Indizieren und suchen mit Jakarta-Lucene

Suchapparat

Mit eigenen Programmen fremde Dateien durchsuchen ist schwierig, die Vielfalt der Formate trübt die Sicht. Die Suche soll schnell und fehlertolerant sein, das Ergebnis ansprechend aussehen. Mit dem Java-API Lucene sind diese Bedingungen zu erfüllen, es ist ein Baukasten für die eigene Search Engine. Bernhard Bablok

Wie bei Google soll es sein: Eine einfache Eingabe bringt das gewünschte Ergebnis. Auch wenn nicht alle Fundstellen befriedigen, hat die Suchmaschine ihre Schuldigkeit getan, denn die gefundenen Seiten enthalten zumindest das gesuchte Wort. Dass schon dies nicht ganz einfach ist, erfährt, wer seine eigene Search Engine basteln will, selbst wenn sie nur Dateien auf der eigenen Festplatte durchsuchen soll. Die Java-Bibliothek Lucene [1] hilft beim Bau und reduziert die Programmierarbeit. Dieser Coffee-Shop führt in Lucene ein und stellt das Projekt Luala (Lucene Application Layer) vor, das den Einsatz der Search Engine demonstriert.

Bei der Suche lassen sich zwei Strategien unterscheiden: Der Anwender sucht innerhalb vorgegebener Kategorien oder er fahndet im gesamten Text nach einem frei gewählten Suchmuster. Beide Verfahren haben Vor- und Nachteile. Bei geeigneten Kategorien ist die Suche schnell und zielgerichtet, der Erfolg aber davon abhängig, ob der Anbieter der Informationen in denselben Kategorien denkt wie der Sucher. Volltextsuche ist von einem Kategorienschema unabhängig, aber deutlich aufwändiger.

Kategorien oder Volltext?

Eine Volltextsuche wird beschleunigt, wenn sie Indizes benutzt, also ein vorher angelegtes Verzeichnis im Text vorkommender Wörter. Die für das Indizieren zuständige Softwarekomponente heißt Indexer. Zum einen legt der Indexer den Index vor der Suche an, die dafür aufgewendete Zeit fällt für den Anwender nicht ins Gewicht. Zum anderen geht die Suche selbst schneller, da der Index schon vorsortiert ist. Der Betreiber der Suchmaschine muss allerdings dafür sorgen, dass der Index immer auf dem aktuellen Stand ist, also jedes Mal neu indizieren, wenn sich die zu durchsuchenden Daten ändern.

Die Qualität einer Volltext-Suchmaschine hängt in erster Linie von ihrem Indizierungsverfahren ab. Wenig nützlich wäre zum Beispiel ein Indexer, der sich bei seiner Arbeit vor allem auf Artikel und andere, häufig vorkommende kleine Wörter konzentriert. Darüber hinaus sind die Suchstrategien, die eine Search Engine bietet, ein Kriterium (boolesche Verknüpfungen, Fuzzy-Search …).

Letztlich richtet sich die Auswahl einer bestimmten Lösung auch danach, wie einfach sich eine Suchmaschine in eigene Anwendungen einbauen lässt. Das Apache-Jakarta-Projekt Lucene erfüllt die meisten dieser Anforderungen, ist allerdings keine fertige Anwendung, sondern eine Bibliothek.

Auspacken und loslegen

Auf der Lucene-Homepage [1] finden sich Binär- und Quellpakete im Zip- oder Tar.gz-Format. Letztere sind normalerweise kleiner und für Linux das Format der Wahl. Nach dem Auspacken stehen die Lucene-Bibliothek »lucene-1.3-final.jar«, die Dokumentation sowie einige Beispiele (auch als Source) zur Verfügung. Wer Lucene selbst kompilieren will, braucht neben einem JDK (mindestens Version 1.2) noch das Build-Tool Ant (ab 1.5). Die Details zum Build-Prozess beschreibt die Datei »BUILD.txt«. Die Übersetzung sollte keine großen Schwierigkeiten bereiten, denn normalerweise genügt ein Aufruf von »ant« im Lucene-Verzeichnis. Lucene verwendet

zum Teil Parser-generierten Code, wer auch diesen Teil neu erstellen will, benötigt zusätzlich Java CC von [2].

Eine weiter gehende Installation ist nicht notwendig. Bei der Entwicklung sowie zur Ausführung von Programmen, die Lucene nutzen, muss der »CLASSPATH« die Jar-Datei enthalten. Die Dokumentation bringt ein kleines Tutorial mit, das anhand des Beispielcodes in grundlegende Konzepte einführt. Außerdem enthält sie Benchmarkergebnisse sowie Verweise auf zusätzliche Programme, die im Lucene-Umfeld nützlich sind.

Strukturen fürs Suchen

Die Klasse »Document« ist der logische Container für die Indizierung durch Lucene. Jedes »Document« enthält eine beliebige Anzahl von Feldern. Ein »Field« besteht aus einem Namen und einem Wert. Die Namen sind Strings, während die Werte entweder Strings oder Objekte der Typen »java.util.Date« oder »java.io.Reader« sind. Bei Letzterem speichert Lucene nicht das Reader-Objekt, sondern den Text, den der Reader liefert. Sinn dieser Struktur ist es, sowohl Texte als auch Metadaten zu verwalten. Typischerweise befindet sich der Index in einem Feld, während die Metadaten wie Dateiname, Datum und Autor in weiteren Feldern gespeichert sind.

Lucene unterscheidet beim Aufbau des Index drei Operationen: die Zerlegung in Teile (Tokenizing), das Speichern (Storing) und die Indizierung (Indexing) – nicht jede ihrer Kombination ist sinnvoll. So ist die Zerlegung eines Textes ohne anschließende Indizierung sinnlos, genauso wie es eine Indizierung ohne Speicherung oder Zerlegung ist.

Eine sinnvolle Anwendung ist es dagegen, die Rohdaten zu zerlegen, zu indizieren und als Ganzes zu speichern. Damit ist es möglich, einen Text mit Lucene zu durchsuchen und ihn auch gleich ausgeben zu lassen. Das Original ist dann nicht mehr erforderlich, aber die Lucene-Datenbasis wird entsprechend groß. Die zugehörige statische Methode ist »Field.Text(String name, String value)«. Der Suchmaschinen-Programmierer muss den zu durchsuchenden Text also in einem String ablegen, bevor er die Methode aufruft.

Die zweite Möglichkeit ist die klassische Variante: Der Text wird zerlegt und indiziert, aber nicht gespeichert. Lucene bietet hierfür zwei statische Methoden: »Field.Text(String name, Reader value)« und »Field.Unstored(String name, String value)«. Erstere ist recht praktisch, da man einfach einen »FileReader« erzeugt und an die Methode übergibt.

Da diese beiden Methoden den Text nicht speichern, sind Metadaten erforderlich, um bei der Suche das Original wiederzufinden. Das kann ein Pfad oder eine URL zum Dokument sein. Für diese Metadaten, die Lucene in seiner Datenbasis ablegt, gibt es zwei Varianten, nämlich sie mit oder ohne Indizierung zu speichern. Die erste Variante erlaubt es, die Metadaten in die Suche einzubeziehen. In Fällen, in denen die Metadaten für den Endanwender keine Bedeutung haben (etwa weil es sich um eine Dokumenten-ID in einer Datenbank handelt), kann man es sich sparen, sie selbst zu indizieren.

Die zugehörigen Methoden haben die Signatur »Field.Keyword(String name, String value)« und »Field.Unindexed(String name, String value)«. Die erste Methode gibt es auch in einer überladenen Version, die ein »Date«-Objekt als zweiten Parameter besitzt.

Gewöhnungsbedürftiges API

Diese Beispiele zeigen einige Schwächen des Lucene-API. An seiner Funktionalität gibt es zwar nichts auszusetzen, aber die Namensgebung und die Struktur der Klassen widersprechen den üblichen Java-Konventionen, sodass eine gewisse Eingewöhnung nötig ist. Eine saubere Implementation hätte eine Basisklasse »Field« mit einer Reihe von Unterklassen für verschiedene, konkrete Typen definiert. Entsprechende Konstruktoren könnten dann die Objekte erzeugen. Die jetzige Lösung verwendet dagegen statische Methoden, deren Namen nicht selbsterklärend sind und mit Großbuchstaben anfangen. Sie sehen aus wie Konstruktoren von internen Klassen, sind es aber nicht.

Ein weiterer Kritikpunkt ist, dass es keinen Satz standardisierter Felder gibt. Ein deutscher Programmierer mag ein Feld „Inhalt“ nennen, während ein englischer eher „Content“ oder „Contents“ bevorzugt. Damit ist die Interoperabilität zwischen verschiedenen Lucene-Anwendungen nicht automatisch gegeben: Jede benötigt ihren eigenen Indexer, denn wenn eine Suche die falschen Felder verwendet, geht sie einfach ins Leere. Abgesehen von diesen Fallstricken bietet Lucene aber wenig Anlass für Kritik.

Der Weg in den Index

Die Klasse »IndexWriter« erzeugt Indizes und ändert sie auch bei Bedarf. Im Normalfall besteht ein Index einfach aus einer Reihe von Dateien in einem Verzeichnis. Dies übergibt der Programmie-

Luala-Funktionen

Der Lucene Application Layer (Luala) bietet höhere Funktionen als Lucene selbst:

IndexEngine: Eine konfigurierbare Klasse, die für die Indizierung zuständig ist. Das ist ein vergleichsweise dünner Wrapper um einen »IndexWriter«. Die »IndexEngine« bindet als Plugins Objekte vom Typ »DocumentFactory« für Dateitypen wie PDF oder ».gz«-komprimierte Textdateien ein.

SearchEngine: Ebenfalls eine konfigurierbare Klasse für die Suche. Listing 2 zeigt die zentrale Methode »search()«. Die Methode liefert ein Array mit Elementen vom Typ »Document«.

ResultRenderer: Dieses Interface definiert die Methode »render()« für die Aufbereitung der Suchergebnisse. Mehrere Klassen implementieren das Interface, zum Beispiel »HtmlResultRenderer«, »TextResultRenderer« oder »StringArrayResultRenderer«. Weitere Renderer wären denkbar, etwa einer, der die Ergebnisse in XML ausgibt.

GUI-Klassen: In diesem Bereich sind zwei Interfaces und mehrere Klassen implementiert. Einmal das Interface »SearchPanel«, zum anderen »ResultView«. Letzteres implementieren die Klassen »HtmlResultView« und »ListResultView«.

Event-Klassen: Die GUI-Komponenten kommen mit zwei Event-Typen aus: »SearchEvent« und »ShowDocumentEvent«. Mittels entsprechender Listener lassen sich Anwendungen nach dem MVC-Paradigma (Model View Controller) bauen. Ein »SearchPanel« erzeugt ein »SearchEvent«, ein »ResultView« bei der Auswahl eines Treffers ein »ShowDocument Event«. Wie die Events verarbeitet werden, ist den Event-Erzeugern egal, die Listener sind das einzige Bindeglied.

rer dem »IndexWriter«. Zusätzliche Parameter sind der so genannte Analyzer und ein Flag, das angibt, ob der Index neu erzeugt oder ein bestehender Index verändert wird.

Die wichtigste Methode von »IndexWriter« ist »addDocument(Document doc)«. Üblicherweise indiziert eine Suchmaschine eine ganze Sammlung von Dokumenten. Für jede dieser Dateien ist ein geeignetes »Document« mit den gewünschten Feldern zu erzeugen und an den »IndexWriter« zu übergeben. Am Ende ruft man die Methoden »optimize()« und »close()« auf, reorganisiert den Index und gibt die Ressourcen von »IndexWriter« frei.

Alle Dateien eines Verzeichnisbaums finden ist eine einfache Übung. Die Klasse »java.io.File« stellt dafür eine Reihe von Methoden zur Verfügung, beispielsweise »listFiles()«. Aus den Dateien ein Lucene-Document erstellen ist dagegen unter Umständen komplexer. Für ganz einfache Textdateien zeigt Listing 1 die notwendigen Zeilen. Das »Document« enthält nur drei Felder, neben dem für den Inhalt ein Feld für den Pfad der Datei sowie ein Typ-Feld.

Listing 1: Erstellen eines Lucene-Document

```
093 public Document createDocument(File f) throws
    IOException {
094
095     Document doc = new Document();
096     doc.add(Field.Text("path", f.getPath()));
097     doc.add(Field.UnIndexed("type", "simple-doc"));
098     doc.add(Field.Text("content", new
                FileReader(f)));
099     return doc;
100 }
```

Listing 2: Suchen im Index

```
171 public Document[] search(String queryString) throws
            IOException,org.apache.lucene.queryParser.
172         ParseException {
174     String dir = iIndexDir;
175     Searcher searcher = new IndexSearcher(dir);
176     Query query = QueryParser.parse
            (queryString,iDefaultField,iAnalyzer);
177     Hits hits = searcher.search(query);
178
179     Document[] results = new
           Document[hits.length()];
180     for (int i = 0; i<results.length; ++i)
181       results[i] = hits.doc(i);
182     searcher.close();
183     return results;
184   }
```

Komplexere Dateien muss man vorverarbeiten, Maildateien (MBox) etwa zerlegen oder PDF-Dateien in Text umwandeln. Für diese Schritte gibt es im Lucene-API aber keine Unterstützung. Der Suchmaschinen-Programmierer muss eigene Klassen schreiben, die solche Dateien verarbeiten, oder fertige Bibliotheken für seine Anwendung anpassen.

Beim Analytiker

Der bereits erwähnte Analyzer verdient eingehendere Betrachtung. Er analysiert den zu durchsuchenden Text und legt fest, was überhaupt als Suchwort in Frage kommt. Bei der Zerlegung und nachfolgenden Indizierung eines Textes geht es ja darum, wichtige Wörter in den Index aufzunehmen und den Rest, also Füllwörter (Stop Words), zu ignorieren. Zu Letzteren zählen Wörter wie „und“, „nur“, Artikel, Pronomina und so weiter. Es ist klar, dass ein Analyzer von der Sprache des Dokuments abhängt. Zurzeit unterstützt Lucene Englisch, Deutsch und Russisch. Außerdem enthält die Dokumentation einen Link auf einen chinesischen Analyzer.

Auch hinsichtlich ihrer Strategie beim Verarbeiten von Wortstämmen unterscheiden sich Analyzer. So mag der eine die Singular- und Pluralform eines Wortes als zwei Suchbegriffe ablegen, während der intelligentere Analyzer daraus nur ein Suchwort macht. Sucht der Endanwender nach der Grundform, gibt nur der zweite Analyzer die Fundstellen beider Formen zurück.

Flexibel finden

Wer den Aufwand nicht scheut, schreibt einen eigenen Analyzer, der auf den jeweiligen Einsatzzweck zugeschnitten ist. Der kann beispielsweise eigene Stoppwort-Listen lesen oder auch mehrsprachige Dokumente richtig verarbeiten – je nach den linguistischen Kenntnissen des Programmierers. Lucene selbst setzt hier jedenfalls keine Grenze.

Neben der Indizierung deckt das Lucene-API auch die Suche ab. Gesucht wird über einen »Searcher« und eine »Query« oder eine ihrer Unterklassen, etwa »FuzzyQuery«. Queries können sehr komplex sein, doch glücklicherweise steht dem Programmierer dafür die Klasse »QueryParser« zur Verfügung. Sie wurde von den Lucene-Entwicklern mittels Java CC [2] automatisch aus der Query-Grammatik erzeugt. Der »QueryParser« wandelt einen Query-String in ein Objekt der Klasse »Query« um. Listing 2 zeigt dessen Einsatz.

Die Query-Syntax ist in der Dokumentation gut beschrieben. Neben den üblichen Operatoren (»AND«, »OR«, »NOT« und Klammerung) unterstützt Lucene auch Wildcards »*«, »?« und Fuzzy-Suche. So findet »Maier~« auch die verschiedenen Schreibweisen Maier, Meier oder Mayer. Leider ist eine Wildcard am Anfang nicht möglich: Eine Suche der Art »*Factory«, um etwa in einem Java-Quellcodebaum nach Factory-Objekten zu suchen, funktioniert nicht. Weitere Suchmöglichkeiten sind:

- Distanzsuche (Proximity Search) mit der maximalen Entfernung der gesuchten Begriffe im Text: »"Programm Java"~5« (die Begriffe „Programm“ und „Java“ dürfen maximal fünf Wörter voneinander entfernt sein).
- Suche in einem vorgegebenen Feld: »title:"Gesucht - Gefunden"« (findet Dokumente, deren Titel-Feld dem Such-Term entspricht).
- Die Bereichssuche (Range Search): »name:[Huber TO Maier]« (der Inhalt des Name-Felds liegt zwischen Huber und Maier; eckige Klammern schließen die Grenzen ein, geschweifte schließen sie aus).

Die Gewichtung der Suchbegriffe erhöht der Boost-Operator »^«. So wird mit dem Suchterm »Java^4 Programm« der Name »Java« für die Suche viermal relevanter als »Programm«. Diese Gewichtungen lassen sich genauso auf Suchphrasen anwenden. Der Boost-Operator wird einfach an die Phrase angehängt: »"Java Programm"^4«.

Lucene Application Layer

Das Lucene-API ist sehr mächtig, bietet aber nur recht einfache Bausteine für die eigene Anwendung. Da sich die Anwendungsfälle auf höherer Ebene immer wieder ähneln, bietet es sich an, diese in

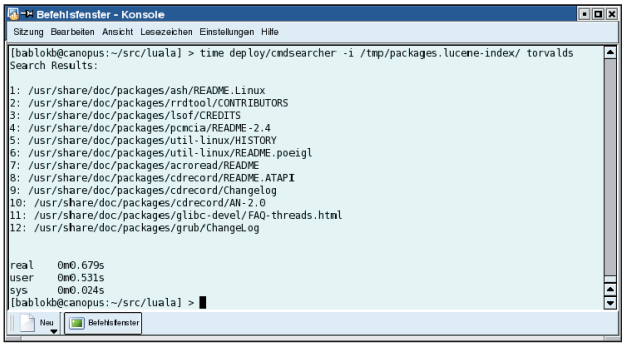

Abbildung 1: Die einfache Textsuche ist mit dem Lucene Application Layer (Luala) sehr schnell, wenn sie einen vorher erzeugten Index benutzt.

Abbildung 2: Der auf Lucene basierende Docsearcher durchsucht mit einer grafischen Oberfläche den Jahrgang 2003 des Linux-Magazins.

einem weiteren Layer zu kapseln. Die Idee zu diesem Projekt wurde aus einer Anfrage des Linux-Magazins geboren: Die Suchmaschine für die Jahres-CD sollte möglichst plattformunabhängig funktionieren. Das Ergebnis ist der Lucene Application Layer (Luala).

Der dafür geschriebene Code demonstriert, wie man mit dem Lucene-API eine eigene, höhere Schicht für eine Suchmaschine programmiert. Die Kernelemente des Application-Layers sind im Kasten „Luala-Funktionen“ näher beschrieben. Der gesamte Quellcode des Projekts sowie alle benötigten Bibliotheken finden sich auf Berlios [3].

Als Proof of Concept, aber auch als eigenständige Anwendung für einfache Bedürfnisse enthält Luala eine Reihe von Beispielanwendungen. So ist ein kommandozeilenorientierter Indexer für die Indexerstellung zuständig. Er liest entweder direkt ein Quellverzeichnis oder man übergibt ihm mit »find« und einer Pipe die zu indizierenden Dateien. Die Suche erfolgt entweder ebenfalls auf der Kommandozeile (siehe Abbildung 1) oder über eine einfache grafische Oberfläche. Bei der ersten Variante lässt sich über eine Option wählen, ob das Programm das Ergebnis als Text oder im HTML-Format ausgibt.

Ein Benchmark

Es bleibt die Frage, was der Aufwand des Indizierens bringt. Ein kleiner Benchmark soll dafür Anhaltspunkte liefern. Indiziert wurde das Verzeichnis »/usr/share/doc/packages«, insgesamt etwa 93 MByte. Das dauerte auf einem Pentium-M mit 1,5 GHz 2:20 Minuten (zweiter Lauf, viele Dateien befanden sich also zu dem Zeitpunkt schon im Cache). Der Index hatte danach eine Gesamtgröße von 19 MByte, was zum Teil auch daran liegt, dass keine speziellen Parser, etwa für HTML-Dateien verwendet wurden.

Die Suche nach „torvalds“ dauerte ungefähr 0,75 Sekunden (etwas schwankend, je nach Systemlast). Dagegen dauerte die Suche mit »rgrep« im Originalverzeichnisbaum 4,5 Sekunden (ebenfalls zweiter Lauf), war also um den Faktor sechs langsamer. Hinzu kommt, dass Grep-Kommandos zwar reguläre Ausdrücke verstehen, es aber schon etwas dauert, um damit komplexere Abfragen zu formulieren – mit der Query-Syntax von Lucene ist man schneller am Ziel.

Projekte im Lucene-Umfeld

Viele Projekte nutzen Lucene und entwickeln es weiter. In der Dokumentation gibt es zwei Abschnitte, einmal Contributions mit Verweis auf entsprechende Webseiten, zum anderen die Lucene-Sandbox mit weiteren Projekten. Der Unterschied ist, dass letztere Projekte im CVS von Lucene gehostet werden und eventuell mit der Zeit Eingang in die Lucene-Distribution finden.

Von den vielen Projekten sei auf eins besonders hingewiesen: Docsearcher von John Brown [4]. Regelmäßigen Lesern des Linux-Magazins ist es von der letzten Jahres-CD (siehe Abbildung 2) bekannt. Docsearcher ist eine gelungene grafische Oberfläche zu den vielfältigen Funktionen von Lucene und bietet Indizierung sowie differenzierte Suchmöglichkeiten. Wer keine Search Engine in eigene Software einbauen will, findet in Docsearcher ein eigenständiges Suchprogramm, das von Haus aus schon viele Dokumententypen kennt. Zum Ausschlachten für eigene Programme eignet sich Docsearcher allerdings nur bedingt. Es enthält zwar viele nützliche Funktionen, leider aber völlig unstrukturiert, es verwendet beispielsweise nicht einmal Packages.

finally{}

Wie so oft im Java-Umfeld lohnt sich Selbermachen nicht. Es ist einfacher, vorhandene Lösungen anzupassen und in eigene Programme einzubauen. Lucene zeigt, wie das Denken in Bibliotheken und APIs die Wiederverwendung erleichtert. Es stellt modulare Komponenten für die Programmierung eigener Search Engines zur Verfügung und lässt sich nach Belieben erweitern. *(ofr)* ■

Infos

[1] Lucene-Homepage: [http://jakarta.apache.org/lucene/]
[2] Java CC, der Java Compiler Compiler: [http://www.webgain.com/products/java_cc/]
[3] Luala-Homepage: [http://luala.berlios.de]
[4] Docsearcher von John Brown: [http://www.brownsite.net/docsearch.htm]

Der Autor

Bernhard Bablok arbeitet bei der Allianz Versicherungs AG im Bereich Data-Warehouse-Systeme. Wenn er nicht Musik hört, mit dem Radl oder zu Fuß unterwegs ist, beschäftigt er sich mit Themen rund um Objektorientierung. Er ist unter [coffee-shop@bablokb.de] zu erreichen.