Indizierung mit Lucene

Dr. Christian Herta

April, 2009

Lernziele - Inhalt

- Indizierungsprozess mit Lucene
- Dokumente: Feldstruktur und Optionen
- (grobe) Index-Struktur und `Directory`
- *Concurrency - Index Locking*
- Transaktionen

Indexing-Prozess

- Extraktion des Textes aus den zu indizierten Dokumenten wie PDF, HTML und Überführen in Felder
- Analysis zur Indizierung wie `LowerCaseFilter`, `StopFilter`, `PorterStemFilter`
- Indizierung: Erzeugen des invertierten Index

Documents und Fields

- `Document` ist die atomare Einheit für die Indizierung und Suche
- `Document` ist Container für `Fields`
- jedes `Field` hat einen Namen zur eindeutigen Identifikation
- Felder haben entweder Text-Werte oder binäre Werte
- Alles was indiziert und durchsucht werden soll, muss in die `Document-Field` Struktur überführt werden
- Beispiele für Felder: Autor, Überschrift, Erstellungsdatum, URL, Abstract, Inhalt, Keywords etc.
- Optionen für `Fields` wie diese bei der Indizierung behandelt werden sollen

Optionen für Felder (`Field`)

- Indizieren oder nicht
- Für indizierte Felder können Term-Vektoren gespeichert werden. Term-Vektoren enthalten alle Token des Feldes.
- Speichern des Feldes. Nur diese Felder werden zum Dokument bei der Suche erhalten

Flexibilität

- Unterschiedliche Dokumente können unterschiedliche Felder haben
- Es muss kein Schema im Voraus festgelegt werden
- Meta-Dokumente könnten Eigenschaften des Korpus beschreiben

Denormalisierung

- Dokumenten-Format ist flach: Document-Field Struktur
- verschachtelte Strukturen oder müssen in diese flache Struktur überführt werden
- wie XML-Verschachtelungen, Datenbank-Strukturen
- dies wird Denormalisierung genannt
- dies wird z.B. bei folgenden Open Soure Projekten umgesetzt: Hibernate-Search, Compass, DB Sight, Browse Engine, Oracle/Lucene

Index-Segmente

- Der Index von Lucene besteht aus einem oder mehreren Segmenten
- Segmente entsprechen eigenen (Sub-)Indizies
- Bei der Suche werden die einzelnen Segmente separat durchsucht und die Ergebnisse kombiniert
- Segmente werden von `IndexWriter` erzeugt: Dabei werden gelöschte und/oder hinzugefügte Dokumente aus dem *Buffer* in ein Segment geschrieben. Dies wird als *Flush* bezeichnet.
- Änderungen am Index (add oder delete) sind erst nach dem *Flush* sichtbar!
- Ein *Flush* wird explizit durch folgende `IndexWriter`-Methoden ausgeführt:
  - `commit()`
  - `close()`
- Jedes Segment besteht aus mehreren Files

Steuern von *Buffering* und *Flashing*

- Kontrolle über Methoden des `IndexWriter` z.B.:
  - `setRAMBufferSizeMB(..)` (default: 16MB)
  - `setMaxBufferedDocs(..)`
  - `setMaxBufferedDeleteTerms(..)`
  - Konstante `IndexWriter.DISABLE_AUTO_FLUSH` kann auch den Methoden übergeben werden

Compound-Files

- Zwei verschiedene Formate bezüglich der Dateien
  - Separate Files für die einzelnen Index-Teile (_X.<ext>)
  - Compound-File (Standard-Verhalten): Hier werden die verschiedenen Dateien in einer Datei zusammengefasst (_X.cfs)
- Compound-Files reduzieren die Anzahl der offenen File-Deskriptoren
- Performance (Such-Geschwindigkeit) von Compound-Files etwas schlechter

Index-Segmente und Merge

- Datei `segments_N` enthält die Referenzen zu den Segmenten, d.h. welche Segmente zum Gesamt-Index gehören
- N ist die Generation des Index. Wird bei jedem Änderung zum Index die *commited* wird um eins erhöht.
- Mit der Zeit entstehen immer mehr Segmente (z.B. beim Öffnen und Schließen des `IndexWriter`)
- Da viele Segmente die Performance bei der Suche verringern, fasst der `IndexWriter` ab und an Index-Segment zusammen. Dies wird als *Merge* bezeichnet. Wann dies passiert wird in der `MergePolicy` festgelegt und vom `MergeScheduler` gesteuert.

Merge und `MergePolicy`

- Merge zum
  - Anzahl der Segmente verringern
  - Größe des Index verringern
- `MergePolicy` - Standard `LogMergePolicy`: Dokumenten- oder Größen-abhängig
  - für `LogSizeMergePolicy`:
  - Levels über `(int) log(max(minMergeMB,size)/log(mergeFactor))`
  - falls ein Level mehr `mergeFactor` oder mehr Segmente hat, so werden diese *gemerged*
- Kontrolle über `setMergeXXX`-Methoden des `IndexWriter`
- falls andere (z.B. zeitabhängige) `MergePolicy` gewünscht, so kann man eine eigene Unterklasse von `MergePolicy` implementieren

Dokumente zum Index hinzufügen

- Methoden des `IndexWriter`
  - `addDocument(Document)`
  - `addDocument(Document, Analyzer)` - aber Vorsicht, gleichen Analyser für die Suche verwenden!
- `Document` besteht dabei aus Feldern, die durch folgende (Beispiel-)Methode von `Document` hinzugefügt werden können:
  - `doc.add(new Field("content", contentString, Field.Store.NO, Field.Index.ANALYSED))`
- Feldoptionen bzg. Speichern und Analyse im Feldkonstruktor

Feld-Optionen für Indexing

- `Field.Index.ANALYZED`: Indizierung und Benutzen des Analysers
- `Field.Index.NOT_ANALYZED`: Indizierung, aber keine Benutzung des Analysers; d.h. Feld ist ein Token
- `Field.Index.NO`: Feld wird überhaupt nicht zum Suchen indiziert

weitere Feld-Optionen für Indexing

- Norm-Informationen (Boost-Informationen zum Ranking) werden normalerweise im Index gespeichert; um Speicher bei der Suche zu sparen können diese abgeschaltet werden
  - `Field.Index.ANALYZED_NO_NORMS`: wie `Field.Index.ANALYZED`, aber keine Norm-Information
  - `Field.Index.NOT_ANALYZED_NO_NORMS`: wie `Field.Index.NOT_ANALYZED`, aber keine Norm-Information
- `Field.setOmitTf(true)`: Kein Abspeichern der Term-Frequenzen - nur Filtern oder Boolean Search; weniger Speicherverbrauch, event. schnellere Suche bzw. Filtern; auch keine Positionen (und Payloads)

Feld-Optionen zum Speichern von Feldern

- `Field.Store.YES`: Speichern des Felds
- `Field.Store.COMPRESS`: Speichern des Felds in komprimierter Form
- `Field.Store.NO`: kein Speichern, typischerweise bei großen Feldern, um den Index nicht aufzublähen

Löschen von Documents

- folgende Funktionen des `IndexWriter` können zum Löschen verwendet werden:
  - `deleteDocuments(Term)`: Löscht alle `Documents`, die den Term enthalten
  - `deleteDocuments(Term[])`: Löscht alle `Documents`, die irgendeinen Term des Term-Arrays enthalten
  - `deleteDocuments(Query)`: Löscht alle `Documents`, welche die `Query` mached
  - `deleteDocuments(Query[])`: Löscht alle `Documents`, welche irgendeine der `Querys` mached

Löschen von Documents (2)

- Will man ein bestimmtes Dokument löschen, identifiziert man typischerweise alle Dokumente über ein ID-Feld (analog *Primary Key* in SQL-Datenbank) und löscht z.B. mittels
  - `writer.deleteDocument(new Term("ID", documentID))`
- Gelöschte Dokumente sind (wie hinzugefügte) erst nach einem *Flush* sichtbar, daher eventuell folgende `IndexWriter`-Methoden aufrufen:
  - `commit()`
  - `close()`
- `expungeDeletes()` um *disk space* zurückzubekommen; eventuell weniger aufwendig als `optimize(..)`

Löschen von Documents (2)

- Blacklisting von gelöschten `Documents`
- `hasDeletions()`
- `maxDocs()`: totale Anzahl von gelöschten und ungelöschten Documents im Index
- `nbDocs()`: totale Anzahl von ungelöschten Documents im Index

Updating

- *Updating* von `Documents` bzw. Feldern wird durch Löschen des `Document` und wiederhinzufügen gelöst
- kann über zwei *Convenience*-Funktionen erfolgen
  - `updateDocuments(Term, Document)`
  - `updateDocuments(Term, Document, Analyzer)`
- Vorsicht: intern `delete` und `add`, daher kann man über `Term` mehr als einen `Document` löschen, daher nur über eindeutige ID, z.B.:
- `writer.updateDocument(new Term("ID", docID), newDoc)`

Term-Vektoren

- Term-Vektoren (`TermVector`) beinhalten die einzelnen Terme
- In der Form, wie sie der Analyser produziert(vgl. Term vs. Type vs. Token)
- Alphabetisch geordnet
- inkl. Häufigkeit (**Term-Frequenz**)
- eventuell mit Positionen

Feld-Optionen für Term-Vektoren

- Optionaler (letzter) Parameter des Feld-Konstruktors:
  - `TermVector.YES`: Speichern des Term-Vektors - ohne Positionen
  - `TermVector.WITH_POSITIONS`: Speichern des Term-Vektors - mit Positionen
  - `TermVector.WITH_OFFSETS`: Speichern des Term-Vektors - mit Positionen und Offset
  - `TermVector.NO`: kein Speichern des Term-Vektors
- Term-Vektoren nur für Felder die auch indiziert werden, d.h. `Index.NO -> TermVector.NO`

Weitere Feld Optionen

- folgende Konstruktoren haben weitere Optionen und feste Einstellungen
- `Field(String name, Reader val, TermVector vec)`
  - *not stored, analysed and indexed*, `vec` für TermVektor-Optionen
- `Field(String name, TokenStream stream, TermVector vec)`
  - *analysed and indexed, not stored*, `vec` für TermVektor-Optionen
- `Field(String name, byte[] val, Store store)`
  - zum Speichern eines binären Feldes, *not indexed, store* für die Store-Optionen: `Store.YES` oder `Store.COMPRESS`

Multi-valued Fields

- Felder mit dem gleichen Namen können mehrmals zu einem Feld hinzugefügt werden
- falls das Feld indiziert wird, werden die Positionen über die Einfügereihenfolge bestimmt
- gespeichert werden sie separat, d.h. man erhält sie zum Dokument als mehrere Feld-Instanzen

Boosting

- `Document` und `Field` können mit Gewichten versehen werden
- Standard Boost-Faktor ist 1
- `Document` Boost-Faktor ändern:
  - `doc.setBoost(1.5f)`
- Dokument-Boost setzt intern die Felder auf den neuen Boost-Faktor
- `Feld` Boost-Faktor ändern:
  - `field1.setBoost(2.0f)`
- **Kein Setzen**: Wird ein neuer Boost-Faktoren gesetzt, so wird eine Multiplikation mit den alten Boost-Faktor vorgenommen.

Norms

- Für jedes Feld in jedem Dokument werden alle Boost-Faktoren zu einem Wert zusammengefasst, wie
  - Kurze Felder haben automatisch höheren internen Boost-Faktor
  - Explizites (Dokument- und) Feld-Boosting
- quantisiert auf ein Byte pro Feld
- Fortgeschritten: mit `setNorm` des `IndexReader` kann dieser auch explizit gesetzt werden, z.B. für *ClickPopularity*
- Norms haben hohen Speicherbedarf daher können diese vor der Indizierung abgeschaltet werden, z.B. durch `Field.setOmitNorms(true)`; dies hat Einfluss auf das Scoring bei Fragen in das Feld
- Norm nicht *Sparse*: Aber falls ein Dokument mit Norm in ein Feld indiziert wurde, haben alle Dokumente für das Feld eine Norm (zumindest nach *Merge*)

Nummern

- Feld mit reiner Zahl: *Nummerisches Feld*
  - Beachte alle Terme im Index sind Strings; für alphanumerische Sortierung
  - daher (eventuell) mit Nullen auffüllen
  - `Field.Index.NOT_ANALYZED`
  - Range-Queries möglich
- falls Nummern wie Terme aus dem Volltext indiziert werden sollen, muss ein entsprechender Analyser verwendet werden

Datum und Zeit

- Beachte alle Terme im Index sind Strings; Alphanumerische Sortierung
- `DateTools`: Klasse zum Konvertieren von Datum in String und umgekehrt: auf Format `YYYYMMDDhhmmss`
- anschließend Suffix-Stripping bis zur benötigten Granularität, da jeder Term einen Index-Eintrag erzeugt

Felder für Sortierung

- `Field.Index.NOT_ANALYZED`
- Wert muss konvertierbar sein zu Integer, Floats oder Strings

Field-Truncation

- `MaxFieldLength` steuert die Anzahl der Token die indiziert werden
- `MaxFieldLength.UNLIMITED`
- `MaxFieldLength.LIMITED`: 10000 Token
- `MaxFieldLength` kann auf gewünschten Wert gesetzt werden
- ob Trunkierungen passieren, kann im Info-Stream gesehen werden
- Vorsicht: Dokumente können so eventuell nicht gefunden werden

Optimierung des Index

- Der Index kann aus mehreren Segementen bestehen.
- bei der Optimierung wird die Anzahl der Segmente verringert, dies führt zu schnellerer Suche
- explizite Funktionen zur Optimierung
  - `optimize()`: Merge des Index zu einem Segment
  - `optimize(int maxNumSegments)`: Merge des Index zur maximalen Anzahl von `maxNumSegments` Segmenten
  - `optimize(boolean doWait)`: Merge im Hintergrund oder nicht
  - `optimize(int maxNumSegments, boolean doWait)`
- Optimierung benötigt viel *diskIO*
- Optimierung benötigt temporär *disk space*

Directory Implementations

- `RAMDirectory`: Speichert Index im RAM
- `FSDirectory`: Standard Festplatte - normaler IO-Zugriff auf Dateien
- `MMapDirectory`: Memory Mapping auf Festplatte (d.h. bei 32-bit Java muss Index kleiner als 4GB sein - Addressraum!)
- `NIOFSDirectory`-Festplatte über Javas native IO-Package (`java.nio.*`); falls in Anwendung viele *Threads* einen *searcher* teilen, ist Performance (wahrscheinlich) besser

Index-Copy zwischen *Directory Implementations*

- Beschleunigen der Suche durch Laden des Index in RAM, z.B.
  - `Directory ramDir = new RAMDirectory(otherDir);`
  - Kopieren des Index vom RAMDir zu anderem DIR (ersetzen aller Files!):
  - `Directory(ramDir, otherDir)`
- falls Index nicht ersetzt, sondern neu indizierte Dokumente hinzugefügt werden sollen, über `IndexWriter.addIndexesNoOptimize`
- ab Lucene 2.3 selbstimplementiertes *Memory-Buffern* zur Beschleunigung des Indizierens nicht mehr nötig (Indizieren in RAMDirectory und kopieren zu Festplatten-Index)!

Lucenes Concurrency Rules

- Beliebig viele `IndexReader` können gleichzeitig auf einem Index geöffnet werden
  - in der gleichen oder auf verschiedenen JVMs
  - auf dem gleichen oder verschiedenen Computern
  - aber aus Performancegründen am besten: verschiedene *Such-Threads* Teilen einen `IndexReader`
- nur ein `IndexWriter` kann auf einem Index geöffnet werden
  - `IndexWriter` erzeugt *write lock*
  - falls `IndexReader` `Documents` löscht oder *Norms* ändert, wird auch ein *lock* erzeugt

Teilen von `IndexReader` und `IndexWriter`

- *Theads* können `IndexReader` und `IndexWriter` teilen
- *Thread friendly*: gute Skalierung durch Minimierung von synchronisiertem Code

Index Locking

- *File-based Lock*: File im *Directory*
- falls Lock schon von anderem Writer gesetzt: `LockObtainFailedException`
- verschiedene Möglichkeiten `Directory` zu locken; mittels `Directory.setLockFactory`:
  - `SimpleFSLockFactory`: Lock-File - falls JVM *crashed* oder `IndexWriter` nicht geschlossen wird bleibt das *Lock-File* erhalten
  - `NativeFSLockFactory`: File mittels `java.nio`; nicht über (geteiltes) Netz-Dateisystem, wie NFS, verwenden!
  - `SingleInstanceLockFactory`: Lock im RAM; Standard bei `RAMDirectory`
  - `NoLockFactory`: Locking ausschalten. Vorsicht!!
- eigene Implementierungen möglich; Lucene bietet *Tool* `LockStressTest` zum Testen

Hilfsfunktion zum *Index Locking*

- `IndexWriter` Funktionen
  - `isLocked(Directory)`: Abfrage ob `Directory` *locked*
  - `unlock(Directory)`: aufheben des *Index-Locks* - Vorsicht!

ACID Transaktionen - Index Konsistenz

- *ACID* Transaktions Modell in Lucene
  - *Atomic*: alle Änderungen zum Index werden entweder *committed* oder nicht
  - *Consistency*: z.B. bei *update* kein *delete* ohne *add*; alle oder keine Indizes hinzugefügt bei `addIndexes`
  - *Isolation*: Änderungen nur nach *commit* sichtbar
  - *Durability*: bei einem *Crash* bleibt der Index konsistent - auf dem Zustand des letzten *commit*
- Einschränkung: nur eine offene Transaktion (ein offener `IndexWriter`)
- `writer.rollback`: Alle Änderungen bis zum letzten *Commit* rückgängig machen

IndexWriter.commit()

- Schritte die der `IndexWriter` beim `commit()` ausführt:
  - *Flush* der *gebufferten* Dokumente und *Deletions*
  - *Sync* aller Files (`Directory.sync`)
  - Schreiben und *Sync* des nächsten `segments_N files`
  - Löschen der alten *Commits* nach der `IndexDeletionPolicy`

IndexDeletionPolicy und alte Index-Zustände

- `IndexDeletionPolicy` beschreibt welche *Commits* veraltet sind. Diese können dann glöscht werden
- Standard: `KeepOnlyLastCommitDeletionPolicy`
  - Löschen aller alten *Commits*, falls neuer *Commit* vollständig ist
- Falls *Point-in-time Snapshots* gewünscht, muss man eigene `IndexDeletionPolicy` implementieren
  - `commit(String commitUserData)` erlaubt *Commits* zu benennen
  - `IndexReader.listCommits()` zählt die zugreifbaren *Commits* auf
  - open-Methode des `IndexReader` kann auf altem *Commit* ausgeführt werden; so ist eine Suche auf einem alten Stand möglich
  - auch `IndexWriter` kann auf einem alten *Commit* geöffnet werden, so ist es möglich auf alten Ständen weiter zu indizieren (*rollback*)

2-Phasen *Commit*

- Phase 1: `prepareCommit()` oder `prepareCommit(String commitUserData)`
  - Vorbereiten des Commits: Schritte 1,2 und das Meiste von 3
  - `segments_N` aber nicht sichtbar (für `Reader`)
- Phase 2: `commmit()` oder `rollback()`
  - schnelle Operation, da alle auffwändigen Schritte im `prepareCommit` stattfanden
  - Fehler finden vor allem beim `prepareCommit` statt
- über 2-Phasen *Commit* ist *distributed 2-Phase Commit* möglich