Lucene Tutorial

Based on

Lucene in Action

Michael McCandless, Erik Hatcher, Otis Gospodnetic

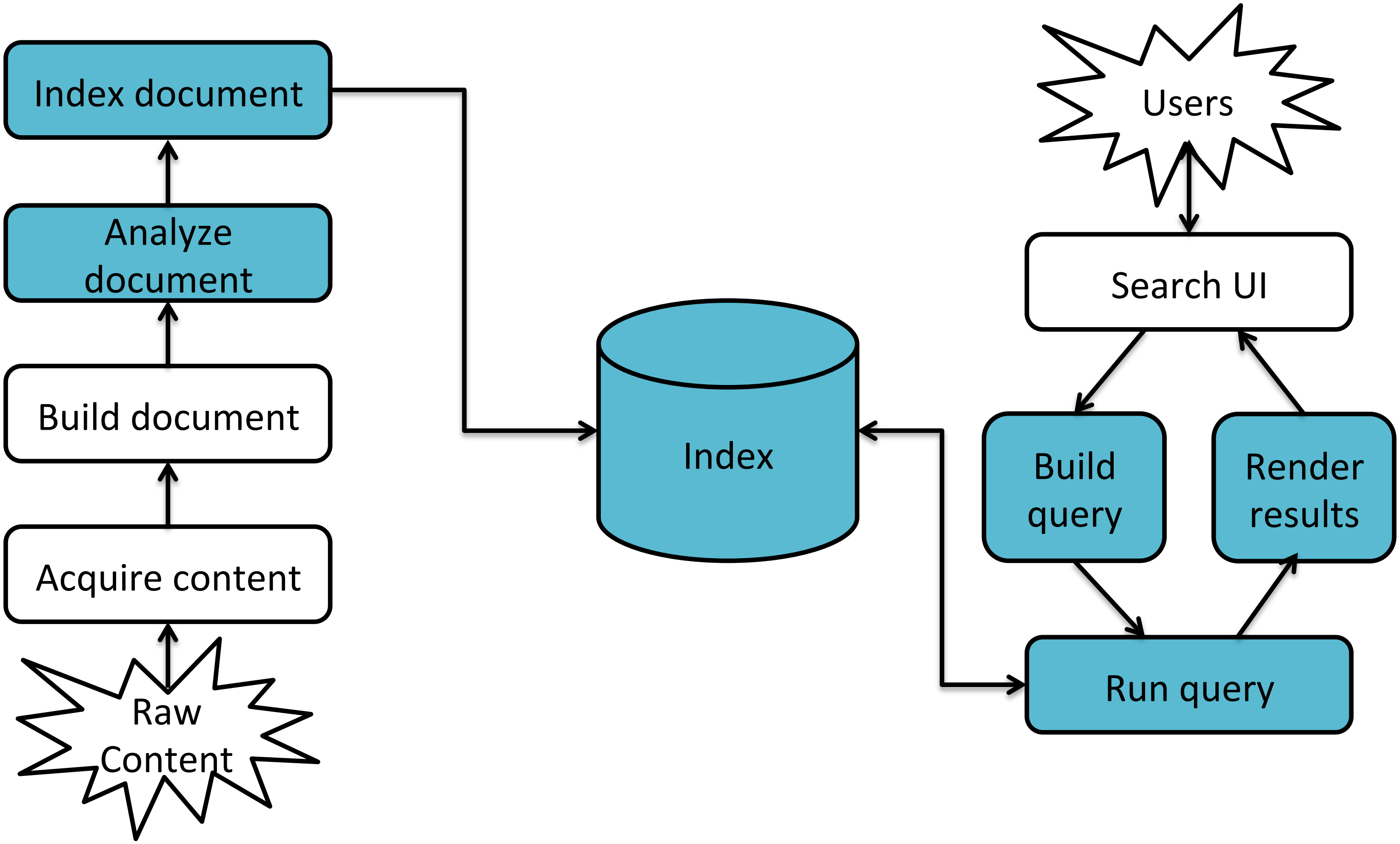
Lucene

- 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`

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);
}
```

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();
}
```

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);
   ...
}
```

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();
}
```

Core indexing classes

- `IndexWriter`
- `Directory`
- `Analyzer`
- `Document`
- `Field`

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 an inverted 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

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]

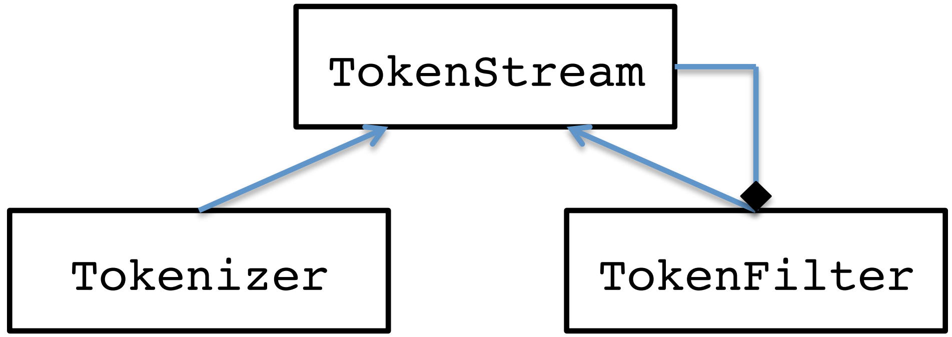
More analysis examples

- “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`

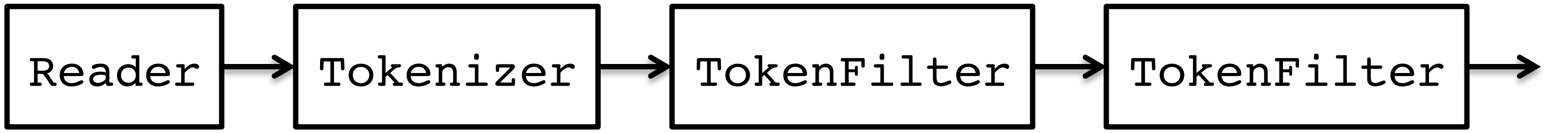
```
public TokenStream tokenStream(String fieldName,
                               Reader reader)
```



TokenStream

Tokenizer

TokenFilter



Reader

Tokenizer

TokenFilter

TokenFilter

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

Core searching classes

- `IndexSearcher`
- `Query`
  - And sub-classes
- `QueryParser`
- `TopDocs`
- `ScoreDoc`

`IndexSearcher`

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

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);
}
```

Scoring

- Scoring function uses basic tf x 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