

Lucene Tutorial

borrowing from: Chris Manning and Pandu Nayak

Open source IR systems

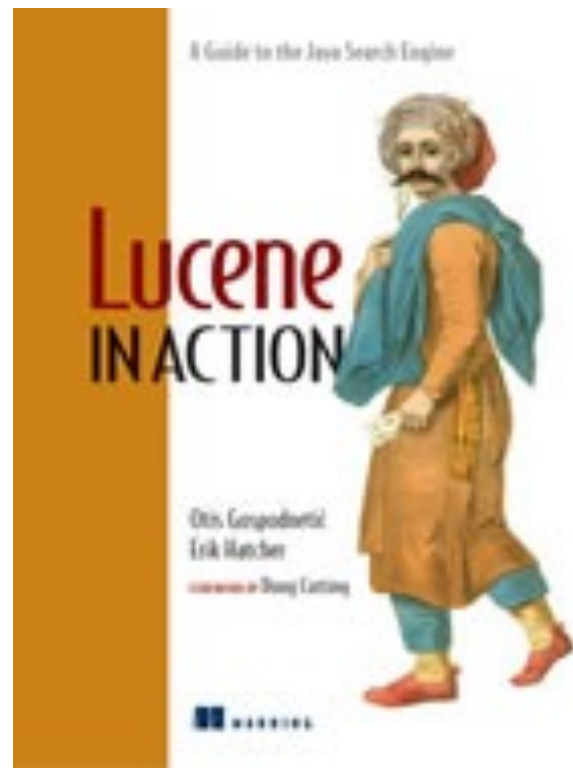
- Widely used academic systems
 - Terrier (Java, U. Glasgow) <http://terrier.org>
 - Indri/Galago/Lemur (C++ (& Java), U. Mass & CMU)
 - Tail of others (Zettair, ...)
- Widely used non-academic open source systems
 - **Lucene**
 - Things built on it: Solr, ElasticSearch
 - A few others (Xapian, ...)

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: Twitter, LinkedIn, Zappos, CiteSeer, Eclipse, ...
 - ... and many more (see <http://wiki.apache.org/lucene-java/PoweredBy>)
- Ports/integrations to other languages
 - C/C++, C#, Ruby, Perl, Python, PHP, ...

Based on “Lucene in Action”

By Michael McCandless, Erik Hatcher, Otis Gospodnetic

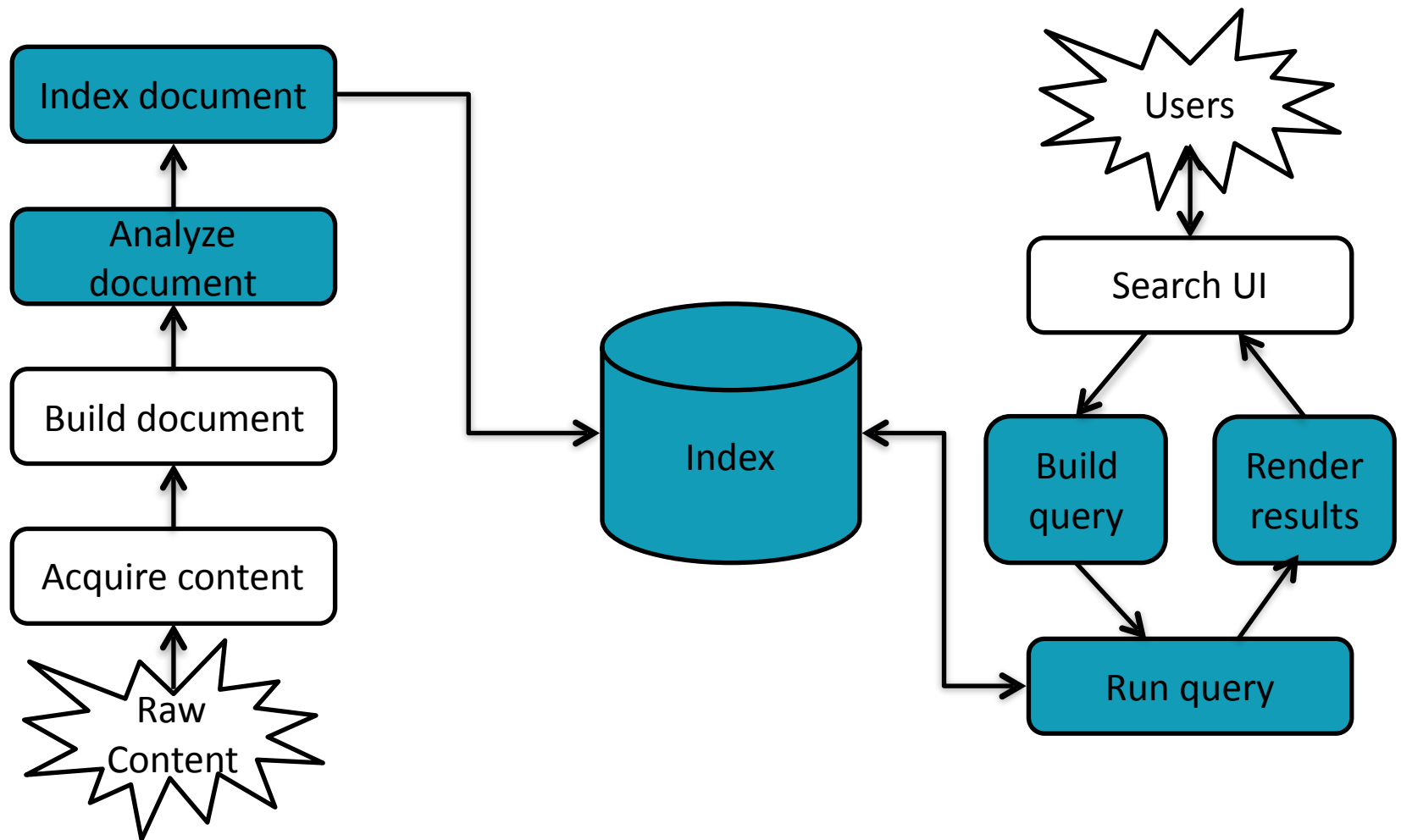


Covers Lucene 3. It's now up to 6

Resources

- Lucene: <http://lucene.apache.org>
- 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 (downloadable from Lucene in Action website)

Lucene in a search system



Lucene demos

- Source files in `lia2e/src/lia/meetlucene/`
 - Actual sources use Lucene 3
 - Code in these slides upgraded to Lucene 6
- Command line **Indexer**
 - `lia.meetlucene.Indexer`
- Command line **Searcher**
 - `lia.meetlucene.Searcher`

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
 - Built on an `IndexWriterConfig` and a `Directory`
- `Directory`
 - Abstract class that represents the location of an index
- `Analyzer`
 - Extracts tokens from a text stream

Creating an IndexWriter

```
import org.apache.lucene.analysis.Analyzer;  
import org.apache.lucene.index.IndexWriter;  
import org.apache.lucene.index.IndexWriterConfig;  
import org.apache.lucene.store.Directory;  
...  
  
private IndexWriter writer;  
  
public Indexer(String dir) throws IOException {  
    Directory indexDir = FSDirectory.open(new File(dir));  
    Analyzer analyzer = new StandardAnalyzer();  
    IndexWriterConfig cfg = new IndexWriterConfig(analyzer);  
    cfg.setOpenMode(OpenMode.CREATE);  
    writer = new IndexWriter(indexDir, cfg);  
}
```

Core indexing classes (contd.)

- `Document`
 - Represents a collection of named `Fields`. Text in these `Fields` are indexed.
- `Field`
 - `StringFields` are indexed but not tokenized
 - `TextFields` are indexed and tokenized

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 TextField("contents", new FileReader(f)))  
    doc.add(new StringField("filename",  
                            f.getName(),  
                            Field.Store.YES));  
    doc.add(new StringField("fullpath",  
                            f.getCanonicalPath(),  
                            Field.Store.YES));  
    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();  
}
```

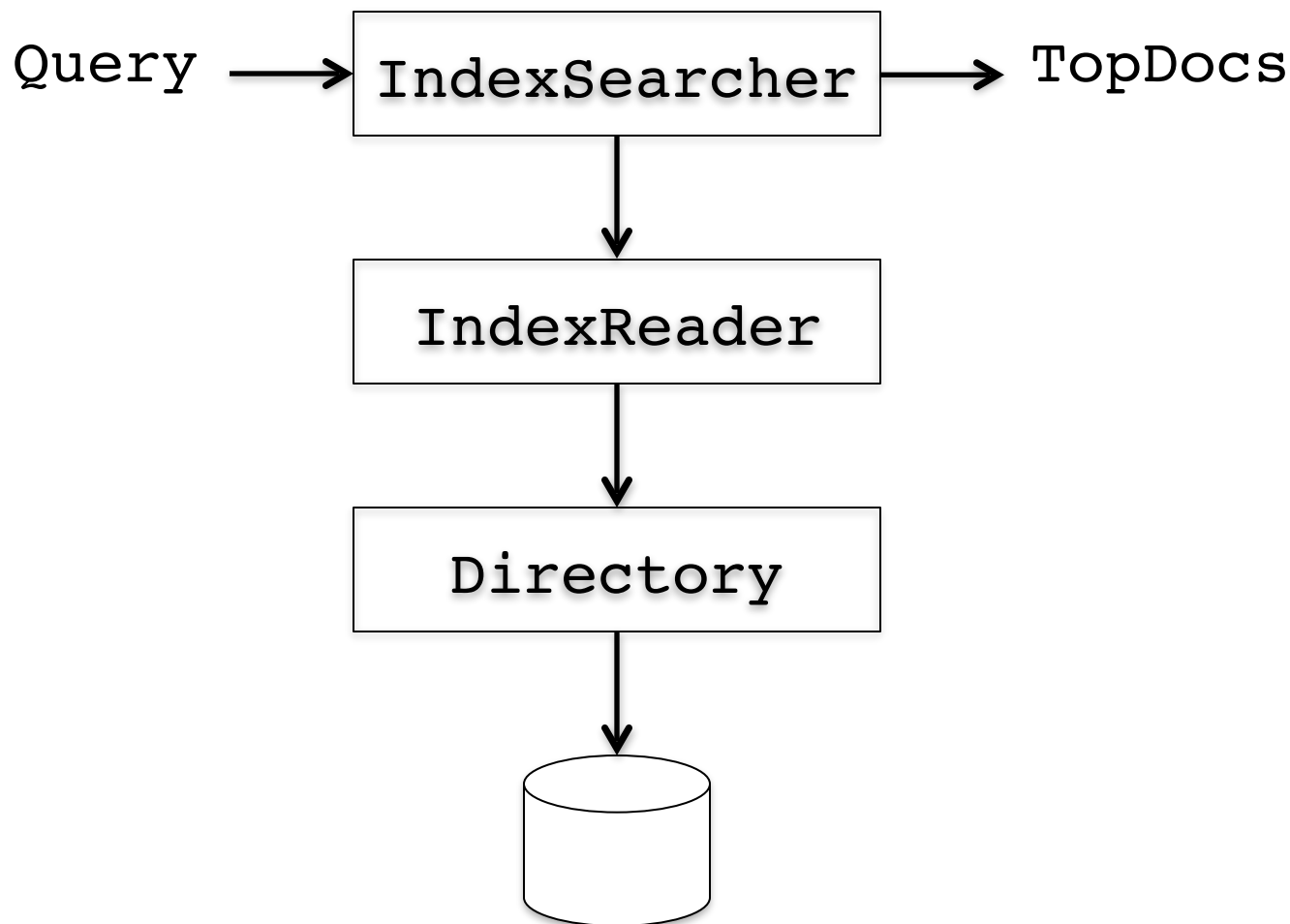
The Index

- The Index is the kind of inverted index we know and love
- default Lucene codec includes term frequencies and positional information
- APIs to customize the codec

Core searching classes

- **IndexSearcher**
 - Central class that exposes several search methods on an index
 - Accessed via an `IndexReader`
- **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

IndexSearcher



Creating an IndexSearcher

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

Query and QueryParser

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

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 in the index or not
 - Useful for fields that you'd like to display to users

Field construction

Lots of different constructors

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

```
Field(String name,
      String value,
      FieldType type);
```

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

FieldType specifies field properties.

Can also directly use sub-classes like TextField, StringField, ...

Using `Field` properties

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

Analyzer

- 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 example

- “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]

Another analysis example

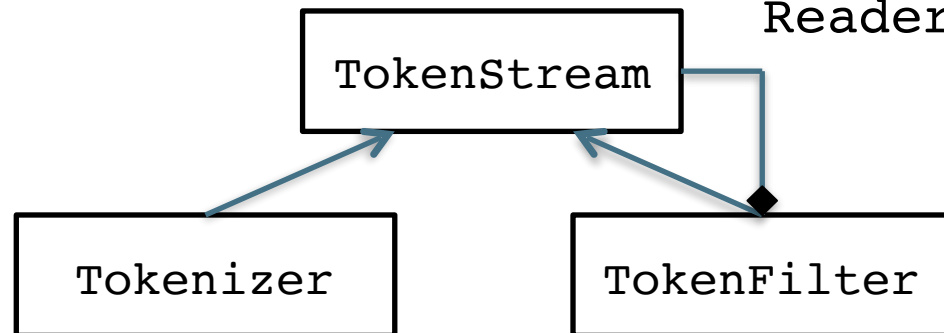
- “XY&Z Corporation – xyz@example.com”
- WhitespaceAnalyzer
 - [XY&Z] [Corporation] [-] [xyz@example.com]
- SimpleAnalyzer
 - [xy] [z] [corporation] [xyz] [example] [com]
- StopAnalyzer
 - [xy] [z] [corporation] [xyz] [example] [com]
- StandardAnalyzer
 - [xy&z] [corporation] [xyz@example.com]

What's inside an Analyzer?

- Analyzers need to return a `TokenStream`
- this is an abstract class, with subclasses

`Tokenizer` and `TokenFilter`

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



`Tokenizer` input is a `Reader`, and `TokenFilter` input is a `TokenStream`

Tokenizers and TokenFilters

- `Tokenizer`

- `WhitespaceTokenizer`
- `KeywordTokenizer`
- `LetterTokenizer`
- `StandardTokenizer`
- ...

- `TokenFilter`

- `LowerCaseFilter`
- `StopFilter`
- `PorterStemFilter`
- `ASCIIFoldingFilter`
- `StandardFilter`
- ...

Adding/deleting Documents to/from an IndexWriter

```
long addDocument(Iterable<IndexableField> d);
```

IndexWriter's Analyzer is used to analyze document.

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

```
// deletes docs containing terms or matching  
// queries.
```

```
long deleteDocuments(Term... terms);
```

```
long deleteDocuments(Query... queries);
```

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 `forceMerge()` to merge segments

Basic merge policy

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

Searching a changing index

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

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

```
DirectoryReader newReader =  
    DirectoryReader.openIfChanged(reader);  
If (newReader != null) {  
    reader.close();  
    reader = newReader;  
    searcher = new IndexSearcher(reader);  
}
```

Near-real-time search

This will update whether the writer's changes have been written yet or not.

```
IndexWriter writer = ...;
DirectoryReader reader =
    DirectoryReader.open(writer, true);
IndexSearcher searcher = new IndexSearcher(reader);

// Now let us say there's a change to the index using writer
writer.addDocument(newDoc);

DirectoryReader newReader =
    DirectoryReader.openIfChanged(reader, writer, true);
if (newReader != null) {
    reader.close();
    reader = newReader;
    searcher = new IndexSearcher(reader);
}
```

QueryParser

- Constructor
 - `StandardQueryParser(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 <i>java</i> in the default field
java junit java OR junit	Contains the term <i>java</i> or <i>junit</i> or both in the default field (<i>the default operator can be changed to AND</i>)
+java +junit java AND junit	Contains both <i>java</i> and <i>junit</i> in the default field
title:ant	Contains the term <i>ant</i> in the title field
title:extreme –subject:sports	Contains <i>extreme</i> in the title and not <i>sports</i> 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 for single terms
java~	Fuzzy matches for single terms
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

IndexSearcher

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

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

- Original 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 5.0+ Scoring

- As well as traditional tf.idf vector space model, Lucene has:
 - BM25
 - drf (divergence from randomness)
 - ib (information (theory)-based similarity)

```
indexSearcher.setSimilarity(  
    new BM25Similarity());  
BM25Similarity custom =  
    new BM25Similarity(1.2, 0.75); // k1, b  
indexSearcher.setSimilarity(custom);
```