Introduction to

Information Retrieval

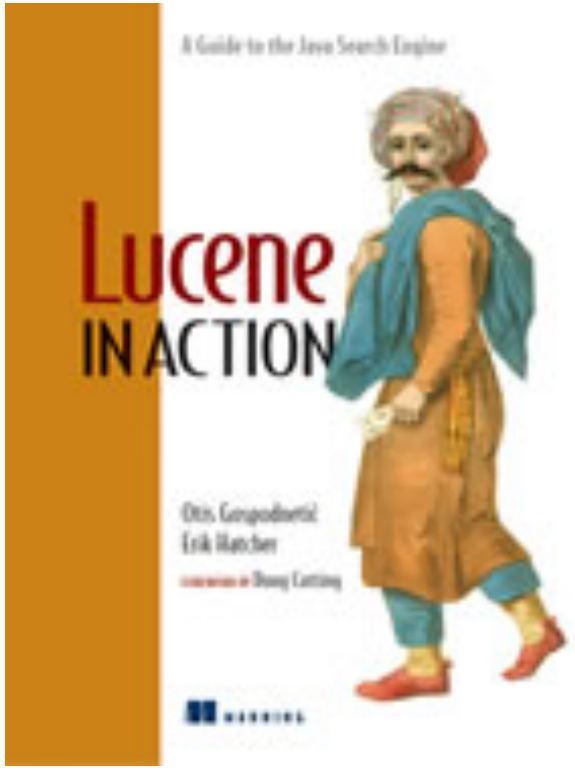
Lucene Tutorial

Chris Manning, Pandu Nayak, and Prabhakar Raghavan

further edited by Hui Shen, Xin Ye, and Razvan Bunescu

Based on "Lucene in Action"

- By Michael McCandless, Erik Hatcher, Otis Gospodnetic



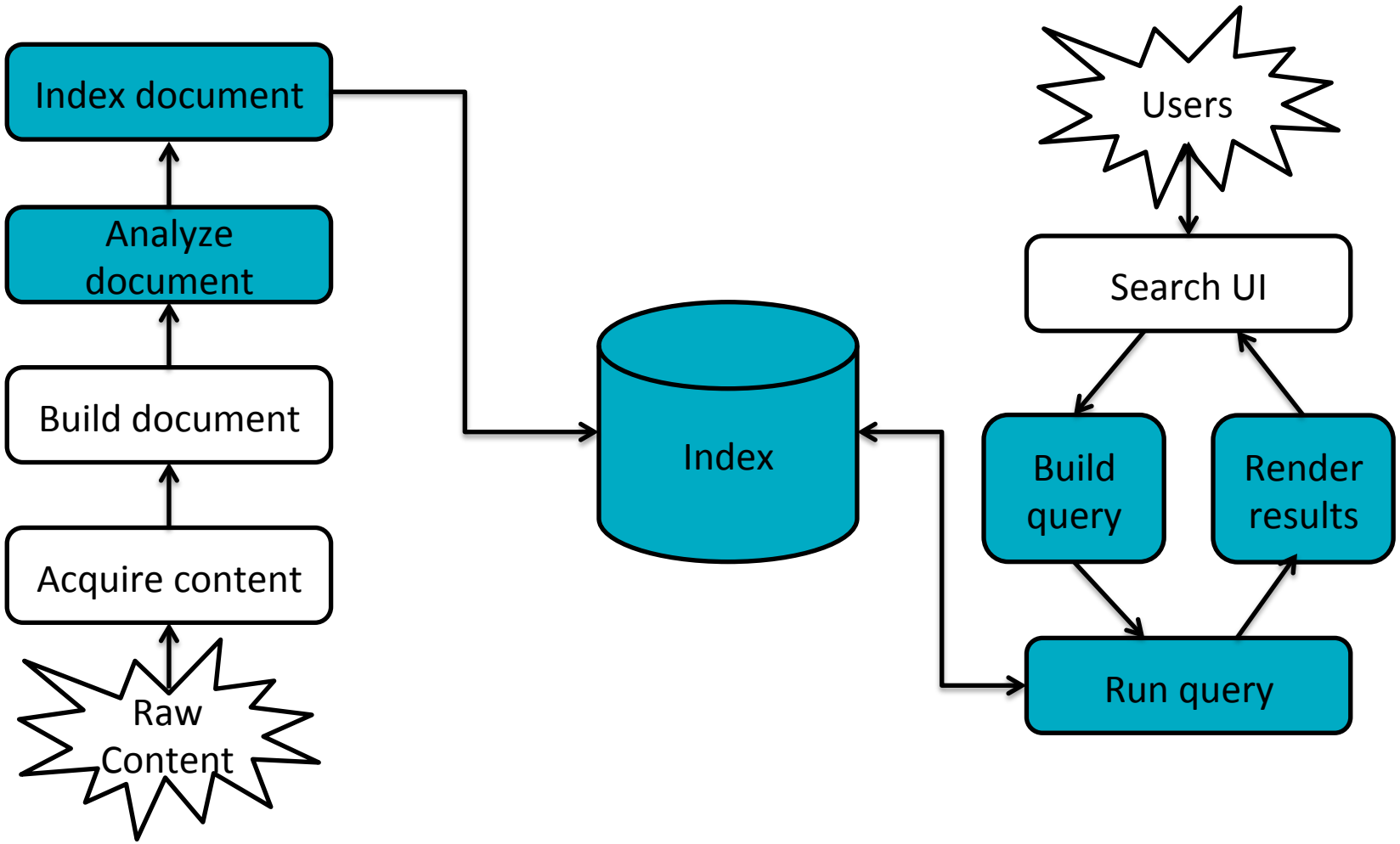
Lucene

- Open source Java library for indexing and searching
  - Lets you add search to your application
  - Not a complete search system by itself
  - Written by Doug Cutting
- Used by LinkedIn, Twitter, …
  - …and many more (see http://wiki.apache.org/lucene-java/PoweredBy)
- Ports/integrations to other languages
  - C/C++, C#, Ruby, Perl, Python, PHP, …

Resources

- Lucene: http://lucene.apache.org/core/
- Lucene in Action: http://www.manning.com/hatcher3/
  - Code samples available for download
- Ant: http://ant.apache.org/
  - Java build system used by "Lucene in Action" code

Lucene in a search system



Lucene in action

- Command line **`Indexer`**
  - `…/lia2e/src/lia/meetlucene/Indexer.java`
- Command line **`Searcher`**
  - `…/lia2e3/src/lia/meetlucene/Searcher.java`

Core indexing classes

- `IndexWriter`
  - Central component that allows you to create a new index, open an existing one, and add, remove, or update documents in an index
- `Directory`
  - Abstract class that represents the location of an index
- `Analyzer`
  - Extracts tokens from a text stream

Creating an `IndexWriter`

```
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
...
private IndexWriter writer;
...
public Indexer(String indexDir) throws IOException {
  Directory dir = FSDirectory.open(new File(indexDir));
  writer = new IndexWriter(
                dir,
                new StandardAnalyzer(Version.LUCENE_30),
                true,
                IndexWriter.MaxFieldLength.UNLIMITED);
}
```

Core indexing classes (contd.)

- `Document`
  - Represents a collection of named `Fields`. Text in these `Fields` are indexed.
- `Field`
  - Note: Lucene `Fields` can represent both "fields" and "zones" as described in the textbook

A Document contains Fields

```
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
...
protected Document getDocument(File f) throws Exception {
    Document doc = new Document();
    doc.add(new Field("contents", new FileReader(f)))
    doc.add(new Field("filename",
                      f.getName(),
                      Field.Store.YES,
                      Field.Index.NOT_ANALYZED));
    doc.add(new Field("fullpath",
                      f.getCanonicalPath(),
                      Field.Store.YES,
                      Field.Index.NOT_ANALYZED));
    return doc;
}
```

Index a Document with IndexWriter

```
private IndexWriter writer;
...
private void indexFile(File f) throws
     Exception {
   Document doc = getDocument(f);
   writer.addDocument(doc);
}
```

Indexing a directory

```
private IndexWriter writer;
...
public int index(String dataDir,
                 FileFilter filter)
      throws Exception {
   File[] files = new File(dataDir).listFiles();
   for (File f: files) {
      if (... &&
          (filter == null || filter.accept(f))) {
         indexFile(f);
      }
   }
   return writer.numDocs();
}
```

Closing the `IndexWriter`

```
private IndexWriter writer;
...
public void close() throws IOException {
   writer.close();
}
```

Core searching classes

- `IndexSearcher`
  - Central class that exposes several search methods on an index
- `Query`
  - Abstract query class. Concrete subclasses represent specific types of queries, e.g., matching terms in fields, boolean queries, phrase queries, …
- `QueryParser`
  - Parses a textual representation of a query into a `Query` instance

Creating an IndexSearcher

```
import org.apache.lucene.search.IndexSearcher;
...
public static void search(String indexDir,
                          String q)
    throws IOException, ParseException {
  Directory dir = FSDirectory.open(
                      new File(indexDir));
  IndexSearcher is = new IndexSearcher(dir);
  ...
}
```

Query and QueryParser

```
import org.apache.lucene.search.Query;
import org.apache.lucene.queryParser.QueryParser;
...
public static void search(String indexDir, String q)
      throws IOException, ParseException
   ...
   QueryParser parser =
      new QueryParser(Version.LUCENE_30,
                      "contents",
                      new StandardAnalyzer(
                            Version.LUCENE_30));
   Query query = parser.parse(q);
   ...
}
```

Core searching classes (contd.)

- `TopDocs`
  - Contains references to the top documents returned by a search
- `ScoreDoc`
  - Represents a single search result

search( ) returns TopDocs

```
import org.apache.lucene.search.TopDocs;
...
public static void search(String indexDir,
                          String q)
    throws IOException, ParseException
  ...
  IndexSearcher is = ...;
  ...
  Query query = ...;
  ...
  TopDocs hits = is.search(query, 10);
}
```

TopDocs contain ScoreDocs

```
import org.apache.lucene.search.ScoreDoc;
...
public static void search(String indexDir, String q)
      throws IOException, ParseException
   ...
   IndexSearcher is = ...;
   ...
   TopDocs hits = ...;
   ...
   for(ScoreDoc scoreDoc : hits.scoreDocs) {
      Document doc = is.doc(scoreDoc.doc);
      System.out.println(doc.get("fullpath"));
   }
}
```

Closing IndexSearcher

```
public static void search(String indexDir,
                          String q)
    throws IOException, ParseException
  ...
  IndexSearcher is = ...;
  ...
  is.close();
}
```

How Lucene models content

- A `Document` is the atomic unit of indexing and searching
  - A `Document` contains `Fields`
- `Fields` have a name and a value
  - You have to translate raw content into `Fields`
  - Examples: Title, author, date, abstract, body, URL, keywords, ...
  - Different documents can have different fields
  - Search a field using name:term, e.g., title:lucene

`Fields`

- `Fields` may
  - Be indexed or not
    - Indexed fields may or may not be analyzed (i.e., tokenized with an `Analyzer`)
      - Non-analyzed fields view the entire value as a single token (useful for URLs, paths, dates, social security numbers, ...)
  - Be stored or not
    - Useful for fields that you'd like to display to users
  - Optionally store term vectors
    - Like a positional index on the `Field`'s terms
    - Useful for highlighting, finding similar documents, categorization

`Field` construction
Lots of different constructors

```
import org.apache.lucene.document.Field

Field(String name,
      String value,
      Field.Store store,  // store or not
      Field.Index index,  // index or not
      Field.TermVector termVector);
```

`value` can also be specified with a `Reader`, a `TokenStream`, or a `byte[]`

Field options

- `Field.Store`
  - `NO` : Don't store the field value in the index
  - `YES` : Store the field value in the index
- `Field.Index`
  - `ANALYZED` : Tokenize with an `Analyzer`
  - `NOT_ANALYZED` : Do not tokenize
  - `NO` : Do not index this field
  - `Couple of other advanced options`
- `Field.TermVector`
  - `NO` : Don't store term vectors
  - `YES` : Store term vectors
  - Several other options to store positions and offsets

Using `Field` options

Index	Store	TermVector	Example usage
NOT_ANALYZED	YES	NO	Identifiers, telephone/SSNs, URLs, dates, ...
ANALYZED	YES	WITH_POSITIONS_OFFSETS	Title, abstract
ANALYZED	NO	WITH_POSITIONS_OFFSETS	Body
NO	YES	NO	Document type, DB keys (if not used for searching)
NOT_ANALYZED	NO	NO	Hidden keywords

Document

```
import org.apache.lucene.document.Field
```

- Constructor:
  - `Document();`
- Methods
  - `void add(Fieldable field); // Field implements // Fieldable`
  - `String get(String name); // Returns value of // Field with given // name`
  - `Fieldable getFieldable(String name);`
  - … and many more

Multi-valued fields

- You can add multiple `Fields` with the same name
  - Lucene simply concatenates the different values for that named Field

```
Document doc = new Document();
doc.add(new Field("author",
                  "chris manning",
                  Field.Store.YES,
                  Field.Index.ANALYZED));
doc.add(new Field("author",
                  "prabhakar raghavan",
                  Field.Store.YES,
                  Field.Index.ANALYZED));
...
```

`Analyzers`

- Tokenizes the input text
- Common `Analyzers`
  - `WhitespaceAnalyzer`
    Splits tokens on whitespace
  - `SimpleAnalyzer`
    Splits tokens on non-letters, and then lowercases
  - `StopAnalyzer`
    Same as `SimpleAnalyzer`, but also removes stop words
  - `StandardAnalyzer`
    Most sophisticated analyzer that knows about certain token types, lowercases, removes stop words, ...

Analysis examples

- "The quick brown fox jumped over the lazy dog"
- `WhitespaceAnalyzer`
  - [The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- `SimpleAnalyzer`
  - [the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- `StopAnalyzer`
  - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]
- StandardAnalyzer
  - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]

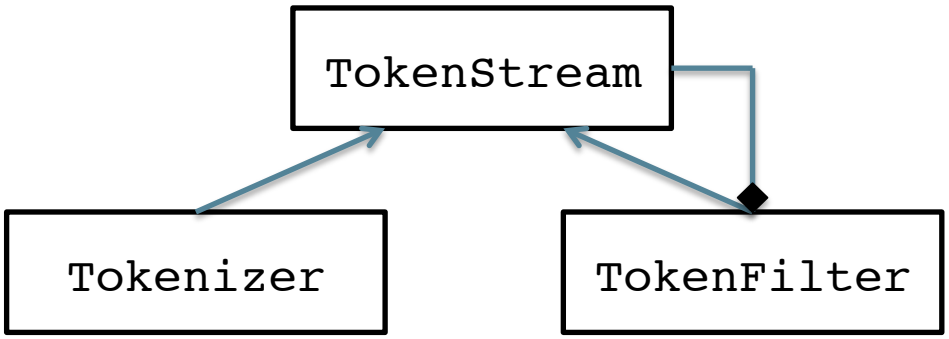
More analysis examples

- "XY&Z Corporation – xyz@example.com"
- `WhitespaceAnalyzer`
  - [XY&Z] [Corporation] [-] [xyz@example.com]
- `SimpleAnalyzer`
  - [xy] [z] [corporation] [xyz] [example] [com]
- `StopAnalyzer`
  - [xy] [z] [corporation] [xyz] [example] [com]
- StandardAnalyzer
  - [xy&z] [corporation] [xyz@example.com]

What's inside an `Analyzer`?

- `Analyzers` need to return a `TokenStream`

```
public TokenStream tokenStream(String fieldName,
                               Reader reader)
```



Tokenizers and TokenFilters

- Tokenizer
  - WhitespaceTokenizer
  - KeywordTokenizer
  - LetterTokenizer
  - StandardTokenizer
  - ...
- TokenFilter
  - LowerCaseFilter
  - StopFilter
  - PorterStemFilter
  - ASCIIFoldingFilter
  - StandardFilter
  - ...

IndexWriter construction

```
// Deprecated
IndexWriter(Directory d,
            Analyzer a,  // default analyzer
            IndexWriter.MaxFieldLength mfl);

// Preferred
IndexWriter(Directory d,
            IndexWriterConfig c);
```

Adding/deleting Documents to/from an IndexWriter

```
void addDocument(Document d);
void addDocument(Document d, Analyzer a);
```

Important: Need to ensure that Analyzers used at indexing time are consistent with Analyzers used at searching time

```
// deletes docs containing term or matching
// query.  The term version is useful for
// deleting one document.
void deleteDocuments(Term term);
void deleteDocuments(Query query);
```

Index format

- Each Lucene index consists of one or more segments
  - A segment is a standalone index for a subset of documents
  - All segments are searched
  - A segment is created whenever `IndexWriter` flushes adds/deletes
- Periodically, `IndexWriter` will merge a set of segments into a single segment
  - Policy specified by a `MergePolicy`
- You can explicitly invoke `optimize()` to merge segments

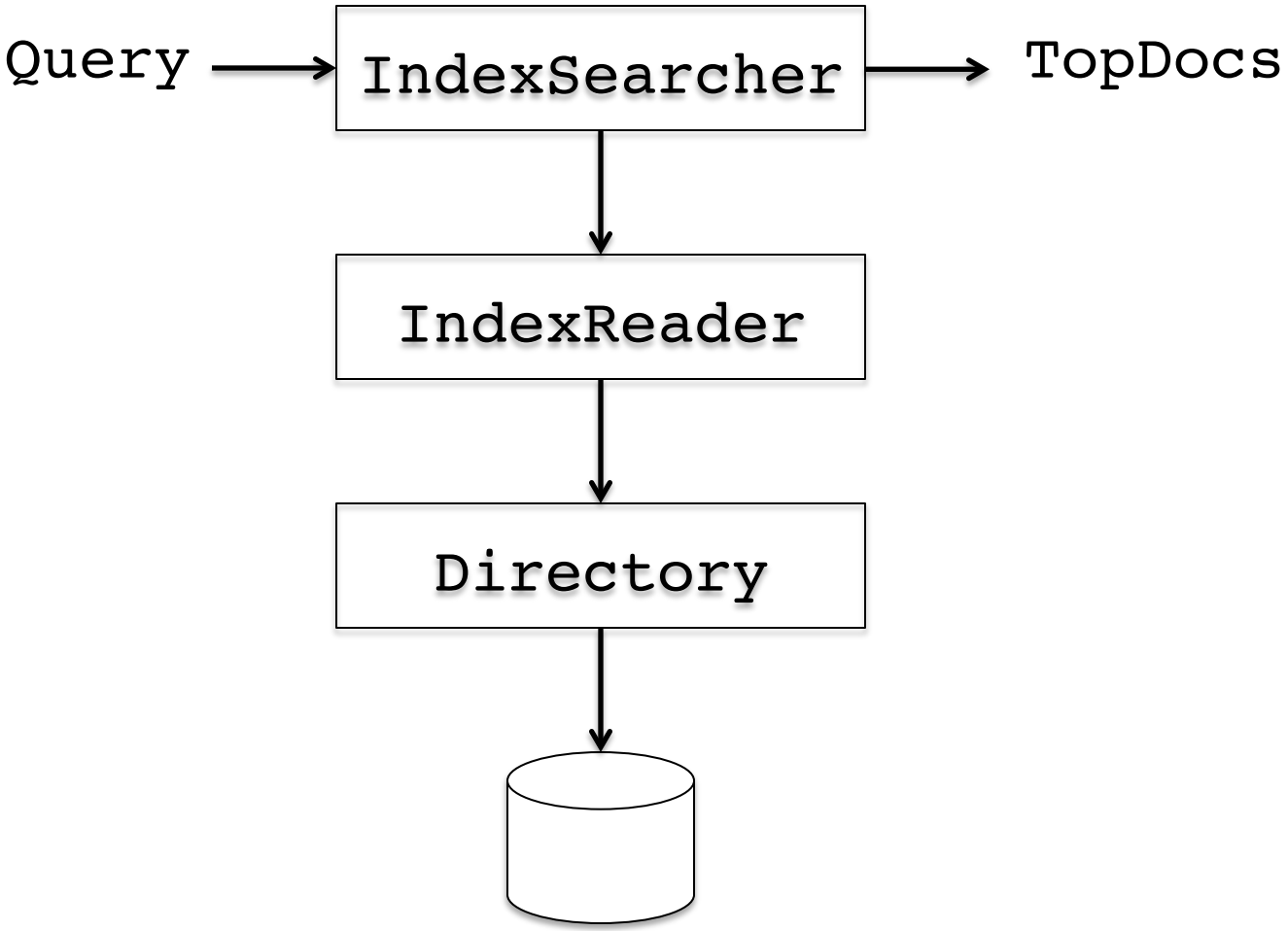
Basic merge policy

- Segments are grouped into levels
- Segments within a group are roughly equal size (in log space)
- Once a level has enough segments, they are merged into a segment at the next level up

IndexSearcher

- Constructor:
  - `IndexSearcher(Directory d);`
    - deprecated

IndexReader



`IndexSearcher`

- Constructor:
  - `IndexSearcher(Directory d);`
    - deprecated
  - `IndexSearcher(IndexReader r);`
    - Construct an `IndexReader` with static method `IndexReader.open(dir)`

Searching a changing index

```
Directory dir = FSDirectory.open(...);
IndexReader reader = IndexReader.open(dir);
IndexSearcher searcher = new IndexSearcher(reader);
```

Above `reader` does not reflect changes to the index unless you `reopen` it.

`Reopening` is more resource efficient than opening a new `IndexReader`.

```
IndexReader newReader = reader.reopen();
If (reader != newReader) {
   reader.close();
   reader = newReader;
   searcher = new IndexSearcher(reader);
}
```

Near-real-time search

```
IndexWriter writer = ...;
IndexReader reader = writer.getReader();
IndexSearcher searcher = new IndexSearcher(reader);
```

Now let us say there's a change to the index using `writer`

```
// reopen() and getReader() force writer to flush
IndexReader newReader = reader.reopen();
if (reader != newReader) {
   reader.close();
   reader = newReader;
   searcher = new IndexSearcher(reader);
}
```

IndexSearcher

- Methods
  - `TopDocs search(Query q, int n);`
  - `Document doc(int docID);`

QueryParser

- Constructor
  - `QueryParser(Version matchVersion, String defaultField, Analyzer analyzer);`
- Parsing methods
  - `Query parse(String query) throws ParseException;`
  - … and many more

`QueryParser` syntax examples

Query expression	Document matches if…
java	Contains the term *java* in the default field
java junit java OR junit	Contains the term *java* or *junit* or both in the default field (*the default operator can be changed to* AND)
+java +junit java AND junit	Contains both *java* and *junit* in the default field
title:ant	Contains the term *ant* in the title field
title:extreme –subject:sports	Contains *extreme* in the title and not *sports* in subject
(agile OR extreme) AND java	Boolean expression matches
title:”junit in action”	Phrase matches in title
title:”junit action”~5	Proximity matches (within 5) in title
java*	Wildcard matches
java~	Fuzzy matches
lastmodified:[1/1/09 TO 12/31/09]	Range matches

Construct Querys programmatically

- `TermQuery`
  - Constructed from a `Term`
- `TermRangeQuery`
- `NumericRangeQuery`
- `PrefixQuery`
- `BooleanQuery`
- `PhraseQuery`
- `WildcardQuery`
- `FuzzyQuery`
- `MatchAllDocsQuery`

`TopDocs` and `ScoreDoc`

- `TopDocs` methods
  - Number of documents that matched the search
    `totalHits`
  - Array of `ScoreDoc` instances containing results
    `scoreDocs`
  - Returns best score of all matches
    `getMaxScore()`
- `ScoreDoc` methods
  - Document id
    `doc`
  - Document score
    `score`

Scoring

- Scoring function uses basic tf-idf scoring with
  - Programmable boost values for certain fields in documents
  - Length normalization
  - Boosts for documents containing more of the query terms
- `IndexSearcher` provides an `explain()` method that explains the scoring of a document

Lucene: Changing the Ranking Function

- Setting the Java environment.
- Creating the index.
- Adding document to the index.
- Reading the index with IndexReader.
- Ranking documents with an IndexSearcher.
- Changing the similarity function used for ranking.

Linux Environment for Lucene (1)

- Include java in your PATH in your Linux account:
  - **PATH=$PATH:${path to jdk}/bin;export PATH**
- Lucene jar files are kept into multiple folders, you have to add the needed jar to your CLASSPATH:
  - **CLASSPATH=$CLASSPATH:.:${path to lucene}/core/***
  - **export CLASSPATH**

Linux Environment for Lucene (2)

- On California, JAVAPATH has been set by default.
  - Lucene jars are under **/usr/local/nlp/tools/lucene**
  - **CLASSPATH=$CLASSPATH:.:/usr/local/nlp/tools/lucene/core/*:/usr/local/nlp/tools/lucene/analysis/common/*:/usr/local/nlp/tools/lucene/queryparser/***
  - **export CLASSPATH**

- You can add the commands to the end of your login script:
  - **$HOME/.bashrc**
  - **$HOME/.cshrc**

Create an Index using Lucene

- Create a directory on disk to hold index:
  - **Directory indexDir = FSDirectory.open(new File(indexPath));**
- Configure an index writer:
  - **Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_45);**
  - **IndexWriterConfig iwc = new IndexWriterConfig(Version.LUCENE_45, analyzer);**
  - **iwc.setOpenMode(OpenMode.CREATE);**
- Create the index writer:
  - **writer = new IndexWriter(indexDir, iwc);**

Adding files to the index

```
Document doc = new Document();
Field nameField = new StringField("name", file.getName(),
                                  Field.Store.YES);
doc.add(nameField);
FileInputStream fis = new FileInputStream(file);
Field contentField = new TextField("content", new BufferedReader(new
                                  InputStreamReader(fis, "UTF-8")));
doc.add(contentField);
fis.close();
writer.addDocument(doc);   // create index for the given file
// repeat from top, for all documents in the collection
```

```
writer.close(); // commits all changes on disk.
```

Searching with Lucene

- Create an index reader and a searcher:
  - **IndexReader reader = DirectoryReader.open(FSDirectory.open(new File(indexPath)))**
  - **IndexSearcher searcher = new IndexSearcher(reader);**

- Use a one term query to search:
  - **Term term = new Term(field, word);**
  - **TermQuery tq = new TermQuery(term);**
  - **TopDocs topDocs = searcher.search(tq, top);**
    - **top** is number of highest ranking results to return.
    - since we use a one term query here we don't need analyzer:
      - using the same analyzer that used to create index.

Extracting the top documents

```
ScoreDoc[] scoreDocs = topDocs.scoreDocs;
for (int i = 0; i < scoreDocs.length; i++) {
        Document doc = searcher_.doc(scoreDocs[i].doc);
         System.out.println(doc.get("name"));
}
```

Changing the Scoring Method (1)

- **Similarity** is the base class that for the scoring method.
- **DefaultSimilarity** is the default scoring implementation:
  - it is a subclass of **TFIDFSimilarity**
    - which is a sublcass of **Similarity**.
- The scoring equation used by **DefaultSimilarity** is:
  - score(q, d) = coord(q, d) * queryNorm(q) *

    ∑(tf(t in d) * idf(t)2 * t.getBoost() * norm(t, d))
    - *coord* is a measure of how many query terms are in the document.
    - *queryNorm* is used to compare different queries.
    - *t.getBoost()* is boost set during search.
    - *norm()* is defined when the index is created.

Changing the Scoring Method (2)

- We can simply use **DefaultSimilarity** as template to create a new scoring method, here we modify the term frequency method:

```
public class WeirdSimilarity extends TFIDFSimilarity {
    public float tf(float freq) {
        // return (float)Math.sqrt(freq);  // original function
        return 1 / (Math.sqrt(freq) + 1);  // use inverse of tf as new tf
        // return freq < 1 ? 0 : 1 + (float)Math.log(freq); // use log scale
    }}
```

- Set similarity in **IndexSearcher** to use new similarity:

```
Similarity similarity = new WeirdSimilarity();
searcher.setSimilarity(similarity);
```

Lucene: Using Fields for Ranking

- QueryParser for querying with a single field.
- Create documents with multiple fields.
- MultiFieldQueryParser for querying with multiple fields.
- Lucene Scoring Formula (TFIDFSimilarity)
- Explainer
- Explanation of the scoring result
- Examples with different boost values

QueryParser for querying with a single field

- When using QueryParser, a query may have multiple terms:

```
/* q is the query */
String q = "This is the query."

/* create a QueryParser, tells the parser that search on the field "content" */
QueryParser parser = new QueryParser(Version.LUCENE_45, "contents",
                                     new StandardAnalyzer(Version.LUCENE_45));

/* create a query */
Query query = parser.parse(q);

/* search */
TopDocs topDocs = indexsearcher.search(query, 10);
```

Create documents with multiple fields

- Each document has 3 fields:
  - "content", "filename", and "fullpath".
  - we want to search a document based on not only its "content" but also on its "filename" field.

```
protected Document getDocument(File f) throws Exception {
  Document doc = new Document();
  doc.add(new TextField("contents", new FileReader(f)));
  doc.add(new TextField("filename", f.getName(), Field.Store.YES));
  doc.add(new StringField("fullpath", f.getCanonicalPath(), Field.Store.YES));
  return doc;
}
```

Replace QueryParser with MultiFieldQueryParser

```
/* tell what fields should be used for searching, and their boost values */
  String[] fields = new String[] {"contents", "filename"};
  HashMap<String, Float> boosts = new HashMap<String, Float>();
  boosts.put("contents", 1.1f);
  boosts.put("filename", 1.2f);

  /* define a MultiFieldQueryParser */
  MultiFieldQueryParser multiparser = new
    MultiFieldQueryParser(Version.LUCENE_45, fields,
                          new StandardAnalyzer(Version.LUCENE_45), boosts);

  /* parse a query */
  Query query = multiparser.parse(queryExpression);

  /* search */
  IndexSearcher searcher = new IndexSearcher(reader);
  TopDocs topDocs = indexsearcher.search(query, 10);
```

Lucene Scoring Formula (TFIDFSimilarity)

- The VSM scoring formula:
  - **Cosine-similarity(q, d) = V(q) * V(d) / (|V(q)| * |V(d)|)**
  - **V(q)** – is the tf-idf vector of the query.
  - **|V(q)|** - is the norm of V(q).

- Lucene conceptual formula:
  - **score(q,d) = coord-factor(q,d) * query-boost(q) * (V(q) * V(d) / |V(q)|) * doc-len-norm(d) * doc-boost(d)**
  - **coord-factor(q,d)** - is a score factor based on how many of the query terms are found in the specified document.
  - **doc-boost(d)** – is the boost value of the query.
  - **doc-len-norm(d)** – is a document length normalization factor.
  - **doc-boost(d)** – is the boost value of the document.

http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

Lucene Scoring Formula (TFIDFSimilarity)

- Lucene practical formula:
  - **score(q,d) = coord(q,d) * queryNorm(q) * $\sum_{t \in q}$ (tf(t in d) * idf(t)^2 * t.getBoost() · norm(t,d))**
  - **coord(q,d)** - is a score factor based on how many of the query terms are found in the specified document.
  - **queryNorm(q)** – is a normalizing factor used to make scores between queries comparable.
  - **tf(t in d)** – is the term frequency of term t in the document.
  - **idf(t)** – is the inverse document frequency of term t.
  - **t.getBoost()** – is the boost value of the query.
  - **norm(t,d)**– encapsulates a few (indexing time) boost and length factors.
- The details are on the official website:
  - http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

Explainer

- By using the Explainer, we can see how scores were calculated in detail:

```
/* search */
TopDocs topDocs = indexsearcher.search(query, 10);

/*
for (ScoreDoc match : topDocs.scoreDocs) {
  /* create an explainer */
  Explanation explanation = indexsearcher.explain(query, match.doc);
  /* output the detailed explanation of the scoring */
  System.out.println(explanation.toString());
}
```

Explanation of the scoring result

```
apache1.1.txt
1.145473 = (MATCH) sum of:
  0.14110807 = (MATCH) sum of:
    0.14110807 = (MATCH) weight(contents:software^1.1 in 1) [NewSimilarity], result of:
      0.14110807 = score(doc=1,freq=17.0 = termFreq=17.0
), product of:
        0.14085886 = queryWeight, product of:
          1.1 = boost
          0.9428416 = idf(docFreq=17, maxDocs=17)
          0.13581656 = queryNorm
        1.0017692 = fieldWeight in 1, product of:
          17.0 = tf(freq=17.0), with freq of:
            17.0 = termFreq=17.0
          0.9428416 = idf(docFreq=17, maxDocs=17)
          0.0625 = fieldNorm(doc=1)
  1.004365 = (MATCH) sum of:
    1.004365 = (MATCH) weight(filename:apache1.1^1.2 in 1) [NewSimilarity], result of:
      1.004365 = score(doc=1,freq=1.0 = termFreq=1.0
), product of:
        0.51176757 = queryWeight, product of:
          1.2 = boost
          3.1400661 = idf(docFreq=1, maxDocs=17)
          0.13581656 = queryNorm
        1.9625413 = fieldWeight in 1, product of:
          1.0 = tf(freq=1.0), with freq of:
            1.0 = termFreq=1.0
          3.1400661 = idf(docFreq=1, maxDocs=17)
          0.625 = fieldNorm(doc=1)
```

field:term

boost

Example with different boost values

- The query: “Apache software program is not a mozilla application”.
- **Set the boost to:**
  boosts.put("contents", 1.0f);
  boosts.put("filename", 1.0f);
- The result was:
  D:\Users\xin\workspace2\LuceneTest\data\mozilla firefox.txt
  D:\Users\xin\workspace2\LuceneTest\data\apache1.0.txt
  D:\Users\xin\workspace2\LuceneTest\data\apache1.1.txt
  D:\Users\xin\workspace2\LuceneTest\data\gpl2.0.txt
  D:\Users\xin\workspace2\LuceneTest\data\lpgl2.0.txt
  D:\Users\xin\workspace2\LuceneTest\data\mozilla_eula_firefox3.txt
  D:\Users\xin\workspace2\LuceneTest\data\mozilla_eula_thunderbird2.txt
  D:\Users\xin\workspace2\LuceneTest\data\gpl1.0.txt
  D:\Users\xin\workspace2\LuceneTest\data\lgpl2.1.txt
  D:\Users\xin\workspace2\LuceneTest\data\mozilla1.1.txt

Example with different boost values

- The query: “Apache software program is not a mozilla application”.
- **Set the boost to:**
  boosts.put("contents", 10.0f);
  boosts.put("filename", 1.0f);
- The result was:

  D:\Users\xin\workspace2\LuceneTest\data\apache1.0.txt
  D:\Users\xin\workspace2\LuceneTest\data\apache1.1.txt
  D:\Users\xin\workspace2\LuceneTest\data\mozilla firefox.txt
  D:\Users\xin\workspace2\LuceneTest\data\gpl2.0.txt
  D:\Users\xin\workspace2\LuceneTest\data\lpgl2.0.txt
  D:\Users\xin\workspace2\LuceneTest\data\mozilla_eula_firefox3.txt
  D:\Users\xin\workspace2\LuceneTest\data\mozilla_eula_thunderbird2.txt
  D:\Users\xin\workspace2\LuceneTest\data\gpl1.0.txt
  D:\Users\xin\workspace2\LuceneTest\data\lgpl2.1.txt
  D:\Users\xin\workspace2\LuceneTest\data\mozilla1.1.txt