Introduction to Information Retrieval

ΜΕ003-ΠΛΕ70: Information Retrieval

Introduction to Lucene

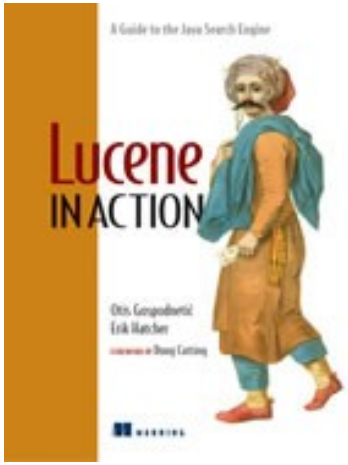
Info

- *https://lucene.apache.org/*
- Open source Java Library for IR (Indexing & Searching)
  - ➔ Written by Doug Cutting
  - ➔ Used by LinkedIn, Twitter Trends, Netflix

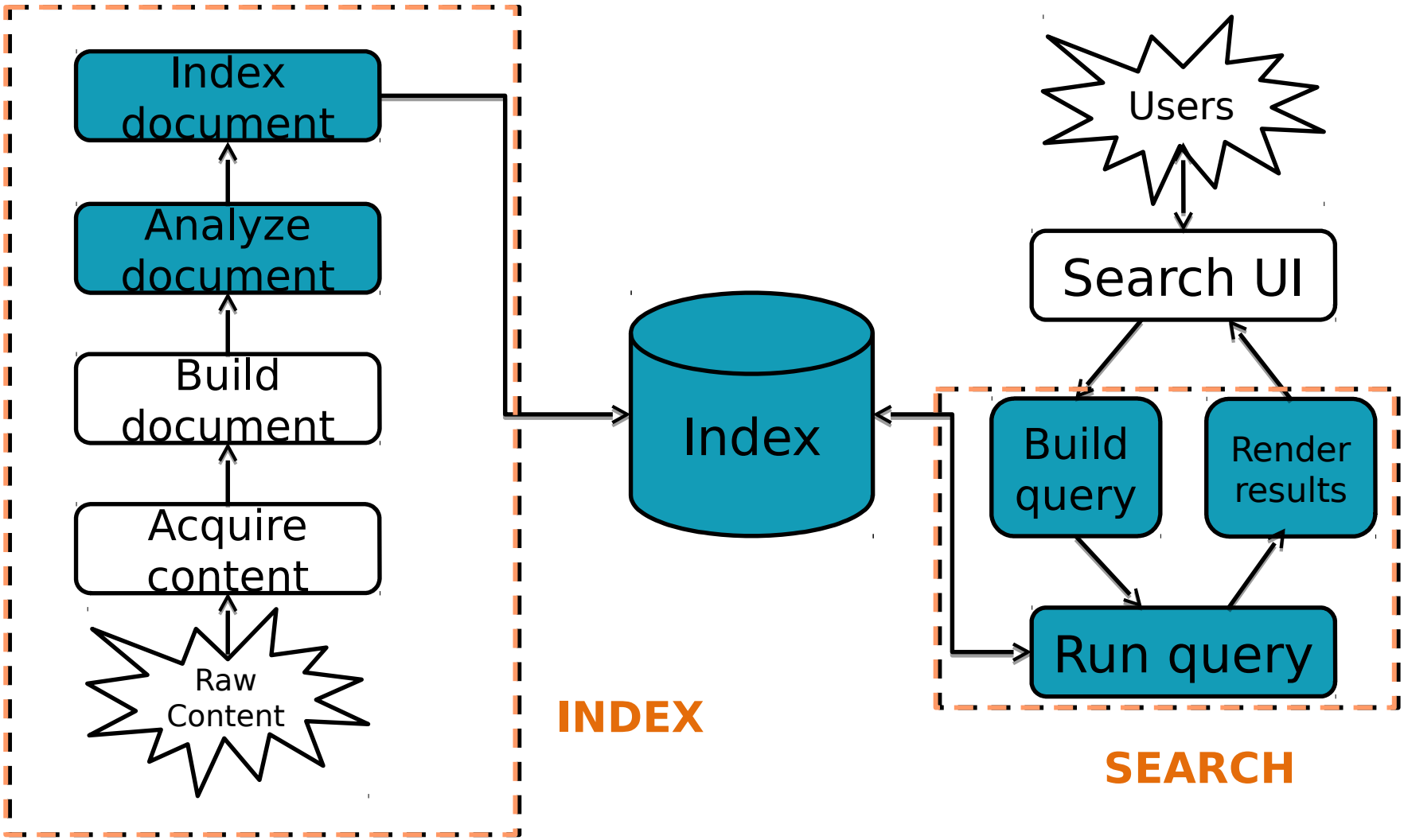
    see *http://wiki.apache.org/lucene-java/PoweredBy*
  - ➔ Ported to other programming languages
    - Python (*http://lucene.apache.org/pylucene/index.html*) C/C++, C#, Ruby, Perl, PHP, ...

Sources

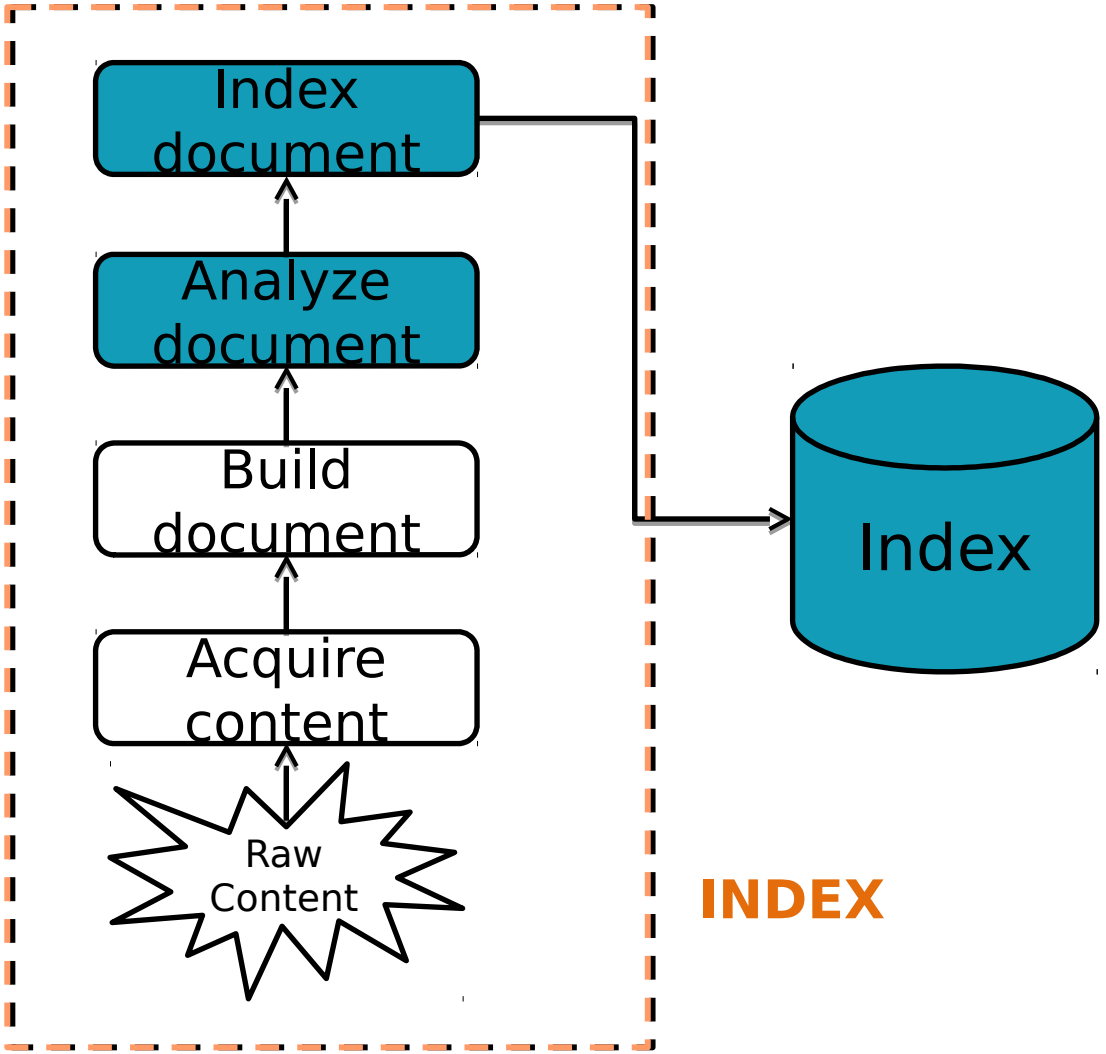
- Lucene: *http://lucene.apache.org/core/*
- Lucene in Action: *http://www.manning.com/hatcher3/*
  - Code samples available for download (*very usefull)*
- *Documentation:*
  - *https://lucene.apache.org/core/6_4_2/index.html*



Lucene in a search system



Lucene in a search system: Index



Steps

1. Acquire content
2. Build content
3. Analyze documents
4. Index documents

Lucene in a search system: Index(1)

- **Acquire content**
  - Depending on type
    - Crawler or spiders (web)
    - Specific APIs provided by the application (e.g., Twitter, FourSquare)
    - Complex software if scattered at various location, etc
  - Complex documents (e.g., XML, relational databases, JSON etc) Using Solr
    - *https://lucene.apache.org/solr/*

Lucene in a search system: Index(2)

- **Build document**
  - A document is the unit of search
  - Each document consists of separately named fields with values (title, body, etc)

What constitutes a document and what are its fields?

Lucene provides an API for building fields and documents

Lucene in a search system: Index(3)

- **Analyse Document**
  - Given a document -> extract its tokens
- Issues
  - handle compounds
  - case sensitivity
  - inject synonyms
  - spell correction
  - collapse singular and plural
  - stemmer (Porter's)

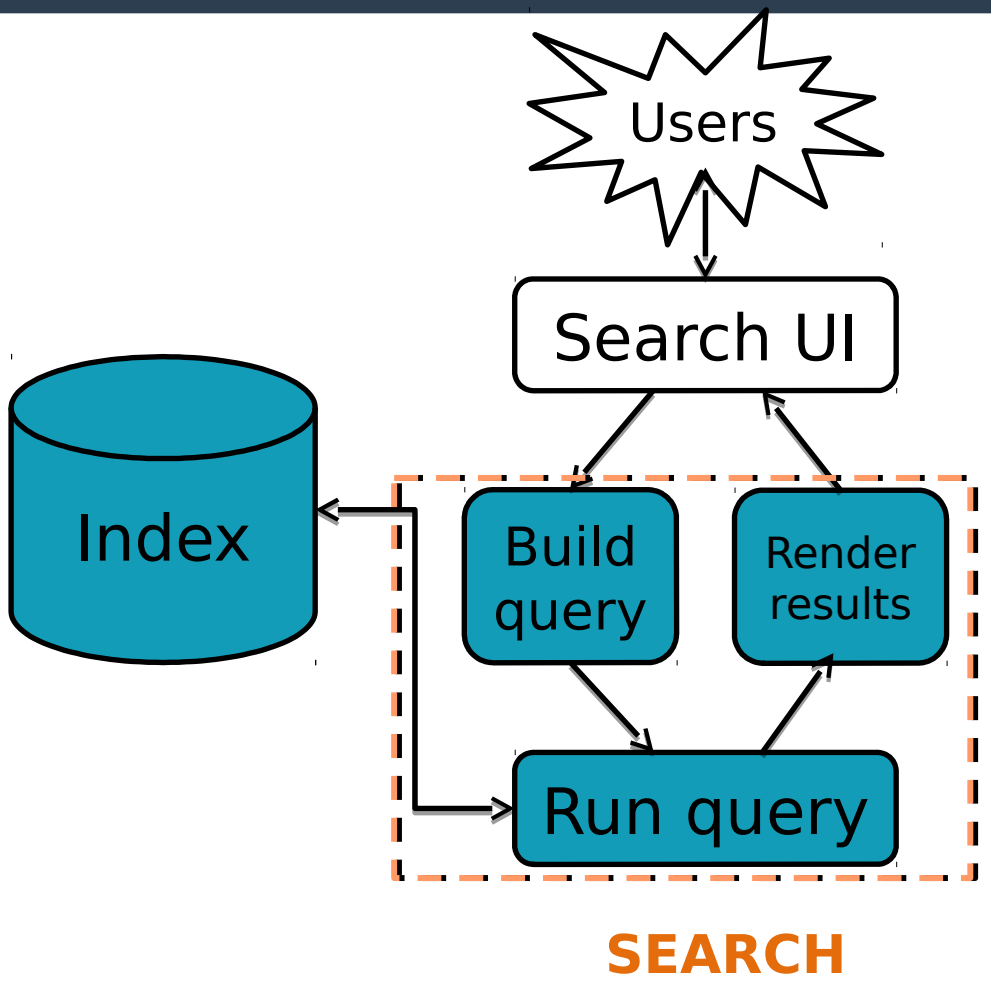
Lucene in a search system: Index(4)

- **Index Document**
  - Details in Chapter 2

Lucene in a search system: Search

STEPS

1.Enter query (UI)
2.Build query
3.Run search query
4.Render results (UI)



Lucene in a search system: Search(2)

- **Search User Interface(UI)**

No default search UI, but many useful *contrib* modules

General instructions

- Simple (do not present a lot of options in the first page)

  a single search box better than 2-step process
- Result presentation is important
  - highlight matches (*highlighter contrib* modules, section 8.3&8.4)
  - make sort order clear, etc
- Be transparent: e.g., explain if you expand search for synonyms, autocorrect errors (*spellchecker contrib* module, section 8.5 , etc)

Lucene in a search system: Search(3)

- **Build Query**

Provides a package *QueryParser*: process the user text input into a *Query* object (Chapter 3)

Query may contain Boolean operators, phrase queries, wildcard terms

Lucene in a search system: Search(4)

- **Search Query**

See Chapter 6

Three models
- Pure Boolean model (no sort)
- Vector space model
- Probabilistic model

Lucene combines Boolean and vector model – select which one on a search-by-search basis

Lucene in a search system: Search(5)

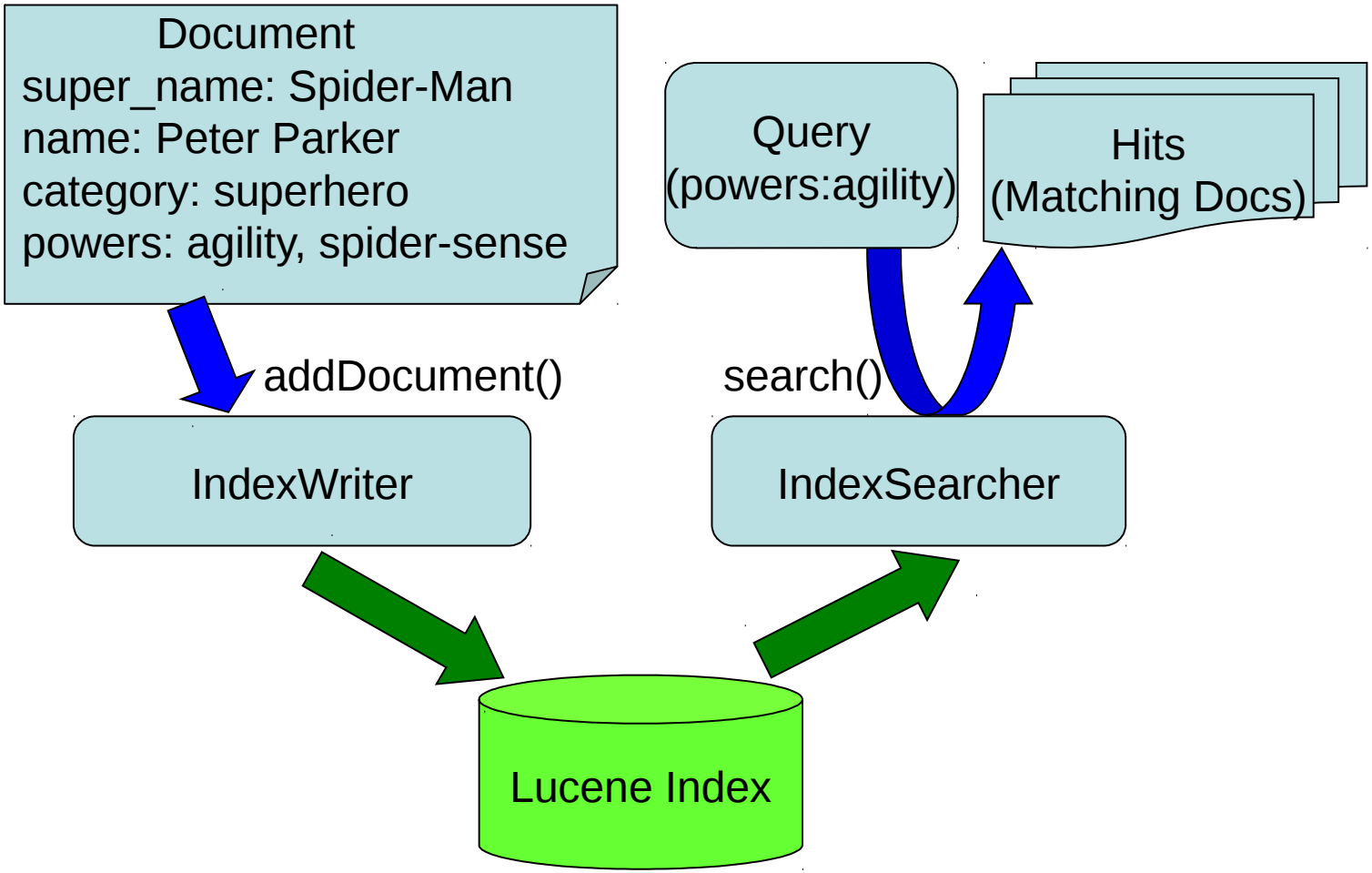
- **Render Results**

UI issues

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
  - Examples: Title, author, date, abstract, body, URL, keywords, ..
  - Different documents can have different fields
- You have to translate raw content into Fields
- Search a field using name:term, e.g., title:lucene

Basic Application



1. Get Lucene jar file
2. Write indexing code to get data and create Document objects
3. Write code to create query objects
4. Write code to use/display results

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

Example code (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(1)

- 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

Document and Fields Example

```
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 example

```
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();
}
```

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

Fields(1)

```
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[]`

Fields(2)

- **`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

Fields(3)

- `TermVector.Yes`
- `TermVector.With_POSITIONS`
- `TermVector.With_OFFSETS`
- `TermVector.WITH_POSITIONS_OFFSETS`
- `TermVector.No`

Multivalued 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));
...
```

Analysers

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, ...

Analysers 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]

Add - Delete Documents

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

// 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 Core Searching Classes

- IndexSearcher
  - Central class that exposes several search methods on an index

  (a class that “opens” the index) requires a Directory instance that holds the previously created index
- Term
  - Basic unit of searching, contains a pair of string elements (field and word)
- Query
  - Abstract query class. Concrete subclasses represent specific types of queries, e.g., matching terms in fields, boolean queries, phrase queries, …, most basic *TermQuery*
- QueryParser
  - Parses a textual representation of a query into a **`Query`** instance

Example 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);
  ...
}
```

Example Query/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);
   ...
}
```

Basic Core Searching Classes(2)

- TopDocs
  - Contains references to the top N documents returned by a search (the docID and its score)
- ScoreDoc
  - Provides access to a single search result

TopDocs & ScoreDocs

- 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`

TopDocs & ScoreDocs(1)

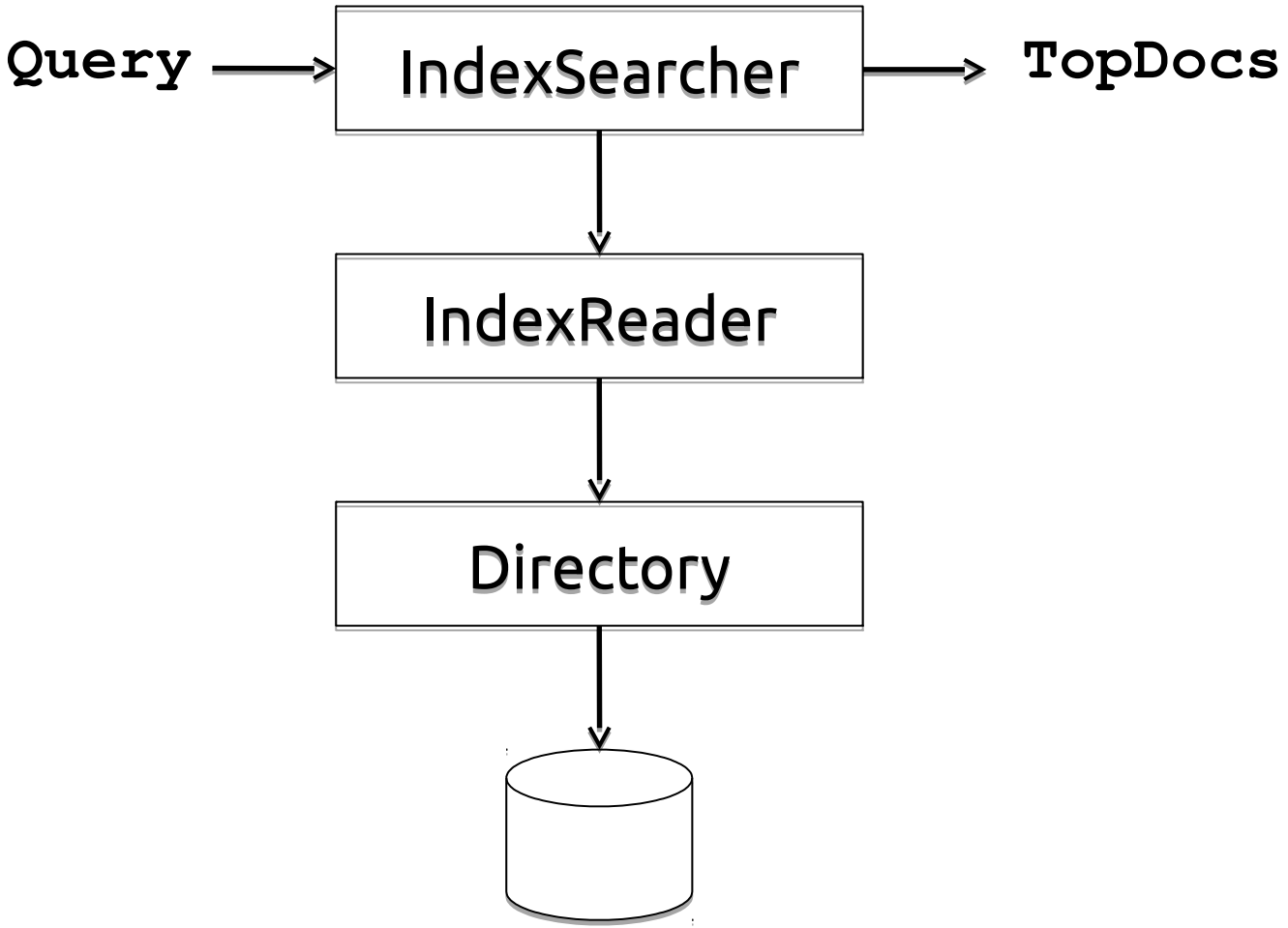
- Scoring function uses basic tf-idf scoring with
  - Programmable boost values for certain fields in documents
  - Length normalization
  - Boosts for documents containing more of the query terms

- IndexSearcher provides an explain() method that explains the scoring of a document

Example 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);
}
```

IndexReader

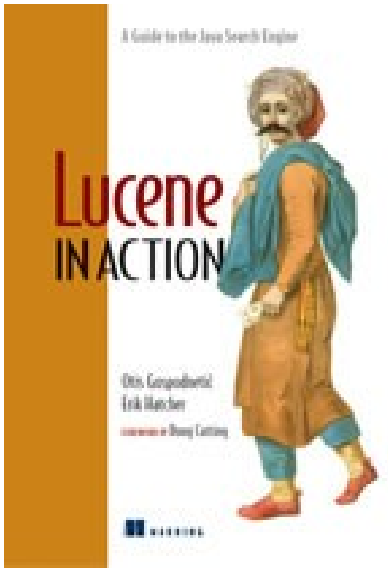


Sources

- Lucene can be downloaded from
  - *http://www.apache.org/dyn/closer.lua/lucene/java/6.4.2*

- Solr can be downloaded from
  - *https://lucene.apache.org/solr/*

- By Michael McCandless, Erik Hatcher, Otis Gospodnetic



Questions??