Seminararbeit

zum Thema

Effizientes Suchen mit Jakarta Lucene

erarbeitet von

Tilman Schneider

betreut durch

Prof. Klaus Gremminger

Inhaltsverzeichnis

1 Einführung....3
2 Grundlagen....5
2.1 Definition: Suchmaschine....5
2.2 Marktübersicht: Welche Suchmaschinen gibt es?....6
2.3 Vorstellung Lucene....7
3 Aufbau eines Suchindex....9
3.1 Umgekehrter Wortindex....10
3.2 Q-Gramme-Index....11
3.3 Bewertung eines Treffers....14
3.4 Indexoptimierung....14
3.5 Das Indexformat von Lucene....15
4 Erstellung eines Suchindex....17
4.1 Aufbereitung des Suchraums....17
4.2 Textanalyse: Vom Text zum Wortkürzel....18
4.3 Documents – Wie man die Textdaten an Lucene übergibt....19
4.4 Codebeispiel....20
5 Die Suche....24
5.1 Die Anfragesyntax von Lucene....24
5.2 Erzeugen einer Trefferliste....27
5.3 Codebeispiel....28
6 Fazit....30
Anhang A Quellen....32

1 Einführung

In vielen Bereichen findet man große Mengen von Texten. Sei es die private Festplatte, das private E-Mail-Archiv, der große Dateiserver einer Firma, auf dem das gesammelte Know-How lagert oder auch das gesamte Internet.

All diese Textarchive sind nach einem bestimmten Schema aufgebaut. So sind die Briefe einer Firma beispielsweise nach Geschäftsjahren oder Kunden in Verzeichnissen gruppiert. Sucht man ein Dokument anhand dieser Struktur, so wird man schnell fündig. Was soll man jedoch tun, wenn man das Jahr oder den Kunden nicht genau kennt, sondern nur nach Briefen sucht, in denen eine bestimmte Person erwähnt wurde?

In solchen Fällen ist man auf eine Volltextsuche angewiesen. Mit der Funktion „Dateien suchen“ eines Dateimanagers stößt man dabei schnell an die Grenzen des Erträglichen, da solche Programme immer alle Dateien von vorne bis hinten durchsuchen müssen und dafür entsprechend viel Zeit brauchen.

Suchmaschinen ermöglichen eine schnelle Volltextsuche über große Datenmengen, indem sie die Daten vor der Suche analysieren und so aufbereiten, dass die spätere Suche sehr schnell wird. Wie das im Allgemeinen funktioniert und wie Lucene das im Speziellen macht, ist Gegenstand dieser Seminararbeit.

Zunächst soll ein Überblick über das Thema Suchmaschinen gegeben werden. Danach werden der Aufbau eines Suchindex und die Vorgänge bei der Indexierung erläutert. Schließlich wird gezeigt, wie der Suchindex bei einer Suchanfrage eingesetzt wird.

Um die einzelnen Schritte besser zu veranschaulichen, wird alles anhand eines Anwendungsbeispiels erläutert: Eine Firma hat in ihrem Firmennetzwerk einen Dateiserver, auf dem sich eine sogenannte „Knowledge Base“ befindet. Dabei handelt es sich um bunte Mischung aus Powerpoint-Präsentationen, Word-Dokumenten, PDF-Dateien und HTML-Seiten, die Informationen enthalten, die von Mitarbeitern für ihre Kollegen zur Verfügung gestellt wurden. Die Dateien haben zusammen eine Größe von 5 GB und enthalten interne Notizen, die beispielsweise bestimmte Geschäftsprozesse oder Projekte

beschreben. Es befinden sich dort auch „nützliche Informationen“, wie z.B. Artikel über neue Technologien oder Benutzerhandbücher.

Um schneller an eine gewünschte Information heranzukommen, will diese Firma nun eine Suchmaschine in ihre Intranet-Seiten integrieren, mit der eine Volltextsuche über die Dokumente in der Knowledge Base ermöglicht wird.

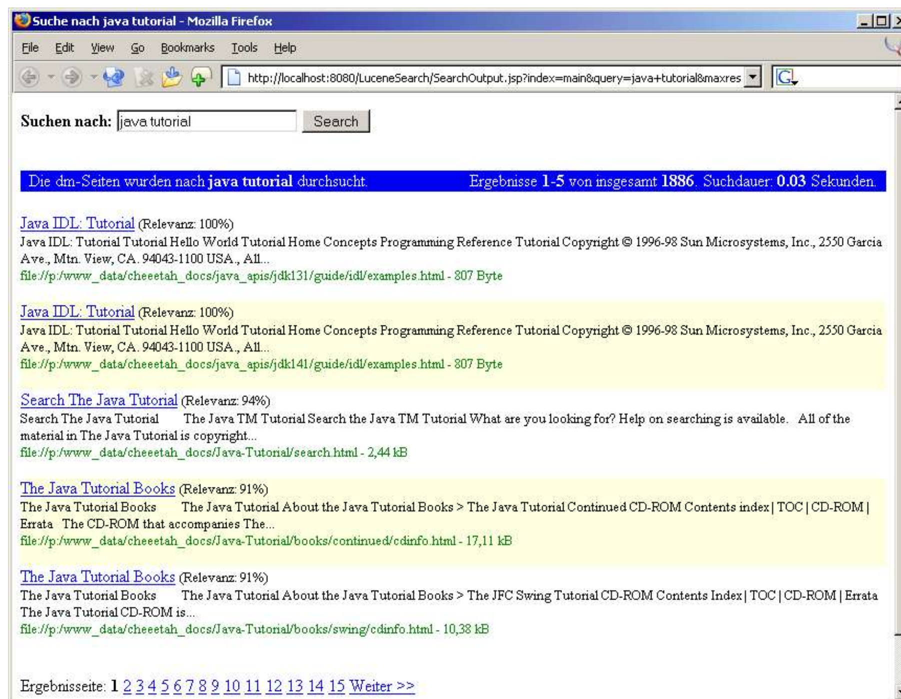

Abbildung 1 Suchmaske im Intanet

2 Grundlagen

Dieses Kapitel erklärt, was eine Suchmaschine überhaupt ist, welche Suchmaschinen es auf dem Markt gibt und wie Lucene darin einzuordnen ist.

2.1 Definition: Suchmaschine

Eine Suchmaschine ist eine spezielle Unterart eines Suchsystems. Ein Suchsystem ist ein System, das dem Anwender hilft, bestimmte Informationen zu finden.

Man unterscheidet dabei zwei Arten von Suchsystemen: Zum einen gibt es menü-orientierte Suchsysteme, sog. Verzeichnisse, zum anderen anfrage-orientierte, auch Suchmaschinen genannt.

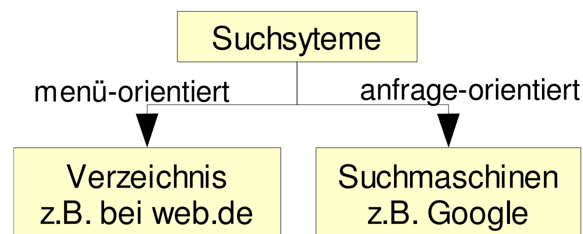

Abbildung 2: Suchsysteme

Verzeichnisse erlauben die Navigation zu bestimmten Informationen anhand einer bestimmten Struktur. So ist beispielsweise ein Dateisystem ein Verzeichnis. Das Dateisystem hält eine logische Baumstruktur aller Dateien. Für jede Datei hält es eine Liste von Blöcken, die zu dieser Datei gehören. Das Betriebsystem kann nun in dieser Baumstruktur navigieren und die Daten der Dateien lesen, indem es rekonstruiert, welche Blöcke zur betreffenden Datei gehören.

Ein anderes Beispiel für ein Verzeichnis ist ein Webverzeichnis, wie man es bei web.de[1] oder Yahoo[2] findet. Hier werden Webseiten in Kategorien und Unterkategorien eingeteilt, in denen der Benutzer dann navigieren kann.

1 Siehe: http://dir.web.de

2 Siehe: http://de.dir.yahoo.com

Eine Suchmaschine dagegen ist ein System zur Volltextsuche in großen Datenmengen. Im Gegensatz zu Verzeichnissen findet der Benutzer die gewünschte Information nicht durch Navigation in einer virtuellen Struktur, sondern indem er dem System eine Suchanfrage stellt. Eine solche Suchanfrage besteht meist aus einer Liste von Stichwörtern und kann weitere Elemente beinhalten, die die Anfrage genauer beschreiben. So kann beispielsweise angegeben werden, dass die gesuchte Information ein bestimmtes Wort *nicht* enthalten soll. Weitere Details dazu finden Sie in Kapitel 5.1 auf Seite 24.

Suchmaschinen können in vielen Bereichen eingesetzt werden: Für die Suche innerhalb einer Webseite oder dem ganzen Internet, für die Suche von E-Mails oder lokalen Dateien oder auch in einer Onlinehilfe. Lucene wird beispielsweise in der Onlinehilfe der Entwicklungsumgebung Eclipse genutzt.

2.2 Marktübersicht: Welche Suchmaschinen gibt es?

Name	***Beschreibung***	***Kosten***
grep	Eigentlich keine Suchmaschine, da kein Suchindex aufgebaut wird. Es müssen daher immer alle Daten durchsucht werden. Jedoch sehr beliebt in der Unix-Welt.	Open Source, Frei
Perlfect Search	Einfaches Tool für Suche in Webseiten. Kann jedoch keine fortgeschrittenen Funktionen wie z.B. Wildcardsuche.	Open Source, Frei
Isearch	Kommandozeilentool das viele verschiedene Dokumentenformate beherrscht. Bietet jedoch nur sehr eingeschränkte Wildcardsuche (Nur am Ende eines Wortes, z.B. Auto*).	Open Source, Frei

Name	Beschreibung	Kosten
glimpse	Kommandozeilentool. Bietet auch Wildcardsuche, sogar Reguläre Ausdrücke und fehlertolerante Suche. Webglimpse setzt auf glimpse auf und bietet eine Webschnittstelle.	Open Source, Frei für nichtkommerzielle Zwecke
ht://dig	Recht mächtige Websuchmaschine. Kann Dateien indizieren, die vom Dateisystem kommen oder über HTTP erreichbar sind und unterstützt viele Dokumentformate	Open Source, Frei
Verity Information Server	Äußerst mächtiges, kommerzielles System. Hat sehr viele Funktionen und kann fast alle Dokumentformate lesen.	Mehrere 10.000 $
Swish-E	Recht mächtige Websuchmaschine. Kann Dateien indizieren, die vom Dateisystem kommen oder über HTTP erreichbar sind und unterstützt viele Dokumentformate	Open Source, Frei

Abbildung 3: Übersicht über Suchmaschinen[3]

2.3 Vorstellung Lucene

Lucene ist ein Projekt der Apache Jakarta Gruppe und wurde 1997/98 von Doug Cutting gegründet. Doug Cutting ist ein Veteran im Bereich Suchmaschinen.

Lucene ist 100% pure Java. Es gibt jedoch mittlerweile auch Implementierungen in C++, .NET, Python und Perl. Eine Implementierung in Ruby ist gerade am Entstehen.

3 Nach Gregor Fischer: http://www2.informatik.uni?wuerzburg.de/staff/holger/lehre/OSWS0001/fischer/os0001_fischer.pdf

Bei der Entwicklung von Lucene wurde darauf geachtet, dass es sehr vielseitig einsetzbar ist. So werden die Eingangsdaten so abstrahiert, dass man schlicht alles, was im Entferntesten mit Text zu tun hat, als Grundlage für die Indexierung nutzen kann.

Lucene ist daher keine Applikation, die sich auf ein bestimmtes Szenario eingeschossen hat, sondern eine API. Das ist gleichzeitig seine Stärke und seine Schwäche. Einerseits kann man Lucene nicht „out of the box" verwenden, andererseits bietet es alles, was man braucht, wenn man in einem Projekt Suchfunktionalität benötigt. Lucene ist dabei so einfach anzuwenden, dass es innerhalb kürzester Zeit integriert werden kann.

© 2004 Tilman Schneider

3 Aufbau eines Suchindex

Eine Volltextsuche erreicht man am einfachsten, indem man den gesamten Suchraum Dokument für Dokument nach den gesuchten Wörtern durchforstet. Dieses Vorgehen funktioniert auch gut bei kleinen Datenmengen und kommt beispielsweise in vielen E-Mail-Programmen zum Einsatz.

Die Suchdauer steigt jedoch linear mit der Größe des Suchraums und wird bei großen Datenmengen inakzeptabel groß. Im Falle der Knowledge Base würde selbst unter der Annahme, dass die vollen 100 Mbit/s des Netzwerks zur Verfügung stehen, eine primitive Suche über 5 GB knappe 7 Minuten dauern.

In der Praxis kommen Protokolloverhead von TCP/IP und dem Dateitransferprotokoll hinzu und die Tatsache, dass sich im Netzwerk auch andere Teilnehmer befinden. Außerdem ist die Vorgehensweise der Suchprogramme nicht auf Netzwerkzugriff optimiert: Die Daten werden meist blockweise gelesen, d.h. es wird abwechselnd ein Block gelesen und dann durchsucht. Selbst unter guten Bedingungen kommen in der Praxis schnell Suchzeiten von über einer halben Stunde zu Stande. Bei einer Suche mit Hilfe eines Suchindex hingegen, beträgt die Suchdauer dagegen nur einige Hunderstel Sekunden.

Es liegt also nahe, den Suchraum in irgendeiner Weise aufzubereiten und in einer geeigneten Datenstruktur abzulegen. Eine solche Datenstruktur nennt man Suchindex. Sie hat das primäre Ziel, möglichst schnell jene Stellen zu identifizieren, an denen ein bestimmtes Wort steht.

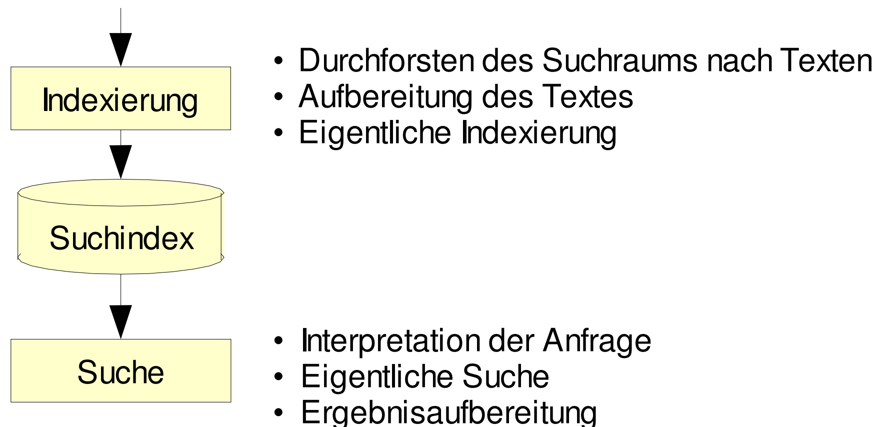

Abbildung 4 Übersicht: Ablauf einer Suche mit Suchindex

3.1 Umgekehrter Wortindex

Beim umgekehrten Wortindex wird jede Fundstelle eines Wortes mit dem Wort als Schlüssel gespeichert.

Ein Beispiel: Es soll der Satz „Blaukraut bleibt Blaukraut und Brautkleid bleibt Brautkleid“ indiziert werden.

Daraus ergibt sich folgender Index:

Wort	***Fundstelle***
blaukraut	0, 17
bleibt	10, 42
brautkleid	31, 49
und	27

Abbildung 5: Beispiel für umgekehrten Wortindex

Als Datenstruktur werden meist Hash-Tabellen oder B-Bäume genutzt. Bei Hash-Tabellen ist der Aufwand des Findens O(1), bei B-Bäumen O(log n), wenn n die Anzahl der Wörter ist.

© 2004 Tilman Schneider

Bei der Suche muss jetzt nur noch der Index abgefragt werden und man erhält die Fundstellen.

Allerdings kann auf diese Weise das Wort „Kraut“ nicht gefunden werden, obwohl es im ursprünglichen Satz vorkommt. Dieses Problem kann man jedoch mit einer Substring-Suche umgehen, indem man innerhalb der Wortliste eine Volltextsuche vornimmt, um Teilworte zu identifizieren.

Die Suche nach Kraut ergäbe in unserem Fall:

„Kraut“ => „Blaukraut“ + 4 => (0, 17) + 4 => (4, 21)

Natürlich wird die Suche dabei sehr viel langsamer, weil man sich nicht mehr die Datenstruktur des Indexes zunutze machen kann: Der Aufwand steigt zu O (n). Allerdings ist sie trotzdem noch wesentlich schneller als eine echte Volltextsuche mit dem Aufwand O(m), da die Menge von Wörtern (n) immer kleiner ist, als der ursprüngliche Suchraum (m), da davon auszugehen ist, dass die meisten Wörter mehrfach vorkommen.

Lucene führt daher nur dann eine Substring-Suche durch, wenn explizit eine Wildcard-Suche gefordert wurde, wenn also „b*kraut“ angefragt wurde. Das führende „b“ ist dabei leider notwendig, weil Lucene keine komplette Substring-Suche durchführt und daher Wildcards am Anfang von Suchwörtern verbietet.

3.2 Q-Gramme-Index

Bei einem Q-Gramme-Index werden nicht ganze Wörter mit einer variablen Länge, sondern Q-Gramme mit der Länge q gespeichert.

Bei der Indexierung werden die Q-Gramme identifiziert, indem ein Fenster der Größe q über die Eingangsdaten geschoben und jedes Q-Gramm mit der Position gespeichert wird.

© 2004 Tilman Schneider

Bei q = 3 ergibt die Indexierung des Satzes „Herr Herrmanns Herz“ folgenden Index:

Wort	***Fundstelle***
_he	4, 14
ann	10
err	1, 6
erz	16
her	0, 5, 15
man	9
nns	11
ns_	12
r_h	3
rma	8
rr_	2
rrm	7
s_h	13

Abbildung 6: Beispiel für Q-Gramme-Index

Bei der Suche wird das Suchwort ebenfalls in Q-Gramme zerlegt. Dabei muss das Fenster jedoch nicht um 1 weitergeschoben werden. Ein Verschieben um q reicht aus. Wenn die Länge des Suchwortes kein Vielfaches von q ist, dann wird das Fenster so weit verschoben, wie es gerade noch geht. Die Differenz zu q wird zusammen mit den Q-Grammen gespeichert.

Das Wort „Herrmann“ ergibt dabei folgende Q-Gramme:

„her“, „rma“, „ann“ (-1)

Mit Hilfe des Index werden nun die Fundstellen zu den Q-Grammen ermittelt. Dann werden die Fundstellen der Q-Gramme verglichen, bei Unstimmigkeiten wird eine Fundstelle verworfen.

- „her“ hat die Fundstellen 0, 5 und 15.

- Das folgende Q-Gramm „rma“ muss nun an den Stellen 0+3, 5+3 und 15+3 zu finden sein.
- Da „rma“ jedoch nur die Fundstelle 8 hat (5+3), werden 0 und 15 verworfen.
- Das nun folgende Q-Gramm „ann“ muss an der Stelle 8+3-1 zu finden sein.
- Das ist auch der Fall: „ann“ hat die Fundstelle 10.
- Die Suche ergibt also einen Treffer, und zwar 5.

Ein Q-Gramme-Index erfordert zwar einen einen kleinen Overhead zum Verifizieren der Fundstellen, dafür bietet er eine echte Volltextsuche. Außerdem können sehr einfach Wildcards unterstützt werden, ohne dass der Aufwand zu O(n) degeneriert.

Allerdings kann nicht nach Worten gesucht werden, die kleiner als q sind. Das ist besonders bedauerlich, weil diese Einschränkung auch für Teilworte einer Wildcardsuche gilt. So kann beispielsweise nicht nach „He*ma*“ gesucht werden.

Ein weiterer, viel entscheidenderer Nachteil ist, dass der Index sehr groß wird. Nach Gregor Fischer[4] ist ein Index der Q-Gramme-basierten Suchmaschine „Q-Gramm-Suche aus HagerROM 2001“ 8 mal größer als die Ursprungsdaten selbst.

Hält man dagegen, dass ein umgekehrter Wortindex meist wesentlich kleiner als die Ursprungsdaten ist, und dass eine Suchmaschine definitionsgemäß für große Datenmengen eingesetzt wird, erkennt man schnell, dass ein Q-Gramme-Index in seiner Urform nicht praxistauglich ist.

Lucene nutzt daher einen umgekehrten Wortindex, unterstützt allerdings auch Wildcardsuchen, mit der Einschränkung, dass keine Wildcards am Anfang eines Suchwortes stehen dürfen.

4 Siehe „Suchmaschinen-Vergleich von Gregor Fischer“

© 2004 Tilman Schneider

3.3 Bewertung eines Treffers

Zusätzlich zur Fundstelle können im Index noch weitere Informationen abgelegt werden, die eine Bewertung der Fundstelle erlauben. So können Treffer, die relevanter für ein bestimmtes Wort sind, im Suchergebnis weiter oben angezeigt werden.

Um weitere Zeit bei der Suche einzusparen, können die Fundstellen eines Wortes bereits bei der Erstellung des Index nach dieser Bewertung vorsortiert werden, so dass sie beim Auslesen bereits in der gewünschten Reihenfolge vorliegen.

Die Bewertung eines Treffers ist eine Wissenschaft für sich. Google[5] beispielsweise kann seinen Erfolg auf eine einfache, aber damals revolutionäre Bewertungweise begründen. Google sieht eine Webseite dann als besonders relevant an, wenn es viele andere Webseiten gibt, die darauf verweisen. Mittlerweile können Firmen auch dafür bezahlen, dass ihre Webseite bei bestimmten Worten sehr weit vorne aufgelistet wird.

Bei Lucene wird für diese Bewertung die Häufigkeit des Wortes im betreffenden Dokument herangezogen. Ein Indexeintrag besteht bei Lucene daher neben Wort und Fundstelle auch aus der Häufigkeit des Wortes im betreffenden Dokument. Um zu verhindern, dass sehr große Dokumente, die naturgemäß viel mehr Worte enthalten, kleinen Dokumenten gegenüber bevorzugt werden, wird dabei die relative Häufigkeit genutzt. Die relative Häufigkeit berechnet sich aus der absoluten Häufigkeit geteilt durch die Gesamtanzahl an Worten im Dokument.

3.4 Indexoptimierung

Wie bereits erwähnt, beträgt die Suche in einem B-Baum O(log n), wenn n die Anzahl Worte ist. Man kann also die Suchgeschwindigkeit erhöhen, indem man die Menge der Wörter verringert. Das hat gleichzeitig den Vorteil, dass der Index weniger Speicherplatz braucht.

5 Derzeit erfolgreichste Internetsuchmaschine, siehe: http://www.google.de

Lucene nutzt dazu sog. „Stoplisten“. Eine Stopliste enthält Worte wie „und“, „dass“ oder „der“, die einerseits sehr häufig vorkommen, andererseits keine Relevanz für die Suche haben, da niemand nach einem solchen Wort suchen würde. Sie blähen den Index nur unnötig auf, weil sie in fast jedem Dokument vorkommen, sind dabei jedoch völlig unnötig, weil sowieso niemand danach sucht.

Stoplisten haben jedoch den Nachteil, dass sie sprachabhängig sind und gepflegt werden müssen. Ein weiterer Ansatz ist, diese Stopwörter dynamisch zu bestimmen. Wenn ein Wort in mehr als beispielsweise 70% der Dokumente vorkommt, dann handelt es sich wahrscheinlich um ein Füllwort und kann aus dem Index entnommen werden.

Auf die gleiche Weise kann man mit Wörtern verfahren, die besonders selten sind. Bei einem Wort, das nur ein oder zweimal vorkommt, kann man annehmen, dass es ein Tippfehler ist, oder so exotisch, dass auch hier niemand danach suchen würde.

Die dynamische Bestimmung der Stopwörter hat wiederum den Nachteil, dass man den Index erst komplett erstellen muss, um bestimmen zu können, wie oft ein Wort darin vorkommt. In der Praxis werden dabei oft beide Techniken gemeinsam genutzt: Die Stopliste enthält die Wörter, die z.B. in 90% Prozent aller Dokumente vorkommen, so dass diese bereits bei der Erstellung des Index ignoriert werden können. Nach der Indexerstellung kann dann noch ein Feinschliff erfolgen, bei dem die restlichen Stopwörter dynamisch bestimmt und dann entfernt werden.

3.5 Das Indexformat von Lucene

Die meisten Suchmaschinen nutzen B-Bäume für ihre Indizes, da sowohl „Einfügen“ als auch „Suchen“ O(log n) Operationen sind, wenn n die Anzahl der Indexeinträge ist.

Lucene geht an dieser Stelle einen etwas anderen Weg. Anstatt nur einen Index zu verwalten, erzeugt Lucene mehrere Segmente und fasst diese nach und nach zusammen. Für jedes neu indizierte Dokument erzeugt Lucene ein neues Indexsegment. Damit die Suche schnell bleibt, versucht Lucene dabei

stets, die Anzahl der Segmente klein zu halten, indem es kleine Segmente zu einem großen vereinigt. Um den Index für schnelles Suchen zu optimieren, kann Lucene auch alle Segmente zu einem großen Segment vereinen. Das macht vor allem für Indizes Sinn, die sich selten ändern.

Um Konflikte bei gleichzeitigen Lese- und Schreiboperationen auf dem Index zu vermeiden, ändert Lucene niemals bestehende Segmente, sondern erzeugt immer nur neue. Beim Vereinigen von Segmenten erzeugt Lucene ein neues Segment und löscht danach die beiden anderen, nachdem alle aktiven Leseoperationen beendet sind. Dieses Vorgehen hat verschiedene Vorteile: Es skaliert gut, bietet einen guten Kompromiss zwischen Indizierdauer und Suchdauer und erlaubt sowohl schnelles Suchen, als auch schnelles Vereinigen.

Da Segmente niemals geändert werden, können die Daten in einfachen, flachen Dateien gespeichert werden, komplizierte B-Bäume sind nicht nötig.

Ein Indexsegment von Lucene besteht aus verschiedenen Dateien:

- Ein Verzeichnisindex: Enthält einen Eintrag für jeden 100. Eintrag im Verzeichnis.
- Ein Verzeichnis: Enthält einen Eintrag für jedes Wort.
- Eine Fundstellendatei: Enthält die Listen der Fundstellen für jedes Wort.

Die Dateien zeigen jeweils mit Offsets auf die nachfolgende Datei. D.h. der Verzeichnisindex enthält Offsets zu Einträgen im Verzeichnis und das Verzeichnis enthält Offsets zu Einträgen in der Fundstellendatei.

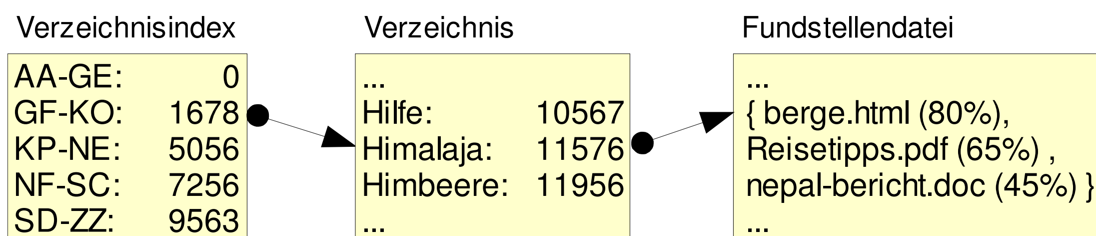

Abbildung 7: Dateien eines Indexsegments von Lucene

Lucene nutzt dabei eine Reihe von Tricks, um das Verzeichnis und die Fundstellendatei zu komprimieren. Dadurch wird der Plattenzugriff verringert, ohne jedoch allzu viel CPU-Overhead zu erzeugen.

4 Erstellung eines Suchindex

Im vorigen Kapitel würde erläutert, wie die Daten über ein Dokument zusammen mit den Worten, die darin vorkommen, so gespeichert werden, dass man schnell jene Dokumente finden kann, die ein bestimmtes Wort enthalten.

Im Folgenden geht es um die Frage, wie man einen Text analysiert, um an die einzelnen Worte zu gelangen und wie man diese Worte so bearbeitet, dass die Qualität des Index verbessert werden kann.

4.1 Aufbereitung des Suchraums

Bei einer primitiven Volltextsuche, wie sie von vielen Dateiverwaltungsprogrammen angeboten wird, werden nur die rohen Daten durchsucht. So kann es passieren, dass Dateien, die das gesuchte Wort enthalten, nicht gefunden werden, weil sie in einem Binärformat wie beispielsweise PDF gespeichert sind.

Bevor der Text eines Dokuments analysiert und indexiert werden kann, muss daher zuerst der eigentliche Text extrahiert werden. Dieser Vorgang hängt sehr stark vom Dateiformat des vorliegenden Dokuments ab.

Bei einem HTML-Dokument muss man beispielsweise die HTML-Tags entfernen, die Zeichenkodierung berücksichtigen und HTML-Entitäten in echte Zeichen übersetzten. Bei einem Worddokument ist die Extraktion des Textes schon etwas schwieriger, weil Word seine Dokumente in einem proprietären Binärformat speichert. Es gibt zwar Open-Source-Projekte, die das Word-Format lesen können, jedoch funktioniert das nicht in jedem Fall fehlerfrei, so dass man – will man auf Nummer sicher gehen – nicht um eine Automatisierung von Word herumkommt.

Lucene erwartet einen String oder einen Reader, der nur noch den reinen Text beinhaltet. Die Applikation, die Lucene nutzt, muss sich also komplett selbst um diesen Schritt kümmern.

4.2 Textanalyse: Vom Text zum Wortkürzel

Bevor ein Dokument in den Index eingetragen werden kann, müssen erst die einzelnen Wörter des Textes extrahiert werden. Bei Lucene übernimmt das ein `Analyzer`. Ein `Analyzer` besteht wiederum aus einem `Tokenizer` und einem oder mehreren Filtern.

Der `Tokenizer` extrahiert die eigentlichen Wörter aus dem Text. Dazu sieht er alle Zeichen, die nicht von Whitespace unterbrochen sind, als ein Wort an. Wenn ein anderes Verhalten gewünscht wird, dann kann – wie überall bei Lucene – der `Tokenizer` durch eine eigene Implementierung ersetzt werden.

Die so extrahierten Wörter, auch „Tokens“ genannt, werden nun je nach `Analyzer` noch gefiltert und bearbeitet.

Lucene stellt bereits drei `Analyzer` zur Verfügung:

- Der `SimpleAnalyzer` wandelt alle Wörter in Kleinbuchstaben.
- Der `StopAnalyzer` filtert zusätzlich alle Wörter aus, die in der Stopwortliste stehen. (Siehe Kapitel 3.4, Seite 15).
- Der `StandardAnalyzer` entfernt nach Kleinschreibung und Stopwortfilter noch Apostrophe und Punkte aus den Wörtern, so wird aus „U.S.A.“ z.B. „USA“.

Nachdem der `Analyzer` den Text in einzelne Wörter zerlegt hat, kommt der `Stemmer` zum Einsatz. Der `Stemmer` hat die Aufgabe, die Wörter in ihre Stammform zurückzuführen. So wird das Wort „Katze“ beispielsweise in „katz“ gewandelt. Das führt dazu, dass eine Suche nach „Katze“ auch Treffer für „Katzen“ oder „Kätzchen“ liefert. Als angenehmer Nebeneffekt kommt dazu, dass so die Anzahl unterschiedlicher Wörter sinkt, was die Suche beschleunigt.

Lucene stellt zwei `Stemmer` von Haus aus zur Verfügung: Den `StandardStemmer` für die Englische Sprache und den `GermanStemmer` für Deutsch. Für andere Sprachen muss man selbst einen Stemmer schreiben, allerdings ist sehr wahrscheinlich, dass künftige Lucene-Versionen weitere `Stemmer` anbieten werden.

4.3 Documents – Wie man die Textdaten an Lucene übergibt

Um Lucene zu sagen, was es wie im Index speichern soll, werden „Document“-Objekte genutzt. Ein Document besteht aus einer Menge von Feldern. Ein Feld besteht zunächst einmal aus einem Namen und einem Wert.

In den meisten Fällen würde man zumindest folgende Felder definieren:

- Der Pfad zum ursprünglichen Dokument, so dass man dem Nutzer nach der Suche das Originaldokument zeigen kann.
- Der Inhalt des Dokuments, worauf sich schließlich die Suche bezieht.

Außerdem wären noch folgende Felder denkbar:

- Der Titel des Dokuments.
- Die Größe des Dokuments.
- Das Datum der letzten Änderung des Dokuments. So kann man bei einer Aktualisierung des Suchindexes feststellen, ob das betreffende Dokument neu indiziert werden muss.

Neben dem Namen und dem Wert hat jedes Feld noch drei Flags: `store`, `index` und `token`.

Der Flag `store`

Normalerweise kann man einen indizierten Text nicht mehr aus dem Index rekonstruieren, da der Originaltext bei der Indexierung in einzelne Wörter zerlegt wird, die nicht mehr im Zusammenhang stehen, sobald sie im Index sind.

Wurde jedoch der `store`-Flag gesetzt, dann speichert Lucene auch den Originaltext im Index, so dass er später wieder ausgelesen werden kann. Da dadurch jedoch auch der Index entsprechend größer wird, sollten nur relativ kleine Felder mit dem `store`-Flag versehen werden.

Der Flag `index`

Dieser Flag gibt an, ob ein Feld überhaupt indiziert werden soll. In manchen Feldern möchte man nur Zusatzinformationen ablegen, auf die man bei der Darstellung des Suchergebnisses zurückgreifen will, nach denen jedoch nicht

gesucht wird. Beispielsweise kann so das Datum der letzten Änderung des Dokuments abgelegt werden.

Der Flag `token`

Mit diesem Flag kann angegeben werden, ob das Feld vor der Indexierung in einzelne Wörter (Tokens) zerlegt werden soll oder ob das Feld als Ganzes als Wort gesehen werden soll. Will man z.B. eine Telefonnummer in einem Feld ablegen, dann will man nicht, dass diese Nummer vorher in einzelne Wörter zerlegt wird.

Für die Erzeugung eines Feldes sieht Lucene fünf verschiedene Fabrikmethoden vor, die die Flags jeweils verschieden setzen:

Methode	***store***	***index***	***token***
Field.Text(String, String)	x	x	x
Field.Text(String, Reader)		x	x
Field.Keyword(String, String)	x	x	
Field.UnIndexed(String, String)	x		
Field.UnStored(String, String)		x	x

Abbildung 8: Fabrikmethoden für die Erzeugung von Feldern

4.4 Codebeispiel

Das folgende Beispiel zeigt, wie in relativ wenigen Zeilen Code ein voll funktionsfähiger Crawler geschrieben werden kann, der einen Suchindex für ein oder mehrere Verzeichnisse erstellen kann. Der Crawler durchsucht die angegebenen Verzeichnisse samt Unterverzeichnisse und fügt jede gefundene Datei zum Index hinzu. Allerdings werden alle Dateien als reine Textdateien behandelt. Dieses Beispiel eignet sich daher nur für Verzeichnisse, die ASCII-Dateien beinhalten.

Für die Knowledge Base müsste der Crawler noch so erweitert werden, dass er verschiedene Dateiformate unterscheiden und daraus den eigentlichen Text

extrahieren kann. Siehe Kapitel 4.1, Seite 17. Die Indizierung braucht dabei etwas länger als eine primitive Volltextsuche. Da die Indizierung im Hintergrund abläuft und kein Benutzer darauf warten muss, ist die Dauer der Indizierung praktisch unerheblich.

Der Aufruf lautet wie folgt:

`java SimpleCrawler` Index Verzeichnis1 Verzeichnis2 ...

Beispiel:

```
java SimpleCrawler c:\temp\testindex "c:\Eigene Dateien"
```

Das Index-Verzeichnis muss dabei bereits existieren.

```java
public class SimpleCrawler {

  public static void main(String[] args) throws Exception {
    // Neuen Index erzeugen
    String indexPath = args[0];
    IndexWriter writer
      = new IndexWriter(indexPath, new GermanAnalyzer(), true);

    // Gegebene Verzeichnisse durchsuchen
    for (int i = 1; i < args.length; i++) {
      System.out.println("Indexing directory " + args[i]);
      File dir = new File(args[i]);
      addToIndex(dir, writer);
    };

    // Index optimieren und schließen
    writer.optimize();
    writer.close();
  }

  private static void addToIndex(File file, IndexWriter writer) {
    if (file.isDirectory()) {
      // Die gegebene Datei ist ein Verzeichnis
      // -> Alle Dateien und Unterverzeichnisse zum Index hinzufügen
      File[] childArr = file.listFiles();
      for (int i = 0; i < childArr.length; i++) {
        addToIndex(childArr[i], writer);
      }
    } else {
      // Die gegebene Datei ist kein Verzeichnis
      // -> Datei auslesen und zum Index hinzufügen
      try {
        InputStream is = new FileInputStream(file);

        // Wir erzeugen ein Document mit zwei Feldern:
        // Eines mit dem Pfad der Datei und eines mit seinem Inhalt
        Document doc = new Document();
        String fileName = file.getAbsolutePath();
        doc.add(Field.UnIndexed("path", fileName));
        doc.add(Field.Text("body", new InputStreamReader(is)));

        // Document zu Index hinzufügen
        writer.addDocument(doc);
        is.close();
      }
      catch (IOException exc) {
        System.out.println("Indexing " + file + " failed");
        exc.printStackTrace();
      }
    }
  }

}
```

Zur Methode `main`:

- Zunächst wird ein neuer `IndexWriter` erzeugt, der einen `GermanAnalyzer` verwendet. Der `IndexWriter` kapselt die komplette Erstellung des Index, wie sie in Kapitel 3, ab Seite 9 beschrieben wurde.
- Danach werden alle Verzeichnisse, die beim Aufruf angegeben wurden zum Index hinzugefügt. Das eigentliche Hinzufügen wird dabei an die Methode `addToIndex` delegiert.
- Schließlich wird der Index noch optimiert und geschlossen.

Zur Methode `addToIndex`:

- Zunächst wird geprüft, ob es sich bei der gegebenen Datei um ein Verzeichnis handelt.
- Bei einem Verzeichnis wird addToIndex für alle Dateien und Unterverzeichnisse rekursiv aufgerufen.
- Handelt es sich um kein Verzeichnis, dann wird ein `Document`-Objekt mit einem Feld für den Pfad und einem für den Inhalt der Datei erzeugt. Dieses `Document` wird dem Index hinzugefügt.

© 2004 Tilman Schneider

5 Die Suche

Wir wissen nun, wie man aus einer Menge von Dokumenten einen Suchindex erstellt. Die Erstellung des Suchindex hatte einen einzigen Zweck: Der Suche sollte möglichst viel Arbeit abgenommen werden, so dass Wartezeit von der Suchanfrage zur Indexerstellung verlagert wird, welche asynchron im Hintergrund abläuft, ohne dass ein Benutzer darauf warten müsste.

Das nun folgende Kapitel wird sich der Frage widmen, was bei der Suche genau passiert und welche Schritte man trotz allem noch bei der Suche vornehmen muss.

5.1 Die Anfragesyntax von Lucene

Dank des Suchindex ist man in der Lage sehr schnell jene Dokumente zu finden, die ein bestimmtes Wort enthalten. Prinzipiell könnte man also den Benutzer auffordern, ein Wort zu nennen, um ihm daraufhin die Fundstellen für dieses Wort zu präsentieren. Jeder, der schon einmal die eine oder andere Internetrecherche durchgeführt hat, weiß, dass es damit nicht getan ist: Oft bekommt man zu viel des Guten und möchte seine Suchanfrage etwas genauer beschreiben. Man möchte eine Reihe von Worten angeben können, manche Worte ausschließen oder auch Worte unscharf angeben, so dass auch ähnliche Worte gefunden werden.

Aus diesem Grund bietet Lucene eine recht umfangreiche Anfragesyntax, die in diesem Abschnitt beschrieben werden soll. Lucene bietet zwar – wie immer - die Möglichkeit, auch eine eigene Syntax zu verwenden, die Standardsyntax sollte jedoch im Normalfall mehr als ausreichen.

An dieser Stelle wird nur auf die wichtigsten Elemente dieser Syntax eingegangen. Weitere Informationen finden Sie in der offiziellen Beschreibung der Anfragesyntax von Lucene[6]

6 Siehe: http://jakarta.apache.org/lucene/docs/queryparsersyntax.html

Terme

Eine Suchanfrage besteht aus Termen und Operatoren. Es gibt zwei Arten von Termen: Einfache Terme und Phrasen.

Ein einfacher Term ist ein einzelnes Wort, wie `Test` oder `Hallo`.

Eine Phrase ist eine Gruppe von Worten, die in Hochkommas eingeschlossen sind, beispielsweise `"Hallo Otto"`. Sie treffen auf Worte zu, die in genau dieser Reihenfolge vorkommen.

Operatoren

Terme werden mit Hilfe von Operatoren verbunden. Die wichtigsten werden nun vorgestellt.

Operator	**OR**
Beschreibung	`OR` verbindet zwei Terme und findet Dokumente, die wenigstens einen der Terme beinhalten.
Hinweis	`OR` ist der Standardoperator, d.h. er wird genutzt, wenn kein Operator angegeben wird.

Beispiel	Um Dokumente zu suchen, die `Otto Maier` oder `Seminar` beinhalten:
	`"Otto Maier" OR Seminar`
	oder einfach:
	`"Otto Maier" Seminar`

Operator	**AND**
Beschreibung	`AND` trifft auf Dokumente zu, die beide Terme irgendwo im Text beinhalten.
Beispiel	Um Dokumente zu suchen, die sowohl `Otto Maier` als auch `Seminar` beinhalten: `"Otto Maier" AND Seminar`

Operator	**+**
Beschreibung	`+` gibt an, dass der folgende Term erforderlich ist.
Beispiel	Liefert Ergebnisse, die `Lucene` enthalten müssen und `Seminar` enthalten dürfen: `Seminar +Lucene`

Operator	**–**
Beschreibung	– schließt Dokumente aus, die den folgenden Term beinhalten.
Beispiel	Sucht Dokumente, die `Seminar` enthalten, `Lucene` jedoch nicht: `Seminar –Lucene`

Wildcards

Man kann Wildcards (auch bekannt als Jokerzeichen) für einzelne oder für viele Buchstaben setzen.

Um ein einzelnes Zeichen offen zu lassen, verwendet man das Fragezeichen (`?`). Um viele Zeichen offen zu lassen, nutzt man den Stern (`*`).

Um `Text` oder `Test` zu suchen, gibt man an:

```
Te?t
```

Um `Test`, `Testergebnis` oder `Testament` zu suchen, schreibt man:

```
Test*
```

Ein Wildcard kann auch in der Mitte eines Terms stehen:

```
Te*t
```

Hinweis: Lucene unterstützt keine Terme, die mit einem `?` oder einem `*` beginnen!

Unscharfe Suche

Um Worte zu finden, die ähnlich buchstabiert werden, kann man an einen Term eine Tilde (`~`) anhängen.

Der folgende Ausdruck findet auch `Mayer`, `Meyer` oder `Maier`:

```
Meier~
```

5.2 Erzeugen einer Trefferliste

Die im vorigen Abschnitt erklärte Syntax wird bei einer Suchanfrage zunächst vom `QueryParser` analysiert. Das Ergebnis ist ein `Query`-Objekt, das den Syntaxbaum der Suchanfrage zugänglich macht.

Jedes atomare Wort wird zunächst wie auch bei der Erstellung des Suchindex vom `Analyzer` bearbeitet. So wird aus `Kätzchen` ein `katz`, was Grundlage dafür ist, dass nun auch Dokumente gefunden werden, die `Katze` enthalten. Damit das reibungslos funktioniert, ist zu beachten, dass sowohl für die Erstellung des Index, als auch für die Suche der gleiche `Analyzer` verwendet wird.

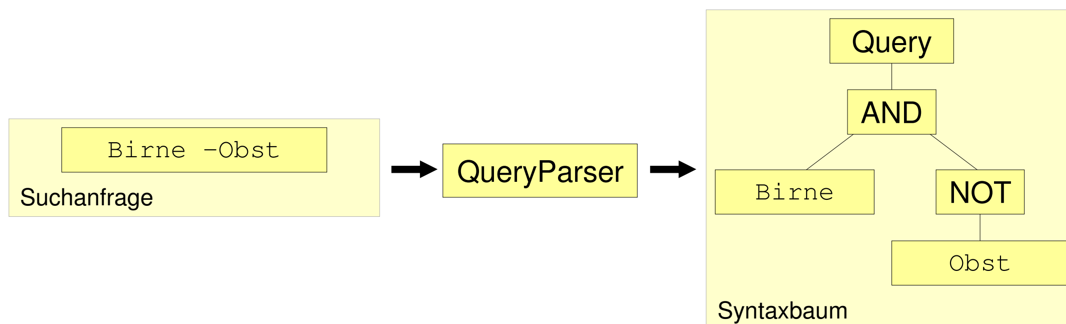

Abbildung 9 Syntaxbaum einer Anfrage

Für jedes der so extrahierten Wortkürzel wird nun eine Ergebnisliste aus dem Index ermittelt.

Diese Ergebnislisten werden nun entsprechend der Regeln des Syntaxbaums miteinander in Verbindung gebracht. Bei einer UND-Operation wird die Schnittmenge gebildet, bei ODER werden beide Listen vereinigt, bei NOT werden aus der einen Liste alle Einträge der anderen entfernt.

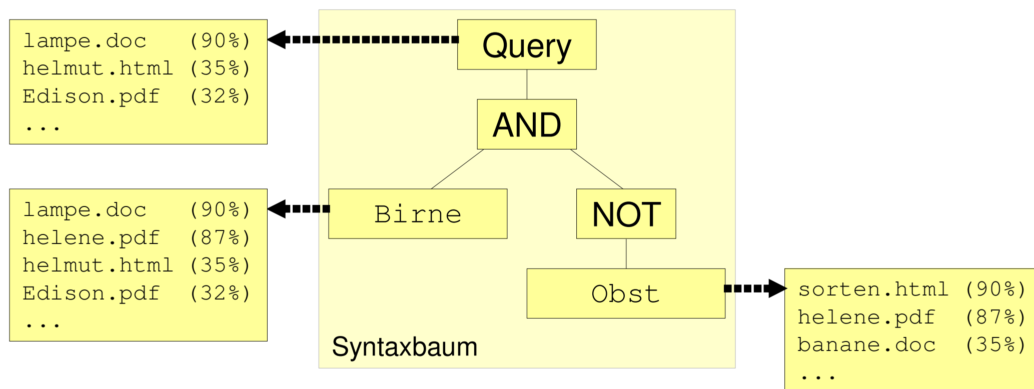

Abbildung 10 Eigentliche Suche mit Hilfe des Syntaxbaums

Die Relevanz eines Dokuments berechnet sich dabei aus dem Durchschnitt der Relevanzen, die es in den ursprünglichen Listen hatte.

Bei einer Wildcardsuche oder einer Unscharfen Suche werden vorher alle passenden Worte aus dem Index ermittelt, um dann die Vereinigung der Ergebnislisten aller Worte zu ermitteln. Eine solche Suche entspricht daher einer Verkettung aller passenden Worte mit dem ODER-Operator.

5.3 Codebeispiel

Das nun folgende Beispiel zeigt, wie ein zuvor erstellter Suchindex in wenigen Zeilen Code durchsucht werden kann.

Der Aufruf lautet wie folgt:

`java SimpleSearcher` Index Query

Beispiel:

```
java SimpleCrawler c:\temp\testindex "Seminar Lucene"
```

Es muss dabei bereits ein Index im Index-Verzeichnis existieren.

© 2004 Tilman Schneider

```
public class SimpleSearcher {

  public static void main(String[] args) throws Exception {
    String indexPath = args[0];
    String queryString = args[1];

    Query query = QueryParser.parse(queryString, "body",
                                    new GermanAnalyzer());

    Searcher searcher = new IndexSearcher(indexPath);
    Hits hits = searcher.search(query);

    for (int i = 0; i < hits.length(); i++) {
      System.out.println(hits.doc(i).get("path")
                         + " (score: " + hits.score(i) + ")");
    };
  }

}
```

Zunächst wird der `QueryParser` genutzt, um ein `Query`-Objekt mit dem Syntaxbaum zu erzeugen.

Dann wird ein `Searcher`-Objekt erzeugt, wobei derselbe `Analyzer` genutzt wird wie auch bei der Erstellung des Index: Der `GermanAnalyzer`.

Das `Query`-Objekt wird dem `Searcher` übergeben, der daraufhin die eigentliche Suche durchführt. Das Ergebnis der Suche wird in der Variablen `hits` gespeichert.

Schließlich wird die Ergebnisliste noch ausgegeben.

6 Fazit

Lucene schafft es, einerseits eine einfache API bereitzustellen, mit der Lucene-Einsteiger sofort zurechtkommen, ohne auf allen Ebenen Möglichkeiten vermissen zu lassen, das Standardverhalten zu beeinflussen: Vom `QueryParser` über den `Analyzer` bis zur Schicht, in der ein Suchindex schließlich abgelegt wird, kann man alle Elemente durch eigene Implementierungen austauschen.

Lucene benötigt extrem wenig Systemressourcen. Das betrifft sowohl Plattenplatz, als auch Prozessor und Arbeitsspeicher. Dabei bietet Lucene einen Suchindex, der an Performance und Funktionalität so manches kommerzielle Produkt in den Schatten stellt.

Lucene stellt allerdings nur den Kern einer Suche zur Verfügung, also die Erstellung und die Nutzung eines Suchindex. Die Darstellung der Suchergebnisse und die Aufbereitung der zu indizierenden Inhalte muss man selbst übernehmen. Gerade der letzte Punkt ist angesichts vieler verschiedener Dokumentenformate nicht gerade trivial.

Für die dm-drogeriemarkt-Kette habe ich eine ähnliche Anwendung entwickelt, wie sie im Beipiel der Knowledge Base beschrieben wurde. Die Anwendung lässt sich für das Durchsuchen von HTTP-Servern, wie auch von Dateisystemen nutzen. Es wird mittlerweile von der dm-drogeriemarkt-Kette sowohl für Intranet-Seiten, als auch für den Internetauftritt[7] produktiv eingesetzt.

Die dm-drogeriemarkt-Kette hat sich dazu bereit erklärt, das Projekt als Open Source bereitzustellen. Es wird in Kürze unter dem Namen regain[8] bei Sourceforge[9] zugänglich sein.

Da Lucene alle Problemstellungen, die sowohl die Erstellung als auch die Nutzung des Index betreffen, hervorragend kapselt, lag das Hauptproblem bei der Unterstützung der einzelnen Dateiformate.

7 Siehe: http://www.dm-drogeriemarkt.de
8 Siehe: http://regain.sf.net
9 Größter Service-Provider für Open-Source-Projekte, siehe http://sourceforge.net/index.php

Offene, textbasierte Formate, wie HTML oder RTF, sind sehr einfach auszulesen. Proprietäre Binärformate, wie beispielsweise Word, bereiteten die meisten Probleme. Manchmal hat man Glück und findet ein Projekt, das dieses Dateiformat bereits fehlerfrei lesen kann. Im Falle von Word habe ich letztendlich über die COM-Schnittstelle Word selbst genutzt, um die Dokumente lesen zu können. Es gibt zwar auch für Word Projekte, die das Format lesen können, allerdings funktionierten diese nicht für jedes Beispiel fehlerfrei.

Man kann daher nur schwer abschätzen, wie lange es dauert, ein bestimmtes Format lesen zu können. Es hängt zu sehr vom Format selbst ab und davon, ob es bereits fertige und auch funktionierende Lösungen gibt. Im Anhang A habe ich einige Links aufgelistet, die sich mit verschiedenen Dateiformaten befassen.

Anhang A Quellen

- Jakarta Lucene Dokumentation:
  http://jakarta.apache.org/lucene/docs/index.html (Abruf am 28.02.04)
- Offizielles Lucene FAQ:
  http://lucene.sourceforge.net/cgi?bin/faq/faqmanager.cgi
- Java Guru: Lucene FAQ Home Page: http://www.jguru.com/faq/Lucene
- Anfragesyntax von Lucene:
  http://jakarta.apache.org/lucene/docs/queryparsersyntax.html
  (Abruf am 21.05.04)
- Java Word-Artikel von Brian Goetz über Lucene:
  http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene.html
  (Abruf am 28.02.04)
- Lucene Tutorial: http://darksleep.com/lucene (Abruf am 28.02.04)
- Suchmaschinen-Vergleich von Gregor Fischer:
  http://www2.informatik.uni?wuerzburg.de/staff/holger/lehre/OSWS0001/fischer/os0001_fischer.pdf (Abruf am 28.02.04)
- Vortrag über Lucene auf der jax2003: www.jax2003.de, jax2003-CD

Dateiformat-Umwandlung:

- Java API. Liest sehr gut MS Excel, weniger gut MS Word:
  http://jakarta.apache.org/poi und
  http://www.mail?archive.com/poi?user@jakarta.apache.org/
- Java-COM-Bridge (Jacob). Eignet sich u.a. zum direkten Zugriff auf alle MS Office Produkte (sehr gut): http://users.rcn.com/danadler/jacob/.
- Jacob-Mailingliste. Enthält u.a. Bugfixes, erfordert jedoch Yahoo-Account: http://groups.yahoo.com/group/jacob-project/
- Code-Generator für Jacob: http://www.bigatti.it/projects/jacobgen/. Wurde verbessert vom STZ-IDA Karlsruhe (http://www.stz-ida.de), die Verbesserung ist jedoch (noch) nicht veröffentlicht.
- Java API zum Lesen von PDF (sehr gut): http://pdfbox.org
- Java API zum Lesen von MS Word (nicht getestet): http://textmining.org
- MS Word -> XML (nicht getestet):
  http://www.xml.com/pub/a/2003/12/31/qa.html