Ausgewählte fortgeschrittene Themen von Lucene

Dr. Christian Herta

Mai, 2009

Outline

Sortierung

- Standard: *Relevance Ranking*
- Für Anwendungen aber auch andere Sortierungen interessant, z.B.:
  - Sortierung nach Preis
  - Sortierung nach Datum
  - Sortierung nach Abstand
- Recap: Suche über `search(Query, Filter, int, Sort)`-Methode möglich

Sort

- Lucene bietet die Möglichkeit mittels der Klasse `Sort` Objekte zu erzeugen nach denen sortiert wird
- Recap: Sortierung nach Feldern möglich, die nicht analysiert wurden (vgl. Vorlesung Indizierung)
  - `Field.Index.NOT_ANALYZED`
  - Wert muss konvertierbar sein zu Intger, Float oder Strings
- Im Konstruktor von `Sort` kann man die Felder angeben
- z.B. `Sort indexDateSort = new Sort("indexDate", true);`

Sortierung mittels mehrerer Felder

- Falls Sortierwerte für Treffer gleich sind, wird nach weiteren Kriterien sortiert; Standard: eindeutige DokumentenID (`doc`), d.h.*Indexorder*)
- Konstruktor von `Sort` kann man ein Sortierfeld-Array (`SortField`) angeben:

```
Sort indexDateSort =
    new SortField[]{
    new SortField("category"),
    new SortField("indexDate",
                  SortField.INT,
                  true),
    Sort.RELEVANCE)
    }
```

SortField

- mittels `SortField`-Array kann mehrere Felder für die Sortierung benutzen
- vordefinierte Sortobjekte: `Sort.INDEXORDER`, `Sort.RELEVANCE`
- Sortierung nach numerischen Typen kostet weniger Speicher, als nach Strings
- Sortierordung: `true` bedeutet natürliche Sortierung (absteigend - außer bei Relevance)

field cache

- (*single value*) Felder können in einem *Cache* gehalten werden
- Viele eingebaute Funktionalitäten von Lucene benutzen den *Field cache* "unter der Haube"
- für native Typen, wie *byte, short, int, float, double, string, StringIndex*
- basiert auf `WeakHashMap`

Outline

Aufgabe: Filter

- Einschränkung des *search space*, z.B.
  - *Security*
  - *Kategorien*
- Caching (mittels `DocIdBitSet`)
- Anwendung in `search`-Methoden mittels Übergabe von `Filter`-Objekten
- Alternative: `BooleanQuery` (Beachte: Ranking und *idf*)

Filtertypen

- `PrefixFilter`: Filtern nach Prefix
- `RangeFilter` (für `String` und num. Werte) und Varianten
  - `FieldCacheRangeFilter`
  - `TrieRangeFilter`
- `FieldCacheTermsFilter`: Filtern nach Termen in *single value*-Feldern
- `QueryWrapperFilter`: Filtern nach `Query`-Objekt
- `CachingWrapperFilter`: Wrapper zur Wiederverwendbarkeit der Filterergebnisse - Performance (intern: `weakHashMap`)

Jenseits der eingebauten Filter

- `ChainedFilter` (Sandbox): Filterketten
- Eigene Filter z.B. um externe Daten einfließen zu lassen

Outline

Was sind Payloads?

- An jede Term-Position kann ein beliebiges *byte-array* gespeichert werden
- z.B. für
  - Änderung des Scorings (z.B. HTML-Überschriften stärker wichten)
  - Entscheiden, welche Dokumente in die Suchergebnisse übernommen werden sollen

Erzeugen von Payload

- mittels `Analyzer`:
  - `PayloadAttribute.setPayload` innerhalb der `incrementToken`-Methode
- Sandbox
  - `NumericPayloadTokenFilter`: für spezielle *Token-Types* *float*-Werte im Payload codieren
  - `TypeAsPayloadTokenFilter`: Encodieren von *Token-Types* im Payload
  - `TokenOffsetPayloadTokenFilter`: Start- und Endposition im Payload ablegen
  - `PayloadHelper`: Statische Methode zur De- und Encodierung von `int`- und `float`-Werten in Payloads

Outline

- `SpanQueries`: flexible Nutzung von Positionsangaben in der Suche
- *Customisierte* Sortierung, wie Geo-Abstand
- *Customisierte* `QueryParser`
- Suchen in mehreren Indizes
- Nutzen von Term-Vektoren, z.B. zur Dokument-Dokument Ähnlichkeit
- *Highlighting* von Suchergebnissen
- `HitCollector` (z.B. für Implementierung von *Group-by*?)