Zürcher Hochschule für Angewandte Wissenschaften

Seminar Information Retrieval

Evaluierung der Retrieval-Leistung einer Search Engine am Beispiel einer privaten MP3-Sammlung

Author:
Philipp Schalcher

Betreuer:
Ruxandra Domenig

9. Mai 2014

Danksagung

Ich möchte mich an dieser Stelle bei allen bedanken, die mich bei dieser Arbeit tatkräftig unterstützt haben. An erster Stelle bedanke ich mich bei Frau Ruxandra Domenig. Das spannende Thema zur Evaluierung der Retrieval Leistung von Lucene anhand MP3-Dateien wurde von Beispielen aus ihrer Sammlung inspiriert. Auch der kleine Exkurs über die Suche nach Ton wurde von ihr vorgeschlagen. Dafür möchte ich mich nochmals herzlich bedanken.

Dann möchte ich Ramona Heiz danken, welche mir als Lektorin gedient hat. Ohne sie wäre dieses Dokument wahrscheinlich unlesbar geworden. Geduldig arbeitete sie sich durch das Dokument und half mir das Bestmöglichste hervorzubringen. Danke für deine Geduld und Arbeit.

Inhaltsverzeichnis

Zusammenfassung

Google hat es vor gemacht. Sie revolutionierten den Gebrauch des Internets mit ihrer Suchmaschine. Der Benutzer hat sich mittlerweile so fest daran gewöhnt (immerhin entstand das Verb googlen), dass sie diesen Komfort nicht mehr missen möchten.
Dokumente von Benutzer müssen auch so durchsucht werden können. Hier bietet Lucene eine mögliche Lösung an. Lucene ist eine Search Engine, welche frei zur Verfügung steht. Sie kann in die Applikation direkt eingebaut werden. Da Benutzer meistens viele MP3-Dateien besitzt, und heute die Lagerung auf CD nicht mehr aktuell ist, ist eine anständige Suchmaschine in einer Musikverwaltung ein muss. Lucene arbeitet mit verschiedenen Analyzern, die Text erkennen und wichtige Inhalte in einen Index schreiben. Bei MP3-Dateien zielt dies auf die ID3-Tags ab, welche in den Dateien eingebettet sind. Je nach Analyzer ist dauert die Indexierung länger, oder die Suchleistung wird verbessert. Das heisst, dass auch Suchanfragen bei einem Analyzer Ergebnisse liefern, bei einem anderen hingegen nicht.
Königsdisziplin im Falle von Musik wäre dann Shazam. Der Musik-Identifizierungsdienst kann mit nur wenigen Sekunden eines Liedes den richtigen Eintrag in einer Datenbank finden und gibt darauf die Informationen preis. Mit einem akustischen Fingerabdruck, der auf spektralen Flachheiten basiert, können so eine Art Hashwert generiert werden, welche mit dem Inhalt der Datenbank abgeglichen wird.
Zum Schluss bleibt, dass Lucene und Information Retrieval eine Wissenschaft für sich sind. Es braucht viel Zeit und Wissen, um das Beste aus Lucene zu holen. Kann man diese investieren und hat ein gewisses Interesse am Themengebiet Information Retrieval, so bietet Lucene ein mächtiges Werkzeug, welches andere Produkte sehr gut ergänzt.

Kapitel 1

Einleitung

In der heutigen Zeit wird der Mensch von einer Fülle an Informationen überflutet. Würde er nicht gewisse Eindrücke selber filtern, könnte das zu einem Kollaps führen. Der Mensch hat das Glück, solche Dinge von der Natur eingebaut zu haben. Im Gegensatz zum Menschen besitzen Informationssysteme keine integrierten Filter. Das beste Beispiel hierfür ist Google. Es gibt eine riesige Menge an Daten, die der Suchmaschine ihr Wissen verleiht.

Lucene ist eine Bibliothek, welche in verschiedene Projekte eingebaut werden kann, um so eine mächtige Suchmaschine auf Basis von Indexen zu bekommen. Lucene enthält alle relevanten Funktionen, die benötigt werden, um Informationen zu durchsuchen. Hier liegt die Herausforderung, eine Suchmaschine für ID3-Tags von MP3-Dateien zu bauen, da Lucene hauptsächlich für Textdateien (PDF,TXT,DOCX,eBooks,usw.) genutzt wird. Für MP3-Dateien stehen andere Probleme an (Wie extrahiere ich die ID3-Tags aus einer MP3-Datei).

Diese Arbeit soll die Information Retrieval Leistung der Suchengine Lucene analysieren. Darin enthalten sind eine Programmierung einer kleinen Suchmaschine, die MP3-Dateien innerhalb eines Ordners indexiert und danach durchsucht. Dabei beschränke ich mich in der praktischen Arbeit auf das indexieren der ID3-Tags. Diese Arbeit soll nicht als Anleitung zur Erstellung einer Suchmaschine dienen!

Da Lucene natürlich auch den Inhalt einer Datei analysiert, muss diese Arbeit ein bisschen angepasst werden. Da MP3-Dateien keinen Text als Inhalt haben, möchte ich daher nur theoretisch aufzeigen, wie anhand von Teilen eines Liedes das entsprechende Lied gesucht werden kann. Dies zu programmieren sprengt den Rahmen der Arbeit, somit werde ich am Beispiel von Shazam nur eine theoretische Lösung aufzeigen.

Kapitel 2

Hauptteil

2.1 Lucene

Was ist Lucene? Dies ist die erste Frage, die ich mir zu Beginn der Arbeit gestellt habe. Da mir das Produkt gänzlich unbekannt war, galt es zuerst Informationen zu sammeln.

Apache Lucene eine Suchengine, die sich auf Text und eine hohe Performance spezialisiert hat. Dabei ist die Engine mittlerweile in verschiedene Sprachen übersetzt worden. Der Apache Lucene Core ist der Hauptteil der Software und ist in Java geschrieben. Das beste an Lucene ist wohl, dass es gratis zur Verfügung steht. Somit kann jeder Entwickler eine mächtige Suchmaschine in seine Programme einbauen.

Bei einer Suchmaschine liegen die Stärken im Resultat welches geliefert wird. Lucene bietet auch hier wieder einige Funktionen, die das Endergebnis schnell und korrekt ergeben sollen. Dazu gehören:

- Ranked Searching - Die besten Resultate werden als Erste zurückgegeben.
- Verschiedene Query-Typen
- Feldsuche (Hier Titel, Album, Künstler, Jahr, Songtext).
- Mehrfache Indexe durchsuchen mit zusammengefasstem Ergebnis.
- Schnell
- Speichereffizient
- Tippfehler-tolerant

Mit diesen und weiteren Gimmicks wird Lucene auf der Webseite `lucene.apache.org/core/` angepriesen. Für meine Arbeit habe ich die Bibliothek in der Version 3.6.2 verwendet, da meine Quelle ebenfalls mit einer 3er-Version gearbeitet hat.

2.2 Information Retrieval

> The IR Problem: The primary goal of an IR system is to retrieve all the documents that are relevant to a user query while retrieving as few non-relevant documents as possible. - Buch: Modern Information Retrieval

Dieses Zitat bezeichnet sehr gut um was es bei Information Retrieval geht. Die Menschheit speichert seit 5000 Jahren Informationen in verschiedenen Systemen, um danach über Indexe oder andere Suchmechanismen an wichtige Informationen zu kommen. Im einfachsten Fall möchte ein User nach einer Suche einen Link zu einer Webseite von einer Organisation, Firma oder sonstigen Quelle. Information Retrieval alleine dreht sich nicht nur um Suchmaschinen. Die Definition aus dem Buch Modern Information Retrieval lautet wie folgt:

> Information retrieval deals with the representation, storage, organization of, and access to information items such as documents, Web pages, online catalogs, structured and semi-structured records, multimedia objects. The representation and organization of the information items should be such as to provide the users with easy access to information of their interest. - Buch: Modern Information Retrieval

Zu Beginn war Information Retrieval nur eine Kategorie, die für Bibliothekare und Informationsexperten interessant war. Dieser Umstand änderte sich aber schlagartig, als das Internet aufkam. Informationen waren nun zugänglich und konnten von fast jedem Menschen abgerufen werden. Mit dem Internet wurde es wichtiger, gute Ergebnisse beim Suchen nach Informationen zu bekommen. Information Retrieval hatte die breite Masse erreicht.

Ein Grundproblem bei IR-Systemen ist, welche Informationen für die gewünschte Suche überhaupt relevant sind. Wie soll die Software entscheiden, welche Informationen am besten auf die Anforderungen des Benutzers zutreffen? Dabei gibt es verschiedene Möglichkeiten. Ein Beispiel ist das sogenannte Ranking. Dabei wird einem Dokument, welches häufiger erscheint, ein höheres Ranking gegeben. Somit weiss die Suchmaschine, dass es wahrscheinlicher ist, dass in diesem Dokument gesuchte Informationen zu finden sind, die mit den Anforderungen übereinstimmen.
Eine weitere Möglichkeit besteht in der örtlichen Ablage von Informationen. So können die "am nächsten liegenden"Informationen auch die richtigen sein. Suche ich zum Beispiel in Google nach einem Detailhändler, möchte ich natürlich die Händler im Ergebnis erhalten, die am nächsten zu mir liegen.
Als Drittes ist die Grösse eines Dokumentes ebenfalls eine Variable. So können kleine Dokumente bevorzugt werden, da sie schnell heruntergeladen und veranschaulicht werden können.

Information Retrieval Systeme müssen komplexe Anforderungen erfüllen. Mit den oben genannten drei Beispielen für Ergebnissen, folgt der Schluss, dass IR-Systeme nie die eine perfekte Lösung anbieten können. Die Anfragen sind zu komplex um alle möglichen Varianten in eine Suchmaschine zu integrieren.

Lucene ist genau solch ein IR-System. Dieses System soll nun analysiert werden. Darunter fällt die Analyse der verschiedenen Analyzer. Konkret handelt es sich um den Standard Analyzer, den Whitespace Analyzer, den Stop Analyzer und den Simple Analyzer. Hier möchte ich mich darauf konzentrieren, wie genau die Informationen in den Index gespeichert werden. Ein ebenfalls wichtiger Faktor ist die Zeit. Einerseits die Dauer der Indexerstellung und danach die Dauer der Suche mit den verschiedenen Einstellungen. Als Letztes möchte ich mich um die Analyse des Ergebnisses kümmern, wobei ich eine Testmenge an MP3 benutze um damit die tatsächlichen Suchergebnisse mit der erwarteten Menge zu vergleichen.

2.3 Lucene Search Engine

Das entwickelte Programm hat den Namen "Lucene Search Engine"und wurde in Java entwickelt. Im Programm kann unter dem Punkt "Pfad"der Ablageort der MP3-Dateien angegeben werden. Mit dem Button Index wird dann in diesem Pfad ein Ordner Index erstellt, in welchem der Lucene-Index erstellt wird. Dabei durchsucht das Programm alle MP3-Dateien im angegeben Pfad, auch in Unterordnern. Aus diesen Dateien werden dann die Informationen Titel, Album, Künstler, Jahr, Songtext, Pfad und Dateiname extrahiert und in den Index geschrieben. Gesucht wird dann in den Feldern Titel, Album, Künstler, Jahr, Songtext und Dateiname. Als Ergebnis wird der Pfad zur entsprechenden Dateien ausgegeben.

2.3.1 Das GUI

In diesem Abschnitt möchte ich ganz kurz die grafische Benutzeroberfläche erläutern. Ich gehe nicht weiter auf die Programmierung ein, da dies nur ein kleiner, eher unwichtiger Teil der Search Engine ist. Zuerst betrachten wir die Menüleiste. Unter dem Punkt Datei

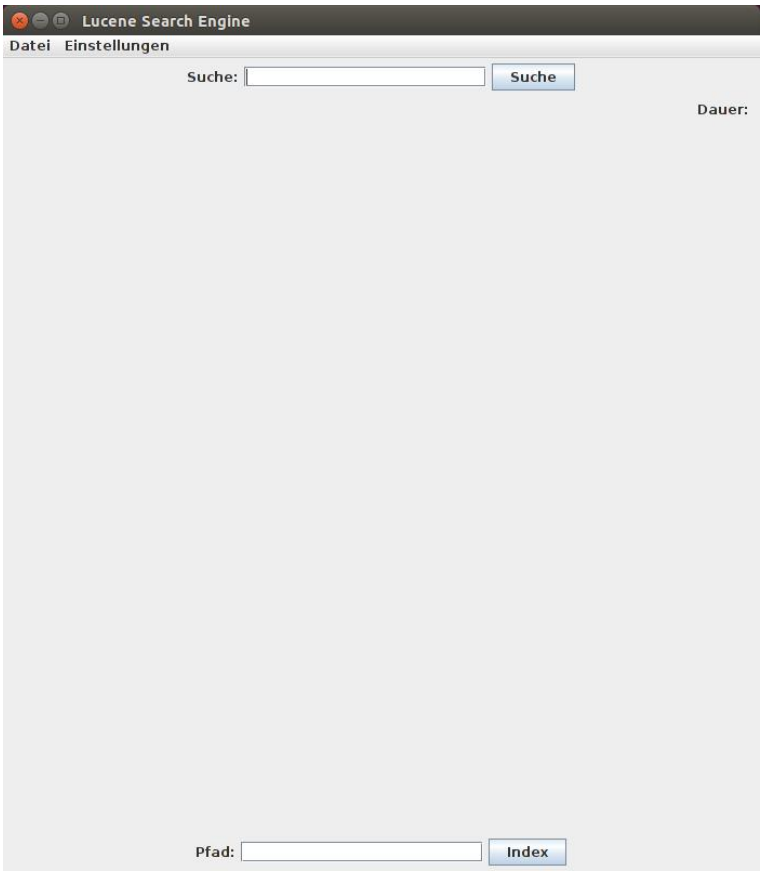


Abbildung 2.1: GUI des Programmes[1]

ist nicht weiteres zu finden als ein Punkt "Schliessen"der das Programm beendet. Interessanter ist da der Punkt Einstellungen. Öffnet man diesen, erhält man eine Auswahl der 4 erwähnten Analyzer. Je nach Auswahl wird der Index auf andere Weise erstellt und die Suche durchgeführt. Es kann jeweils nur ein Analyzer ausgewählt werden. Falls man ihn ändert sollte man auch den Index neu erstellen.

Indexieren

Am unteren Rand der Applikation befindet sich das Feld Pfad. Dort wird der Pfad zu den abgelegten Dateien eingefügt. Dies funktioniert für Linux wie für Windows und Mac. Sobald dies erledigt wurde, kann mit dem Knopf die Indexierung gestartet werden. Als

[1] Quelle: Screenshot

Analyzer wird die Auswahl aus den Einstellungen genommen. Der Index wird im gleichen Pfad erstellt und in einem Unterordner mit dem Namen "Index"abgelegt. Dies geschieht automatisch und kann nicht geändert werden. Sobald die Indexierung beendet ist, wird auf der rechten Seite die Dauer in Millisekunden angezeigt. Je mehr MP3-Dateien indexiert werden müssen, desto länger dauert der Vorgang. Nach der Indexierung ist die Suchma-

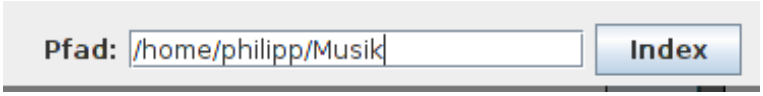


Abbildung 2.2: Beispiel eines angegebenen Pfades in Ubuntu[2]

schine bereit und kann verwendet werden. Das praktische an dieser Lösung liegt darin, dass die MP3-Dateien zum Beispiel auch auf einem externen Speichermedium liegen können und der Index auf diesem Medium erstellt werden kann (gilt natürlich nicht für CD/DVDs). So kann theoretisch für jeden Ablageort eine eigener Index erstellt werden. Genaueres zur Indexierung und wie sie gelöst wurde ist im Abschnit "Der Indexer"zu lesen.

Suchen

Als Erstes muss gesagt werden, dass für die Suche das Pfadfeld weiterhin den Eintrag enthalten muss. Dies ist wichtig, da sonst die Engine nicht weiss, wo sich der Index befindet. Am oberen Rand befindet sich das Suchfeld. Der Benutzer kann hier seine Suchbegriffe eingeben, welche danach im Index gesucht werden. Wieder aktiviert man mit einem Klick auf Suche die Aktion. Sobald die Engine die Ergebnisse bekommen hat, wird die Dauer der Suche am rechten Rand angezeigt. Unterhalb des Suchfeldes wird eine Tabelle ausgegeben, welche die gefundenen Resultate beinhaltet. Lucene bietet integrierte Funktionen, um

Abbildung 2.3: Beispiel eines eingegebenen Suchbegriffes[3]

die Suche besser auf seine Wünsche anzupassen. Hier ist eine Liste, wie man mit der Suchmaschine detaillierter suchen kann:

- Eingabe: song1
  Sucht in den indexierten Feldern nach diesem Begriff
- Eingabe: song1 song2
  Sucht in den indexierten Feldern nach den beiden Begriffen. Gibt Resultate aus, welche Begriff 1 oder Begriff 2 oder beide beinhalten.

[2]Quelle: Screenshot
[3]Quelle: Screenshot

- Eingabe +song1 +song2
  Sucht in den indexierten Feldern nach den beiden Begriffen. Gibt Resultate aus, welche beide Begriffe beinhalten.

- Titel:song1
  Sucht in dem indexierten Feld Titel nach dem Begriff.

- Titel:song1 -Album:album1
  Sucht in den indexierten Feldern nach dem Begriff. Gibt als Resultat die Dateien zurück, welche den Titel besitzen aber nicht zum Album gehören.

- (song1 OR song2) AND album1
  Sucht in den indexierten Feldern nach den Begriffen. Gibt als Resultat die Dateien zurück welche song1 oder song2 im besitzen und in album1 zu finden sind.

- "song1"
  Sucht in den indexierten Feldern nach exakt diesem Wort oder Text.

- song1*
  Sucht in den indexierten Feldern nach Werten, die mit song1 beginnen.

- song1∼
  Sucht in den indexierten Feldern nach Werten, die ähnlich wie song1 sind.

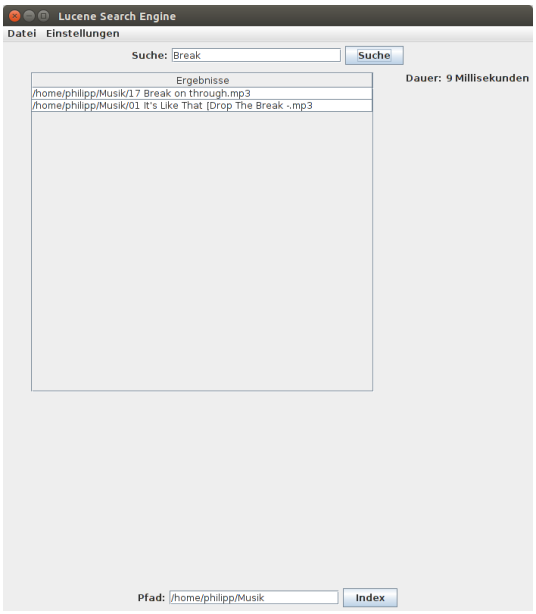


Abbildung 2.4: Beispiel gesuchter Begriff[4]

[4]Quelle: Screenshot

2.3.2 Der Indexer

In diesem Teil möchte ich kurz auf die Klasse Index eingehen, welche nach den MP3-Dateien sucht und die Informationen aus den ID3-Tags zieht. Die Klasse Index wird durch das GUI gestartet, indem man den Kopf "Index"betätigt. Als Parameter erhält er den Inhalt des Textfeldes "Pfad"und welcher Analyzer genutzt werden soll. Nun wird der Pfad für den Index definiert. Dieser liegt immer im angegeben Verzeichnis im Unterordner "Index". Danach initialisiert die Klasse den Indexer und erstellt den Analyzer. Im Anschluss startet er die Messung der Zeit und beginnt die Dateien zu suchen und die Informationen zu extrahieren. Dies geschieht in der Methode index(String dataDir, FileFilter filter). Falls er einen Ordner findet, öffnet die Klasse den Ordner und sucht weiter nach Dateien.

Hat die Methode eine MP3-Datei gefunden, wird sie mit der Methode indexFile(File f) in ein Document-Typ gewandelt, welcher die Informationen aus den ID3-Tags enthält. Per writer.addDocument(doc) wird das MP3 in den Index aufgenommen. Die ID3-Tags werden in der Methode getDocument(File f) einzeln aus jedem MP3 ausgelesen. Dabei standen verschiedene Lösungen zur Verfügung. Dazu aber später mehr im Kapitel "ID3-Tags".

Sobald alle MP3-Dateien aufgenommen wurde, berechnet die Klasse als Letztes, wie lange der Vorgang gedauert hat. Dieses Ergebnis wird dann, wie schon erwähnt, auf dem GUI ausgegeben. Der Index wurde erstellt und kann nun zur Suche genutzt werden.

2.3.3 Der Searcher

Ein Index alleine nützt uns nichts, wenn nicht ein Searcher implementiert wird, der diesen Index auch durchsuchen kann. Dazu ist in diesem Projekt die Klasse "Searcher"vorhanden. Wie schon beim Indexer, erhält der Aufruf per Knopfdruck auf den Button "Suche"die Parameter Pfad, Analyzerart und die Anfrage. Der Pfad wird direkt aus dem Indexpfad gelesen. So findet das Programm den korrekten Index. Danach startet die Methode search(String indexDir, String q, String analyzerArt).

Die Methode search öffnet den vorgegebenen Pfad um den Index zu lesen. Danach wird ein Parser erstellt, der mehrere Felder absuchen kann (MultiFieldQueryParser). Je nach Analyzerart wird eine andere Version erstellt. Dieser Parser wandelt die Benutzeranfrage (hier q) so um, dass Lucene die Anfrage versteht. Die Methode durchsucht den Index und ergibt maximal 100 Ergebnisse. Die einzelnen gefundenen Inforationen werden wieder in den Typ Document umgewandelt. Zum Schluss wird eine ArrayList mit Document zurückgegeben, welche alle Ergebnisse beinhaltet. Die Liste wird jedesmal bei der Suche gelöscht und wieder von neuem befüllt. Der Inhalt der ArrayList wird zum Schluss im GUI ausgegeben, sowie auch die Dauer der Suche.

2.3.4 ID3-Tags

ID3-Tags sind Informationen, welche direkt in die Dateien eingebettet wurden. Sie beinhalten wertvolle Informationen wie Titel, Künstler, Album, Jahr, Genre, usw. ID3 hat sich mittlerweile zum Standard für MP3 entwickelt und wird von Programmen wie iTunes, Windows Media Player und VLC genutzt. Leider können nicht alle Programme mit jeder ID3-Version umgehen. Dies kann zu Komplikationen führen.

Um in meiner Applikation diese Daten zu extrahieren, standen drei Möglichkeiten zur Auswahl. Diese waren manuelles extrahieren in Java, benutzen einer Bibliothek und einsetzen von Tika.

Manuelles extrahieren

Das auslesen von ID3v1 ist relativ einfach. Die Informationen sind immer fest in einem Block von 128 Bytes am Ende der Datei gespeichert. Bei Offset 3 mit einer Länge von 30 wird zum Beispiel der Songtitel gespeichert. Folgende Tabelle gibt die Felder und deren Position wieder:

Offset	**Länge in Bytes**	**Bedeutung**
0	3	Kennung eines ID3v1-Blocks
3	30	Songtitel
33	30	Künstler
63	30	Album
93	4	Jahr
97	30	Kommentar
127	1	Genres

Diese können so direkt angesprochen werden. Schwieriger wird es mit der Version 2. Ab Version zwei wird im Header der Datei angegeben, ob es sich um ID3 der Version 2 handelt. Die Tags werden dann so kodiert, dass ein Player sie nicht versteht und als fehlerhafte Informationen interpretiert. So werden die Teile der Datei übersprungen und nicht abgespielt. Neu können auch Bilder in diese Feldern abgelegt werden. Allerdings ist die Implementation sehr mühsam.

Da diese Lösung viel Zeit verschlingen würde, da für jede Version der ID3-Tags auch eine Methode geschrieben werden müsste, die die entsprechenden Tags ausliest, habe ich diese Lösung nicht in Betracht gezogen. Ausserdem muss ich das Rad nicht neu erfinden, wenn schon Möglichkeiten vorhanden sind, welche meine Anforderungen erfüllen. So kommen wir zum nächsten Kapitel.

Tika

Tika ist ein Framework, das im Oktober 2008 zur Lucene-Familie hinzugefügt wurde. Es besitzt die gleichen Standard-APIs wie Lucene. Tika selber ist nur eine Ergänzung, welche

verschiedene Parser für Dateien beinhaltet. Darunter auch MP3. Allerdings können laut Buch "Lucene in Action"nur ID3v1-Tags ausgelesen werden. Dies erschwert unser vorhaben wieder, da verschiedene Versionen eingesetzt werden können. Ausserdem muss Tika auf andere Open-Source Projekte aufsetzen, um Dateien zu suchen und zu öffnen. Dies kann das Framework nicht selbst tätigen.

Aus diesen Gründen konnte ich Tika aus meinen Möglichkeiten streichen, da so der Aufbau der Applikation nur komplizierter wurde. Und mit der Einschränkung auf ID3v1 war Tika nicht wirklich eine gute Wahl für mein Vorhaben. Ausserdem exportiert Tika die Informationen in eine XHTML-Datei. Das heisst, dass ich zusätzlich zum Index nochmals Dateien ablegen müsste. Somit blieb mir die letzte Lösung.

Extraktion mit Library

Im Internet gibt es verschiedene Bibliotheken, die frei zur Verfügung stehen und sich um solche Dinge, wie die Extraktion von ID3-Tags, kümmern. Meine Wahl fiel auf die Lösung JAudioTagger. Dies ist ein kleines JAR-File, welches problemlos in das Projekt integriert werden konnte. Mit wenigen Code-Zeilen ermöglichte mir die Klasse, dass die Informationen aus den MP3-Dateien in den Index geschrieben werden konnten.

Mit 3 Zeilen wird der Vorgang initialisiert.

```
AudioFile audioFile = AudioFileIO.read(f);
Tag tag = audioFile.getTag();
AudioHeader header = audioFile.getAudioHeader();
```

Zeile 1 liest die Datei ein und wandelt sie in den Typ AudioFile. Durch diesen Typ können nun mit der Methode getTag() in Zeile 2 die ID3-Tags ausgelesen werden. Als Zusatz (aber hier nicht notwendig) wird der Header für weitere Funktionen aus dem AudioFile ausgelesen. Um nun die bestimmten Tags zu bekommen braucht es nur noch den folgenden Befehl:

```
tag.getFirst(FieldKey.TITLE)
```

Mit diesem Befehl wird der spezifische Eintrag bei Titel gelesen. Diese kann dann direkt beim Indexer in ein Feld für den Typ Document gespeichert werden. Natürlich kann anstatt TITLE auch ALBUM, YEAR, ARTIST, usw. genutzt werden. Somit habe ich alle relevanten Informationen aus den Dateien extrahiert und in meinen Index geschrieben.

2.4 Analyse der Leistung

Die Suchleistung der Engine möchte ich über zwei Werte vergleichen. Einerseits wäre das natürlich die Zeit, welche benötigt wird, um zu indexieren und zu suchen. Andererseits muss auch das Ergebnis überprüft werden, ob auch die erwarteten Resultate zurückgegeben werden. Die Algorithmen welche verwendet werden, sind bei Lucene die Analyzer. Daher werde ich die vier gezeigten Analyzer überprüfen und danach die Werte vergleichen und ein Fazit ziehen.

2.4.1 Vorbereitung

Um eine korrekte Messung zu bekommen, müssen für alle Analyzer die gleichen Umstände herrschen. Das heisst, dass auf einem USB-Stick eine Auswahl an Dateien gespeichert sind. Diese werden indexiert (ergibt ersten Wert). Dann wird dieser Index (der danach ebenfalls auf dem USB-Stick liegt) wird mit bestimmtem Suchabfragen gefüttert, um dann die Ergebnisse und die benötigte Zeit zu bekommen. Die Tests werden unter Windows 8.1 durchgeführt, auf einem Notebook mit einer 256GB Solid State Disk. Der USB-Stick besitzt 8GB Speicherplatz und ist von der Marke disk2go.com

Testmenge

Für den Test wurden drei Ordner aus der Musikbibliothek ausgewählt. Insgesamt enthält der USB-Stick 960 Dateien und 55 Ordner. Dabei wurden Bilddateien oder ähnliche nicht entfernt. Für die Gegenüberstellung der Suchergebnisse wird das Programm MediaMonkey auf dem Computer installiert und mit der Testmenge gefüttert. Im optimalsten Fall müsste die Search Engine die gleichen Ergebnisse liefern wie das spezifizierte Suchprogramm. MediaMonkey wird in der Version 4.1.1 eingesetzt. Laut MediaMonkey sind auf dem USB-Stick 29 Genres, 257 Interpreten und 51 Alben vorhanden.

2.4.2 Zeitmessung

In diesem Abschnitt möchte ich mich ausschliesslich um die Zeitmessungen bei den verschiedenen Analyzern kümmern. Als erstes beginnen wir mit der Messung der Indexierung. Danach kommt die Messung der Suchanfragen.

Standard Analyzer

Wir beginnen mit dem Standard Analyzer. Zuerst möchte ich erklären, wie dieser bei der Indexierung vorgeht. Man kann den Standard Analyzer nur als umfassend einsetzbar bezeichnen. Er enthält JFlex basierten Grammatik. Somit kann er alphanummerische Werte, Akronyme, Email-Adressen, Webseiten, IP-Adresse, usw. erkennen. Ausserdem besitzt er die Stopp-Wörter, die auch der Stopp-Analyzer verwendet. Bei Stopp-Wörtern werden

uninteressante Wörter wie Der, Die, Das ausgeblendet und nicht in die Indexierung aufgenommen. Ausserdem ist er der einzige Analyzer, der Strings wie X&Y oder xy@zhaw.ch direkt in den Index übernimmt.

Nun indexieren wir mit dem Standard Analyzer die Dateien auf dem USB-Stick: Wie wir

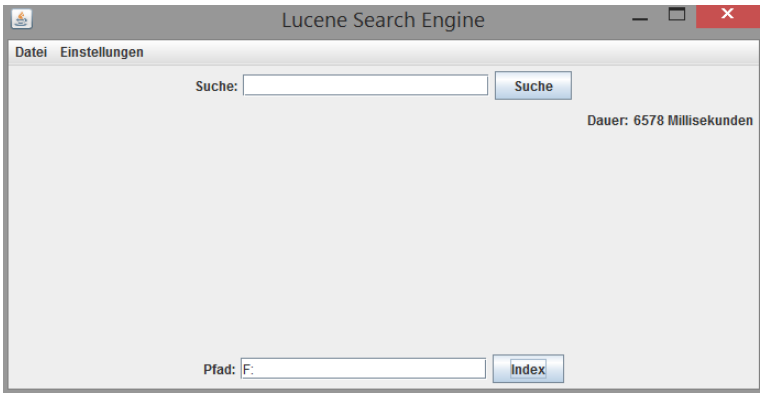


Abbildung 2.5: Indexierung abgeschlossen[5]

auf dem Screenshot sehen brauchte die Indexierung 6578 Millisekunden um den Ordner und den Index auf dem USB-Stick zu erstellen und die Dateien zu durchforsten. Umgerechnet sind das ungefähr 6,6 Sekunden.

Simple Analyzer

Der Simple Analyzer arbeitet auf eine sehr rudimentären Basis. Da Text bei den Analyzern immer in Tokens umgewandelt (also geteilt) werden muss, gehen die Algorithmen auf verschiedene Arten vor. Der Simple Analyzer überprüft den Text nur auf Zeichen die keine Buchstaben sind. Sobald er ein solches Zeichen gefunden hat, teilt er den String an dieser Stelle. Aus dem Beispiel des Standard Analyzers, folgt also das X&Z zu X und Z werden würden und xy@zhaw.ch würde zu xy, zhaw und ch werden. Der Analyzer ist sehr einfach aufgebaut, müsste aber nach meiner Erwartung dafür schneller sein als der Standard Analyzer. Nach meiner Überlegung müsste nur der Whitespace Analyzer noch schneller sein. Später aber mehr dazu.

[5]Quelle: Screenshot

Der Index-Ordner wird auf dem USB-Stick wieder gelöscht und erneut erstellt mit dem Simple Analyzer:

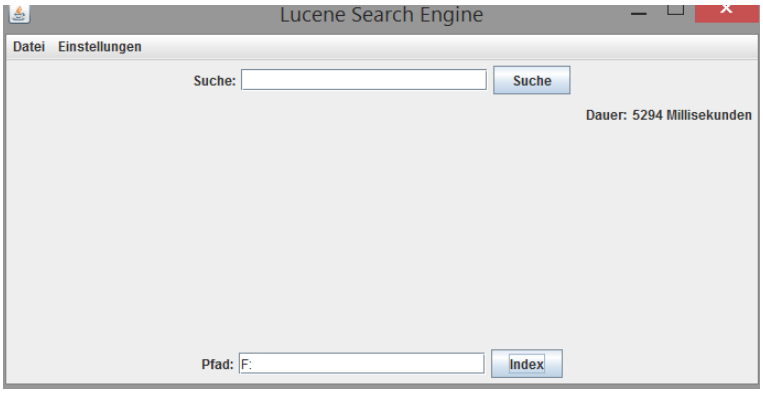


Abbildung 2.6: Indexierung abgeschlossen[6]

Auch hier können wir dem Screenshot wieder ablesen, dass die Indexierung insgesamt 5294 Millisekunden gedauert hat. Die Arbeiten, die das Programm machen musste, waren die Gleichen wie schon beim Standard Analyzer. In Sekdunden umgerechnet dauerte das indexieren als ungefähr 5,3 Sekunden. Wie erwartet war dieser Analyzer schneller als der vorherige.

Whitespace Analyzer

Wie der Simple Analyzer ist der Whitespace Analyzer sehr einfach aufgebaut. Der Text wird in diesem Fall nicht einfach nach Nicht-Buchstaben durchsucht, sondern die Tokens werden aus den Leerschlägen (Whitespaces) im Text erstellt. Wieder am Beispiel von X&Z xy@zawh.ch würde er X&Y und xy@zhaw.ch erkennen. Allerdings hat er, wie der Simple Analyzer das Problem, dass keine Stopp-Wörter erkennt werden. Das heisst, unwichtige Wörter wie “und”werden ebenfalls in den Index aufgenommen. Da diese Füllwörter sehr häufig vorkommen, könnten sie im Index eine höhere Gewichtung erhalten, ohne dass sie wirklich Informationen über den Text oder die Datei liefern.

[6]Quelle: Screenshot

Es werden wieder die gleichen Schritte durchgeführt wie zuvor:

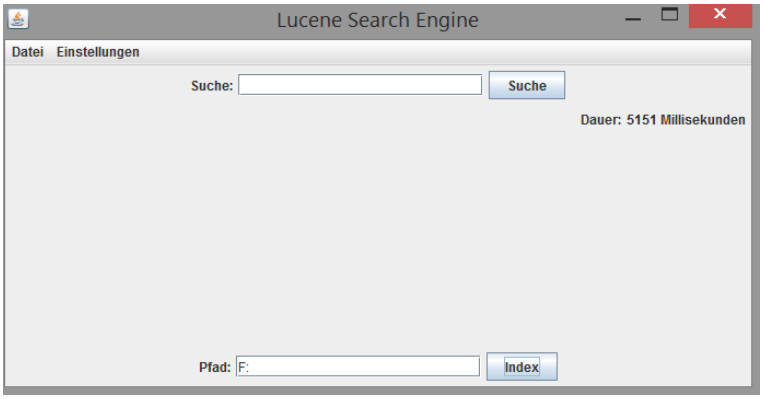


Abbildung 2.7: Indexierung abgeschlossen[7]

Dieses Mal hat der Indexierungsvorgang 5151 Millisekunden gedauert. Wie erwartet war der Whitespace Analyzer noch ein bisschen schneller als der Simple Analyzer. Ich schätze, dass dies aus der Limitierung auf Leerzeichen kommt, da er nicht nach verschiedenen Zeichen suchen muss. Sprich für jedes Zeichen im Text muss nur eine einzige Abfrage gemacht werden. Umgerechnet sind dies ungefähr 5,2 Sekunden.

Stop Analyzer

Der Stop Analyzer hat die Besonderheit, dass er unwichtige Wörter im Text erkennt und automatisch nicht in den Index aufnimmt. Diese Stopp-Wörter können mit Listen erweitert werden, was aber in dieser Arbeit weggelassen wurde. So beschränkt sich der Stop Analyzer auf Wörter wie “The”oder “and”. Da die meisten MP3 auf Englisch sind, sowie deren Songtexte, ist die Standardimplementierung keine schlecht Wahl.

[7]Quelle: Screenshot

Ein letztes Mal wird der Index gelöscht und die Zeit nochmals gemessen:

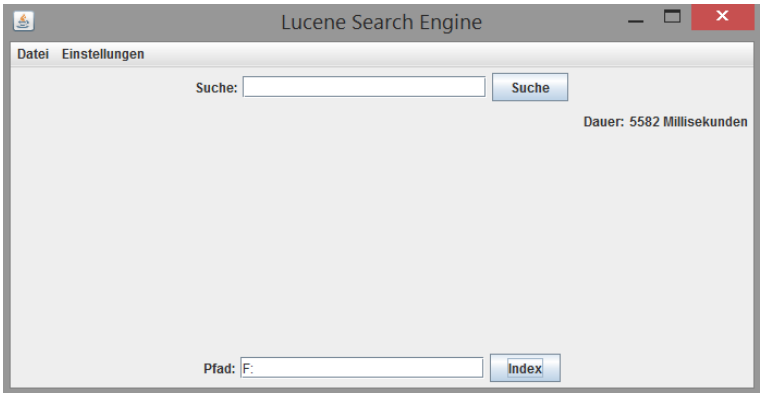


Abbildung 2.8: Indexierung abgeschlossen[8]

Beim letzten Test ging die Indexierung 5582 Millisekunden. Im Vergleich zum Simple und Whitespace Analyzer hat sich die Dauer wieder erhöht. Umgerechnet in Sekunden sind das ungefähr 5.6 Sekdunden. Mit dieser Zeit liegt der Stop Analyzer aber immer noch unterhalb des Standard Analyzers. Ich erkläre mir das so, dass der Standard Analyzer komplexere Muster zu erkennen hat. Dadurch wird die Indexierung entsprechend länger brauchen.

Fazit Indexierungsdauer

Im Vergleich haben wir nun gesehen, dass der Whitespace und der Simple Analyzer die schnellsten Verfahren sind. Mit 5,2 und 5,3 Sekunden liegen sie einiges weiter vorne als der Stop Analyzer und der Standard Analyzer. Da wir hier im Test mit einer kleinen Menge an Dateien arbeiten könnten wir das Ergebnis hochrechnen:
Gehen wir davon aus, dass nur MP3-Dateien in den Ordnern liegen. bei 960 Dateien mit der Gesamtgrösse von 4.6 GB, können wir abschätzen, dass eine MP3-Datei zirka 4.9 MB gross ist. Pro MP3-Datei hat der Analyzer in unserem Fall folgende Zeiten benötigt:

- Standard Analyzer: 6.85 Millisekunden
- Simple Analyzer: 5.5 Millisekunden
- Whitespace Analyzer: 5.3 Millisekunden
- Stop Analyzer: 5.8 Millisekunden

Sagen wir nun, dass die durchschnittliche MP3-Sammlung einer Person an die 25 GB Speicherplatz braucht (Wert aus Erfahrung und Referenz der eigenen Sammlung). Dann folgt daraus, dass ungefähr 5224 MP3-Dateien vorhanden sind. Dadurch können wir sagen, wie lange die Indexierung bei dieser Menge brauchen würde:

[8]Quelle: Screenshot

- Standard Analyzer: 35.8 Sekunden
- Simple Analyzer: 28.7 Sekunden
- Whitespace Analyzer: 27.7 Sekunden
- Stop Analyzer: 30.3 Sekunden

Wir sehen, je grösser die Sammlung wird, desto grösser wird der Unterschied in der Dauer der Indexierung. Immerhin ist der schnellste Analyzer um fasst 20% schneller als der langsamste Analyzer (langsamer Analyzer wurde als 100% gewertet).

2.4.3 Suchmessung

Die Messung der Suchleistung teile ich in zwei Arten. Die eine Art kümmert sich um die Dauer der Suchanfrage, die andere konzentriert sich auf die gefundene Menge und ob diese Menge der erwarteten entspricht. Dazu brauchen wir aber noch eine kurze Vorbereitung, in der definiert wird, welche Suchanfragen überprüft werden. Die folgende Liste soll uns als Orientierung dienen:

Suchanfrage	**Ergebnis**
long way	Pfad zu: 14-Do You Remember.mp3 04 Flyover.mp3 01 Hip Hop Bommi Bop (Tap Into Ameri).mp3 01 It's A Long Way To The Top (If Yo).mp3 01 It's A Long Way To The Top (If Yo).mp3 13 Long way from Liverpool.mp3 36 Stand up.mp3 13 Viva La Revolucion.mp3 18 Whole Wide World.mp3
Break	Pfad zu: 03 All for the sake of love.mp3 01 Ballbreaker.mp3 02 Blow Me Away.mp3 17 Break on through.mp3 08 Breaking The Rules.mp3 14 Disneyland.mp3 23 Do Anything You Wanna Do.mp3 14 Do You Remember.mp3 08 Hand of Blood.mp3 01 Heatseeker.mp3 10 I Faught The Law.mp3 01 It's Like That [Drop The Break -.mp3 03 Life Burns.mp3 13 Love Machine.mp3 04 Lovesong.mp3 09 More & More.mp3 11 No Escape.mp3 05 Problem Child.mp3 05 Pushed Again.mp3 03 Super Shooter.mp3 20 The Great Escape.mp3 07 The Guns Of Brixton.mp3 25 The World.mp3 27 Top of the World.mp3 10 Two's Up.mp3 10 Wake the Dead.mp3 01 Who Let The Dogs Out.mp3

Suchanfrage	**Ergebnis**
instrumental	Pfad zu: 03 All for the sake of love.mp3 06 Diary of a lover.mp3 09 Helmstedt Blues.mp3 16 Imperial March.mp3 13 Love Machine.mp3 09 More & More.mp3 10 My Land.mp3 12 Perfect Criminal.mp3 01 Tote Hose.mp3 10 You Shook Me All Night Long.mp3
Bells Album:Black	Pfad zu 04 Hells Bells.mp3
“break on through”	Pfad zu: 17 Break on through.mp3
Bells OR Hit Album:Black	Pfad zu: 01 Back In Black.mp3 02 Given The Dog A Bone.mp3 03 Have A Drink On Me.mp3 04 Hells Bells.mp3

Dies sind die Abfragen, welche ich verwenden werde, um die Engine zu testen. Ich lasse bewusst die Möglichkeit nach ähnlichen Wörtern zu suchen weg, da das Referenzprogramm dies ebenfalls nicht unterstützt.

Standard Analyzer

Wir starten mit dem Standard Analyzer. Da MediaMonkey bei Suchen wie long way die beiden Wörter mit AND verknüpft, wird die Abfrage in der Engine angepasst, so dass die Logik stimmt. Daher werden solche Anfragen zu long AND way. Die Screenshots zeigen die jeweiligen Ergebnisse. Die Dauer wird in der Bildunterschrift vermerkt:

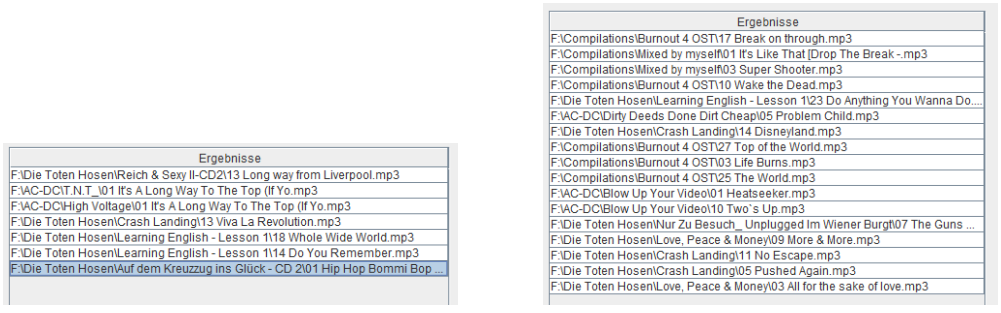


Abbildung 2.9: Links: Suche 1 nach 3ms, Rechts: Suche 2 nach 2ms[9]

Abbildung 2.10: Links: Suche 3 nach 1ms, Rechts: Suche 4 nach <1ms[10]

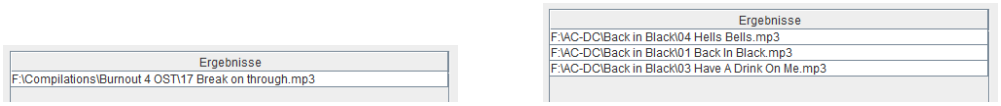


Abbildung 2.11: Links: Suche 5 nach 5ms, Rechts: Suche 6 nach 1ms[11]

Simple Analyzer

Nun wiederholen wir die Probe mit dem Simple Analyzer:

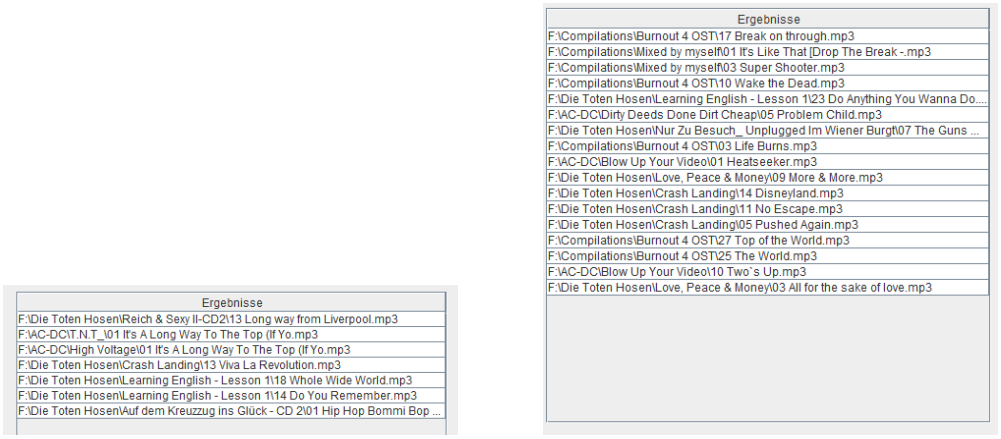


Abbildung 2.12: Links: Suche 1 nach 28ms, Rechts: Suche 2 nach <1ms[12]

[9] Quelle: Screenshot
[10] Quelle: Screenshot
[11] Quelle: Screenshot
[12] Quelle: Screenshot
[13] Quelle: Screenshot
[14] Quelle: Screenshot

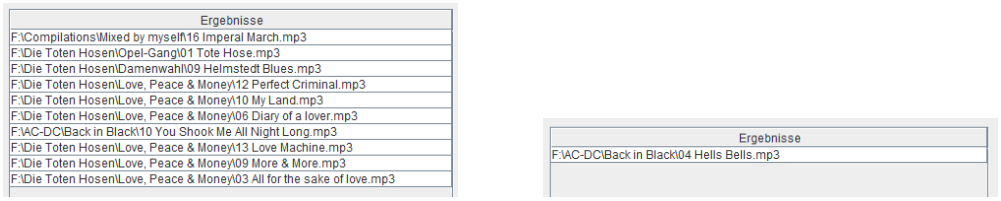


Abbildung 2.13: Links: Suche 3 nach 4ms, Rechts: Suche 4 nach <1ms[13]

Abbildung 2.14: Links: Suche 5 nach 15ms, Rechts: Suche 6 nach <1ms[14]

Whitespace Analyzer

Dieses Prozedere wiederholen wir nun nochmals für den Whitespace Analyzer:

Abbildung 2.15: Links: Suche 1 nach 4ms, Rechts: Suche 2 nach 16ms[15]

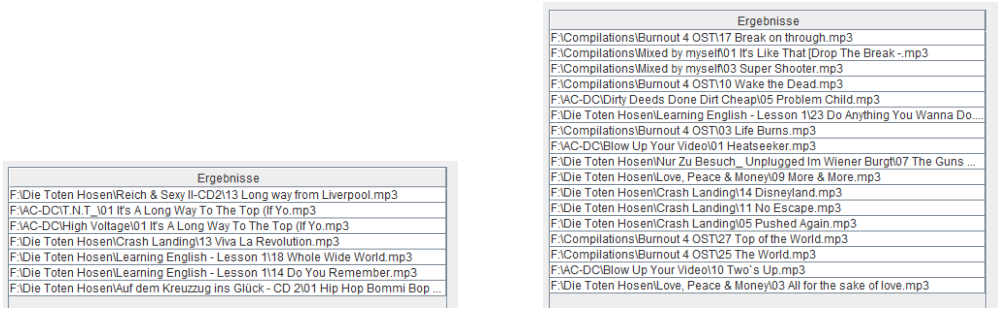
Abbildung 2.16: Links: Suche 3 nach 4ms, Rechts: Suche 4 nach 16ms[16]

Abbildung 2.17: Suche 6 nach 15ms[17]

[15]Quelle: Screenshot
[16]Quelle: Screenshot
[17]Quelle: Screenshot

Stop Analyzer

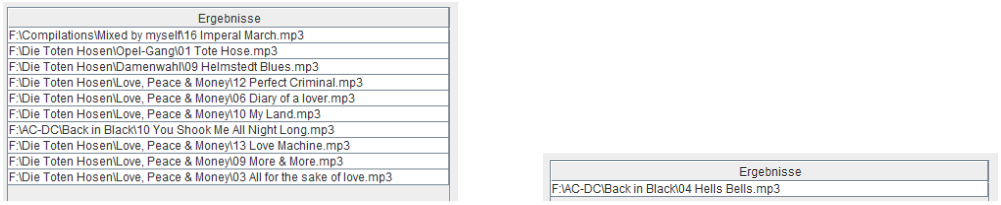
Ein letztes Mal führen wir nun die Analyse durch:



Ergebnisse
F:\Die Toten Hosen\Reich & Sexy II-CD2\13 Long way from Liverpool.mp3
F:\AC-DC\T.N.T_\01 It's A Long Way To The Top (If Yo.mp3
F:\AC-DC\High Voltage\01 It's A Long Way To The Top (If Yo.mp3
F:\Die Toten Hosen\Crash Landing\13 Viva La Revolution.mp3
F:\Die Toten Hosen\Learning English - Lesson 1\18 Whole Wide World.mp3
F:\Die Toten Hosen\Learning English - Lesson 1\14 Do You Remember.mp3
F:\Die Toten Hosen\Auf dem Kreuzzug ins Glück - CD 2\01 Hip Hop Bommi Bop ...

Ergebnisse
F:\Compilations\Burnout 4 OST\17 Break on through.mp3
F:\Compilations\Mixed by myself\01 It's Like That [Drop The Break -.mp3
F:\Compilations\Mixed by myself\03 Super Shooter.mp3
F:\Compilations\Burnout 4 OST\10 Wake the Dead.mp3
F:\AC-DC\Dirty Deeds Done Dirt Cheap\05 Problem Child.mp3
F:\Die Toten Hosen\Learning English - Lesson 1\23 Do Anything You Wanna Do....
F:\Compilations\Burnout 4 OST\03 Life Burns.mp3
F:\AC-DC\Blow Up Your Video\01 Heatseeker.mp3
F:\Die Toten Hosen\Nur Zu Besuch_ Unplugged Im Wiener Burgt\07 The Guns ...
F:\Die Toten Hosen\Love, Peace & Money\09 More & More.mp3
F:\Die Toten Hosen\Crash Landing\14 Disneyland.mp3
F:\Die Toten Hosen\Crash Landing\11 No Escape.mp3
F:\Die Toten Hosen\Crash Landing\05 Pushed Again.mp3
F:\Compilations\Burnout 4 OST\27 Top of the World.mp3
F:\Compilations\Burnout 4 OST\25 The World.mp3
F:\AC-DC\Blow Up Your Video\10 Two`s Up.mp3
F:\Die Toten Hosen\Love, Peace & Money\03 All for the sake of love.mp3

Abbildung 2.18: Links: Suche 1 nach 16ms, Rechts: Suche 2 nach <1ms[18]



Ergebnisse
F:\Compilations\Mixed by myself\16 Imperal March.mp3
F:\Die Toten Hosen\Opel-Gang\01 Tote Hose.mp3
F:\Die Toten Hosen\Damenwahl\09 Helmstedt Blues.mp3
F:\Die Toten Hosen\Love, Peace & Money\12 Perfect Criminal.mp3
F:\Die Toten Hosen\Love, Peace & Money\06 Diary of a lover.mp3
F:\Die Toten Hosen\Love, Peace & Money\10 My Land.mp3
F:\AC-DC\Back in Black\10 You Shook Me All Night Long.mp3
F:\Die Toten Hosen\Love, Peace & Money\13 Love Machine.mp3
F:\Die Toten Hosen\Love, Peace & Money\09 More & More.mp3
F:\Die Toten Hosen\Love, Peace & Money\03 All for the sake of love.mp3

Ergebnisse
F:\AC-DC\Back in Black\04 Hells Bells.mp3

Abbildung 2.19: Links: Suche 3 nach 16ms, Rechts: Suche 4 nach <1ms[19]

Ergebnisse
F:\Compilations\Burnout 4 OST\17 Break on through.mp3

Ergebnisse
F:\AC-DC\Back in Black\04 Hells Bells.mp3
F:\AC-DC\Back in Black\03 Have A Drink On Me.mp3
F:\AC-DC\Back in Black\01 Back In Black.mp3

Abbildung 2.20: Links: Suche 5 nach 16ms, Rechts: Suche 6 nach 15ms[20]

[18] Quelle: Screenshot
[19] Quelle: Screenshot
[20] Quelle: Screenshot

Auswertung und Fazit

Vergleichen wir nun die verschiedenen Werte der Suche, kommen wir auf folgendes Ergebnis:

	Standard	**Simple**	**Whitespace**	**Stop**
Hits Suche 1	7/9	7/9	2/9	7/9
Hits Suche 2	17/27	17/29	3/29	17/29
Hits Suche 3	10/10	10/10	2/10	10/10
Hits Suche 4	1/1	1/1	1/1	1/1
Hits Suche 5	1/1	1/1	0/1	1/1
Hits Suche 6	3/4	3/4	1/4	3/4
Dauer Suche 1	3	28	4	16
Dauer Suche 2	2	<1	16	<1
Dauer Suche 3	1	4	4	16
Dauer Suche 4	<1	1	16	<1
Dauer Suche 5	5	15	-	16
Dauer Suche 6	1	<1	15	15

Wir erkennen also, dass kein Analyzer alle Einträge gefunden hat. Besonders bei Suche 2 gehen viele verloren. Zu erwähnen ist der Whitespace Analyzer, da dieser beim Indexieren zwar sehr schnell war, hingegen bei der Suchleistung und der benötigten Zeit sehr schwach ist. So hat er bei Suche 5 kein Resultat geliefert. Obwohl der Standard Analyzer lange indexiert, ist dafür seine Suchleistung sehr gut. Er hat den grossen Teil der Dateien gefunden und das in einer sehr guten Zeit.

Schlussendlich muss man sagen, dass es wohl nicht den einen Analyzer gibt, welche alle Fälle sehr gut abdeckt. Steht der Fokus auf einer schnellen Indexierung, ist der Whitespace Analyzer wohl die erste Wahl. Hingegen bei einer optimalen Suchleistung würde ich in diesem Fall zum Standard Analyzer raten.

2.5 Suchen nach Musikmustern

Die hier programmierte Suche Engine prüft nur die ID3-Tags von MP3-Dateien. Für eine Suchmaschine innerhalb einer Musikverwaltung reicht eine solche Lösung vollkommen aus. Faalls man hingegen in einer Musikbibliothek nach einem Teil eines Liedes suchen möchte, stösst diese Lösung an seine Grenzen. Es wäre also eine praktische Erweiterung, wenn man Teile eines Liedes in die Suchmaschine laden könnte und dieses Segment dann in der Bibliothek gesucht wird. Für Mobiltelefone und Tablets gibt es bereits ein ähnliches System. Shazam ist eine kleine Applikation, mit welcher ein Teil von einem Lied aufgenommen werden kann und das gesuchte Lied identifiziert wird. Aber wie genau arbeitet Shazam?

2.5.1 Shazam

Shazam wird als Musik-Identifizierungsdienst bezeichnet. Der Dienst wird von der englischen Firma Shazam Entertainment Limited angeboten. Dessen Hauptsitz ist in London zu finden. In der Anfangsphase des Dienstes konnten Benutzer per SMS einen Teil der Lieder an die Firma senden, welche den Teil analysierte und mit einer Datenbank abglich. Dieser Dienst war aber nur für England verfügbar. Seit 2002 können per Mobiltelefon oder Tablet Musikteile aufgenommen werden und direkt mit der Datenbank abgeglichen werden. So erhält der Benutzer Titel, Künstler, Album und weitere Informationen.

Abbildung 2.21: Logo Shazam[21]

In Ergänzung ist zu erwähnen, dass im Jahr 2014 herausgefunden wurde, dass Shazam Informationen von Android-Mobilgeräten heruntergeladen hatte. Die Informationen wurden an Werbefirmen weitergeleitet.

Der akustische Fingerabdruck

Der akustische Fingerabdruck kann verglichen werden mit einem Barcode für Musik. Durch diesen ist es möglich, Datenbanken zu erstellen, welche diesen Abdruck speichern und in Verbindung bringen mit anderen Informationen. Dazu kann auch nur ein Teil des Liedes genutzt werden.

Wieso kann Shazam mit wenigen Sekunden eines Liedes den korrekten Abdruck von einem Lied mit einer Dauer von 3 Minuten finden? Nun, dies liegt in der Natur des akustischen Fingerabdrucks. Bei Menschen entspricht er dem normalen Fingerabdruck. Ein Lied wird

[21] Quelle: `http://de.wikipedia.org/wiki/Shazam_(Dienst)`

bei der Berechnung in verschiedene Frequenzbänder unterteilt. Jedes dieser Frequenzbänder wird dann analysiert und eine "spektrale Flachheit"berechnet. Dieser bleibt über die Dauer eines Liedes gleich. Mit Hilfe dieses Abdrucks kann ein Lied exakt identifiziert werden.

Spektrale Flachheit

Die spektrale Flachheit ist die grundlegende Berechnung, um den Fingerabdruck zu bekommen. Dabei gilt, dass harte rhythmische Änderungen als Beispiel einen Wert von 1 haben, während gleichmässiger Singsang eher in die Richtung 0 tendiert. Durch diesen Wert reichen wenige Sekunden aus, um einen solchen Abdruck zu bestimmen.

Kapitel 3

Fazit

Als Erstes muss gesagt werden, dass Lucene eine sehr positive Überraschung war. Die Implementation war schlussendlich einfacher, als zu Beginn angenommen. Natürlich muss beachtet werden, dass in meiner Lösung nur eine grundlegende Realisierung genutzt wurde. Je mehr man Lucene personalisieren möchte, desto komplexer wird das Thema. Unter anderem muss man sich zum Beispiel nur Gedanken machen, welches optimale Stopp-Wörter sind. Dann müssen diese aufgenommen und in das Programm integriert werden.

Als zweiter Punkt ist die Abschätzung des Anwendungsbereichs wichtig. Schnelle Indexierung oder schnelle Suchergebnisse sind ein Aspekt des Anwendungsbereiches. Eine einfache Antwort erschliesst sich mit der Erkenntnis, dass Endbenutzer die Suchmaschine benutzen sollen. In unserem Beispiel würde ich also zum Standard Analyzer raten. Er lieferte ein gutes Suchergebnis in einer guten Zeit. Liegt der Fokus aber nicht in einer sehr guten Suchmaschine, sondern nur in der Unterstützung (zum Beispiel für ein Intranet in einem Geschäft) des Benutzers, ist es wichtiger, dass die Indexierung schnell durchläuft. Da der zweite Anwendungsfall darauf hinausläuft, dass die Indexierung immer wieder durchgeführt wird. Da ist ein schneller Durchlauf höher gewichtet, als die effektive Suchleistung. In solch einem Fall würde ich doch eher zum Whitespace Analyzer raten.

Information Retrieval ist ein komplexes Thema. In Verbindung mit Lucene könnte man komplette Bücher schreiben (wie dies ja auch geschehen ist). Lucene hat seine Arbeit sauber ausgeführt, auch die Erkennung von MP3-Dateien war gut zu implementieren. Obwohl Lucene optimiert wurde für Textdateien, konnte mit der Hilfe vom JAudioTagger die Aufgabe gut gelöst werden. Lucene ist ein sehr mächtiges Tool. Aber es muss gut analysiert werden, wie die Suchmaschine eingesetzt werde soll.

Die Indexierung von Ton hingegen ist nochmals ein ganz anderes Thema. Akustischer Fingerabdruck und spektrale Flachheit sind nur kleine Einblicke in ein ganz grosses Themengebiet der Akustik. Aber das Beispiel von Shazam zeigt schön, wie genial die Idee ist, die die Hersteller bedient haben. Mit mehr als 400 Millionen Benutzer haben sie ein Bedürfnis entdeckt. So einfach und zuverlässig wie Shazam, wurde eine Top-App gebaut.

Glossar

MP3
Bezeichnung für MPEG-1 Audio Layer III oder MPEG-2 Audio Layer III. Ist ein Verfahren der verlustbehafteten Kompression.

Query
Bezeichnet die Abfrage in einer Sprache.

Java
Java ist eine objektorientierte Programmiersprache.

GUI
Steht für Graphical User Interface und ist die Abkürzung für die Benutzeroberfläche.

Methode
Bezeichnet in diesem Fall ein Code-Teil in Java, welcher einen Ablauf von Operationen durchführt.

Parser
Analysiert und wandelt eine Eingabe in ein anderes, für die Weiterverarbeitung brauchbares, Format um.

Offset
Bezeichnet den Abstand zum Beginn innerhalb einer Datei auf Byte-Ebene.

Solid State Disk
Ist eine Harddisk, welche ohne beweglichen Teile auskommt und eine sehr hohe Lese- und Schreibgeschwindigkeit besitzt.

Literaturverzeichnis

[1] Diverse. Akustischer fingerabdruck. Webseite. http://de.wikipedia.org/wiki/Akustischer_Fingerabdruck besucht am 09.05.2014.

[2] Diverse. Id3-tag. Webseite. http://de.wikipedia.org/wiki/ID3-Tag besucht am 07.05.2014.

[3] Diverse. Mp3. Webseite. http://de.wikipedia.org/wiki/MP3 besucht am 09.05.2014.

[4] Diverse. Query. Webseite. http://de.wikipedia.org/wiki/Query besucht am 09.05.2014.

[5] Diverse. Shazam (dienst). Webseite. http://de.wikipedia.org/wiki/Shazam_(Dienst) besucht am 09.05.2014.

[6] Mike McCandless Erik Hatcher, Otis Gospodnetic. *Lucene in Action*. Manning, 2010.

[7] flaviomartins. Analyzerutils.java. Webseite. https://github.com/behas/lucene-skos/blob/master/src/test/java/at/ac/univie/mminf/luceneSKOS/util/AnalyzerUtils.java besucht am 07.05.2014.

[8] Fraunhfer-Gesellschaft. äkustischer fingerabdruck"verrät musiktitel. Webseite. http://www.scinexx.de/wissen-aktuell-1802-2004-10-26.html besucht am 09.05.2014.

[9] Berthier Ribeiro-Neto Ricardo Baeza-Yates. *Modern Information Retrieval - the concepts and technology behind search*. Addison Wesley, 2011.

[10] Edda Schlager. Kampf der namenlosen ohrwürmer. Webseite. http://www.scinexx.de/dossier-detail-341-11.html besucht am 09.05.2014.

[11] Unknown. Apache tika. Webseite. http://tika.apache.org/ besucht am 07.05.2014.

[12] Unknown. Id3v2. Webseite. http://www.id3.org besucht am 07.05.2014.

[13] Unknown. Jaudiotagger. Webseite. http://www.jthink.net/jaudiotagger/ besucht am 04.05.2014.

[14] Unknown. Lucene intro. Webseite. https://today.java.net/pub/a/today/2003/07/30/LuceneIntro.html besucht am 04.05.2014.

[15] Martin Weigert. Shazam hat mehr als 400 millionen nutzer. Webseite. http://netzwertig.com/2013/12/09/musikalischer-riese-shazam-hat-mehr-als-400-millionen-nutzer/ besucht am 09.05.2014.

Appendices

Anhang A

Main.java

```
package ch.phischa;

public class Main {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		new GUI();
	}

}
```

Anhang B

GUI.java

```
package ch.phischa;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Vector;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.table.DefaultTableModel;

import org.apache.lucene.document.Document;

public class GUI {
```

```
private JFrame mainFrame;
private JTextField suche, pfadFeld;
private JButton sucheButton, indexButton;
private JLabel dauer, dauerTitel, ergebnis;
private JScrollPane scroll;
private ButtonGroup bgroup;
public GUI()
{
        createFrame();
}

private void createFrame()
{
        mainFrame = new JFrame("Lucene Search Engine");
        JMenuBar menubar = createMenuBar();

        JLabel sucheLabel = new JLabel("Suche:");
        JLabel indexLabel = new JLabel("Pfad:");
        suche = new JTextField(20);
        sucheButton = new JButton("Suche");
        dauer = new JLabel();
        dauerTitel = new JLabel("Dauer:");
        pfadFeld = new JTextField(20);
        indexButton = new JButton("Index");

        JPanel northPanel = new JPanel();
        final JPanel ostPanel = new JPanel();
        JPanel southPanel = new JPanel();
        final JPanel centerPanel = new JPanel();
        northPanel.add(sucheLabel);
        northPanel.add(suche);
        northPanel.add(sucheButton);
        ostPanel.add(dauerTitel);
        ostPanel.add(dauer);
        southPanel.add(indexLabel);
        southPanel.add(pfadFeld);
        southPanel.add(indexButton);

        mainFrame.add(northPanel,BorderLayout.NORTH);
        mainFrame.add(ostPanel,BorderLayout.EAST);
        mainFrame.add(southPanel,BorderLayout.SOUTH);
```

```
mainFrame.setJMenuBar(menubar);

mainFrame.pack();
mainFrame.setSize(700,800);
mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
mainFrame.setVisible(true);

sucheButton.addActionListener(new ActionListener(){

        @Override
        public void actionPerformed(ActionEvent arg0) {
                // TODO Auto-generated method stub
                String analyzer = bgroup.getSelection().get
                Searcher searcher = new Searcher(pfadFeld.g
                dauer.setText(String.valueOf(searcher.getDa
                ArrayList<Document> results = searcher.getI
                String cols[] = {"Ergebnisse"};
                DefaultTableModel tableModel = new DefaultT
                JTable table = new JTable(tableModel);
                tableModel.getDataVector().removeAllElemen
                int i = 0;

                if(results.isEmpty())
                {
                        JOptionPane.showMessageDialog(new J
                }
                else
                {
                        for (Document doc:results)
                        {
                                tableModel.addRow((Vector)
                                tableModel.setValueAt(doc.g
                                i++;
                        }
                }
                scroll = new JScrollPane(table);
                centerPanel.removeAll();
                centerPanel.add(scroll);
                mainFrame.add(centerPanel);
                mainFrame.repaint();
        }

});
```

```
        indexButton.addActionListener(new ActionListener()
        {

                @Override
                public void actionPerformed(ActionEvent arg0) {
                        // TODO Auto-generated method stub
                        String analyzer = bgroup.getSelection().get
                        Index indexer = new Index(pfadFeld.getText
                        long dauerZeit = indexer.getTime();
                        dauer.setText(String.valueOf(dauerZeit) + "
                }

        });
}

private JMenuBar createMenuBar()
{
        JMenuBar menubar = new JMenuBar();
        JMenu datei, einstellungen;
        JMenuItem schliessen,analyzerDemo;

        JRadioButtonMenuItem saButton = new JRadioButtonMenuItem("S
        JRadioButtonMenuItem wsButton = new JRadioButtonMenuItem("W
        JRadioButtonMenuItem stButton = new JRadioButtonMenuItem("S
        JRadioButtonMenuItem smButton = new JRadioButtonMenuItem("S

        saButton.setActionCommand("Standard Analyzer");
        wsButton.setActionCommand("Whitespace Analyzer");
        stButton.setActionCommand("Stop Analyzer");
        smButton.setActionCommand("Simple Analyzer");

        bgroup = new ButtonGroup();
        bgroup.add(saButton);
        bgroup.add(smButton);
        bgroup.add(wsButton);
        bgroup.add(stButton);

        datei = new JMenu("Datei");
        einstellungen = new JMenu("Einstellungen");

        schliessen = new JMenuItem("Schliessen");
        analyzerDemo = new JMenuItem("Analyzer Demo");
```

```
                analyzerDemo.addActionListener(new ActionListener()
                {

                        @Override
                        public void actionPerformed(ActionEvent arg0) {
                                // TODO Auto-generated method stub
                                try {
                                        new AnalyzerDemo();
                                } catch (IOException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                        }

                });

                schliessen.addActionListener(new ActionListener(){

                        @Override
                        public void actionPerformed(ActionEvent e) {
                                // TODO Auto-generated method stub
                                mainFrame.dispose();
                        }

                });

                datei.add(schliessen);
                einstellungen.add(saButton);
                einstellungen.add(smButton);
                einstellungen.add(wsButton);
                einstellungen.add(stButton);
                einstellungen.add(analyzerDemo);

                menubar.add(datei);
                menubar.add(einstellungen);

                return menubar;
        }
}
```

Anhang C

Searcher.java

```
package ch.phischa;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;

import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.WhitespaceAnalyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.queryParser.MultiFieldQueryParser;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Searcher {

                private static long dauer;
                private static ArrayList<Document> results = new ArrayList<
```

```
public Searcher(String pfad, String query, String analyzerA
{
        String indexDir = pfad + File.separator + "Index";

        try {
                search(indexDir,query, analyzerArt);
        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } catch (ParseException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
}

@SuppressWarnings("deprecation")
public static void search(String indexDir, String q, String
{
        Directory dir = FSDirectory.open(new File(indexDir)
        @SuppressWarnings("resource")
        IndexSearcher is = new IndexSearcher(dir);

        MultiFieldQueryParser parser;

        if(analyzerArt.equals("Whitespace Analyzer"))
        {
                parser = new MultiFieldQueryParser(Version.
        }
        else if(analyzerArt.equals("Stop Analyzer"))
        {
                parser = new MultiFieldQueryParser(Version.
        }
        else if(analyzerArt.equals("Simple Analyzer"))
        {
                parser = new MultiFieldQueryParser(Version.
        }
        else
        {
                parser = new MultiFieldQueryParser(Version.
        }
```

```
                Query query = parser.parse(q);
                long start = System.currentTimeMillis();
                TopDocs hits = is.search(query, 100);
                long end = System.currentTimeMillis();
                dauer = end - start;

                System.out.println("Gefunden " + hits.totalHits + "

                results.clear();

                for(ScoreDoc scoredoc : hits.scoreDocs)
                {
                        Document doc = is.doc(scoredoc.doc);
                        results.add(doc);
                }

                for (Document doc:results)
                {
                        System.out.println(doc.get("fullpath"));
                }
        }

        public long getDauer()
        {
                return dauer;
        }

        public ArrayList<Document> getResult()
        {
                return results;
        }
}
```

Anhang D

Index.java

```
package ch.phischa;

import java.io.File;
import java.io.FileFilter;
import java.io.FileReader;
import java.io.IOException;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.WhitespaceAnalyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;
import org.jaudiotagger.audio.AudioFile;
import org.jaudiotagger.audio.AudioFileIO;
import org.jaudiotagger.audio.AudioHeader;
import org.jaudiotagger.audio.mp3.MP3AudioHeader;
import org.jaudiotagger.audio.mp3.MP3File;
import org.jaudiotagger.tag.FieldKey;
import org.jaudiotagger.tag.Tag;
import org.jaudiotagger.tag.id3.AbstractID3v2Tag;
import org.jaudiotagger.tag.id3.ID3v1Tag;
```

```
import org.jaudiotagger.tag.id3.ID3v24Frames;
import org.jaudiotagger.tag.id3.ID3v24Tag;

public class Index {

	public IndexWriter writer;
	long dauer;

	public Index(String pfad, String analyzerArt)
	{

		String pfadIndex = pfad + File.separator + "Index";
		int numIndexed = 0;
		try {
			Indexer(pfadIndex,analyzerArt);
		} catch (IOException e1) {
			// TODO Auto-generated catch block
			e1.printStackTrace();
		}

		long start = System.currentTimeMillis();

		try{
			numIndexed = index(pfad, new MPFileFilter());
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} finally {
			close();
		}

		long end = System.currentTimeMillis();

		System.out.println("Indexing " + numIndexed + " in " + (end
		dauer = end - start;
	}

	@SuppressWarnings("deprecation")
	public void Indexer(String indexDir, String analyzerArt) throws IO
	{
		Directory dir = FSDirectory.open(new File(indexDir));

		if(analyzerArt.equals("Whitespace Analyzer"))
```

```
        {
                writer = new IndexWriter(dir,new WhitespaceAnalyzer
        }
        else if(analyzerArt.equals("Stop Analyzer"))
        {
                writer = new IndexWriter(dir,new StopAnalyzer(Versi
        }
        else if(analyzerArt.equals("Simple Analyzer"))
        {
                writer = new IndexWriter(dir,new SimpleAnalyzer(Ve
        }
        else
        {
                writer = new IndexWriter(dir,new StandardAnalyzer(V
        }



}

public void close()
{
        try {
                writer.close();
        } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
}

public int index(String dataDir, FileFilter filter) throws Exceptio
{
        File[] files = new File(dataDir).listFiles();

        for(File f:files)
        {
                if(f.isDirectory())
                {
                        index(dataDir+File.separator+f.getName(),fi
                }
                else if(!f.isHidden() && f.canRead() && f.exists()

                {
```

```
                        indexFile(f);
                }
        }

        return writer.numDocs();
}

private static class MPFileFilter implements FileFilter
{
        public boolean accept(File path)
        {
                return path.getName().toLowerCase().endsWith("mp3")
        }
}

protected Document getDocument(File f) throws Exception
{
        AudioFile audioFile = AudioFileIO.read(f);
        Tag tag = audioFile.getTag();
        AudioHeader header = audioFile.getAudioHeader();

        System.out.println(tag.getFirst(FieldKey.ARTIST));

        Document doc =  new Document();
        doc.add(new Field("filename",f.getName(),Field.Store.YES,F
        doc.add(new Field("fullpath",f.getCanonicalPath(),Field.Sto
        doc.add(new Field("KÃ¼nstler",tag.getFirst(FieldKey.ARTIST)
        doc.add(new Field("Album",tag.getFirst(FieldKey.ALBUM),Fiel
        doc.add(new Field("Titel",tag.getFirst(FieldKey.TITLE),Fiel
        doc.add(new Field("Jahr",tag.getFirst(FieldKey.YEAR),Field.
        doc.add(new Field("Songtext",tag.getFirst(FieldKey.LYRICS),

        return doc;
}

private void indexFile(File f) throws Exception
{
        System.out.println(f.getName());
        System.out.println("Indexing " + f.getCanonicalPath());
        Document doc = getDocument(f);
        writer.addDocument(doc);
}
```

```
        public long getTime()
        {
                return dauer;
        }
}
```

Anhang E

AnalyzerDemo.java

```
package ch.phischa;

import java.io.IOException;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.WhitespaceAnalyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.util.Version;

public class AnalyzerDemo {

        private static final String[] examples = {"The quick brown fox jump
        @SuppressWarnings("deprecation")
        private static final Analyzer[] analyzers = new Analyzer[] {
                new StandardAnalyzer(Version.LUCENE_30),new SimpleAnalyzer(
        };

        public AnalyzerDemo() throws IOException
        {

                String[] strings = examples;

                for(String s:strings)
                {
                        analyze(s);
                }
```

```
        }

        private static void analyze(String string) throws IOException
        {
                System.out.println("Analyzing \"" + string + "\"");

                for(Analyzer analyzer:analyzers)
                {
                        String name = analyzer.getClass().getSimpleName();
                        System.out.println("     " + name + "
");
                        System.out.println("     ");
                        AnalyzerUtils.displayTokens(analyzer,string);
                        System.out.println("\n");
                }
        }


}
```

Anhang F

AnalyzerUtils.java

```
package ch.phischa;

import java.io.IOException;
import java.io.StringReader;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.tokenattributes.CharTermAttribute;
import org.apache.lucene.analysis.tokenattributes.OffsetAttribute;
import org.apache.lucene.analysis.tokenattributes.PayloadAttribute;
import org.apache.lucene.analysis.tokenattributes.PositionIncrementAttribut
import org.apache.lucene.analysis.tokenattributes.TypeAttribute;
import org.apache.lucene.util.BytesRef;

/**
 * Utils for displaying the results of the Lucene analysis process
 */
public class AnalyzerUtils {

  public static void displayTokens(Analyzer analyzer, String text)
      throws IOException {
    displayTokens(analyzer.tokenStream("contents", new StringReader(text)))

  }

  public static void displayTokens(TokenStream stream) throws IOException {

    CharTermAttribute term = stream.addAttribute(CharTermAttribute.class);
    while (stream.incrementToken()) {
```

```
      System.out.println("[" + term.toString() + "] ");
    }

  }

  public static void displayTokensWithPositions(Analyzer analyzer, String t
      throws IOException {

    TokenStream stream = analyzer.tokenStream("contents",
        new StringReader(text));

    CharTermAttribute term = stream.addAttribute(CharTermAttribute.class);
    PositionIncrementAttribute posIncr = stream
        .addAttribute(PositionIncrementAttribute.class);

    int position = 0;
    while (stream.incrementToken()) {

      int increment = posIncr.getPositionIncrement();
      if (increment > 0) {
        position = position + increment;
        System.out.println();
        System.out.print(position + ":");
      }

      System.out.print("[" + term.toString() + "] ");

    }
    System.out.println();

  }
}
```