

FACULTY OF INFORMATICS, MASARYK UNIVERSITY

Indexing Very Large Text Data

MASTER'S THESIS

Lucie Molková

Brno, spring 2011

Declaration

Thereby, I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing this thesis, as well as any quoted materials, are properly cited, including full references to their sources.

Advisor: RNDr. Michal Batko, Ph.D.

Acknowledgements

I would like to thank my advisor RNDr. Michal Batko, Ph.D. for the help and motivation he provided throughout my work on this thesis, and to RNDr. Vlastislav Dohnal, Ph.D. for mediating this interesting topic. I am also grateful to Mgr. Martin Janík for his time and patience while introducing the NetBeans IDE to me, and to Bc. Tomáš Janoušek for his \LaTeX support. Last but not least, many thanks go to B.Sc. Patrick Donley for the proof-read and Czenglish to English translation.

Abstract

With the emergence of digital libraries with non-textual content, there is a clear need for improved techniques to organize large quantities of information. It appears, that the textual descriptions associated with non-textual content are an important source of information when judging topical relevance. The CoPhIR (Content-based Photo Image Retrieval) data set is an example of a multimedia collection that serves as the basis of the experiments, including over 100 million images with associated metadata. The main objective of this thesis is to study Lucene technology for indexing text data, and, using this technology, implement indexing and content-based image searching of the CoPhIR collection. The procedure of creating an index from the initial data set is described in detail, including possible pitfalls. This work also surveys efforts that focus on information retrieval and presents some challenges for retrieval in both image indexing and searching.

Keywords

Lucene, CoPhIR, Information Retrieval, Text Retrieval, Text-search, Parsing, Similarity Searching, Searching, Relevance, Index, Indexing, Analysis, Web Service

Contents

1	Introduction	6
1.1	Objectives	7
1.2	Contents	7
2	Information Retrieval	9
2.1	Information Retrieval History	9
2.2	Information Retrieval Strategies	10
2.3	Text Retrieval	12
3	Lucene	15
3.1	What Is Lucene?	15
3.2	Lucene Architecture	15
4	Parsing XML Documents	19
4.1	Parsing Approaches	19
4.2	Parsing CoPhIR Data	22
5	Analysis	26
5.1	Lucene Analyzers	26
5.2	Field Options	28
5.3	Analyzing CoPhIR Data	29
6	Indexing	32
6.1	Lucene Indexing	32
6.2	Indexing CoPhIR Data	35
7	Searching	39
7.1	Lucene Document Searching	40
7.2	Lucene Queries	40
7.3	Lucene Scoring	41
7.4	Searching CoPhIR Data	42
8	Conclusions	46
8.1	Improvements and Further Steps	47

Chapter 1

Introduction

Since the beginning of civilization, human beings have focused on written communication. Today, with the emergence of digital libraries and electronic information exchange, there is a clear need for improved techniques to organize large quantities of information. [5]

The information is mainly stored on computers, but the role of computers has changed in the last decades. We no longer use computers for just their raw computing abilities [1]: they also serve as communication devices, multimedia players, and media storage devices. Those uses require the ability to quickly find a specific piece of data. So far, humans have invented categorizations, classifications, and other types of hierarchical organizational schemes in order to speed up the search process.

The problem arises when an increasing portion of the information is non-textual in format: consider, for example, the emergence of YouTube and Flickr and many similar systems as web-based sources of video and audio information. Many applications that handle information on the internet would be completely inadequate [6] without the support of information retrieval technology. New problems and challenges for information retrieval come up constantly. One of the contemporary challenges is to make rich media - such as images, video, and audio files in various formats - easy to locate.

Applied and theoretical research and development in the areas of information processing, storage, and retrieval is of interest to all sectors of the community. This thesis surveys recent efforts that focus on information retrieval, with emphasis on content-based image search.

Images are sought after in many contexts. The amount of visual information available has grown dramatically in the past few years. The nature of visual information creates some challenges for retrieval in both image indexing and searching. Images may be selected on the basis of content, salient features, quality, visual appearance, level of accuracy, semantic value, image tone, physical size, layout, date, and anonymity [6]. The textual descriptions associated with images are an important source of information when judging the topical relevance of images.

This work dwells on the problem of indexing a very large collection of image

metadata. The metadata is textual by nature, and it contains descriptive information about the image, including image name, description, and tags. Having this information available, it is easy to perform topical searches. Textual pieces of information act as a basis for content-based searches, providing the global idea about the picture contents. Textual image search can be either used stand-alone, or in combination with similarity searches based on visual characteristics.

1.1 Objectives

The main objective of this work is to study Lucene technology for indexing text data and using this technology to implement searching and indexing of 100 million XML files from the CoPhIR (Content-based Photo Image Retrieval) data set. The CoPhIR data set [10] is a test collection that serves as the basis of the experiments on content-based image retrieval techniques and their scalability characteristics, in the context of the SAPIR (Search on Audio-visual content using Peer-to-peer Information Retrieval) project. The data collected within CoPhIR represents the world largest multimedia metadata collection available for research purposes, containing visual and textual information regarding over 100 million images.

The purpose of this work is to allow users of the MUFIN (Multi-Feature Indexing Network) system a convenient full text search method. MUFIN [9] is a system that organizes 100 million image collections based on the CoPhIR data set. MUFIN adopts the metric space as a very general model of the similarity. Its indexing and searching mechanisms are based on the concept of structured peer-to-peer networks, which makes its approach highly scalable and independent of the specific hardware infrastructure on which it runs.

The resulting index has to be optimized for searching in descriptive image data. Before the actual indexing, data cleansing has to be executed. Subsequently, an index can be built and optimized for user searches. The user interface has to be prepared and adjusted in order to provide users a simple piece of software. Since this search is intended for MUFIN users, the user interface has to be integrated into the existing MUFIN website.

Outputs of the work include: This paper describing both the theoretical background of information retrieval and technical reports, a text index of the CoPhIR data set including software used for its generation, and an implementation of a user interface for the MUFIN system.

1.2 Contents

The following chapter introduces information retrieval in more detail, providing the reader an overview of information retrieval development, and introducing different strategies. Consecutively, it focuses on text retrieval, and presents several solutions from this area, choosing Lucene technology for further usage.

The third chapter serves as an opening passage for Lucene. The technology is introduced elaborately, allowing the reader to grasp the fundamentals of Lucene indexing and searching. Lucene architecture is accompanied by a step-by-step description of the tasks that have to be undertaken during the information retrieval process.

After the theoretical preliminary introduction to information retrieval and Lucene, chapters four to seven become more practical, focusing on parsing the input data, analysis, indexing, and searching. Each chapter begins with a short theoretical background with respect to the specifics of Lucene technology. It describes the fundamentals of the given task and then switches to practice. The described processes are then applied to the CoPhIR data. Each chapter explains the performed procedures and supplements the theory with performance measurements, encountered issues and further recommendations.

The last chapter summarizes the outcomes of the whole thesis, emphasizes the most important aspects of indexing and searching, and concludes the findings. The whole thesis is enclosed by suggesting further steps for information retrieval improvement.

Chapter 2

Information Retrieval

The term *information retrieval* [5] refers to a search that may cover any form of information: structured data, text, video, image, sound, musical scores, DNA sequences, etc. In this chapter we will briefly describe the information retrieval history, strategies, and existing implementations of those strategies.

2.1 Information Retrieval History

For thousands of years, people have realized the importance of archiving and finding information. The oldest and simplest way of doing information retrieval is called a brute force method; looking at every item in the data set, and determining whether it satisfies the information need. No ordering, sorting or pre-processing is necessary.

The first more advanced technique appeared in libraries, with their attempts to locate printed materials by different criteria such as author, title, and subject. Many catalogues were built to act as access points to the collections, and to facilitate the information retrieval process.

Cataloging, called indexing today, then experienced very little advancement until the beginning of computer era. Card catalogues that were easier to maintain began to appear, and single terms became the target of indexing instead of whole titles or sentences. However, no automation appeared yet.

The field of information retrieval, as it is known today, was born in the 1950s out of this necessity for automation. It was after 1945 when Vannevar Bush [7] published a ground-breaking article titled *As We May Think*. The first automated information retrieval systems were introduced a few years later, ready to undertake the human labor.

Several works emerged engaging in the topic of information retrieval. One of the most influential techniques was described by Hans Peter Luhn [7] in 1957. He proposed KWIC (Key Words In Context) indexing with word overlap as a criterion for retrieval. Later, he worked out many of the commonly used techniques including full-text processing, hash codes, and auto-indexing.

In the 1960s Gerard Salton [7], with his students, initiated the development of

the SMART (System for the Mechanical Analysis and Retrieval of Text) system at Harvard University. Salton published several articles about the theories of indexing, automatic indexing, and the vector space model. He also introduced the TF-IDF (term frequency - inverse document frequency) model for scoring documents.

In the 1970s, many developments were built on the evaluation methodology introduced earlier as well as new models for doing document retrieval. These new models were experimentally proven to be effective on collections of several thousand articles [7], but there was no certainty about their performance on larger text collections.

In 1992, the US Department of Defense, along with NIST (the National Institute of Standards and Technology), organized TREC [7] (the Text Retrieval Conference) and encouraged research in information retrieval from large text collections. Over the years, TREC has expanded into related fields, including retrieval of spoken information, non-English language retrieval, information filtering, and user interactions with a retrieval system.

The introduction of web search engines in the 1990s boosted the need for very large scale retrieval systems. Search engines became the most common, and maybe the best instantiation of information retrieval models, research, and implementation. They are now used on an everyday basis by a wide variety of users. New challenges appear almost constantly, including the need for searching in non-textual data like pictures, audio, and video.

2.2 Information Retrieval Strategies

While presenting the history of information retrieval, we came across vector space models, and we mentioned that more models, or strategies, were developed over the years. In this section, we are going to introduce the most significant strategies that have emerged.

A *retrieval strategy* [5] is an algorithm that takes a query Q composed of a finite set of terms $T = \{t_1, t_2, \dots, t_n\}$ and defined keywords. For each document from finite set $D = \{D_1, D_2, \dots, D_m\}$, $D_i \in 2^T$ it returns a similarity coefficient that reflects how the document corresponds to the query. In other words, a query is issued and a set of documents that are deemed relevant to the query are ranked based on their computed similarity to the query. Numerous strategies exist to identify how these documents are ranked.

Boolean Model

In a *boolean model* [1], the user specifies his or her information need using a query in conjunctive normal form. The boolean model then simply returns a set of matching documents. In a pure boolean model, the similarity coefficient is either 0 or 1, match or no match.

The boolean model has several disadvantages. It may be very hard for a user to form a good search request using the conjunctive normal form. Furthermore, there is

no relevance scoring, and the documents are retrieved unordered. Due to increasing requirements for precision, a pure boolean model is rarely used alone, and is usually replaced or supported by relevance ranking models.

Vector Space Model

In a *vector space model* [6] the documents and queries are represented as vectors embedded in a high-dimensional Euclidean space, where each term is assigned a separate dimension. The occurring term has non-zero values, and omitted terms are assigned zero values. Similarity or relevance is a vector distance between the query and the document.

Several different ways of computing these values have been developed. TF-IDF (term frequency – inverse document frequency) weight is one of the widely used approaches. The best relevance in TF-IDF is achieved by having a high term frequency in the given document, and a low document frequency of the term in the whole collection of documents.

Metric Space Model

The *Metric space model* is the most general data model, but it can still be used for designing an index structure. Thanks to its high abstraction it is more flexible than the vector space model. The metric space model concept treats the dataset as unstructured objects together with a distance or dissimilarity function measurable for every pair of objects.

The similarity search based on the metric space data model has recently become a standalone research stream, which arouses greater and greater interest. It is being used in multi-feature similarity searches [8] where several similarity sub-queries are executed on different features of the objects stored in the database and the obtained subresults are combined to compute the overall similarity of the respective object.

Probabilistic Model

Probability theory is another field that concerns with term weighting. In a *probabilistic retrieval* [6] the similarity is based on the likelihood that a term will appear in a relevant document. Probabilistic retrieval defines the sample space as a set of all possible outcomes, where each sample space is either relevant or irrelevant. Consecutively it defines the probability that a document contains given term with the sample space. Joint probability distribution denotes the similarity.

Inference Network Model

In *inference network model* [7] the document retrieval is modeled as an inference process in an inference network. A document instantiates a term with a certain strength, and the credit from multiple terms is accumulated given a query to compute the equivalent of a numeric score for the document.

Latent Semantic Indexing Model

The *Latent semantic indexing model* [5] represents the occurrence of terms in documents with a term-document matrix. Decomposition is defined to filter out the noise from the document, hence two documents with the same semantics are placed close to each other in a multi-dimensional space.

Other Models

We can find many more models that are somehow used for information retrieval. The *Language model* [5] computes the likelihood that each document will generate the given query, while the *fuzzy set model* [5] associates each element of a document a certain number that indicates its strength of membership.

We can find attempts of modeling the optimal query out of the entered query using evolutionary principles and genetic algorithms as well as attempts to train *neural networks* [5] to trigger the correct links between queries and documents.

2.3 Text Retrieval

Text retrieval [5] refers to the process of searching for documents, information within documents, or metadata about documents. It is devoted to finding relevant documents, not just simple matches to patterns.

Text Retrieval Solutions

The techniques described earlier are adopted by many commercial as well as many open source solutions in the domain of information retrieval. When thinking about indexing and searching applications, one may choose between numerous products available on the market. Basically, the products can be grouped into two major categories. The first of these categories are information retrieval libraries that can be easily customized and embedded into any application. The second consists of ready to use indexing and searching applications that are typically designed to work with particular types of data, and are therefore less flexible.

For the purpose of this work, we need a text retrieval solution that can be obtained and used for free. We also require a solution that can be easily adjusted in order to comply with our needs (i.e. indexing 100 million XML files and embedding the search into MUFIN). Following product reviews present the most common text retrieval solutions that more or less meet the stated objectives.

Xapian

Xapian¹ is an open source search engine library written in C++ that is released under the GPL. It supports bindings for many languages including Perl, Python, PHP,

¹<http://xapian.org/>

and Java. Xapian is actively developed software. It's currently at version 0.8.3, but it has a long history behind it, and is based on decades of experience in the information retrieval field. The noteworthy features include ranked probabilistic search, relevance feedback, and phrase and proximity searching. It supports a full range of structured boolean search operators, wildcard searches and synonym searches. Xapian has a parser for several rich document types at its disposal. In addition to providing an IR library, Xapian comes with a web site search application called Omega, which can be downloaded separately.

Minion

Minion² is a quality search engine from Sun Labs written in Java. In addition to the standard document retrieval operations, it provides for relational querying in conjunction with boolean and proximity querying. It also provides document similarity measures, result and document clustering, and automatic document classification capabilities. The engine is designed to be highly configurable and is intended to be used in research as well as in production environments.

The main disadvantage of this project is that it is quite recent. The only version available for download is Minion 1.0. The information on the developer website states that the documentation is still being put together.

Lucene

Lucene³ is sometimes considered the de-facto commercial standard for search, since it is the most widely used information retrieval library in industry. It is written in Java and it comes with several built in analyzers that can handle compound words, case sensitivity, and spell correction. Its search combines the vector space model and the boolean model approaches. Lucene was originally written by Doug Cutting and is now supported by the Apache Foundation. It is the very active developer community that is Lucene's biggest strength. Lucene has widespread industry adoption. It is used by Amazon's Search Inside This Book, NetFlix, Digg, MySpace, LinkedIn, Fedex, the Hathi Trust Digital Library, and many more [1].

Egothor

Egothor⁴ is a full-text indexing and searching Java library that uses core algorithms very similar to those used by Lucene. Egothor supports an extended boolean model, which allows it to function as both the pure boolean model and the vector model. The required model can be specified via a simple query-time parameter. Egothor features a number of different query types, supports similar search syntax, and allows multithreaded querying, which comes in handy when working on a multi-CPU

²<https://minion.dev.java.net/>

³<http://lucene.apache.org/>

⁴<http://www.egothor.org/>

computer or searching remote indices. Egothor also provides parsers for several rich text document formats, such as PDF and Microsoft Word documents. However, at the time of this writing, the Egothor project is, after more than ten years, leaving the area of full-text search engines. According to the current information, a new version is already under development that will serve as a novel universal database system.

Solr

Solr⁵ is an open source enterprise search platform from the Apache Lucene project. Since its target is enterprise development, it provides several features that can help in dealing with gigantic indexes. Solr can replicate indexes from one machine to another, which can help with failover or performance issues. It automatically distributes queries across multiple shard servers. With `DataImportHandler`, Solr can crawl a SQL database and index the data contained therein without additional code, just an XML configuration. Solr can also be easily customized, since it is entirely written in Java.

Conclusions

We have now introduced several information retrieval solutions with comparable set of features. Of course, more of them can be found, but they are less appropriate for the purposes of this work. Typically, they are difficult to customize, and build in existing applications. Some of them are also meant only for specific types of data. Namely, Microsoft Search Server 2010 Express is designed for intranet searches, Sphinx is meant primarily for database searches, and Zettair is supposed to index HTML collections. Furthermore, all commercial solutions have been skipped, since we need to download and use the solution for free.

When summarizing the findings, Lucene occurs to be the number one choice for this work. The facts that it is entirely written in Java and that it is widely used and supported by the Apache developers community are big advantages. Egothor's main disadvantage lies in its redirection to a database system. Xapian is primarily written in C++, and even though a Java binding exists, it would be more difficult to embed it in the Java-based MUFIN. Minion appears to be a rather recent project, and at the time of this writing, it is neither well documented, nor widely used. Solr might be a good alternative to Lucene, or rather a successor of Lucene, when the index becomes too large and distributed computing becomes necessary.

In this chapter we have presented the basics of information retrieval and reviewed software solutions to it. We have given some reasons for the choice of Lucene. Lucene has also been shortly introduced, but for further reading we need to introduce Lucene in more detail.

⁵<http://lucene.apache.org/solr/>

Chapter 3

Lucene

A common misconception is that Lucene is an entire search application. In fact, it is a Java search library that serves only as the core indexing and searching component that can be easily added to any application. Lucene [1] powers the search features behind many websites and desktop applications and is one of the most widely used information retrieval libraries.

3.1 What Is Lucene?

Lucene is a free, open source project implemented in Java, and a project in the Apache Software Foundation, licensed under the liberal Apache Software License.

Lucene's core [1] itself is a single JAR (Java Archive) file, less than 1 MB in size and with no dependencies, and integrates into the simplest Java stand-alone console program as well as the most sophisticated enterprise application.

Lucene has high performance and high scalability. It can index and make searchable any data that text can be extracted from. Lucene is format independent, which means it can index and search data stored in files of almost any type - simple text files, Microsoft Word documents, XML documents, HTML documents, PDF files and many more.

Lucene's primary goal is to facilitate information retrieval. The emphasis on retrieval is important. Indexing and searching steps have to be supplemented with parsing and analysis in order to achieve the best search results. The whole process of information retrieval can be divided into several sequential steps. Let us now take a closer look at Lucene's architecture with an emphasis on these steps.

3.2 Lucene Architecture

As we said, Lucene acts as the core component of a search application. It is important that Lucene does not provide all the functionality of a search application. Figure 3.1 shows us a scheme of typical search application. Note that only certain

processes are handled by Lucene; the remaining tasks have to be handled separately by the application.

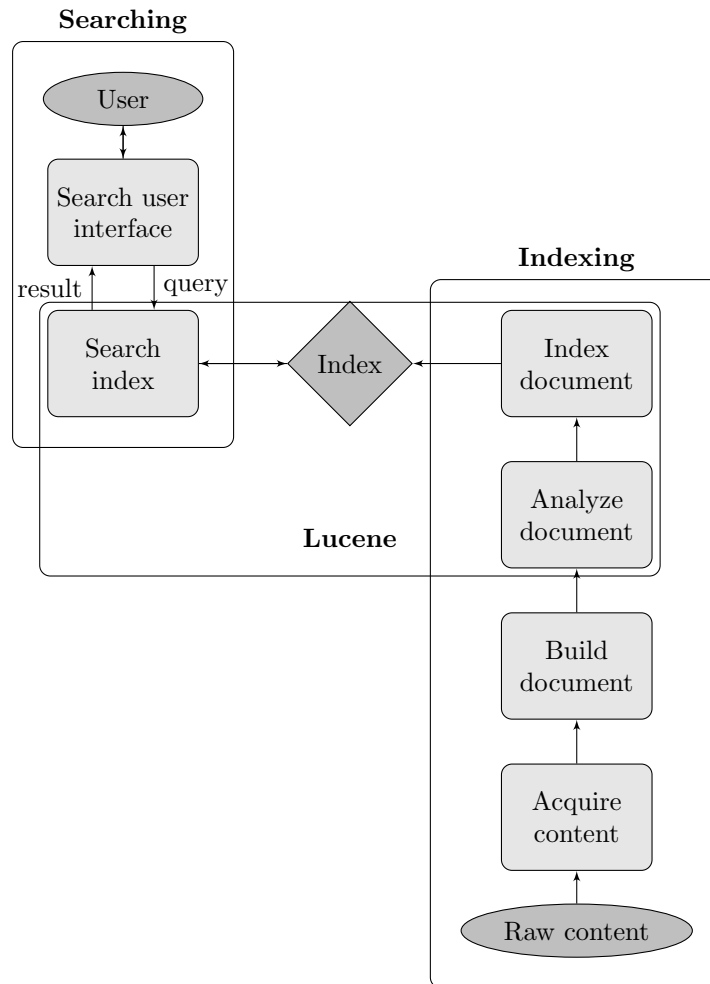


Figure 3.1: Typical components of search application architecture with Lucene components highlighted

The sequence of steps in a search application is important. Table 3.1 lists the basic steps sequentially, and indicates whether Lucene or the application is responsible for the step. For better understanding of the process, all the steps may be grouped into two major categories. The first four steps, from acquiring the content to indexing the document, are part of indexing, while the following steps, from obtaining the query to presenting the search results, are part of searching. Other tasks, like updating and deleting, have to be involved in the search application in case the application works with a non-static data set.

Step	Responsibility
Acquire content	application
Build document	application
Analyze document	Lucene
Index document	Lucene
Obtain query	application
Search index	Lucene
Present search results	application

Table 3.1: Responsibilities for search steps

Indexing

Indexing is the initial part of all search applications. Its goal is to process the original data into a highly efficient cross-reference lookup in order to facilitate rapid searching.

The first step of indexing is *acquiring the content*. This process gathers and scopes the content that needs to be indexed. The job is simple when the content is already textual in nature and its location is known. In other cases, sometimes it is necessary to use crawlers to locate relevant files. Sometimes parsers need to be built to extract required content from files that contain markups. Lucene itself does not provide any functionality to support acquiring content.

The next step is to *build documents* out of the content. The raw content that needs to be indexed has to be translated into the units (usually called documents) used by the search application. The document typically consists of several separately named fields with values, such as title, body, abstract, author etc. The design of documents and fields plays an important role in searching for the relevant document. An important part of creating fields is computing their values. The documents or fields that are deemed more or less important may be boosted. Note that boosting may be done statically at indexing time, or dynamically during searching. Building the document from the raw content is up to the application.

The textual fields in a document cannot be indexed directly. Rather, the text has to be broken into a series of individual atomic elements called tokens. This happens during the *document analysis* step. Each token corresponds roughly to a word in the language, and the analyzer determines how the textual fields in the document are divided into a series of tokens. There are several issues that the analyzer has to handle. Namely, dealing with singular and plural, compound words, spell correction, and synonym injection. Lucene [1] provides an array of built-in analyzers to control this process. It also allows for the building of a custom analyzer, or for creating arbitrary analyzer chains combining Lucene's tokenizers and token filters.

The final step is to *index the document*. During the indexing step, the document is added to the index. Lucene provides everything necessary for this step.

Searching

Once the index is created, we can perform searching tasks on it. Searching is the process of looking up words in an index to find documents where they appear.

To know what to search for, we need to *obtain the search query*. When the user interacts with the search application, he or she is presented with a user interface in the web browser or desktop application. Once a user interacts with the search interface, he or she has to be allowed to submit a search request. This request is then translated into an appropriate query object for the search engine. Lucene does not provide any default search user interface. It is entirely up to the application to create one. However, it does provide a query parser package to process the request into a query object according to a common search syntax.

After obtaining and processing the search request, Lucene is ready to *execute the search* request to retrieve documents matching the query, sorted in the requested sort order. There are several approaches to this that will be discussed later in this work. Lucene's approach combines the vector space and pure boolean models.

Presenting the search results is the final step of searching process. Once a set of documents that match the query are returned, sorted in the right order, the application has to display them to the user in an intuitive and consumable manner.

Lucene extensions

We saw that the indexing process consists of gathering content, extracting text, creating documents and fields, analyzing the text into a token stream, and adding to an index. We have also discussed the aspects of searching the index.

Beyond Lucene's core functionality, there are a number of extension modules that offer useful add-ons. These include the spellchecker and highlighter modules among others. These modules are housed in a contrib area, and developers may find them handy when creating their own search applications.

Chapter 4

Parsing XML Documents

Lucene, as a core search library, does not provide any functionality to support acquiring content. This has to be handled by the application, or a separate piece of software. This chapter is about parsing the structure of a document to extract the content information contained in the document. We will cover various parsing approaches, implement them in order to meet our objectives, discuss the advantages and disadvantages of each approach, and finally, make a decision about the most appropriate one.

4.1 Parsing Approaches

In order to obtain content from an XML file, we first need to apply a filter that removes the tag marks and other unwanted content. The process of analyzing a text in order to determine its structure is called *parsing*. Parsing is the most fundamental aspect of processing an XML document.

When an application parses an XML document, it may need to ensure that the document is well formed, or check whether the document conforms to a specified structure. In our case, we may skip these steps and focus on another aspect of parsing. We are going to need to access various elements and attributes specified in the document and obtain their textual content.

Several parsing approaches exist that can be used for obtaining the content. They differ in some major aspects and we are going to introduce them in more detail, see how they can be used in our case, and compare their respective performance.

DOM Approach

The first approach we are going to discuss is the Document Object Model (DOM). The DOM is a language independent convention for interaction with XML documents. It specifies interfaces for accessing and manipulating content and specifies the structure of generalized document. The DOM represents a document as a tree of `Node` objects. Some of these `Node` objects have child `Node` objects, others are

leaf objects with no children. The `Node` objects may be addressed and manipulated within the syntax of the programming language in use.

To represent the structure of an XML document, the generic `Node` type is specialized to other `Node` types, and each specialized `Node` type specifies a set of allowable child `Node` types. Sample `Node` types are listed in Table 4.1.

Type	Description
Document	Represents an XML document
Element	Represents an element
Attr	Represents an attribute
Text	Represents text, including whitespace

Table 4.1: Sample Specialized DOM `Node` Types

Under the DOM approach, an XML document is parsed into a random-access tree structure [3] in which all the elements and attributes from the document are represented as distinct nodes.

Note that in the DOM approach, the tree representation of the complete document has to be constructed before the document structure and content can be accessed. Afterwards, the document nodes can be accessed randomly and the random access to any tree node is fast. Parsing the complete document before accessing any node, however, reduces efficiency.

SAX Approach

An alternate way of reading XML documents is the push parsing approach. A push parser generates synchronous events as a document is parsed, and these events can be processed by an application using a callback handler model. An API (Application Programming Interface) for the push approach is available as SAX (Simple API for XML). SAX parsers operate on each piece of the XML document sequentially, therefore their use is recommended if no modification or random-access navigation of an XML document is required.

The SAX API defines a `Content Handler interface` [3], which may be implemented by an application to define a callback handler for processing synchronous parsing events generated by a SAX parser. You can see commonly used methods of `SAX Content Handler` in Table 4.2.

Note that compared to DOM parsers, SAX parsers have certain benefits. The quantity of memory that a SAX parser must use in order to function is typically much smaller. Processing XML documents can often be faster because of the event driven nature of SAX. The disadvantage of SAX parsers is that they process the document sequentially, and do not allow random access to the document.

Method	Description
startDocument	Start of a document
startElement	Start of an element
characters	Character data
endElement	End of an element
endDocument	End of a document

Table 4.2: Sample SAX Content Handler Event Methods

Commons Digester

A Commons Digester [4] is a way of parsing an XML document that avoids using either the DOM or the SAX API directly. The Digester allows the mapping of an XML document structure to an object model in an external XML file containing a set of rules telling the Digester what to do when specific elements are encountered.

Note that the Digester is a shortcut for creating a SAX parser and, therefore, has similar characteristics and similar performance as a SAX parser. The advantage of using a Digester is its ease of connection with Lucene.

StAX Approach

So far we have introduced two approaches. A tree-based DOM allows unlimited, random access and manipulation, while an event based SAX allows a single pass through the document. The pull model was designed as a median between these two opposites.

Under the pull approach, events are pulled from an XML document under the control of the application using the parser. StAX (Streaming API for XML) [3] is similar to the SAX API in that both offer event based APIs. However, StAX differs from the SAX API in several respects. Most importantly, in StAX it is the application rather than the parser that controls the delivery of parsing events. StAX can be used both for reading and for writing XML documents. Sample Event Types are shown in Table 4.3.

Event Type	Description
START_DOCUMENT	Start of a document
START_ELEMENT	Start of an element
CHARACTERS	Character data
END_ELEMENT	End of an element
END_DOCUMENT	End of a document

Table 4.3: Sample StAX Events Types

Note that the pull model StAX has an advantage over the push model SAX. In

the push model, the parser generates events as the XML document is parsed. In the pull model, the application generates the parse event and the parse events can be generated as required.

4.2 Parsing CoPhIR Data

CoPhIR Collection

The CoPhIR collection [10] is a test collection for scientific research on scalable similarity search techniques in the context of the SAPIR project. The collected data represents the world's largest multimedia metadata collection available for research purposes, containing visual and textual information regarding over 100 million images.

The images come from the Flickr¹ photo sharing site that also provides a significant amount of additional metadata about the photos hosted. Flickr assigns each photo a unique ID that can be used to unequivocally devise an URL for accessing the associated photo.

In addition to Flickr user information, MPEG-7 descriptors [10] were extracted using the MPEG-7 eXperimentation Model, the official software certified by the MPEG group that guarantees the correctness of the extracted features. For each entry of the collection we have the following pieces of data available [10]:

- The link to the corresponding entry into the Flickr web site
- The photo image thumbnail
- An XML structure with the Flickr user information: title, location, GPS, tags, comments, etc.
- An XML structure with five extracted standard MPEG-7 image features: scalable colour, colour structure, colour layout, edge histogram, and homogeneous texture

CoPhIR XML Structure

After introducing parsing approaches and CoPhIR collection, we are going to delve more deeply into the structure of CoPhIR XML files, and we are going to extract textual information from these files. Each photo from Flickr is described in one XML file. The root element in this file is `SapirMMObject` that has three child elements:

1. **MediaLocator**: holds identification information about the photo uniform resource identifier. This identifier allows us to link and retrieve the corresponding image on the Flickr web site.
2. **Photo**: contains the image textual data and metadata; for instance, its author, title, description, GPS location, tags, comments, view count, etc.

¹<http://flickr.com/>

3. Mpeg7: contains information related to the five standard MPEG-7 visual descriptors.

A simplified structure of the XML file is shown below and lists important elements, excluding their attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<SapirMMObject>
<MediaLocator>
  //Uniform resource identifier
</MediaLocator>
<photo>
  <owner />
<title>Title</title>
<description>Description</description>
<visibility />
<comments>No. of comments</comments>
<notes />
<tags>
<tag>Tag</tag>
...
</tags>
<urls>
<url>Url</url>
</urls>
  <comments>
    <comment>Comment</comment>
    ...
  </comments>
</photo>
<Mpeg7>
  //Multimedia content description
</Mpeg7>
</SapirMMObject>
```

All the aforementioned data are available for our indexing application. Thus, we might be able to index title and description, identification and location of the author, user-provided tags, comments of other users, GPS coordinates, notes related to portions of the photo, the number of times it was viewed, the number of users who added the photo to their favourites, the upload date, and, finally, all the information stored in the EXIF (exchangeable image file format) header of the image file. However, it is not wise to index every single piece of information. In order to minimize the index size and increase the search speed, we need to carefully pick the most important elements and attributes for indexing. The question is, Which pieces of data are the most relevant for our similarity search? .

We are going to need a unique identifier for each XML document. For this purpose, we can simply use the `mediauri` element from the XML file. The photo `title` and its `description` can be used to obtain the elementary information. Furthermore, the information from `tags` and `comments` is relevant for our similarity search. In addition to the information about the picture, we might need data about the owner (attributes `username` and `realname` of the owner element) and about the dates (attribute `taken` of the dates element).

The MPEG-7 data will not be needed since it only contains visual descriptors. Visual descriptors characterize a particular visual aspect of the image and they can be, therefore, used to identify images which have a similar appearance, but they are of no use when searching for specific textual information associated with the photo.

Performance

All approaches described earlier can be used for parsing XML documents. However, they may or may not be suitable in our case, when we need to walk through millions of XML documents and acquire certain pieces of information about all of them.

To decide on the best solution, we have designed parsers of all types that process XML files. The first part of processing the document is obtaining the file identifier and textual information from the following elements: photo `title`, photo `description`, all `tags`, and all `comments`. The second part of processing the file is adding the element content into a Lucene index.

The key performance indicator for us is the runtime. For comparison, it is sufficient to process only 200,000 XML files. Three different runs were executed on the same set of documents. Table 4.4 compares the runtimes of different parsing approaches.

	DOM	SAX	Digester	StAX
runtime 1	56.972 s	31.706 s	58.040 s	35.278 s
runtime 2	56.844 s	38.762 s	57.660 s	35.662 s
runtime 3	57.648 s	36.266 s	59.840 s	36.326 s
average	57.155 s	35.578 s	58.513 s	35.755 s

Table 4.4: Runtimes of different parsing approaches on 200,000 XML Files

We can see that DOM performance is bad since it is likely to be best suited for applications where the document must be accessed repeatedly or out of sequence order. We are not using these advantages of the navigation and random access and at the same time the DOM disadvantages appear in their full extent. The need to parse the entire document and memory intensity makes DOM an unacceptable solution.

An important aspect of our parsing is that we know exactly which elements and attributes we are interested in and we know their position in the documents. That allows us usage of the strictly sequential and one pass SAX approach that is

more efficient than DOM with lower memory consumption. There is no need for navigation, random access or modification, therefore the SAX approach is the best match for our needs with the best runtimes.

Although Digester is built on the basis of the push model and is supposed to perform well together with Lucene, its runtimes are comparable with the DOM approach, and therefore not reasonable for our purposes. Thus, its only advantage remains in ease of use and effortless connection with Lucene.

StAX has pros and cons similar to the SAX parser. No need for random access makes it a reasonable solution to our problem. Its low memory consumption makes its runtimes comparable to the SAX approach, yet they are a little bit longer. In conclusion, the push approach using the SAX API performs the best, and is going to be used and referred to further in this work.

Chapter 5

Analysis

No search engine indexes text directly: rather, the text must be broken into a series of individual atomic elements called *tokens*. A stream of tokens is the fundamental output of the analysis process [1]. In simple terms, a token corresponds roughly to an individual word of the text.

Each token carries with it its text value as well as some metadata. When we combine tokens with their associated field name, we get *terms*. Terms can be considered the most fundamental indexed representation of the original text and they are used during the search process to determine what documents match a given query.

The *analyzer* plays the most important role during the analysis process. It determines the division of textual fields into a series of tokens. The analyzer's responsibility is to handle problems that may appear with case sensitivity, compound words, spelling, synonym injection, or collapsing singular and plural forms.

The analyzer has full control over which tokens, extracted during the analysis, are the terms inserted into the index. This is a crucial characteristic since only the terms that are indexed become searchable.

Before we take a closer look at the analysis process inside Lucene, we would like to emphasize that analysis is not only used prior to indexing, but also prior to querying the index. The analyzer has to be applied to the search query entered by the user. It is a good habit to use the same analyzer for both indexed documents and user queries in order to obtain corresponding terms.

5.1 Lucene Analyzers

We have already indicated that Lucene indexes `Document` objects that are represented by collection of `Field` objects. Each `Field` object is a name and value pair. We need to realize that analyzers do not help in field separation, because they only deal with a single field at a time. Instead, parsing these documents prior to analysis is required as described in the previous chapter.

We may think of an analyzer as an encapsulation of the analysis process. The analysis process consists of number of operations performed on the initial text.

Analyzers differ from each other in number of operations they perform and in type of those operations.

Choosing the right analyzer is a crucial development decision. With Lucene, one may choose between using one of built-in analyzers, or creating a custom analysis solution. Lucene provides five core ready-to-use analyzers [1] that we are going to describe in more detail. No matter which analyzer we end up using, we need to keep in mind that the resulting tokens are very analyzer dependent.

Whitespace Analyzer

As the name implies, the **Whitespace Analyzer** performs only one operation, which is splitting the input text into tokens on whitespace characters. There are no other tasks, like lowercasing or removing redundant words, that would normalize the tokens.

Simple Analyzer

The **Simple Analyzer** is still very simple, yet it is smarter than the **Whitespace Analyzer**. Similar to the **Whitespace Analyzer**, it splits the tokens, however it uses non-alphabetic characters instead of whitespaces. This results in discarding all non-alphabetic characters including phone numbers, dates etc. The second operation that the **Simple Analyzer** performs is lowercasing each token.

Stop Analyzer

The **Stop Analyzer** is built upon the **Simple Analyzer**, therefore it performs the same operations. Furthermore, it removes common words. By default, it removes common words specific to the English language. The default set of stop words [1] follows: a, an, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or, such,that, the, their, then, there, these, they, this, to, was, will, with. This default set is used unless otherwise specified. Note that removing words decreases the index size, but can have a negative impact on precision querying.

Keyword Analyzer

We might say that the **Keyword Analyzer** performs no operations on the input text and treats the entire text as a single token. This may seem useless at first, but the **Keyword Analyzer** becomes handy when treating special fields that need to remain untouched, for example, identifiers.

Standard Analyzer

The **Standard Analyzer** is Lucene's most sophisticated core analyzer. It applies quite a bit of logic to identify certain kinds of tokens. It recognizes patterns of

company names, email addresses, acronyms, and hostnames. As a mature analyzer, it also removes stop words and punctuation and lowercases the tokens.

5.2 Field Options

Lucene indexes documents that contain fields. Fields are very important since they carry the actual value to be indexed. Numerous options [1] regarding the analysis and indexing process can be specified for each field, dictating to Lucene what to do with the fields when adding the document into the index.

It is appropriate to set up at least two basic options controlling the searchability and reachability of the field. Options for `Field.Index.*` control how the value of the field will be searchable, and if it will be searchable, after the index is created. Options for `Field.Store.*` determine whether the exact value of the field will be stored or not. Storing the value increases the index size, but allows for retrieving the value during searching without looking up the whole document.

The introduced options have many theoretical combinations. In practise, it is helpful to remember at least the configurations shown in Table 5.1 that list the most frequently used option combinations and their common usage.

Field.Index.*	Field.Store.*	Sample usage
NOT_ANALYZED_ _NO_NORMS	YES	Identifiers (filenames, primary keys), telephone and Social Security numbers, URLs, personal names, dates, and textual fields for sorting
ANALYZED	YES	Document title, document abstract
ANALYZED	NO	Document body
NO	YES	Document type, database primary key (if not used for searching)
NOT_ANALYZED	NO	Hidden keywords

Table 5.1: Sample option configurations along with common usage examples

Per Field Analyzer

The fact that a document is composed of multiple fields, with diverse characteristics, introduces some interesting requirements to the analysis process. The need for the use of different analyzers for different fields or skipping analysis entirely for certain fields is another frequently encountered analysis challenge. If the documents have diverse fields, it would seem that each field may require a unique analysis.

Lucene has a helpful built-in utility class, `Per Field Analyzer Wrapper` [2], that makes it easy to use different analyzers per field. We create a `Per Field Analyzer Wrapper` by providing it with a default analyzer. We may then choose to

assign each field a new analyzer. In case we do not assign any analyzer, the default one is automatically picked.

5.3 Analyzing CoPhIR Data

Analysis, while only a single facet of using Lucene, is the aspect that deserves the most attention and effort. Remember that only the tokens produced by the analyzer are searchable, unless the field is indexed with `Field.Index.NOT_ANALYZED` or `Field.Index.NOT_ANALYZED_NO_NORMS`, in which case the entire field's value, as a single token, is searchable.

Since we do not have any special needs, we may get along with a built in analyzer. For those applications that use a core analyzer, `Standard Analyzer` is likely the most common choice. `Standard Analyzer` [1] holds the honor of being the most generally useful built-in analyzer. A JFlex-based¹ grammar underlies it, tokenizing with cleverness for the following lexical types: alphanumeric, acronyms, company names, email addresses, computer hostnames, numbers, words with an interior apostrophe, serial numbers, IP addresses, and Chinese and Japanese characters. These lexical types are sufficient for our needs, and `Standard Analyzer` is therefore a clear choice.

In the chapter about parsing, we gave reasons for our choice of elements and attributes for indexation. Let us remember that we selected the following pieces of information: `mediauri` as an identifier, `title`, `description`, `tags`, `comments`, `username`, `realname`, and the date when the photo was `taken`. These values, extracted from CoPhIR XML files, are the inputs of our analysis process.

Each XML file from the CoPhIR dataset will be represented in a single Lucene `Document` comprising multiple `Fields`. We begin with creating the `Document`. Afterwards, we can create `Field` with values obtained from the XML element or attribute and add it to the `Document`. New fields are added into the same document until the end of the XML document is reached. The core of the actual code behind this process is shown below in a simplified way. Note that the whole process is preceded by defining an analyzer; in our case, `Standard Analyzer`. The analyzer is only defined once at the very beginning, since we use the same analyzer for all fields.

```
luceneDocument = new Document();
for (field : XMLFile) {
    Field luceneField = new Field(
        field.name,
        field.value,
        Field.Store.VALUE,
        Field.Index.VALUE);
    luceneDocument.add(luceneField);
}
```

¹JFlex: lexical analyzer generator, <http://jflex.de/>

Each field has to be assigned with values of `Field.Store.*` and `Field.Index.*` that determine whether the field will be stored in the index, and whether the field will be searchable via the inverted index. The `mediauri` is an identifier that we need to store in the index in order to uniquely define the document for further operations, like updates and deletions. We cannot allow any analyzer to change its value, and, in fact, we do not even need to make this field searchable.

We also wish to store the fields with `title`, `description`, `tag`, `username`, and `realname`. Unlike the `mediauri`, we want these fields to be analyzed, and thus made searchable. We also want to analyze and search `comments` that can provide further specifications regarding the photos. We need to realize that the comments may often be very long, and we do not need to store them in the index and retrieve them in their original form.

The last value we have to index is `taken`, the date when the photo was taken. We want to store this value in the index and we need the field to be treated as a single token. Fortunately, Lucene has an option for fields that we want to search on but should not be broken up. In addition to that, we may decide we do not need to store all necessary information to implement the vector space model on this value. This can be done in cases when the field is only used for filtering or sorting, but not for phrase searches. This option is frequently used on dates, and we are going to use it as well. The specific configurations for all fields are shown in Table 5.2.

field name	<code>Field.Store.*</code>	<code>Field.Index.*</code>
<code>mediauri</code>	YES	NO
<code>title</code>	YES	ANALYZED
<code>description</code>	YES	ANALYZED
<code>tag</code>	YES	ANALYZED
<code>comment</code>	NO	ANALYZED
<code>username</code>	YES	ANALYZED
<code>realname</code>	YES	ANALYZED
<code>taken</code>	YES	NOT_ANALYZED_NO_NORMS

Table 5.2: Configurations used for particular fields

Note that our documents have a `tag` field, but sometimes there is more than one tag for a document. There are basically two ways of handling such a situation. The first one would be to loop through all the tags, appending them into a single `String` that can be inserted into a single field. Another, perhaps more elegant way is to keep adding the same field with different values. This solution creates what are called *multivalued fields* [1]. Internally, whenever multiple fields with the same name appear in one document, the tokens of the field will be logically appended to one another, in the order the fields were added. The same approach is also used with `comments`, for they legitimately have multiple values.

We have discussed the main aspects of analysis, but have another step ahead. The following chapter will introduce some magic behind the indexing procedures and describe the CoPhIR data indexing process, including its performance measures.

Chapter 6

Indexing

The process of *indexing* [1] can be described as processing the original data into a highly efficient cross-reference lookup in order to facilitate rapid searching. The conversion process is called indexing, and its output is called an index. During the indexing step, the document is added to the index, and thus made searchable.

6.1 Lucene Indexing

Lucene is often compared to a database, because both can store content and retrieve it later. But there are important differences. The first one is Lucene's flexible schema. In a database, each table is described by its schema that defines the names of attributes, their domains, and other constraints. Data that do not conform to the table structure cannot be held in the database. Thanks to its flexible schema, Lucene's single index can hold documents that represent different entities.

The second major difference between Lucene and databases is that Lucene requires the content to be flattened, or denormalized, during indexing. No nesting or recursion is allowed within a Lucene index. Relational databases allow an arbitrary number of joins, via primary and secondary keys, relating tables to one other, yet Lucene documents are flat. Potential recursions and joins have to be denormalized when creating the index, which may result in the replication of the same pieces of information again and again.

Inverted Index

To quickly find terms in the document collection, we need an efficient cross-reference lookup. Lucene scans the entire collection first, identifying all the unique terms, building a structure known as an *inverted index* [1]. The set of unique terms is usually referred to as a *dictionary*. For each dictionary entry, a *posting list* is created. The posting list is a list of documents that contain a given term.

The structure is inverted because it uses tokens extracted from input documents as lookup keys, instead of treating documents as the central entities. The inverted

index concept is similar to classical book indexes; the book index references the page number(s) where a given word or phrase occurs. The structure enables quick answers to the question “Which documents contain term x ?”. The structure of an inverted index is illustrated in Figure 6.1.

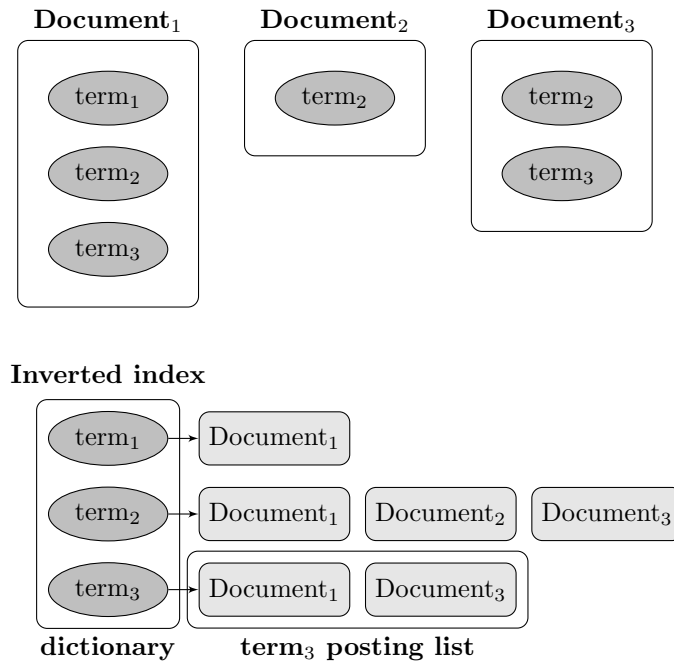


Figure 6.1: Inverted index structure for given documents

When Lucene builds the inverted index, by default it stores all necessary information to implement the vector space model. Additionally, an entry in the posting list contains a manually or automatically assigned weight for the term in the document. This weight is used in computations that generate a measure of relevance to the query. This model requires the count of every term that occurred in the document, as well as the positions of each occurrence. The position of the term in the document is also used when facilitating proximity searches or phrase searches.

Clearly, the construction of an inverted index is expensive [5], but once built, queries can be efficiently implemented. Upon receiving a query, the index is consulted, the corresponding posting lists are retrieved, and the algorithm ranks the document based on the contents of the posting lists. Note that an inverted index makes efficient use of disk space while allowing quick keyword lookups.

Index File Formats

In this section we would like to present definitions of the index file formats used in Lucene version 3.0 including their internal structure. The fundamental concepts in

Lucene are index, document, field and term in the following hierarchy:

- An index contains a sequence of documents.
- A document is a sequence of fields.
- A field is a named sequence of terms.
- A term is a string.

Internally, Lucene refers to documents by an integer document number. The first document added to an index is numbered zero, and each subsequent document added gets a number one greater than the previous. Note that a document's number may change. This can happen especially when merging segments of one index or when deleting documents and subsequently removing the numbering gaps.

Remember that the same string in two different fields is considered a different term. Thus, terms are represented as a pair of strings, the first naming the field, and the second containing the text within the field.

Furthermore, an index can be divided into more segments. The index (or its segments respectively) maintain the following:

- Field names - the set of field names used in the index
- Stored field values - pairs of field names and values (keyed by document number)
- Term dictionary - all terms used in all indexed fields of all documents, the number of documents containing the term, and pointers to the term's frequency and proximity data
- Term frequency data - numbers of all the documents containing that term, and the frequency of the term in each document
- Term proximity data - positions in which the term occurs within each document
- Normalization factors - a per field value that is multiplied into the score for hits on that field
- Term vectors - a per field value consisting of the term's text and term's frequency
- Deleted documents - an optional file indicating which documents are deleted

Boosting

Lucene indexes documents that may represent different entities. A mechanism is available allowing us to assign higher importance to some documents. We can even assign higher importance to individual fields within a document. Hence, not all documents and fields are created equal.

Boosting [1] may be done during indexing or during searching. Let us now delve deeper into the index time boosting. The default boost factor is 1, and it can be changed to any number, as long as it stays positive. By setting the boost factor lower than 1, the document or field is assigned lower importance. The importance is relative with respect to other documents in the index when computing relevance.

Indexing Numbers

Let us now take a closer look at how Lucene supports indexing numeric fields in order to facilitate efficient range searching and numeric sorting. Note that when talking about numeric values we also mean dates and times, since their values can be easily converted to an equivalent int or long value type, and then indexed as numbers. When a numeric value is encountered, it is indexed using a *trie structure* [1].

The term trie comes from *retrieval* and trie structure is, in fact, a prefix tree, where all descendants of a certain node share a common prefix. Trie logically assigns a single numeric value to larger and larger predefined brackets. At search time, the requested range is translated into an equivalent union of these brackets, resulting in a high-performance range search or filter.

Index Segments and Optimization

When building an index, especially when indexing many documents or using multiple sessions, many separate segments are created. Each segment is a standalone index, holding a subset of all indexed documents. When searching a segmented index, Lucene has to walk through each segment separately and merge the results. It works perfectly as long as there are only a few segments and the whole index is not too large. After exceeding tolerable amount of segments or the index size, the performance is cut back noticeably.

Indexes evolve by either creating new segments for newly added documents, or by merging existing segments. To reduce performance issues, optimization can be performed that merges several index segments into a single segment. After optimization completes, the index will consume less disk space, and the searches will be faster than at the beginning. Note that during optimization, the index requires substantial temporary disk space, up to three times its starting size. Optimization only improves searching speed, not indexing speed.

You saw Lucene's conceptual model for documents and fields, including a flexible but flat schema (when compared to a database). Let us now delve into the details of our implementation of an index and look at some interesting statistics.

6.2 Indexing CoPhIR Data

The disk space requirement for our CoPhIR collection [10] consists of 231.4 GB for the XML data, 51.0 GB for the image content index, and 335.4 GB for the image thumbnails. These numbers are slightly smaller compared to the original CoPhIR collection sizes. The reason for this is that the data available at the Faculty of Informatics is not complete and the collection there consists of only 100 million XML files instead of the original 106 million.

Our objective is to index the XML files that store descriptive data about the photos. The statistics state that each image is associated on average with 0.52 comments and 5.02 tags. However, the distribution of comments and tags among

images is highly skewed, and follows a typical power law. It turns out that the 87.76 % of the images have no comments, and 29.96 % of the images have no comments and no tags associated with them, while only the 14.16 % of the tagged images have at least one comment. Only 1.14 % of the images have at least six comments and six tags associated with them.

The distribution of comments and tags between the photos is irrelevant regarding indexing. On the other hand, we need to realize it may influence the searching. The more information associated with the picture, the easier it should be to find it, assuming the information is relevant. We can see that almost one third of the photos have no tags or comments associated with them. These photos will be difficult to find and the chance of hitting them will depend on the accuracy of their title and description.

We know we have 245.3 GB of original XML data, and our goal is to index it. Since the analysis is already done, we have some easy work to do here. We need to instantiate `IndexWriter` and assign it four parameters; the directory for index storage, the analyzer to use, whether the index is newly created or not, and whether or not to limit field lengths. In our case, we will create the index newly and leave the field lengths unlimited.

Once the writer is instantiated, we call an `addDocument` method for each `Document` we want to add to the index. At the end, we have to remember to close the writer. The closure commits all changes to an index and closes all associated files. Simplified code that is responsible for index creation resembles the following:

```
IndexWriter writer = new IndexWriter(  
    directory,  
    new StandardAnalyzer(Version.LUCENE\_CURRENT),  
    true,  
    IndexWriter.MaxFieldLength.UNLIMITED);  
for (LuceneDocument : documentCollection) {  
    writer.addDocument(luceneDocument);  
}  
writer.close();
```

Indexing of such an amount of data is expected to take a few hours, and it would not be wise to wait until its end to save and make visible all changes. Instead, we can make the changes visible sooner by calling the `commit` method. This is done after every million XML files is indexed. It does slow down the indexing process a little bit, but on the other hand it guarantees data integrity in case an exception or an error occurs. The method commits all pending changes, including added and deleted documents, optimizations, segment merges, added indexes, etc. to the index, and synchronizes all referenced index files, such that a reader will see the changes.

```
writer.commit();
```

Many segments are created during the indexing. The more segments the index includes, the longer the searches take. Fortunately, we can request Lucene to execute an optimize operation on an index, priming the index for the fastest available search. Using the default merge policy means merging all segments into a single segment; individual merge policies may implement optimize in different ways. It is recommended that this method be called upon completion of indexing, and that is why we are going to call this method right before we close the writer.

```
writer.optimize();
```

It is now time to think about future usage of the index, since there are several possibilities regarding how to create it, all of them having some advantages, as well as some disadvantages. The first option suggests itself. We have earlier described the elements and attributes from the XML files that we want to index. A simple approach would be creating corresponding fields within the document, but there is another approach that might cross our mind, and that is creating single document field, joining together all the distinct values, no matter which element they come from.

What are the pros and cons of these approaches? Creating a single field for all values would result in less terms, because a term is a field and value pair. The fewer terms an index contains, the smaller it is, and the faster the search time it should achieve. A smaller index should also be faster to create.

On the other hand, separate fields for distinct values provide potential scalability of the search. The more fields the index contains, the more advanced the search can be, providing the users with options to search by different fields. Furthermore, the results can be more precise, since shorter fields, like tag or name, are by default assigned higher importance.

To obtain relevant numbers and make a responsible decision about which indexes to use, we will create both of the aforementioned indexes. The index with a single field will join the field's `title`, `description`, `tags`, `comments`, `username`, and `realname` into a single one, leaving only fields `mediauri` and `taken` separate. The index with separate fields isolates each distinct attribute and element.

	single field	separate fields
real	324 min 42.185 s	345 min 3.324 s
user	325 min 59.298 s	334 min 27.386 s
system	12 min 11.730 s	13 min 57.280 s
indexing time	305 min	319 min
optimize time	20 min	26 min

Table 6.1: Comparison of indexing times for different indexes

The Table 6.1 lists the indexing times for both indexes. The runtime is over five hours in both cases. The index unifying more fields into a single one is slightly

faster to create and also to optimize after creation. Remember, that the optimize function merges multiple segments into one, resulting in a faster search of the index.

To complete the information about created indexes, the index size has to be presented. The index with a single field is smaller, as expected, having a total size of 17.52 GB. The larger index with separate fields is 18.19 GB in total. The aggregate numbers are important, and there is the last piece of information missing to round off the indexing process; the progress of the index creation will be introduced shortly.

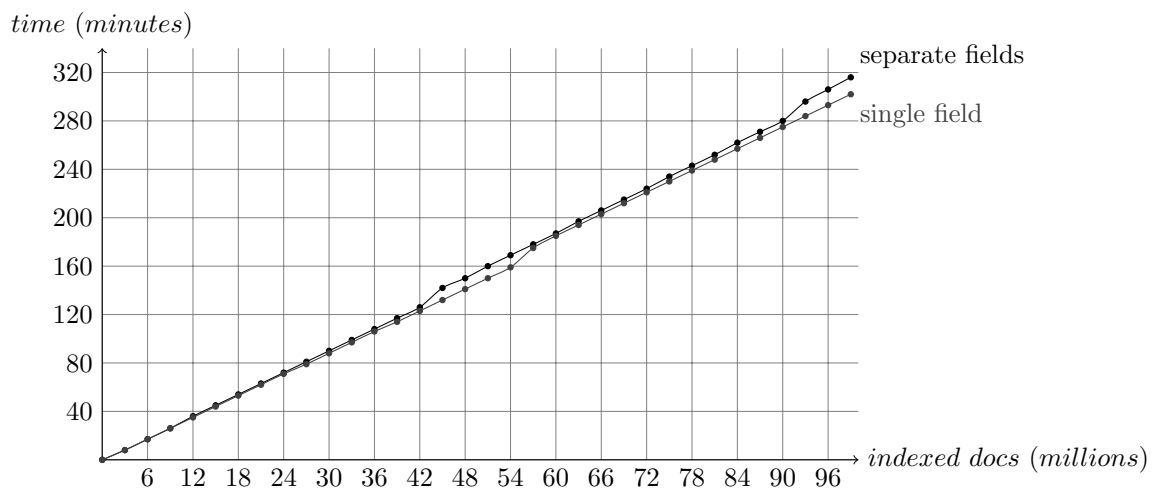


Figure 6.2: Progress of indexing

Figure 6.2 maps the progress of indexing performance for both indexes. The XML files are being streamed from .tar (tape archive) files, where each .tar file contains approximately one million XML files. The .tar file format as a form of an archive bitstream is often used together with a compression method, such as gzip (GNU zip). These files are appropriate for sequential reading that is performed during the indexing process.

The graph shows that the indexing time grows linearly and that there is not a big difference between the two indexes. The linearity is caused by the method of how the documents are added into the index. The index is not created as a single file: Instead, segments are created that hold only a subset of the index. When the number of segments exceeds the limit, or whenever the optimize method is called, the segments are merged into a single compound file. In this case, the optimize method is called at the very end, and Table 6.1 shows that the merging takes non-trivial time (about 20 minutes). After this process, the index is optimized and ready to use.

In order to use the index, there is only the last step missing, which is the search. The next chapter introduces the Lucene scoring formula and the usage of different queries for obtaining relevant results.

Chapter 7

Searching

Searching [1] is the process of looking up words in an index to find documents where they appear. The quality of a search is typically described using precision and recall metrics. Recall measures how well the search system finds relevant documents; precision measures how well the system filters out the irrelevant documents.

Precision [5] is the ratio of the number of relevant documents retrieved to the total number retrieved. It indicates the quality of the answer. Recall [5] is the ratio of the number of relevant documents retrieved to the total number of documents in the collection that are believed to be relevant. Recall considers the total number of relevant documents. Figure 7.1 illustrates the relationship between relevant, retrieved, and relevant retrieved documents.

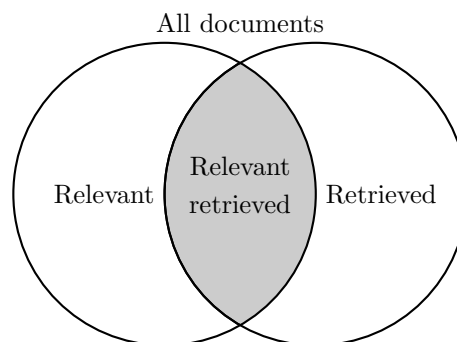


Figure 7.1: Relationship between relevant, retrieved and relevant retrieved documents

$$\text{Precision} = \text{Relevant retrieved} / \text{Retrieved}$$

$$\text{Recall} = \text{Relevant retrieved} / \text{Relevant}$$

A perfect retrieval system [6] would retrieve only the relevant documents and no irrelevant documents. However, perfect retrieval system does not exist, and will

never exist, because search statements are necessarily incomplete, and relevance depends on the subjective opinion of the user.

A typical search method visits every single document that is a candidate for matching the search, and only accepts documents that meet every condition of the query. Finally, it gathers the top results and returns them to the user.

Any information retrieval system is only as good as its search capabilities. It is not only about searching, though. Crucial conditions for initiating a search include having a well-prepared index, as discussed in previous chapter.

7.1 Lucene Document Searching

Documents indexed by Lucene typically have quite a few fields, such as title, author, date, abstract, text, keywords, etc. When searching for a term, you have to specify which field you want to search. This is a great feature for advanced searches where you need to provide different searches by author, by title, etc. Sometimes this could be tricky, because ordinary users have no idea about the fields used internally, and we do not even want them to have this inside look. In this case, it might be a good solution to create a catch-all field and use it to combine all of the text into a single field for searching.

Another source of confusion presented is the store option. When retrieving a document from the index, only the stored fields are present. Fields that were indexed but not stored will not be returned. To obtain these fields, we have to access the original documents directly.

7.2 Lucene Queries

Dealing with human-entered queries is the primary purpose of search engines. The queries may be composed of a single term, or they can be very rich and complex, depending on what the user's information needs are.

The query is passed to the searcher that first activates the `QueryParser` to preprocess the query. To allow the parser the query preprocessing, syntax has to be defined that the user has to respect. Assuming the query is syntactically correct, the `QueryParser` passes the query to the analyzer to break pieces of the query into terms. These are then compared to indexed terms, and the resulting documents are returned. We have already mentioned that in most cases, the same analyzer has to be applied to the search query that was used to create the index itself.

Lucene provides a diverse selection of built-in queries. These search queries include¹ searching by specific term, by range (numeric or textual), by prefix or wildcard, by phrase, or by fuzzy term matching. For the queries, Lucene offers a standard syntax that most users are familiar with. The syntax overview is shown in Table 7.1. Note that some characters have special functions. When we want to use

¹http://lucene.apache.org/java/2_3_2/queryparsersyntax.html

them as a part of our query, we need to escape them using the backslash character (\). The characters that require escaping are as follows: \+ - ! () : ^] { } ~* ?

Search	Syntax
Single terms	Lucene
Phrases	"Lucene in Action"
Fields	field_name:"Lucene in Action"
Wildcard searches	te?t, test*
Fuzzy searches	roam~
Proximity searches	"jakarta apache"~10
Range searches	field_name:[1 TO 100], field_name:{Aida TO Carmen}
Boosting a term	jakarta^4 apache
Boolean operators	AND, OR, NOT, +, -

Table 7.1: Different search types and their syntax

7.3 Lucene Scoring

Every time a document matches during a search, Lucene computes a score (a numeric value of relevance) and assigns the score to the document. The score reflects how good the match is. Higher scores reflect stronger similarity.

There are three common theoretical models of search: the pure Boolean model, the vector space model, and the probabilistic model. Lucene’s approach combines the vector space and pure Boolean models, and offers you controls to decide which model you’d like to use on a search-by-search basis.

Earlier in this work we have introduced several information retrieval strategies. Before we reveal what is under the hood of Lucene, let us remember two of those strategies.

The Boolean model needs a query in conjunctive normal form. Then it simply returns a set of matching documents. There is no relevance scoring; the result is simply match, or no match.

In a vector space model, the documents and queries are represented as vectors embedded in a high-dimensional Euclidean space, where each term is assigned a separate dimension. The occurring term has non-zero values, and omitted terms are assigned zero values. The similarity, or relevance, is a vector distance between the query and the document.

In other words, we can say that Lucene uses Boolean model approval with vector space model scoring. The similarity scoring formula uses TF-IDF weight to measure the similarity between a query and each document that matches the query. We are now going to present Lucene’s similarity scoring formula. The score is computed for each document (d) matching each term (t) in a query (q).

$$score(q, d) = coord(q, d) \cdot queryNorm(q) \cdot \sum(tf(t \in d) \cdot idf(t)^2 \cdot boost(t.field \in d) \cdot lengthNorm(t.field \in d))$$

We can see that TF (term frequency) and IDF (inverse document frequency) are calculated and somehow normalized. To understand better the magic of Lucene scoring, we need to describe all the functions used in the formula. All of them are listed in Table 7.2.

Function	Explanation
$tf(t \in d)$	normalized number of times the term appears in a document (term frequency)
$idf(t)$	total number of documents/ number of documents containing the term (inverse document frequency)
$coord(q, d)$	how many of the query terms are found in the document (coordination factor)
$queryNorm(q)$	normalizing factor assuring comparable scores between queries (normalization value for query)
$boost(t.field \in d)$	index time boost, specified during indexing
$lengthNorm(t.field \in d)$	normalizing factor assigning shorter fields bigger boost (normalization value of a field)

Table 7.2: Explanation of functions used in Lucene scoring formula

The computed score is a raw score, which is a non-negative floating-point number. It is a good manner to normalize the scores by dividing all scores by the maximum score for the query.

7.4 Searching CoPhIR Data

This section describes the process of searching in a created index of CoPhIR data. After introducing a single query search, we will delve into large-scale searching and measuring the performance of the searches. At the end, the user interface for entering queries and obtaining results is presented.

To perform the search operation, the target directory containing the index has to be passed as a parameter to the `IndexReader` class. In the case of the CoPhIR collection, the data are static and the reader will only be used to read the index, therefore the second parameter is `read-only true`. After obtaining the query, the query is parsed and passed to the searcher. The searcher searches the index and returns the top documents matching the query. Sample source code is shown below.

```
IndexReader reader = IndexReader.open(FSDirectory.open(file), true);
Searcher searcher = new IndexSearcher(reader);
```

```

QueryParser parser = new QueryParser(Version.LUCENE_CURRENT,
                                     fieldname, analyzer);
Query query = parser.parse(strLine);
TopDocs topDocs = searcher.search(query, 10);

```

It is great to have a search available, but there is a clear need to measure the performance of searches. Two different indexes were described in previous chapter, and it is now time to compare their search capabilities.

Search Benchmark

Once the indexes are built, the most straightforward way to measure and compare their response time is to prepare and run a benchmark. A benchmark, in general, is the act of running a computer program, a set of programs, or other operations, in order to assess their relative performance. With respect to information retrieval, the benchmark is a set of queries that are executed as a batch. The response times are measured that serve for further elaboration. This section describes the created Lucene benchmark in more detail.

The goal of the Lucene benchmark is to run a number of queries, measuring the response time of every single query. The main concern lies in the performance of parsing the query and searching the index. Therefore the reader is instantiated only once, at the very beginning, and it is closed after all searches are executed.

Expected queries within further usage are predominantly one or two term queries. These are the most important and should be tested in the first place. To obtain relevant results, a series of queries have to be searched: A reasonable number would be 100 of each. To create the test set of queries, the official list of popular tags from Flickr was used² for one term queries; two term queries were inspired by them, too.

The queries were executed on both aforementioned indexes: the first one having separate fields for distinct values, the second one unifying all values into a single field. The measurement was executed twice; Before the first run disk caches were cleaned, and for the second run the content of the caches was retained. The actual measurements are summarized in Table 7.3.

	separate (no cache)	separate (cached)	single (no cache)	single (cached)
1 term query	0.268 s	0.079 s	0.061 s	0.030 s
2 term query	0.331 s	0.151 s	0.121 s	0.084 s
average	0.300 s	0.115 s	0.091 s	0.057 s

Table 7.3: Measured search times of both indexes, with and without cached data

²<http://www.flickr.com/photos/tags/>

The index with a single field unifying values from distinct elements and attributes was expected to respond faster, and this presumption was proved by the benchmark measurements. The average search time was 91 milliseconds with empty caches, and 57 milliseconds with cached data.

It has been proven that caching plays an important role within the search. The response times with cached data were about half of the response times with cleared caches. Another observation made concerns the appropriateness of individual indexes. The index with single field performance is about twice higher, resulting in half response times. The main reason for this high performance difference lies in the way the terms are stored and searched within the Lucene index. Remember, that term, as represented in the index, is a field name and value pair. When searching across multiple fields, more terms have to be consulted to obtain the result.

User Interface

To make use of the search abilities, users have to be able to simply enter their requests and obtain the results in a transparent form. This is the main responsibility of the *user interface*. It was mentioned that Lucene itself does not provide any user interface and it is up to the application to create one.

One of the objectives of this thesis is to integrate the Lucene search into an existing MUFIN system. The existing system already disposes with user interface, accessible via any internet browser at <http://mufin.fi.muni.cz/imgsearch/>. There are no complaints about this interface, therefore it is going to be used in a full extent for the purposes of this work. The appearance of the user interface is shown in Figure 7.2.



Figure 7.2: MUFIN user interface

One of the ways of incorporating the Lucene search into an existing interface, is creating a *web service*. As the name suggests, a *web service* [12] is a kind of web application, that is, an application typically delivered over HTTP (Hyper Text Transfer Protocol). A web service is thus a distributed application whose compo-

nents can be deployed and executed on distinct devices. The client of a web service is also known as a consumer or requester.

An `Http Server` class implements a simple HTTP server [13] that is bound to an IP address and a port number, and listens for incoming TCP connections from clients. An `Http Context` then describes mapping between a root URI path and an `Http Handler` that is invoked to handle those requests targeting the path. The sample code below illustrates creation of HTTP server and creating the search context.

```
HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);
server.createContext("/search", new MyHandler(searcher, parser));
server.start();
```

The `Http Handler` instantiates the query parser and performs the search. The search results are then returned to the requester for further processing and displayed to the user. The information returned by the handler include `mediauri` as an image identifier, and the relevance of the image, also known as a score. The existing interface presents the actual images in a well arranged manner.

Searching and presenting results to the user is the last part of the information retrieval process, as perceived in this thesis. The last chapter reminds us of the crucial principles of information retrieval with respect to the Lucene technology and summarizes the observation made during parsing, analysing, indexing, and searching the CoPhIR data.

Chapter 8

Conclusions

The main objectives of this thesis were to study the Lucene technology, to create an index of CoPhIR data, and to integrate the search into the MUFIN application, in order to allow its users content-based image retrieval. The following paragraphs briefly summarize the process of achieving these goals.

Initially, background knowledge on the architecture of search applications was presented, as well as the core Lucene knowledge. It should be clear now that Lucene is an information retrieval library, not a ready-to-use standalone product, and that it does not contain a data parser, document filter, or a search user interface.

The introductory chapters provided a general overview of information retrieval and Lucene. Each successive chapter systematically delved into a specific area, beginning with parsing the input data. Several parsing approaches were tested. The SAX approach was proved to be faster than DOM, StAX, and Digester. It was established to use the SAX parser for obtaining the required pieces of information from the CoPhIR descriptive XML files.

Having the indexation data prepared, analysis came into play. The **Standard Analyzer** was chosen, as the most advanced built in analyzer, to analyze the data from the parser. The decision was made about which values should be made searchable and stored in the index.

After an explanation of what happens to the text indexed with Lucene, two indexes were created. The first of them conformed to the structure of original XML files and had a separate field for every distinct element or attribute. On the other hand, the second index only had one field for identifier and a single field for each distinct value. The latter index took 305 minutes to create and consumed 18.19 GB of disk space after optimization.

The searching is where Lucene shines; it was shown in the chapter dedicated to searching. A benchmark was created to run a batch of 200 different queries. The best achieved search times ranged around 57 milliseconds with the best configuration. The work was successfully rounded off by adding the search to the MUFIN application.

The goals of this thesis were accomplished. Theoretical background of infor-

mation retrieval was researched, and a functional content-based image search was prepared and integrated within the MUFIN search. The work briefly described each of the Lucene classes used in the certain task, and showed snippets of code where necessary.

The final section was devoted to discussing the future of the CoPhIR collection index. The possible further steps were suggested, and improvements were indicated in order to improve Lucene's performance. These recommendations do not relate only to the CoPhIR index, but they can be applied in any similar project.

8.1 Improvements and Further Steps

One of the steps taken in improving Lucene's performance could be adding threads and concurrency. Threads can be used during both indexing and searching. There is no need to use threading during indexing with a static set of data, but threading might help to improve search performance once it becomes necessary. The only way to find the right number of threads is empirical. Generally, the throughput improves by adding more threads. When hitting the right number of threads, adding more will not improve the throughput due to context switching costs. Lucene claims to be thread-safe [1] and the important classes including `Index Searcher` can be shared across multiple threads.

Other simple performance-tuning steps can be added. Using the latest release of both Lucene and Java is good practice, as well as not reopening the `Index Reader` any more frequently than absolutely required. The next steps according to [1] are hardware improvements, and they include usage of solid-state disks, more physical memory and budgeting memory, CPU, and file descriptors for peak usage.

Lucene has its restrictions and it may turn out that it will reach its limit one day. Fortunately, a more mature solution from the Apache Foundation exists. Solr is an open source enterprise search platform from the Apache Lucene project. The transition from Lucene to Solr should be easy, since they use the same core.

Solr uses the Lucene Java search library at its core for full-text indexing and search. In addition to standard Lucene, it provides several advanced features that may become handy when dealing with large indexes. Solr [11] offers highly scalable distributed search with sharded index across multiple hosts and user-extensible caching performance optimizations. It provides server statistics including statistics on cache utilization, updates, and queries. These statistics form a basis of background autowarming, a process that re-populates the most recently accessed items in the caches of the current searcher in the new searcher, enabling high cache hit rates across searcher changes. Solr is enriched with geospatial search, it can handle multiple search indices and provides real data schema with numeric types and unique keys.

Bibliography

- [1] Michael McCandless, Erik Hatcher, Otis Gospodnetić *Lucene in Action, Second Edition*. Manning, 2010, 475 pages.
- [2] Otis Gospodnetić, Erik Hatcher *Lucene in Action*. Manning, 2004, 421 pages.
- [3] Ajay Vohra, Deepak Vohra *Pro XML Development with Java Technology*. Apress, 2006, 451 pages.
- [4] Timothy M. OBrien *Jakarta commons cookbook*. O'Reilly Media, Inc., 2005, 377 pages.
- [5] David A. Grossman, Ophir Frieder *Information retrieval: algorithms and heuristics*. Springer, 2004, 332 pages.
- [6] Ayşe Göker, John Davies *Information Retrieval: Searching in the 21st Century*. John Wiley and Sons, 2009, 295 pages.
- [7] Amit Singhal *Modern Information Retrieval: A Brief Overview*. Bulletin of the Technical Committee on Data Engineering (Vol. 24 No. 4). IEEE Computer Society, December 2001, <http://singhal.info/ieee2001.pdf>.
- [8] Michal Batko, David Novák, Pavel Zezula *MESSIF: Metric similarity search implementation framework*. DELOS Conference, ser. LNCS, vol. 4877. Springer, 2007, pp. 1–10.
- [9] Michal Batko, Vlastislav Dohnal, David Novák, Jan Sedmidubský *MUFIN: A Multi-Feature Indexing Network*. 2nd International Workshop on Similarity Search and Applications. Los Alamitos, CA 90720-1314 : IEEE Computer Society, 2009, pp. 158-159.
- [10] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, Fausto Rabitti *CoPhIR: a test collection for content-based image retrieval*. CoRR, vol. abs/0905.4627, 2009, <http://cophir.isti.cnr.it/>.
- [11] David Smiley, Eric Pugh *Solr 1.4 Enterprise Search Server*. Packt Publishing, 2009, 336 pages.
- [12] Martin Kalin *Java web services: up and running*. O'Reilly Media, Inc., 2009, 297 pages.
- [13] Jeff Friesen *Beginning Java Standard Edition 6 platform: From Novice to Professional*. Apress, 2007, 485 pages.