let's see how this query is parsed. q= web AND search , According to the QueryParserBase class. The first term is parsed as MUST, and the returned operator is "+" so we obtain the parsed query as **q= +contents:web +contents:search ,** each term is associated to its field's name . The tree representation of this query is given in section 3.3.1. If the first term is prohibited , let's say we have q= NOT web AND search. Then, the resulting query parsed would be **q= -contents:web +contents:search**, this means match all documents where web MUST_NOT occur and search MUST occur.

6.2.8 FuzzyQuery

As an example suppose the query q= literature~ .The Queryparser recognizes the tilde at the end of the term, it thus understand it as the start of a fuzzy_slop. And call the getFuzzyQuery(contents,literature,0.5) method, to transform this into **q=contents:literature~0.5.**

The field name is contents, and the term query concern is literature. Since no similarity is given, the default one will be consider this is 0.5.

6.2.9 WildcardQuery

By the same token, consider an example wildcard query q=s*e?, according to the QueryParser class, no query term should start or contain characters like star(*) or question mark(?), if this is the case, then the query is consider as a wildcard query. Therefore, the corresponding method for processing wildcard queries is called. For this particular example a new query object would be created by getWildcardQuery("contents","s*e?"). This call returns **q = contents:s*e?.**

All things considered, one can say that Lucene include both keyword-based and pattern matching languages. The term- as well as the Boolean Query has respectively the same language concept as the single-word language and the boolean query language defined in Information retrieval. In addition, the Wildcard query and the Fuzzy query are based on regular expression pattern matching. With this knowledge of the query construction, let's move to the query retrieval process.

6.3 Using the Index for Search

We know from the previous chapters, that the IndexReader accesses the Index in a read only way and returns information about fields. In this part we want to demonstrate how documents are found in the index to match a given Lucene query. Unlike the chapter 5.1.2 about the index construction, we will start this part with the algorithm implemented by the index search.

6.3.4 The index search algorithm

The index search algorithm was describe in a paper written by Doug Cutting(the Lucene founder) and Jan O. Pedersen [9].

Considering a given query *q* , let scores be an array of length *N*,that will contain the result. Let *queue* be a queue containing the id (*id*) and the scores(*scores[id]*) of the matching documents, let k be the maximal number of results to display to the user. Here goes the index search algorithm :

```
inverted_search(query) =
   scores = an array of length N initialized to zero
   queue = an empty queue of (id,score) pairs ordered by ascending score
   for (t ∈ q) ; iterate over terms in query
     ps = postings(t) ; a posting stream for term t
     while (p = nextposting (ps)) ; iterate over postings
       id = p.id, weight = p.weight
       scores[id] = scores[id] + q_t * weight
       if (length(queue)=k + 1) pop(queue)
       insert((id,scores[id]),queue)
     end
   end
   pop(queue)
   return(the contents of queue in descending order)
end
```

Figure 45: The inverted index search algorithm(Source: [D.Cutting,O.Pedersen, 1997])

For each term t in a query q, the p postings list of that term, are reading sequentially from the inverted index. For each document in the postings, the document id and the weight of that term is store. Then the partial score (*score[id]*)for the document is updated and stored together with the query into the queue. A partial score is calculated each time the term occurs in the document, therefore, *scores[id]=scores[id]+q_t*weight* for the document number id is increased .Once the size of the queue reaches k, the old content of the queue is pop out, and the current (id,score[id]) pairs are pushed in. At the end of the process, the queue is return to the user in a descending order of id. According to [9] this algorithm reduce the disk access because only the postings of the query terms are reading. If there are p postings , then the algorithm has a complexity of $O(p)$. But the amount of disk need for partial scores is $O(N)$ where N is the length o the score array. This depend on the number of results the user wants to display. In short, the Lucene index search algorithm can be divided into three steps:

Step1 : Accessing the index and find posting list for a given query term. This process is demonstrated in section 6.3.2

Step2: iterating over postings list and find the corresponding documents. This step is handled in sections 6.3.3 and 6.3.4

Step3: The last step is already done within step two, but here, we will find out how scores are computed. This part is handled in section 6.4.

We know from the previous chapters, that the Lucene IndexReader accesses the Index in a read only way and returns informations about fields. The search process starts with the creation of an indexSearcher, Lucene provides four different implementations of IndexSearcher, one of them is *IndexSearcher(IndexReader r)*. An IndexReader provides the searcher with the index where search will occur. Therefore, the IndexSearcher uses one of the search methods supplied by the abstract class Searcher,to find results for a given query. Obviously, all those search methods turn into a single one, namely the

search (weight,filter,collector) method Where: *weight* is the query weight, *filter* is a query filter and collector is the set of results gathered for the given user query.

In essence, except the *Searcher*,*Collector*,*weight* and *query filter*, the other Lucene components involved in the search process are: *DocsEnum*, *TopScoreDocCollector*,*TermScorer*,*Similarity* and *TopDocs*. For a better understanding of this algorithm, here is a step-by-step explanation about which responsibility each component has in the processing.

6.3.5 How the Index is used for search

Let us sketch the dependencies between components involved in the index access.

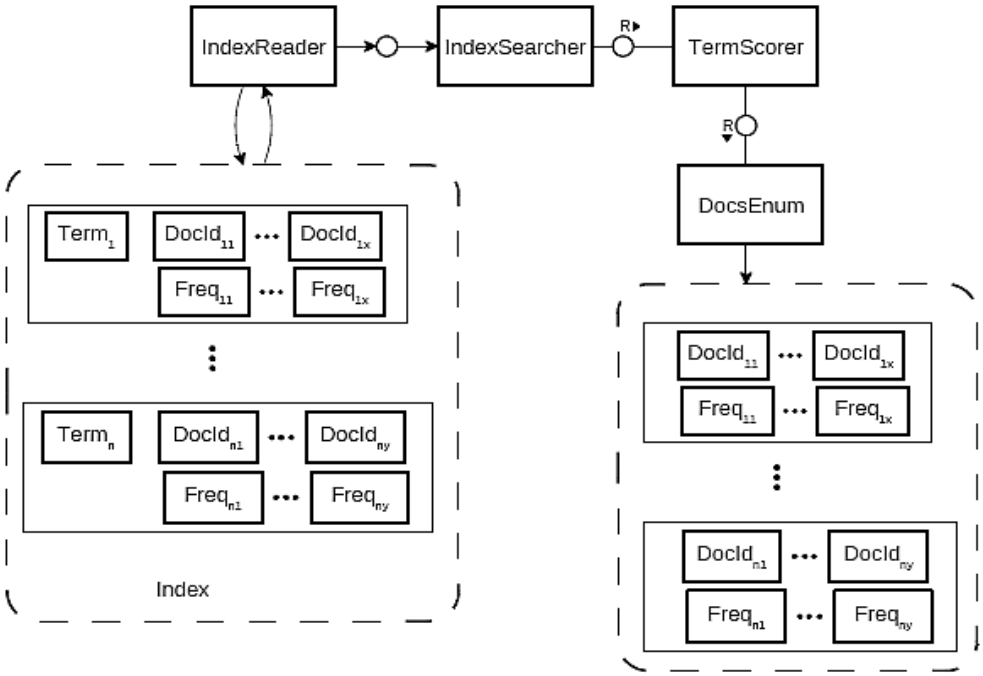


Figure 46: IndexSearcher using the provided IndexReader to access the Index, call a TermScorer, which uses DocsEnum to iterate over documents matching the term

A user provided IndexReader , gives a reference to the index to be used by the search application, to create an IndexSearcher.

Within the IndexSearcher.search(Weight weight, Filter filter, Collector collector) method the *scorer.score(collector)* calls, give the processing to the Scorer and supply him with a collector . The Scorer is the based class for scoring different kinds of query, for instance, the BooleanScorer for Boolean query, TermScorer for term query, Phrase scorer for phrase query[36] and so on. The responsibility of the scorer, is first of all, to request from an iterator the whole documents matching the provided query (the query is represented by its weight). For this purpose, the TermScorer buffer those result using the *DocsEnums.read(int[],int[])*.

A *DocsEnum* is a class of the Index package, that causes an empty or an old buffer to be filled with current or new documents Ids and frequencies stored in the index. In fact, DocsEnum browse documents number in the memory, and returns the count read to the TermScorer. A document number , and its frequency are represented as `IntsRef` which is a piece of three values : (int[],int,int), this representation is implemented in the Lucene util package. If DocsEnum returns a count=0, then the end is reached, there is no more documents in the index.

36 Another query type that matches documents containing a particular sequence of terms. Available in the Lucene library at http://lucene.apache.org/java/3_0_1/api/core/org/apache/lucene/search/PhraseQuery.html

With this count, the TermScorer can initialize the maximum size of the documents collection : pointerMax = docsEnum.read(); then use its *nextDoc()* method to advance to the next document matching the query . Before the TermScorer goes to the first document, it first set the nextDoc() to -1, this means, there is nothing in the results buffer.

We have just demonstrated how the IndexSearcher accesses the Lucene documents store in the index. To sum up, the IndexSearcher receive, the path to the index, throw the IndexReader, with this information, the IndexSearcher creates a Scorer, providing him the query weight, the filter chosen by the user, an a collector to collect result. Then, the scorer while calling nextDoc() use the *DocsEnum.read(int[],int[])* to get the Lucene documents numbers and frequencies. Recalling the first step of the index search algorithm, we notice, that the *TermScorer.nextDoc()* iterates over documents matching the query, in other words it browse the posting list of a given query, this is the *nextPosting(ps)* call in the algorithm. Each time nextDoc() is called, the next matching document is read from the index through the DocsEnum.read(int[],int[]). But how is the matching document found ? Here goes the search within postings list.

6.3.6 Search within the Terms- dictionary

Let's assume we want to search for the query q = "**web".** In this part, we would see the mechanism used to retrieve the terms in the index, that matches q. For better understanding, here is an overview of the concept behind term retrieval in Lucene.

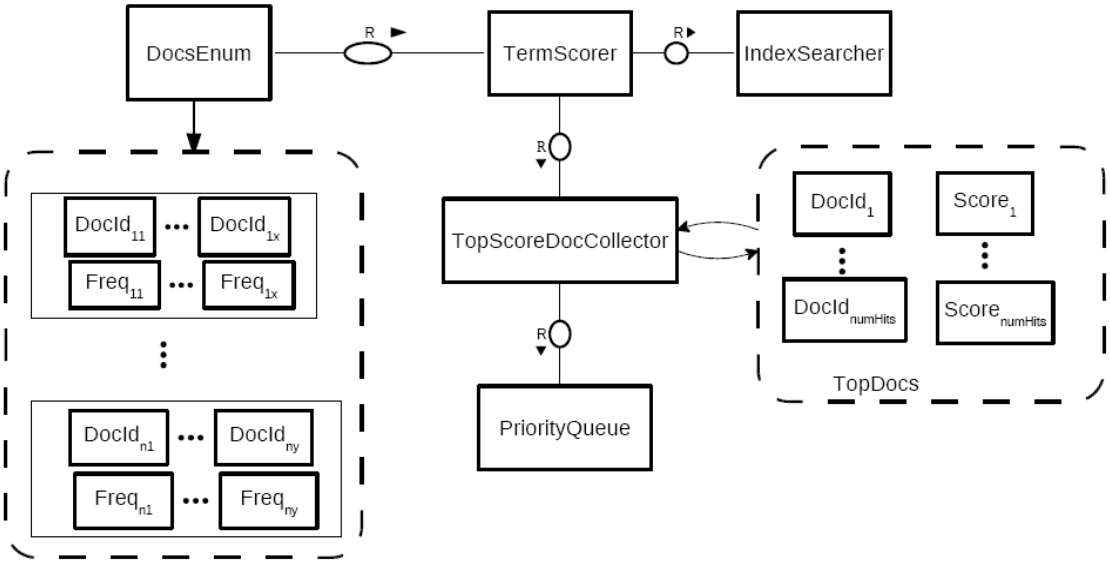


Figure 47: Searching within the terms dictionary

The first observation that can be done, is that TermScorer, IndexSearcher and DocsEnum, are the most important actors of the process. As recall from the previous section, where the relationship between Index and search was demonstrated, we can say that, the IndexSearcher releases the search mechanism by requesting Top documents from the TermScorer. In its turn, the TermScorer receive through the DocsEnum, the postings list of all terms for each stored field. The TermScorer can iterates over the provided list using its *nextDoc()*method. For each document number in the index, TermScorer compute the score of that document, to see if one or more terms match the query; for this, it uses its score (collect, int,int) method :

```
protected boolean score(Collector c, int end, int firstDocID) throws IOException {
  c.setScorer(this);
  while (doc < end) {                           // for docs in window
    c.collect(doc);                     // collect score
    if (++pointer >= pointerMax) {
      refillBuffer();
      if (pointerMax != 0) {
        pointer = 0;
      } else {
        doc = NO_MORE_DOCS;                 // set to sentinel value
        return false;
      }
    }
    doc = docs[pointer];
    freq = freqs[pointer];
  }
  return true;
}
```

Table 14: TermScorer calling the scorer.score(c,end,firstDocID) (source: Lucene svn at http://svn.apache.org/repos/asf/lucene/dev/trunk/lucene/src/java/org/apache/lucene/search/TermScorer.java)

The TermScorer define the DocsEnum's count as the maximum value, that the documents ' pointer should reach. Each time the pointer is incremented, the next matching document, and its frequency are pushed in to the results buffer using refillBuffer() method. This is done until the buffer reach its end, in other words, when pointer =DocsEnum.read() = 0 . Each time a matches is found, it is gather in a collector using collector.collect() method below:

```
public void collect(int doc) throws IOException {
    float score = scorer.score();

    // This collector cannot handle NaN
    assert !Float.isNaN(score);

    totalHits++;
    doc += docBase;
    if (score < pqTop.score || (score == pqTop.score && doc > pqTop.doc)) {
      return;
    }
    pqTop.doc = doc;
    pqTop.score = score;
    pqTop = pq.updateTop();
  }
```

Table 15: The Lucene TopScoreDocCollector collect(doc) method gather the top-scoring hits for a document with id = doc. The priority queue pq maintains a partial ordering of docIDs . Code available at http://svn.apache.org/repos/asf/lucene/dev/trunk/lucene/src/java/org/apache/lucene/search/TopScoreDocCollector.java

for each Lucene Document (doc), the collector calls the TermScorer's method score(), to calculates the score of that document for the given term query. Once the score is returned, the collector uses a PriorityQueue(pq) to sort the document partial scores in increasing order

of docIDs. Each new docID is compare to an old one. If the docID at the top of the queue is smaller than the current docID, or if the score of the current document is smaller than or equal to the score of the document at the top of the queue, then the PriorityQueue should update the queue and insert the new document id and its score at the top. Obviously, the PriorityQueue implements the following functions of the index search algorithm: insert((id,score[id]),queue) of. Now that we have an idea of how the partial scores are buffer, let's see how the score is computed for a that single term query.

To compute scores, TopScorer uses its score() method like this:

```
  public float score() {
    assert doc != NO_MORE_DOCS;
    float raw =                           // compute tf(f)*weight
      freq < SCORE_CACHE_SIZE                    // check cache
      ? scoreCache[freq]                  // cache hit
      : getSimilarity().tf(freq)*weightValue;       // cache miss

    return norms == null ? raw : raw * getSimilarity().decodeNormValue(norms[doc]); // normalize for
field
  }
```

Table 16: Compute the score of a document

The TopSCorer define a default cache size of 32 for the score of a term for instance t= “web”. Since the cache is not full, the partial score are buffer into. A score for a single term in a document is:

*scoreCache[freq] = getSimilarity().tf(freq) * weightValue;* where:

getSimilarity is the similarity between the document' s terms and query term .We get into the computation of the similarity in section 6.4

tf(freq) is the frequency of a term within a document.

Weightvalue is the query weight.

In fact the score() method is call for each term stored in the index, to calculate the similarity to the term query. Each similar term has it's score stored in a priority queue. This computation is the same as the assignment :*scores[id]=scores[id]+q_t*weight* done in the search algorithm.

Each time the queue is full, its content is pop out and insert by the Collector in a result buffer as TopDocs. TopDocs returns the array of documents number with their corresponding scores. Note that the size of the result buffer is defined by the number of hits found for the query (numHits).

Section 6.4 goes deep into the computation of score[id] for a term in the query. But first of all let's take a look at other search functionalities.

6.3.7 Special search: Numbers, date and synonym

User application like web search engine or library search engine need special search implementations. For instance, searching for the year of publication of a book, retrieving the cheapest price of a product in an on-line shop search engine , or looking for synonyms of words. For those type of search, Lucene proposes different solutions:

Searching for numbers and dates:

In chapter4 we saw that a Field's value can be a number, and that Lucene creates a NumericField to store such field. Because Lucene supports the Java primitive types int,long,

double and float, any numeric Field's value is converted into one of those type, like Date field are converted into long value, prices or floating point number may be converted to an integer or a float value. Those fields can be added to the index in this way:

```
...
doc.add(new NumericField("date of creation").setIntValue(cal.get(Calendar.DAY_OF_MONTH)));
...
w.addDocument(doc);
```

Note that Lucene represent each numeric value as a *trie* structure. In this structure, each term is logically assigned a precision representation of its value, this is called Precision step. A Precision step is measure in bits. The default value for each data types is 4, but it is mostly 64 bits for long or double or 32 bits for int and float value . Once the numeric field is indexed, one can search for its value by using a NumericRangeQuery .This is a type of Lucene query that matches numeric values within a specified range. This is an example of creation of a NumericRangeQuery:

```
NumericRangeQuery<Integer> numericquery = NumericRangeQuery.newIntRange("date of creation", 4, null, null, false, false);
```

The syntax used is
NumericRangeQuery.NewIntRange(field,PrecisionStep,min,max,minInclusive,maxInclusive);

In our example, the precision step is 4 which is the default one. Min and max are set to null, this mean half setting range like <= or >= are considered. The two boolean values *minInclusve* and *maxInclusive* bounds the results between min and max, we set them to false to obtain all the matching documents.

Synonym retrieval:

In the core of the current Lucene Version(3.0.2), there is a package named lucene.wordnet[37], which provides classes and method to implement a synonym search. Lucene uses synonyms provided by *Wordnet*[52]. Wordnet is a large lexical database of English words group in sets of cognitive synonyms called *synset*, each set of synonym express a distinct concept. Lucene uses the synset provided by the Wordnet prolog package Wnprolog [38]to create an index suitable for synonym search and query expansion. To perform that, Lucene creates a Syns2Index class that transform wn_s.pl prolog files contaning the sets of synonym into a Lucene suitable index . The resulting index contain fields named F_Word and F_SYN both of type String. More precisely, for a single F_word field named "word", there are F_SYN fields named "syn" for every synonym of that word. To look up for the synonym of a word, Lucene has a class called SynExpand, which expand query by looking up for every term in the index created before. According to the Lucene Wordnet package, the Syntax for a Query expansion is:

expand(String query, Searcher syns, Analyzer a, String f, float boost)

query is the user query which synonym are looked up, syns is a Searcher over the index created by Syns2Index, this Searcher can be used like an IndexSearcher, a is the analyzer

37 Lucene Wordnet package is available at http://lucene.apache.org/java/3_0_2/api/all/org/apache/lucene/wordnet/package-summary.html#package_description

38 The Wornet Prolog package is available at http://wordnetcode.princeton.edu/3.0/WNprolog-3.0.tar.gz

used to parse the user query, if nothing is specified, then the StandardAnalyzer is used, f is the field name to search into and boost is the boost factor applied to that query's synonym.

As an Example,a search for the synonym of "*business*" will cause the searcher to look up in the field F_Word, once business is found, the F_SYN fields for that word are retrieved in the corresponding F_SYN fields. The resulting synset obtained are sort by concept:

concept1: business, concern

concept2: commercial enterprise, business enterprise

concept3: occupation, business, job, line work, line

and so on, and online synonym search is available on the Wordnet homepage at: http://wordnetweb.princeton.edu/perl/webwn .

All things considered, we have a knowledge on how index is accessed for search, example was given with a Term query retrieval. The next paragraph shows how to compute similarity between a term in the query and a term in the index.

6.4 Computation of search results

An Information retrieval model consist of: the representation of documents, the representation of a query and a modeling framework for the document and the query, including the relationship between them[39]. Singhal [53]describe in its paper "Modern Information retrieval: A Brief overview" different retrieval models and implementations. The three most important one are the **vector space model**, the **probabilistic model** and the i**nference network model**. Concerning Lucene retrieval's model,Fang and Zhai [54], demonstrate that, Lucene similarity computation is derived from the vector space model; In addition, Lucene processes Boolean queries using another retrieval model called the **Boolean retrieval model**. Both Boolean and vector space model would be described in the following paragraphs.

Before starting with the retrieval model, here is an overview of the search mechanism for different type of queries:

i. Given a user query **q**, use a QueryParser to parse the query into a Lucene suitable query **Q**: The QueryParser detects the type of the query by using the Lucene query language . From this step on, the user application knows if a Query is a Boolean query, a span query , Fuzzy query, phrase query and so on.

ii. For each term **t** in the Lucene query Q, find its postings <docIDs,postings>

iii. Process the postings according to the type of the query Q

iv. return the top scores for the given query

6.4.4 "The Boolean model"

The Boolean model is an exact match model . In this model, a query is based on set of terms combine with Boolean operators. The concept behind this model, is to retrieve documents that satisfies the query according to operations of the Boolean algebra. In fact the Boolean retrieval can not match documents containing just a part of the query , it does not help sorting the results. Moreover, Baeza-Yates and Ribeiro-Neto [45] say it is not easy for the user, to translate its information need into a Boolean expression. Additionally, exact matching of documents leads to few or too many results, and there is no index term weighting. But Lucene still use the Boolean model to find the resulting postings for a Boolean query.

39 IR Models: The boolean Model(Lecture6). Available at http://www.cs.umbc.edu/~ian/irF02/lectures/06Models-Boolean.pdf

As an illustration, consider the user query q = “web AND search”, and consider four pdf files we have indexed. The first thing to do is to parse the query into a Lucene query. Here we obtain, Q = +contents:web+contents:search. We observe, that Q is a Boolean Query, with two Boolean operators PLUS(+) which means, both term have an occurrence equal MUST.

The next step is to retrieve the postings for each term we have:

Web	[0](0.24276763)	[1](0.24276763)	[2](0.099495865)	[3](0.06392489)
search	[1](0.113885775)	[2](0.44194174)	[3](0.17980424)	

Postings for each term holds the docIDs and the score for that term in the document. We know from section 6.3.3, that the TermScorer provides the implementation of score for each term. The computation of the score is provided by the result formula in section 6.4.3

Afterward, an intersection of the two posting list is done. According to the Boolean algebra, the AND of two conditions is true if both conditions are true at the same time. This yield to the following results set:

Web And search	[1](0.15097556)	[2](0.49793825)	[3](0.18120934)

Notice, that the score for each document have been updated. The Boolean model can also be extended to conjunctive query, which is used for AND like queries like web AND search AND spider, the Boolean model is also used for disjunction query, which is a OR like query. Those search methods was outlined by Doug Cutting in its Lecture at Pisa [17].

Lucene is not limited to Boolean model, it goes further by using the vector space model which is one of the most used model in information retrieval.

6.4.5 “The Vector space model”

This retrieval model was created 1970 by Gerard Salton[40], while developing the SMART information retrieval system. Salton consider a document space consisting of documents D_i, each document is identified by one or more index terms T_j. He represent each document as a t-dimensional vector $D_i = (d_{i1}, d_{i2}, \ldots, d_{it})$ where d_{ij} represents the weight of the jth term of the document. For two document's index Vectors, it is possible to compute a similarity coefficient $S(D_1, D_2)$ between them, which is the degree of similarity between terms and terms weights of those two documents. This similarity measure is evaluated as the inner product of the two vectors, or as the inverse function of the angle Θ between the two vectors, this is call the *cosine similarity*. By the same token, considering a query Q as a vector , one can calculate the cosine similarity between the query vector and the document vector for a given document in the index $S(D_j, Q)$ [45] like this :

$$S(D_j, Q) = \frac{\vec{D}_j \cdot \vec{Q}}{|\vec{D}_j| \cdot |\vec{Q}|}$$

Where the Query vector $\vec{Q}$ is define as $\vec{Q} = (W_{1Q}, W_{2Q}, \ldots W_{tQ})$ where each w_{tq} is the weight associated with each term in the query Q.

and the document vector $\vec{D}_j$ is define as $\vec{D}_j = (W_{1j}, W_{2j}, \ldots W_{tj})$ where t is the total number of terms stored in the index.

The numerator is the inner product between the two vector, this is the dot product between .

$\vec{D}_j$ And $\vec{Q}$. The denominator is the product of their euclidean length[40]. This leads to the following formula of the cosine similarity:

$$S(D_j, Q) = \frac{\vec{D}_j \cdot \vec{Q}}{|\vec{D}_j| \cdot |\vec{Q}|} = \frac{\sum_{i=1}^{t} W_{ij} \cdot W_{iQ}}{\sqrt{W_{ij}^2} \cdot \sqrt{W_{iQ}^2}}$$

The cosine similarity has value between 0 and 1, this means the smaller the angle Θ is, the more similar the terms in the document and those in the query are. The resulting document set is therefore more precise, than in the Boolean model, because the user information is better matched[45]. A document can be retrieved, although a part of it matches the query. An observation of the cosine similarity formula, shows that the index terms weighting has a great importance on the efficiency of the search. The term weight is the importance of the term in the document or in the query; its computation can be done in many different ways. Different term-weighting methods where define by [G. Salton, M. J. McGill,1986].The one used by Lucene is called the Mean Average Precision(M.A.P) which is the average of the precision value obtained after every relevant documents is returned[55]. The next section deal with the term weighting in Lucene and the computation of the cosine similarity.

6.4.6 Term weighting and result formula

6.4.6.a)Term weighting

To evaluate the similarity between a document and a given Lucene query, we first need to compute term weights. According to Salton and Buckley[56], the best terms display as result for a search, should have high term frequencies but low overall collection frequencies. Moreover, term weighting depends on three factors :

- the frequency of the term t_i in a document D_j : which represent the number of occurrences of that term in that document. This value shows how important the term is in the document. Its value is computed by tf_{ij}
- The inverse document frequency of that term frequency which is the importance of the term in the collection. This is a kind of discrimination factor, because one can assign a higher weight for less frequent terms. This frequency is computed by idf_i
- The importance of the document : Some document may be longer than other one, in this case, their tf_{ij} factor are normalized. If several documents have the same relevance, the shorter one are prefer. Despite, long documents holds more terms, they treat different topics or speak more extensively about a single topic. In this case they have the best score.

Referring to the definition of the term weight by Baeza-Yates and Ribeito-Neto[45], let's consider N as the number of documents in the index, let n_i be the number of documents that holds the term t_i occur. We can say that n_i is the length of the number of docIDs in the postings list for the term t_i , let $freq_{ij}$ be the number of times t_i occurs in the document D_j .Let $maxl\,freq_{lj}$ be the maximum number of terms mentioned in the text

40 A definition of the Euclidian length is available here http://en.wikipedia.org/wiki/Euclidean_distance

of the document D_j The normalize value of the frequency of t_i in the document D_j is given by the formula:

$$tf_{ij} = \frac{freq_{ij}}{maxl\,freq_{lj}}$$

Then the next value is the inverse document frequency for the term t_i this is compute by:

$$idf_i = \log\left(\frac{N}{n_i}\right)$$

The term weight is therefore the product $tf_{ij} \times idf_i$

which means for a document the t_i in a document, the term weight is given by:

$$W_{ij} = \frac{freq_{ij}}{maxl\,freq_{lj}} \times \log\left(\frac{N}{n_i}\right)$$

Furthermore, the same computation have been done before by Salton and Buckley[56] for term weighting in the query, they suggest the following formula should be the best one to be used:

$$W_{iQ} = (0.5 + \frac{0.5 \cdot freq_{iQ}}{maxl\,freq_{lQ}}) \times \log\left(\frac{N}{n_i}\right)$$

Examples of the computation of terms weighting and scores are given the book “introduction to Information retrieval”[4]

6.4.6.b)Lucene score formula

Since Lucene uses the Vector space model for the computation of similarities, the previously given formula is the basis of Lucene retrieval function .This scoring function is defined in the Lucene Library as:

$$score(q,d) =$$

$$coord(q,d) \cdot queryNorm(q) \cdot \sum_{t \in q} (tf(t \in d) \cdot idf(t)^2 \cdot t \cdot getBoost() \cdot norm(t,d))$$

Parameter of this formula are implemented in the Similarity.java class

The **coord(q,d)** is the number of occurrence of a query term in the specified document. Comparatively to the normalized frequency formula , coord(q,d) for a term in the query q is implemented by the method **coord**(int overlap,int maxOverlap)

, this method returns a float value which may be computed by the following formula:

$tf_{iq} = \frac{freq_{iq}}{maxl\,freq_{lq}}$ where $maxl\,freq_{lq}$ is the total number of term in the query called maxOverlap , and $freq_{iq}$ is the number of query term in the document called overlap and expressed in term of frequency.

The **queryNorm(q)** represent the normalization value of a query, which default value is the Euclidean norm of the query weight .It is implemented by **queryNorm**(float

sumOfSquaredWeights), where the *sumOfSquaredWeights* is compute in the TermQuery.java class like this:

```
public float sumOfSquaredWeights() {
    queryWeight = idf * getBoost();          // compute query weight
    return queryWeight * queryWeight;        // square it
  }
```

Table 17: sumOfSquaredWeights compute the weight of a term query

This means the queryNorm is given by:

$$queryNorm(q) = W_{iQ} * t.getBoost()$$

or

$$queryNorm(q) = [(0.5 + \frac{0.5 \cdot freq_{iQ}}{maxl\,freq_{lQ}}) \times \log(\frac{N}{n_i})] * t.getBoost()$$

Recall from section 6.4.3.a, the values of $tf(t \in d)$ and $idf(t)$ are equal to those of tf_{ij} and iff_i .

If a boost factor has been defined for a term in the query, then *t.getBoost()* holds that value; if nothing is given, then the default value is 0.5.

Additionally, some other boost and length factors can be added to the scoring through the $norm(t,d)$; these are:

the boost factor for the document, the field boost, and the total number of terms contained in a field. When a document is indexed, all those factors are multiplied together.

Although the Lucene scoring function is said to be practical and efficient, Fang and Zhai[55] proposed a derived retrieval function which they consider more robust than other retrieval function they called it Axiomatic retrieval function:

$$S(Q,D) = \sum_{t \in Q \cap D} \frac{(c(t,Q)) \cdot (\frac{N}{df(t)})^{0.35} \cdot c(t,D)}{c(t,D) + 0.5 + \frac{0.5 \cdot |D|}{avdl}}$$

where S(Q,D) is the relevance score of document D for query Q; c(t,Q) is the number of occurrences of term t in query Q, df(t) is the number of document that contain term t, c(t,D) is the number of occurrences of term t in document D, |D| is the length of document D and avdl is the average document length in the collection.

To sum up, the term-weighting scheme of the vector space model improve the retrieval performance of Lucene,it allows the retrieval of documents which partially matches the user request . The concept of similarity makes it possible to find the most relevant documents matching the query, furthermore Lucene also provides a modified or better said an extended vector space model by implementing the Boolean model together with the vector space model. Particularly, during the retrieval of Boolean , conjunctive or disjunctive queries. In fact, the computation of query differs from one query kind to another.

chapter 7 Case Study: Adaptation of Lucene

At the beginning of this thesis, there was a job offer from *D.k.d*[41] to incorporate their Solr team as Typo3 developer. From this proposal, the idea of studying the architecture of Lucene came out. Due to the fact that Solr is being based on Lucene, the knowledge of Lucene helps to understand Solr. By the way, the *D.k.d* is an Internet agency and Typo3 system house that offers different services to their clients like: web application analysis, consulting, web development and design. The idea of D.k.d was to create a search application based on Solr, and integrate this application into Typo3[42] as an extension. This extension is available in the typo3 repository at http://typo3.org/extensions/repository/view/solr/current/ .The Solr project and more information about Solr for typo3 is available at http://www.typo3-solr.com . In this part we will first discuss about how Solr uses Lucene to implement search, then we will examine the D.k.d Solr search algorithm.

7.1 Adaptation of Lucene in Solr

The first thing is to understand the concept behind Solr, then to see how Solr uses Lucene. Afterward, a comparison between Solr and Lucene will be done, in relation to quality.

7.1.4 What is Solr?

Solr is an open source enterprise search server from the Apache Lucene project,it was created by Yonik Seeley[43] and donated 2006 at the ASF(Apache software foundation). Its major features include indexed search, highlighting, faceted search, dynamic clustering, database integration, and rich document (e.g., Word, PDF) handling [23]. Solr runs within a servlet container like Tomcat[44] , Lucene is the basis of its full-text search. Solr receives documents in XML format or via REST-HTTP request[45] and uses HTTP, XML and JSON[46] Libraries during indexing and querying.

41 D.k.d. Internet services (www.dkd.de)
42 The enterprise open source content management system typo3 available at www.typo3.org
43 Yonik Seeley blog at http://yonik.wordpress.com
44 Apache Tomcat is an open source software for Java servlet and JavaServer Pages source: http://tomcat.apache.org/
45 Representational State Transfer (REST) is a set of architectural principles by which one can design some web services. A definition of REST-HTTP request is given here https://www.ibm.com/developerworks/webservices/library/ws-restful/
46 JSON:JavaScript Object Notation official web page http://www.json.org/

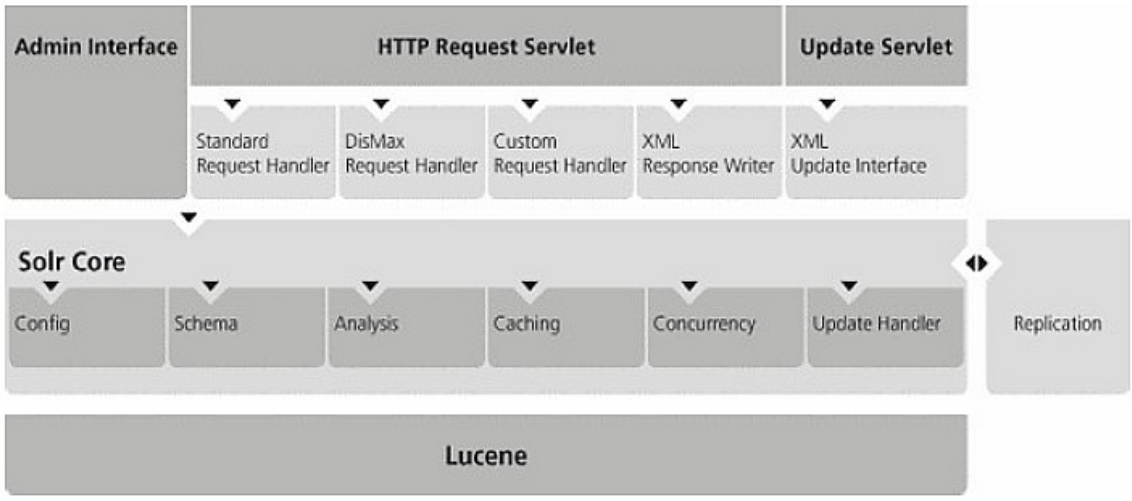


Figure 48: Using Apache Solr for search(source:http://www.typo3-solr.com/en/what-is-solr/)

Figure 48 illustrates how an application based on Solr works. According to [23], Solr also extends Lucene with other features. In the next part, we will focus on the usage of Lucene components, that Solr implements, and those that it extends. Version 4.0 of Solr will be used.

7.1.5 Application of Lucene components in *Solr*

The *Solr* indexing and search structure is similar to Lucene. Those Solr components that uses Lucene library classes or extends them, are outline in this part.

A Solr Document:

A *Solr* Document is also a set of fields stored as a list of <key-value> pairs. A Solr field is a name-value pair, where the name is almost a string . Unlike Lucene, whose fields are either String, Bytes, reader or numbers, a Solr Field's value is an Object, that can match the type defined in the XML_schema. A Solr field would be <*"filename","Broder_websearch.pdf"*> where *"Broder_websearch.pdf"* takes the type define in the user XML schema

To add a field to a Solr Document, the method *addField(name,value)* is used. A field with the same name can also have several values like in Lucene. the same method is used to add a new value to a field.

Solr analysis:

The Solr uses the Lucene analyzer to build its own analyzer called *SolrAnalysis*. This one is said to be a "*declarative analyzer* "[23]. In essence,a *SolrDocument* can be processed directly by *Tokenizers* and Tokenfilters. two factories are available in its library to create Tokenfilters and *Tokenizers* ; these are *TokenFilterFactory* and the TokenizerFactory. Like in Lucene, the *TokenStream* first breaks the *SolrDocument* into tokens,then passes it to the *Tokenfilter* which removes terms with respect to the kind of filter selected by the user application. A variety of natural language *Tokenfilters* are provided in the Solr library such as: ArabicNormalizationFilterFactory, BrazilianStemFilterFactory,TrimFilterFactory, some tokenizers are also available for processing some languages like the Chinese and Arabic. Other Tokenizer Factories like the *KeyWordTokenizerFactory* and the *WhitespaceTokenizerFactory* are derived from the Lucene Keyword- and

WhitespaceTokenizer.

The *Solr* IndexWriter:

All methods of the *Solr IndexWriter* class are inherited from the Lucene indexWriter class. This means the index writing process is the same. An *IndexWriter* configuration class in Solr contains constant values for an optimization of the search quality.These constants are : mergeFactor, maxFieldLength and maxMergeDocs.

The *Solr* IndexReader:

To add, delete, commit changes or optimize the index, one can use the updateHandler class, which implements Lucene indexReader methods. The Deletion policy , based on the Lucene IndexCommit, gather index commits points for a certain time, in the aim of supporting index replication. Besides, an Update request processor is implemented to process the document before it is indexed. This one passes the processing to the next processor responsible for handling the user request. For instance, the processing is passed to MergeIndexesCommand each time an index is to be merged into a segment. Other processors are the CommitUpdateCommand, DeleteUpdateCommand and AddUpdateCommand

SolrQueryParser:

This class deviates from the Lucene QueryParser. Solr uses a default field name given by the user, together with a Lucene QueryParser and/or with an analyzer, to create its own SolrQueryParser instance. If the default field is null, then the Solr schema is used to search for a default field name. Other Solr types of queries are Lucene-like for instance: *RangeQuery*, *PrefixQuery* and *WildcardQuery* can be recognize. To handle those queries,Solr defined a *LuceneQParserPlugin* class. A *QueryUtils* based on the Lucene BooleanQuery and Lucene Query implementations, is define to recognize negative and positive queries. A negative query is a query whose clauses are all prohibited. A positive query is the inverse of a negative query , they are both Boolean queries. If *q= web OR search* is the positive query, then *q = -Web - search* would be the negative query, in this case all documents must not contain the terms web or the term search

Here are some examples of queries, that Solr can handle:

- A query like *date of creation:[2000 To 2010]* will find all dates of creation between the year 2000 and 2010. And date of creation *[* TO 2010]* would find all those date of creation, where the field value is less than 2010.
- A negative query *-author:Broder* finds in all its author field's values where the author is not Broder.
- A wildcard query *s*h* would finds all documents which contains terms starting with *s* and ending with h like search, searches
- A Prefix query *inform** would retrieve all documents whose terms starts with inform.

Solr also includes new types of queries called FunctionQuery. A FunctionQuery is the actual value of a field with a defined function used in a relevancy score. A field is defined by the user, but functions are available in the Solr library . A list of more than 27 functions is listed in the Solr wiki [58]. A function can be a constant, a filed value ,another function or a parameter substitution(in Solr version 4.0). A function syntax starts with the keyword "*func*" such as: {!func} sum(n,5) which returns the sum of n and 5.

The *Solr* search:

In addition to the Query creation, the search with Solr includes other components like faceting, more like this, highlighting, statistics and debugging. These would be described in the next section.

7.2 Solr versus Lucene: Quality criteria

Some nice-to-have functionality, makes Solr user friendliness better than Lucene. But Lucene remains the "core", it is already a ready-to use packet for search engine developers, it is extensible, scalable and fast,Lucene supplies a cross platform solution with implementations in various programming languages like C++,.NET,PHP5,Perl,Python,Ruby and so on. Solr benefits from all those features because it's core indexing and search is based on Lucene. Lucene incremental indexing algorithm is highly efficient: It can index a data of 20 MB in size on a computer with a 1,5 GHZ processor speed[47] in one minute By relying on the comparison of Lucene and Solr by Ingersoll[57], we can say that:

- Lucene and Solr share many common features like:
  - the creation of similar document format data type: Lucene document and Solr document. Both are sets of fields, and field always have a name and one or several values
  - The processing of user query with a Query parser
  - The indexing and search algorithms
  - cross-platform solution in different programming languages
  - Both have been donated to Apache Software foundation as open source projects: Lucene in 2001 and Solr in 2006
  - They are both fast, scalable and thread-safe

- Lucene must be embedded in a search application, but Solr is a complete search application
- Lucene has no container, while Solr is built on a server and uses HTTP request to receive web query
- To control Lucene indexing and search features, one need to access the source code and modified constants or implements new methods. Solr supplies an easy setup and configuration through an administration interface and an Xml Schema.
- Solr includes features like replication, caching, faceting and highlighting. These are not available in Lucene.
  - **Replication**: When changes are done in the index, through deletion, add or update, of the commits points are reserved for an amount of time (minutes) by the *IndexDeletionPolicy*. Before this is done a Replication handler is created when there are two indexes in which one is set as the master, while the other one is stored as slave. These setting are done in the Solr configuration file *Solrconfig.xml*. A master can have many slaves and is totally unaware of them. When a modification is done on the master, the replication handler reads the file names associated to each commit points. If the value of ReplicateAfter=commit in the configuration file, then the changes are committed into the master. The slave

47 Apache Lucene features available at http://lucene.apache.org/java/docs/features.html

checks frequently if the index has a newer version. If the version is newer, the slave launch a replication process to get the new index.

- **Caching**: This concept is associated with index searching, While a current index search is used for getting queries, a new index search can be opened, its data are cached or auto warmed until the old IndexSearcher is closed. At that moment, the new indexSearcher is ready to use and would be registered as current. There two different caching implementations: the ***LRUCache***(Least Recently Used in the memory) and the ***FastLRUCache***. The last one handles concurrency during caching. Caching can also be applied to query filters, documents and user defined data.
- **faceting**: This is a most user friendly and nice-to have feature of Solr. Faceting is a break down of search results based on some criteria. To add faceting to the search, the faceting parameter called ***facet*** should be set to have the value "true", in order to enable facet counts in the query response. The Solr Search handler uses a FacetComponent class to support the faceting parameters. Some of those parameters are:
  - facet.field: allow the choice of a field which should be treated as facet. For instance a solr search can choose fields like "price", "date" and so on to generate the facet counts of field that contains that term . Note that such field should be set as "indexed"
  - facet.sort: determines the alphabetical order of the facet fields
  - facet.mincount includes the minimum count for facet fields.

    More about those parameters are found on the Solr wiki pages [48]
- **Highlighting**: This feature is also applied to search results. It includes highlighted matches in the field value.Also, the Search handler which is the base class for query handling, uses the HighlightingComponent class to support highlighting. To give a highlighting options to a field, one can use the syntax: *f.<fieldName>.<originalParam>=<value>* where f.<*fieldname*> is the name of the field, value is the <*value*> of the highlighting parameter and *o<originalParam>* is one of the highlighting parameters provided in the Solr library. Here goes an example of highlighting:
  - f.contents.hl.snippets = 4: this means in the result, the maximum of highlighted snippets generated would be 4.
  - More parameters are given in the Solr wiki pages[49]

Some important Solr features have been described briefly, and a comparison with Lucene gives a better understanding of Solr features. Let's see a use case in which Solr is being implemented for search.

48 Simple facet parameters are available at http://wiki.apache.org/solr/SimpleFacetParameters#Parameters

49 Highlighting Parameters at http://wiki.apache.org/solr/HighlightingParameters

7.3 The D.k.d. Solr for Typo3 search

As said at the beginning of this thesis, that the D.k.d. Had the idea of building a Sas (Software as a service) by combining Solr with Typo3. Their motivations for this was:

- They needed a better search engine for their Typo3 clients. Because the most popular typo3 search extension called *indexed search*[50], couldn't fit their expectancies, they had to replace it. In fact, for more than 500 web pages, Indexed search fail to display effective results
- The *indexed search* typo3 extension retrieves just front end pages, back end pages will not be found.
- There is no locking functionality implemented in the Typo3 *indexed Search* extension
- Indexed search is linked to typo3, it can not be extended apart from the typo3 field.

So they choose to implement a new typo3 extension better than the indexed search, and one that is an Open Source , because D.k.d strongly believes in Open Sources. Here are the reasons why the D.k.d choose to implement a *Solr* search engine:

- Solr is based on Lucene, which is a powerful, scalable and extensible search engine library
- Solr uses XML document, this is also a widely used document format in the web development- as in the typo3 world.
- Solr has a PHP implementation.

The figure below shows the architecture of the D.k.d Solr search for typo3.

We can observe that, the data pool contains XML, CSV and data from data base. Those data are passed to the Solr server. The developer provided Solr with an XML configuration file, where he defines, the type of data he wants to index, in particular, the Object type, that a field value should take and the fields options to be considered during indexing and search. Those fields' options includes facet parameters, a list of German or English stop words and much more. Once Solr receives these configurations, it uses Lucene core to create *Solr* Documents and to index them with respect to the use specifications.

Data is now available in form of fields whose names are words(String) and whose values are defined by the user configurations. This data is then passed to the Typo3 content management system(CMS). Once a query is taken out from the search form, it is passed to Solr, for search.

50 Indexed search typo3 extension

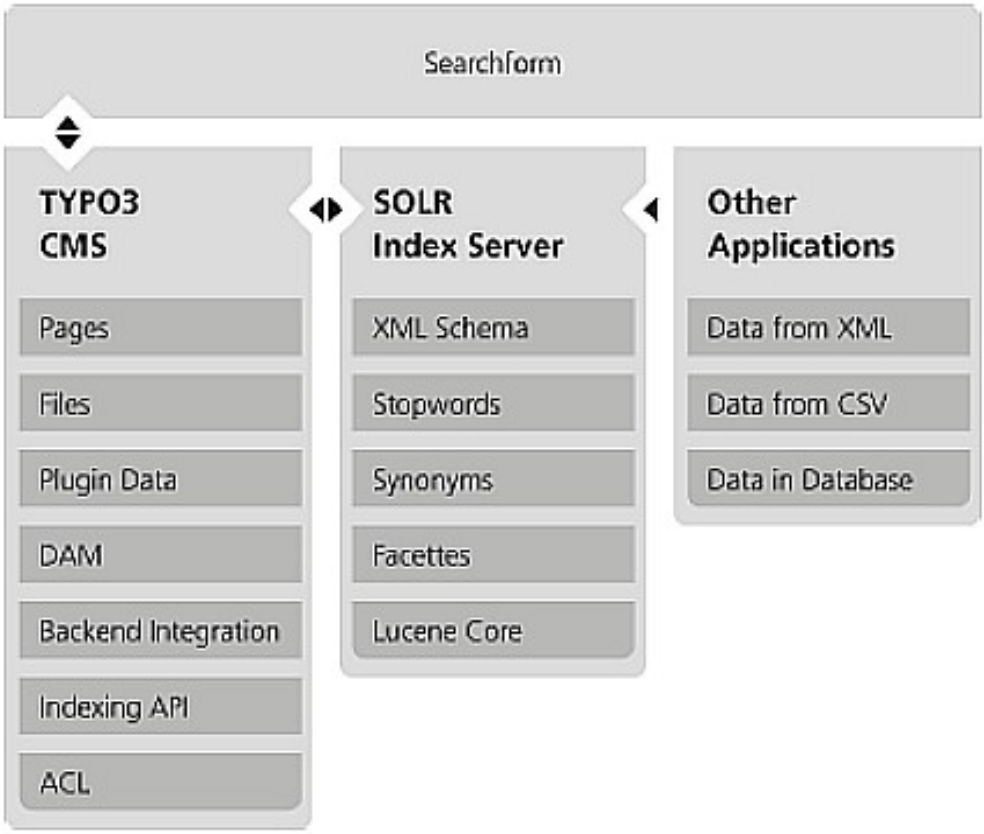


Figure 49: The Dkd Apache Solr for typo3 search (source:http://www.typo3-solr.com/en/solr-for-typo3/)

chapter 8 Summary and view

8.1 Summary

The efficiency of a search engine based on Lucene depends on the quality of the indexing and the effectiveness of the search. This Thesis focus on the architecture of Lucene internal components, and on the implementation of those components. The Lucene architecture overview designed, shows that indexing precedes search. Any Lucene based search application must first creates Lucene Documents. For this, it is recommended to use the Lucene provided document Handler interface. Nevertheless, the handler can be extends by according to needs. This is done by using a common document parser like *PDFBox* or *SAX* libraries, each document handler have to populate Lucene Document with fields. A field is a name value pair, where the name is almost a String and the value is either a String, a number, a byte array or a reader. Before indexing, each Lucene Document is analyzed with accordance to user defined parameters like "*indexed*", "*stored*"or "*term vectors*", these determine the manner with which a field must be indexed or search. Some of those parameters like the *boost factor*, *norms* or *payload* can be omitted.

A design of the Analysis process architecture shows that, the analyzer creates *TokenStream* object to keep tokens extracted from a Lucene Document by a Tokenizer. Afterward, a Tokenfilter is applied to the resulting stream of token in order to purge them from contents like stop words, white spaces, hyphens, and much more. It was demonstrated, that the analysis process is based on the decorator pattern whose central component is the TokenStream. Moreover, the internal architecture of a token produced by an analyzer has been figured . Such a token is define by its attributes which are: term, offsets, type, , text,position increment and flag. The text of a token is a term, it constitutes the central component of the indexing and search process. Besides the list of token's terms build the terms dictionary, which is part of the index. To each term correspond a postings list which hold document numbers , each posting is a reference to Lucene Document containing that term. For search reason, it is necessary to add the term frequency of each term to its postings list . A demonstration of the Lucene index building was made in chapter5. It has also been prov, that the analyzer is a strategy used by the IndexWriter to add Lucene documents to the index. A comparative study between the Lucene indexing algorithm and some information retrieval algorithms such as suffix arrays, signature files and inverted files has shown that Lucene uses the last one which is the best one in term of indexing time, and ability to process large index. The purpose of Lucene indexing algorithm is to merge indexes of the same generation or level into a single segment, once their number have reached the value of the merge factor. An example of this algorithm has been illustrated. It has been described, that merging is an index optimization method, a certain number of optimization possibilities have also been explained among which: the control of the buffer used by Lucene, the uses of multiple indexes in multiple threads, the limitation of the number of fields to indexed, and much more. The process of updating an index has been described as being a deletion followed by an addition of Lucene documents .

Unlike the indexation, a Lucene based application can fully use the QueryParser provided to transform user query into Lucene suitable query. The QueryParser is generated by the Java compiler compiler, which uses the Lucene query language to create Lucene Query. Various types of queries are available such as the TermQuery, BooleanQuery, WildcardQuery, FuzzyQuery and much more. It has been demonstrated, that these queries have a tree structure, a design of the structure of some of them is given in chapter 3. After the creation of a Lucene query, it is forwarded to the IndexSearcher which triggers the search process. A demonstration of the search process including the relationship between search components, has been made with the help of FMC diagrams. At the beginning of the search, IndexSearcher creates a Scorer and a collector for the query. The scorer in its turn creates an instance of documents enumerator called *DocsEnum*, the latter has the responsibility to

read documents number and frequencies out of the Lucene index. Once those information are returned to the scorer it uses them to calculate the similarity between the Lucene query, supplied by the IndexSearcher , and the documents pointed by the provided documents number. The computation of search result in Lucene is based on the vector space model created by Gerard Salton [40]. According to him, document and query should be consider as vectors of terms they holds. The similarity between the document and the query is then the cosine of the angle between the two vectors. The more the cosine is small, the more the document is similar to the query. This document is then consider as then consider as the result of the query. An analyze of the Lucene scoring formula is made in section 6.4.3.b

At the end of this thesis, a case study of Lucene is made. This study shows the Lucene components involved in Apache Solr search application; a comparison of the quality of Lucene and Solr is also made. An example usage of Solr by the Dkd company is outline.

8.2 View

During the redaction of this thesis, some gaps where identified in Lucene functionalities. For further work on Lucene, It can be interesting to:

- To use more Lucene optimization capabilities in the Dkd Solr search like
- analyze this Lucene architecture and extend it for newer Lucene components
- demonstrate if and how Lucene can include some of Solr functionality as faceting, caching and replication.
- Do the further development of the document handler interface, in order to introduce functionality like XML and Pdf document parsing.

Bibliography

[1] Y. Le Coadic:" *Le besoin d'information – formulation, négociation, diagnostic* 2eme edition". ADBS.ISBN 978-2-84365-097-0, january 2008

[2] E. Hatcher , O.Gospodnetić. "*Lucene in Action"*. Manning publication, January 2005. ISBN 1932394281, 2005

[3] M. McCanless, E. Hatcher , O.Gospodnetić ."*Lucene in Action.Second edition"*.Manning publication. ISBN 978-1-933988-17-7, 2010

[4] C. D. Manning, P. Raghavan, H. Schütze : "*Introduction to Information Retrieval"*. Cambridge University Press. ISBN 0521865719, July 2008

[5] Manfred Hardt and Fabian Theis. "*Suchmaschinen entwickeln mit Apache Lucene 1*. *edition*. " Entwickler Press . ISBN 3935042450, August 2004.

[6] N. Fielden: *"History of Information Retrieval Sytems & Increase of Information over Time".* Available at http://userwww.sfsu.edu/~fielden/hist.htm ,May 2002

[7] Doug Cutting. Open source search – "*Information retrieval Seminar* ", IBM Research Lab., Haifa / Isreal. Available at http://www.haifa.ibm.com/Workshops/ir2005/papers/ DougCutting-Haifa05.pdf, December 05, 2005

[8] Andrei Broder. "*A taxonomy of web search" –IBM Research*. Available at http://delivery.acm.org/10.1145/800000/792552/p3-broder.pdf? key1=792552&key2=1385728721&coll=GUIDE&dl=GUIDE&CFID=95859812&CFTOKEN=36441287 , fall 2002

[9] Douglas R. Cutting and Jan O. Pedersen. "*Space Optimizations for total Ranking"*. Montreal. Available at http://lucene.sourceforge.net/papers/riao97.ps , June 1997

[10] Justin Zobel and Alistair Moffat. *Inverted files for text search engines*. Volume 38. ACM publication, New York. ISSN 0360-0300. 2006

[11] Umberto Eco. "*Wie man eine Wissenschaftliche Arbeit schreibt*", 12. Auflage. Verlagsgruppe Hüthig Jehle rehm GmbH. ISBN 978-3-8252-1512-5, 2007

[12] Apache Software Foundation. "*Apache Lucene official web page* ". URL http://lucene.apache.org/ , 2006

[13] Apache Software Foundation. Lucene 2.4.1 API. URL, http://lucene.apache.org/java/2_4_1/api/index.html, 2000- 2009

[14] Apache Software foundation Lucene 3.0.2 -API. URL http://lucene.apache.org/java/

3_0_2/api/core/index.html , 2000-2010

[15] Aaron Wall ."*Search Engine History"*.URL, http://www.searchenginehistory.com.

[16] Deng Peng Zhou. "*Delve inside the Lucene indexing mechanism"*. URL http://www.ibm.com/developerworks/library/wa-lucene/ , 2006.

[17] Doug Cutting. "*Lucene lecture at pisa"*. URL http://lucene.sourceforge.net/talks/pisa , November24,2004

[18] Dr. Martin Porter and Richard Boulton." *The Porter Stemming algorithm"*. URL, http://snowball.tartarus.org/algorithms/porter/stemmer.html ,2001 2002

[19] Zend Technologies Ltd." *Programmer's reference guide"*, Chapter 46. Zend_Search_Lucene. URL, http://framework.zend.com/manual/en/zend.search.lucene.html, 2006-2009

[20] Grant Ingersoll. Lucene Boot Camp. *Lucid Imagination*. Atlanta, Georgia. URL, http://www.slideshare.net/GokuID/lucene-bootcamp , November 12, 2007

[21] Lucid Imagination. "*Introduction to Apache Lucene and Solr".* URL, http://www.lucidimagination.com/Community/Hear-from-the-experts/Articles/Intoduction-Apache-Lucene-and-Solr , 2009

[22] Grant Ingersoll. "*Search smarter with Apache Solr", Part1: Essential features and the Solr Schema*. URL, https://www.ibm.com/developerworks/java/library/j-solr1/ , May 2007

[23] Apache Software Foundation. "*Apache Solr official web page"*. URL, http://lucene.apache.org/solr/ , 2007

[24] Luke- Lucene Index Toolbox. URL, http://www.getopt.org/luke ,2008

[25] MoinMoin Wiki Engine. *Lucene-Java Wiki*. URL, http://wiki.apache.org/lucene-java/HowTo , 2009

[26] Javacc[tm] grammar files. URL,https://javacc.dev.java.net/doc/javaccgrm.html , 2010

[27] Bernhardt Treutwein. «*Algol-60-BNF».* URL, http://www.lrz-muenchen.de/~bernhard/Algol-BNF.html,1979

[28] Martin Porter. "*The Porter Stemming algorithm"*, URL, http://tartarus.org/~martin/PorterStemmer/ ,2006

[29] Martin Porter, Richard Boulton, "*Stemming Algorithms for various European Languages",* URL, http://snowball.tartarus.org/texts/stemmersoverview.html , 2001,2002

[30] Grant Ingersoll, Ozgur Yilmazel, "*Advanced Lucene"*, ApacheCon, Europe

2007.URL, http://www.cnlp.org/presentations/slides/dvancedluceneeu.pdf ,2007

[31] J.R.Arnold, W.F.Libby, "*Age determination by radiocarbon content:checks with sample of known age"*, Available at URL: http://hbar.phys.msu.ru/gorm/fomenko/libby.htm ,december 1943

[32] Till Zoppke, "*The ENIAC simulation 2003/2004"*, available at http://www.zib.de/zuse/Inhalt/Programme/eniac/about.html ,september 2010

[33] bibliotheque nationale de France,*dossier pédagogique*, Available at Http://calsses.bnf.fr/dossitsm/biblroya.htm, 2010

[34] Society for Anglo-Chinese Understanding. Available at Http://www.sacu.org/greatinventions.html ,2004

[35] Susan K. Soy, University of texas, *Class Lecture Notes: H.P. Luhn and Automatic indexing*. Available at http://www.ischool.utexas.edu/~ssoy/organizing/l391d2c.htm , 2006

[36] Vatican Library, available at http://www.vaticanlibrary.va/home.php?pag=storia&ling=en&BC=11 2010

[37] Eugene Garfield," *A tribute To Calvin N. Mooers, A Pionner of Information Retrieval* ", The Scientist, Vol:11,#6,P.9, March 1997

[38] Ayşe Göker, John Davies, "*Information Retrieval. Searching in the 21st Century".* John Wiley & Sohns,Ltd. . ISBN 978-0-470-02762-2, 2009

[39] Chris Buckley, James Allan, and Gerald Salton. "*Automatic routing and Retrieval using SMART:TREC-2.".* Elsevier Science Ltd. Information Processing & Management, Vol. 31, No.3, PP.315-326, 1995. Available at http://www.sciencedirect.com/science/journal/03064573 , 1995

[40] Gerard Salton, A. Wong, C.S. Yang. "*A vector space model for automatic indexing"*. ACM Inc. Pages 613-620,1975

[41] Yahoo! Inc.The history of Yahoo!- "*How it All started"*. URL: http://docs.yahoo.com/info/misc/history.html , 2005

[42] David A. Vise, Mark Malseed. "*The GOOGLE Story"*. Published by Bantam Dell A Division of RandomHouse, Inc. New York . ISBN-10:0-553-80457-X, ISBN-13:978-533-80457-7, November 2005

[43] Doug Cutting Interview by Philipp Lenssen. 28. Available at http://blogoscoped.com/archive/2004_05_28_index.html, May 2004

[44] William B. Frakes, Ricardo Baeza-Yates. "*Information Retrieval: Data structures and Algorithms*." Prentice Hall. ISBN-10: 0134638379, June 22 ,1992

[45] Ricardo Baeza-Yates, Berthier Ribeiro-Neto," *Modern Information Retrieval"*. Addison Wesleyby the ACM press, A Division of the Association for Computing Machinary ,Inc(ACM). ISBN 0-201-39829-X, 1999

[46] Gaston H. Gonnet, *Unstructured data bases or very efficient text searching* by ACM, ISBN: 0-89791-097-4 , 1983

[47] Klemens Böhm. "*Vorlesung Datenbankeinsatz"*, Universität Karlsruhe, 2005

[48] Eric W. Brown, James P. Callan, W. Bruce Croft. "*Fast incremental indexing for full-text information retrieval*.Department of computer science", University of Massachusetts. Proceedins of the 20th International conference on Very Large Data Bases(VLDB), Santiago, Chile, 1994. Pages 192-202.

[49] Chris Faloutos, Stavros christodoulaskis. "*Signature files: An Access method for Documents and its analytical performance Evaluation*.". University of Toronto . Pages 267-288. ACM Ny, USA. ISSN: 1046-8188 .Available at : http://portal.acm.org/citation.cfm?id=357411, September 1984

[50] Stefan Büttcher and Charles L.A. Clarke. "*Indexing time vs query time Trade-offs in Dynamic information Retrieval systems"*. Pages 317,318. University of Waterloo, Canada, Bremen, Germany. ACM 1-59593-140-6/05/0010 ,2005

[51] The Apache Software foundation, *PDFBox user guide*. Available at http://pdfbox.apache.org/userguide/index.html ,2008-2010

[52] The Apache Software foundation, *The QueryParser.Token class*, available at http://lucene.apache.org/java/3_0_1/api/core/index.html? org/apache/lucene/search/TopDocs.html ,2000-2010

[53] Wordnet , a lexical database for Englisch. Princeton University. URL: http://wordnet.princeton.edu/wordnet/ 2010

[54] Amit Singhal, Google,Inc. URL.http://singhal.info/ieee2001.pdf ,2001

[55] Hui Fang, ChengXang Zhai, "*Evaluation of the Default Similarity Function in Lucene"*, URL: http://www.ece.udel.edu/~hfang/lucene/Lucene_exp.pdf ,2009

[56] Gerard Salton, Michael J. McGill, "*Introduction to modern information retrieval"* .McGraw-Hill, Inc. New York, NY, USA. ISBN:0070544840 ,1986

[57] Grant Ingersoll, "*Apache Lucene and Solr*.", Triangle Java user's group , November 2007

[58] Yonik Seeley, "*FunctionQuery"* available at

http://wiki.apache.org/solr/FunctionQuery ,2010

[59] M.E. Maron, J.L. Kuhns. Journal of the ACM, Volume 7, pages 216-244 ,1960

[60] C. Buckley. "*Implementation of the SMART information retrieval System".* Cornell University Ithaca,NY,USA. ,1985

[61] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *"Design Patterns:Elements of Reusable Object-Oriented Software ".Pages 59-63*. Addison-Wesley Longman, Amsterdam, 1st Ed.,October 1994