Suche mit Lucene

Dr. Christian Herta

Mai, 2009

Lernziele - Inhalt

- Prozess der Suche
- Klassen der Suche
- Query-Objekte und Query-Syntax
- Scoring mit Lucene

Outline

Übersicht: Wichtige Klassen für die Suche

- `IndexSearcher`: Zentrale Klasse für den Zugang zum Index für die Suche
- `Query`: Klasse zur internen Repräsentation einer Anfrage; Subklassen repräsentieren dabei verschiedene Query-Typen
- `QueryParser`: Zum Überführen eines Suchstrings (Nutzereingabe) in ein `Query`-Objekt
- `TopDocs`: Container für den Zugriff der Treffer einer Suchanfrage
- `ScoreDocs`: Zugriff zu den einzelnen Treffern, die in den `TopDocs`-Instanzen gehalten werden

IndexSearcher

- Zentrale Klasse für den Index zur Suche
- Index wird über `search`-Methode des `IndexSearcher` angefragt
- Öffnen des `IndexSearchers` ist eine aufwändige Operation, da interne Datenstrukturen vom Index initialisiert werden müssen -> Wiederverwenden!

Initialisierung des IndexSearcher

- Angabe des Verzeichnisses in dem den Index liegt über:
  - Pfadangabe "/path/to/index"; benutzt dann intern `FSDirectory`
  - über `java.io.File`
  - `Directory`-Instanz
  - über `IndexReader`

search-Methoden des IndexSearcher

- Zugriff auf die Treffer mittels `TopDocs`
  - `TopDoc search(Query q, int n)`: *n* Anzahl der besten Treffer, die zurückgeliefert werden
  - `TopDoc search(Query q, Filter filter, int n)`: Filtern der Resultate mittels *filter*
  - `TopDoc search(Query q, Filter filter, int n, Sort sort)`: *Customisierte* Sortierung mittels *sort*
- Zugriff auf die Treffer mittels `HitCollector` für angepasste Logik (zur Anzeige der Dokumente)
  - `void search(Query q, HitCollector hits)`
  - `void search(Query q, Filter filter, HitCollector hits)`

Hinweis

`Filter` und `HitCollector` werden in späteren Vorlesungen behandelt

Aktualität

Index-Aktualität

Die Aktualität des durchsuchten Index bezieht sich auf den Index, der zur Zeit der Instanziierung des `IndexSearcher` vorhanden war.

- Neuere Dokumente oder Löschungen sind bei der Suche nicht sichtbar, bzw. erst nach:
  - Index-*commit* z.B. `IndexWriter.commit()`
  - Instanziierung eines neuen `IndexSearcher`

Zugriff auf die Treffer: `TopDocs`

- `search(q,n)` Methoden liefern `TopDocs`-Objekt zurück siehe
- über `TopDocs` kann auf die Suchergebnisse zugegriffen werden
- Resultate sind (in der Regel) absteigend sortiert (*Ranking*) nach Relevanzkriterium (*Cos-Similarity*) (oder Sortierung - spätere Vorlesung)

TopDocs-Methoden

- `totalHits`: Rückgabewert ist Anzahl der Dokumente der Treffer
- `scoreDocs`: Rückgabewert ist *Array of* `ScoreDoc`-Instanzen zum Zugriff auf die Resultate
- `getMaxScore()`: Rückgabewert ist höchster Score-Wert aller Treffern. Der höchste Score muss nicht in den Top `n`-Dokumenten vorhanden sein (Sorting-Kriterium)

Einzelne Treffer: `ScoreDoc`

- `ScoreDoc` dient für den Zugriff auf die einzelnen Treffer
- `ScoreDoc` besteht aus einer Dokument-ID `int doc` und einem Score-Wert `float score`
- mittels `IndexSearcher.document(doc)` kann auf die gespeicherten Felder zugegriffen werden

Begriffsklärung

unter *Paging* versteht man das Blättern in den Suchergebnissen

verschiedene Möglichkeiten zur Umsetzung:

- Genügend Resultate per `search(..)` holen und Instanzen von `ScoreDocs` und `IndexSearcher` offen halten (nicht *stateless*)
- Neue Anfrage falls der Nutzer blättert (*stateless*)
  - bevorzugte Methode in Web-Anwendung
  - (IO-)Caching macht wiederholte Anfragen schneller
  - Suchanfrage z.B. in Link-URL oder *hidden* HTML-Feld halten

Outline

Query-Objekte

- zur internen Repräsentation der *Query*
- `search`-Methoden des `IndexSearcher` nehmen ein `Query`-Objekt entgegen siehe
- Query-Objekte können verschiedenartig erzeugt werden:
  - Direkte Instanzierung von Query-Objekten, Subklassen von `Query`, wie `TermQuery`, `RangeQuery` etc.
  - mittels `QueryParser`
  - *XML Query-Parser Package* (Lucene Sandbox)
- Die verschiedenen Query-Objekten können mittels boolescher Ausdrücke zu einem neuen Query-Objekt kombiniert werden

Übersicht: Wichtigste Subklassen von Query

- `TermQuery`
- `RangeQuery`
- `PrefixQuery`
- `PhraseQuery`
- `WildcardQuery`
- `FuzzyQuery`
- `MatchAllDocsQuery`
- `BooleanQuery`

TermQuery

- Erzeugung mittels:
  - `Term t = new Term("contents","java");`
  - `Query q = new TermQuery(t);`
- keine Analyse (Stemming, *Lowercasing*, Normierung etc.)
- geeignet für Felder mit `Field.Index.NOT_ANALYSED`, d.h. z.B. zum Filtern, Dokumenten-IDs wie URIs etc.

RangeQuery

- *Range* über lexikographische Ordnung
- vom Start-Term bis zum End-Term (inklusiv oder exklusiv mittels zweier Konstruktor-Parameter)
- (intern) verschiedene Möglichkeiten:
  - expandiert zur OR-Query (Standard); Nachteile:
    - langsam falls zu vielen Terme expandiert
    - unintuitives Ranking wegen *idf*
  - über `setConstantScoreRewrite(true)` wird ein internes *bit set* verwendet; Konstanter Score gleich dem *Query-Boost*
  - beste Performance mittels `TrieRangeQuery` in Lucenes Sandbox

Hinweis

meist will man Range-Filtern (in späteren Vorlesung)

PrefixQuery

- *Match* aller Dokumenten, die mit einem Präfix beginnen
- Beispiel:
  - `Term t = new Term("category","/science/")`
  - `PrefixQuery q = new PrefixQuery(term)`
  - *Query-Match* z.B. bei *category: "/sciene/physics"*
- effizienter konstanter Score zum Ranken mittels `setConstantScoreRewrite(true)`, wie bei `RangeQuery`

PhraseQuery

- Phrasen-Suche über Positionslisten, d.h. nur für Felder **ohne** `omitTF(true)`(siehe Vorlesung: Indizierung)
- Stopworte können nicht genutzt werden!
- über *slop distance* (default: 0) kann eine *near query* durchgeführt werden; *move*
- Reihenfolge der Terme auch über *slop distance*, z.B. von "A B" zu "B A" entspricht *slop distance*: 2
- auch *multiple term phrases*(Vorlesung: Suche); *slop distance* bezieht sich auf die totale Anzahl der *moves*
- Scoring berücksichtigt *invers* die Anzahl der *moves*

$$\frac{1}{distance + 1} \tag{1}$$

WildcardQuery

- *Wildcard*-Operatoren
  - "*": *match* von keinem oder mehreren beliebigen Charaktern
  - "?": *match* von einem Charakter
- `setConstantScoreRewrite(true)`
- automatisches *lowercasing* als Standard-Verhalten

Achtung

Performance (vgl. Indexstruktur)

- mittels **Levenshtein Abstand** (Edit-Distance)
- z.B. Abstand ist 1 für "Haus" und "aus"
- Angabe eines *threshold* t, der die Länge der Strings berücksichtigt:

$$t = 1 - \frac{distance}{\min(textlen, targetlen)} \tag{2}$$

Achtung

Performance (vgl. Indexstruktur)

MatchAllDocsQuery

- *match* von jedem Dokument
- konstanter Score 1

Kombination von Query-Objekten

- Zusammenschalten verschiedenen Query-Objekte mittels Boolescher Ausdrücke (**AND**, **OR**, **NOT**)
- über Klasse `BooleanQuery` und Methode
  - `public void add(Query q, BooleanClause.Occur occur)`
  - `occur` kann die Werte `BooleanClause.Occur.MUST`, `BooleanClause.Occur.SHOULD`, `BooleanClause.Occur.MUST_NOT` annehmen
- Gruppieren und Verschachtelungen möglich, da auch andere `BooleanQuery`-Instanzen hinzugefügt werden können
- `TooManyClausesException` falls zu viele Ausdrücke hinzugefügt werden (*default*: 1024); dies kann auch intern geschehen (vgl. `RangeQuery`)
- mit `setMaxClauseCount(int)` kann der Wert erhöht werden; Achtung: Performance

Outline

QueryParser

- Klasse zum Parsen der Nutzeranfragen, z.B. Suchfeld in Web-Anwendung
  - `Query parser = new QueryParser(String field, Analyser analyser)`
- Benötigt einen `Analyser` im Konstruktor. Dieser muss kompatibel zum `Analyser` der Indizierung sein!
- Der `String field` ist *Default-Field* für die Suche: Falls kein explizites Feld für einen Ausdruck angegeben wird, wird in diesem gesucht.

parse-Methode

- Methode des `QueryParser`:
- `public Query parse(String query) throws ParseException`
- `String query` ist in der Regel die Nutzereingabe (Suchstring)
- Methode konvertiert `query`-String in `Query`-Objekt
- `query`-String muss der Lucene Query-Syntax entsprechen, falls nicht
  - `ParseException`, die beschreibt warum das Parsing nicht geklappt hat

Settings für `QueryParser`

- Default-Suchoperator (Standard: OR)
- Codierung von Datum-Angaben (*locale*)
- *Phrase-Slope*
- Minimum Ähnlichkeit und Präfix-Länge für *Fuzzy-Queries*
- Granularität des Datums
- *Lowercase of Wildcard-Queries*: ja oder nein
- etc.

Query-Syntax

- Die verschiedenen Query-Typen werden durch verschiedene spezielle Charakter ausgedrückt
- to escape special character: Backslash \
- Beispiel für einen *Query-String:*
  - `+pubdate:[20060301 TO 20081231] Java AND (Apache OR Jakarta)`
- *`Query.toString()`-Methode liefert Einblick, wie die Query aussieht*

Kombination von Klauseln

- `Default-Operator:` OR
- `parser.setOperator(QueryParser.AND_Operator)`
- `a AND b` oder `+a +b`
- `a OR b` oder `a b`
- `a AND NOT b` oder `+a -b`

Query-Syntax (1)

- `Gruppieren mittels Klammern: Perl AND (JAVA OR C++)`
- Feld auswählen: `author:(Goethe OR Schiller)`
- `PrefixQuery: Java*` *default lowercasing*
- `PhraseQuery mittels Anführungszeichen: "java program"`
  - Stopworte!
  - * ergibt keine *Wildcardquery*
  - *slope factor* mittels: ~an abschließende Anführungszeichen hängen

Query-Syntax (2)

- `WildcardQuery: Jav?pr*g`
  - keine *Wildcard* am Anfang erlaubt
  - *default lowercasing*
- `FuzzyQuery: JAVA~`
- Boosting mittels Carat: `junit^3.0 testing`

RangeQuery

- inklusive `[A TO F]`
- exklusive `{1041 TO 2010}`
- bei Datum `[1/1/04 TO 31/12/09]`
  - `parser.setDateResolution` auf gleiche Granularität wie beim Indexing einstellen
  - mittels `setLocale` auf lokale Interpretation einstellen (Internationalisierung mittels **I18N**), z.B. mittels `HttpServletRequest`

Query Parsing

- API mächtiger als `QueryParser`
- Kombination von Parsing mit expliziten `Query`-Objekten möglich über `BooleanQuery`
- eigene Subklassen von `QueryParser` zur Anpassung
- eigener Query-Parser z.B. über `ANTLR` oder `JFlex`

Outline

Scoring mit Lucene

- Prinzip:
  - erst mittels (erweiterten) Boolean-Modell die Dokumenten-Menge erhalten, die dem Boolean-Ausdruck entsprechen
  - anschließend *Scoring* mittels Vektorspace-Modell
  - *Ranking* durch Sortierung nach *Scores*
- Wie kann das Vektor-Space Modell bei einer Suche über mehrere Felder erweitert werden?

Recap: Vektor-Space Modell

- Score-Berechung:

$$score = \frac{\vec{d} \cdot \vec{q}}{|\vec{d}| * |\vec{q}|} \tag{3}$$

  - $\vec{d}$ bzw. $\vec{q}$ Dokument- bzw. Frage-Vektor
  - $|\vec{d}|$ bzw. $|\vec{q}|$ Längen-(Norm) des Dokuments bzw. der Frage

- Vektoren und Normen können unterschiedlich definiert werden. Hier wird meist mit dem *tf-idf* gearbeitet.
- Der *idf* dient dabei als Wichtung der Terme.
- Zusätzlich sind weitere explizite Wichtungsfaktoren (Boost) *b* sowohl im Dokument als auch in der Anrage denkbar.

Recap: *tf-idf*

- Term-Frequenz (*tf*) bei Lucenes `DefaultSimilarity`:
  - $tf_{d,t} = \sqrt{f_t}$: für das Dokument d und den Term t
  - f_t: (rohe) Termfrequenz, d.h die Anzahl des Vorkommen des Terms t im Dokument d
- *Inverse Document Frequency (idf)* mittels `DefaultSimilarity`:
  - $idf_t = 1 + \ln(N/df_t)$
  - Gesamtzahl der Dokumente: N
  - Dokumentenfrequenz: df_t
- $tf - idf$ ist Multiplikation von tf mit idf

Erweiterung des Vektor-Space Modells auf Felder

- Pro Feld ein Vektor-Raum und Addition der einzelnen Scores:

$$score_{d,q} = \sum_{f} score_{f,q_f} = \sum_{f} \left(b_f * \frac{\vec{f} \cdot \vec{q_f}}{|\vec{f}| * |\vec{q_f}|} \right) \quad (4)$$

  - f steht für die Felder der Dokumente
  - q_f ist der Anteil der Query der sich auf Feld f bezieht
  - b_f: Feld-Boost, um die Bedeutung der unterschiedlichen Felder anwendungsspezifisch zu wichten
- Ein Vektorraum für Feld-Term Paare
  - Eine Dimension wird durch ein Feld-Term Paar bestimmt, z.B. sind *content.Haus* und *abstract.Haus* zwei unterschiedliche Terme
  - In die Dokumentennorm $|\vec{d}|$ gehen somit auch Felder ein, die nicht abgefragt werden.

Implementierung des Scorings

- In den Subklassen der abstakten `Similarity`-Klasse
- Subklasse `DefaultSimilarity` wird verwendet, falls nicht anders definiert

Scoring bei Lucene mittels DefaultSimilarity

$$score_{d,q} = |\vec{q}|^{-1} * c_{q,d} * \sum_{f.t \in q} (tf_{f.t,d} * idf_{f.t}^2 * bq_{f.t} * n_{f.t}) \quad (5)$$

- $f.t$ steht für den Term bezüglich des Feldes f
- $bq_{f.t}$: Boost-Faktor des Terms $f.t$ der Query, um diesen in $\sum_f score_{f,q_f}$ der Anfrage zu wichten.
- $n_{f.t} = bi_{f.t}/|\vec{f}|$: Feld-Normalisierung und Boost-Faktor des Terms $f.t$ des Feldes (inkl. Dokument-Boost); $|\vec{f}| = \sqrt{n_f}$ mit n_f Anzahl der *types* (**unterschiedlichen** Terme) des Feldes
- $c_{q,d}$: Koordination-Faktor; je größer, desto mehr Terme der Query in den Feldern vorkommen

Lucene Scoring und Vektor-Space Modell

- Lucene Scoring entspricht im Wesentlichen dem pro 'Feld ein Vektorraum-Modell', wie leicht gezeigt werden kann:

$$score_{d,q} = |\vec{q}|^{-1} * c_{q,d} * \sum_{f.t \in q} (tf_{f.t,d} * idf_{f.t}^2 * bq_{f.t} * n_{f.t}) = \quad (6)$$

$$\frac{c_{q,d}}{|\vec{q}|} \sum_{f} \frac{\sum_{f.t}((tf_{f.t,d} idf_{f.t} bi_{f.t})(tf_{f.t,q} idf_{f.t} bq_{f.t}))}{|\vec{f}|} \quad (7)$$

- Hier gibt es keine feldgenaue Fragenormierung $|\vec{q_f}|$. Dies wird aber einfach durch eine andere Wahl von $bi_{f.t}$ ausgeglichen.
- $|\vec{q}|^{-1}$ dient nur dazu *Scores* verschiedener Anfragen vergleichen zu können, und hat keine Auswirkungen auf das Ranking.
- $c_{q,d}$ als zusätzlicher Boost

idf für Phrasen

- .....

Explanation

- `IndexSearcher` liefert über `explain(query, docID)`-Methode ein `Explanation`-Objekt
- mittels `Explanation`-Objekt können die Details bezüglich des Scorings abgefragt werden (wie *tf, idf, boost-Faktoren, Query-Norm etc.*)
- mittels `.toString()` kann man dies in einem formatiertem *String* überführen und anzeigen lassen; oder `.toHtml()`