Introduction to

Information Retrieval

ΜΕ003-ΠΛΕ70: Ανάκτηση Πληροφορίας

Διδάσκουσα: Ευαγγελία Πιτουρά

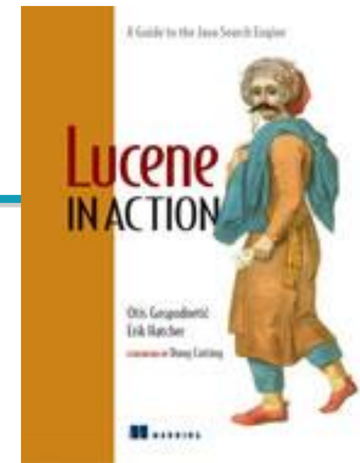
Εισαγωγή στο Lucene.

Τι είναι;

- *Open source Java library* for IR (indexing and searching) http://lucene.apache.org/
  - Lets you add search to your application, not a complete search system by itself -- *software library not an application*
  - Written by Doug Cutting
- Used by LinkedIn, Twitter Trends, Netflix …

  and many more (see http://wiki.apache.org/lucene-java/PoweredBy)
- Ports/integrations to other languages
  - Python (http://lucene.apache.org/pylucene/index.html) C/C++, C#, Ruby, Perl, PHP, …
- Beyond core jar, a number of extension modules
  - *contrib* modules

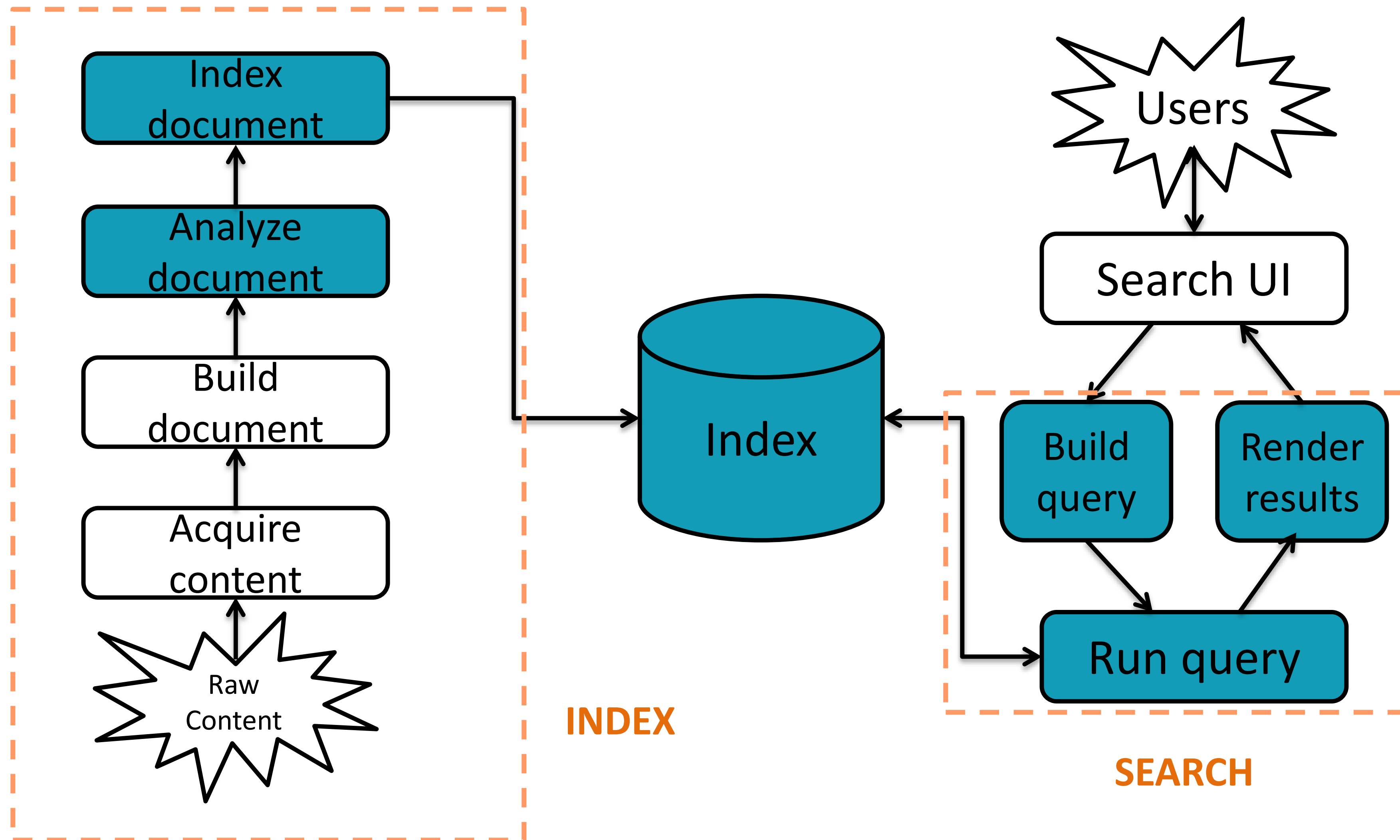
Πηγές



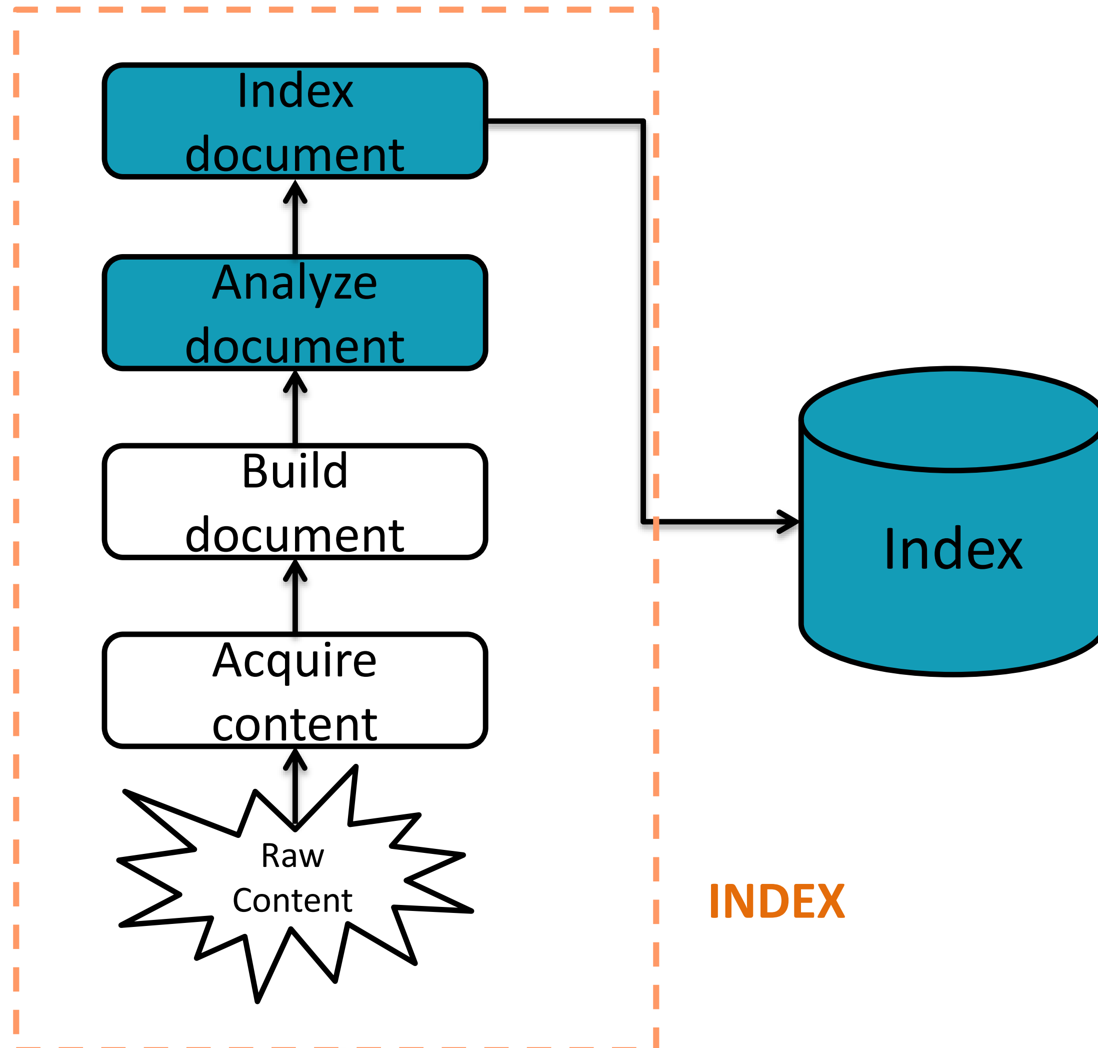
- Lucene: http://lucene.apache.org/core/
- Lucene in Action: http://www.manning.com/hatcher3/
  - Code samples available for download

    πολύ χρήσιμο
  - JUnit: http://junit.org/
    - Some examples are JUnit test cases
    - Automatically executes all methods with *public void test-XXX()* signature

Lucene in a search system



Lucene in a search system: index



INDEX

Steps

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

Lucene in a search system: index

Acquire content (not supported by core Lucid)

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

Additional issues

- Access Control Lists
- Online/real-time

Complex documents (e.g., XML, relational databases, JSON etc)

Solr (Tika, chapter 7)

Lucene in a search system: index

Build document (not supported by core Lucid)

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

Other issues (not handled)

- Extract text from document (if binary)
- Handle markups (XML, HTML)
- Add additional fields (semantic analysis)
- Boost individual files
  - At indexing time (per document and field, section 2.5)
  - At query time (section 5.7)

Lucene in a search system: index

Analyze document (supported by core Lucid)

Given a document -> extract its tokens

Details in Chapter 4

Issues

- handle compounds
- case sensitivity
- inject synonyms
- spell correction
- collapse singular and plural
- stemmer (Porter’s)

Lucene in a search system: index

Index document (supported by core Lucid)

Details in Chapter 2

Lucene in a search system: search

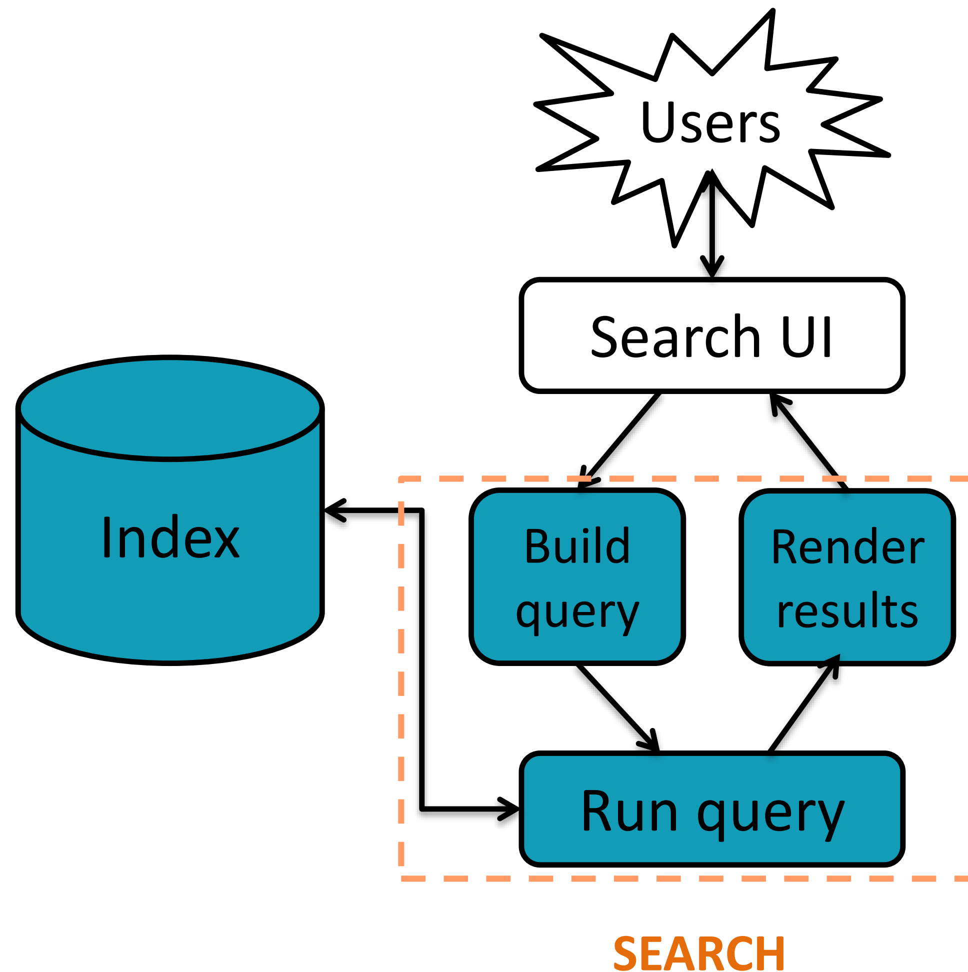
STEPS

Enter query (UI)

Build query

Run search query

Render results (UI)



Lucene in a search system: search

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

Build query (supported by core Lucid)

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

Search query (supported by core Lucid)

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

Customize

Lucene in a search system: search

Render results (supported by core Lucid)

UI issues

Lucene in action

Get code from the book

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

How Lucene models content

- A `Document` is the atomic unit of indexing and searching
  - A `Document` contains `Fields`
- `Field`s 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

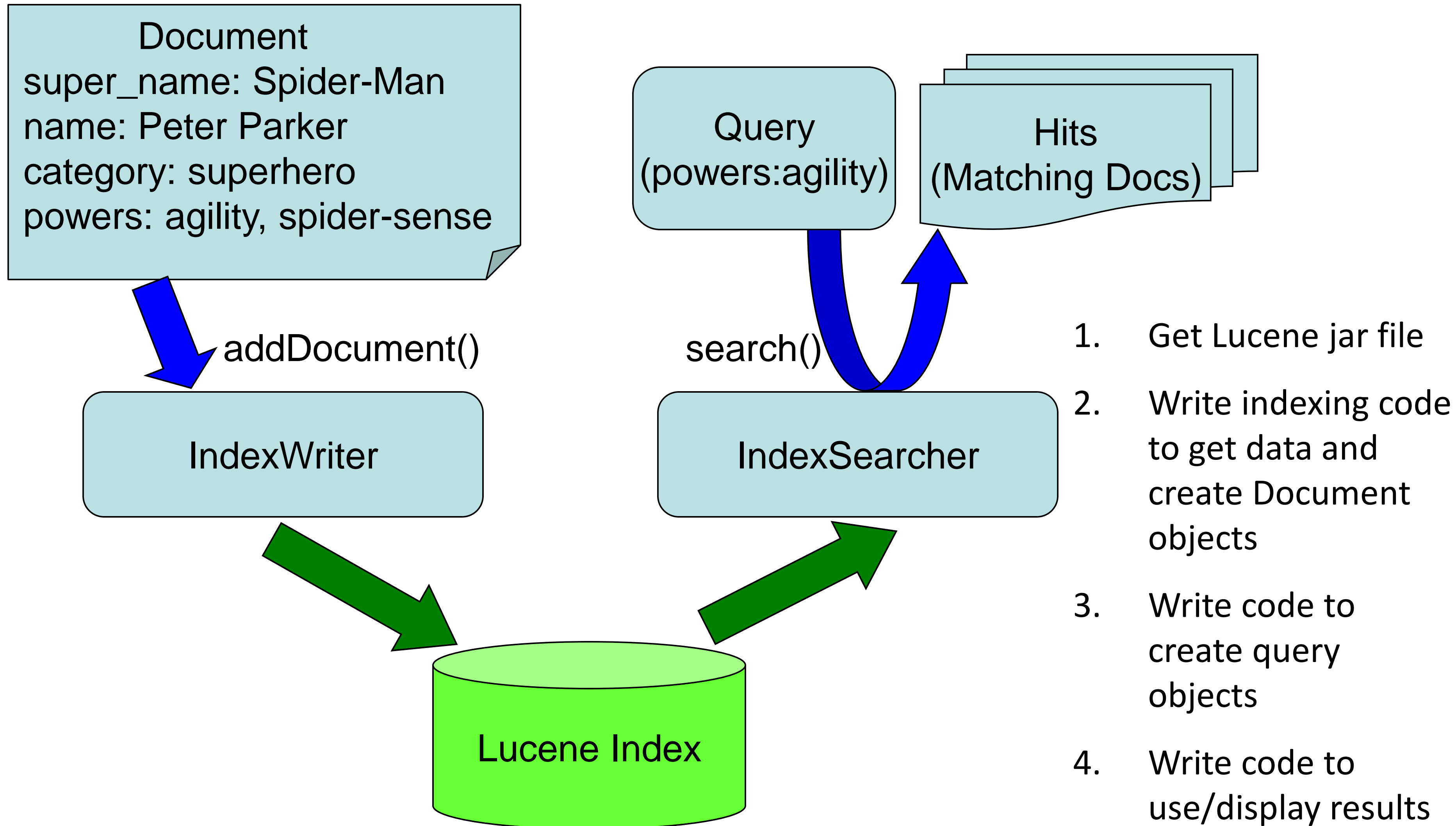
Documents and Fields

Parametric or zone indexing

There is one (parametric) index for each field

Also, supports *weighted* field scoring

Basic Application



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

- `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();
}
```

`Field`s

`Field`s 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

`Field vector` options

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

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 `Field`s 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]

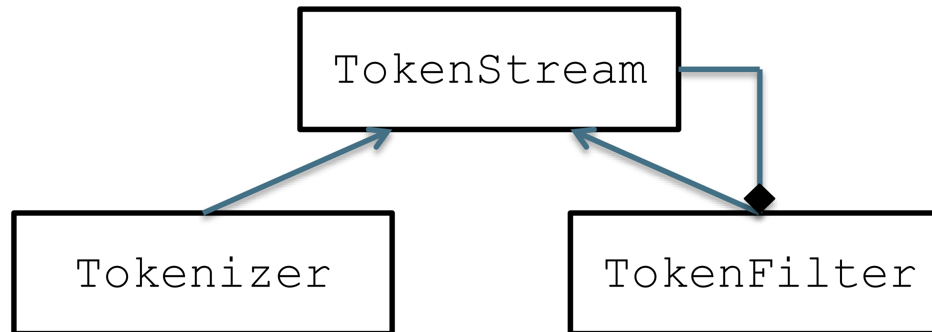
More analysis examples

- "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
  - ...

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

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

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 N documents returned by a search (the docID and its score)
- `ScoreDoc`
  - Provides access to 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"));
   }
}
```

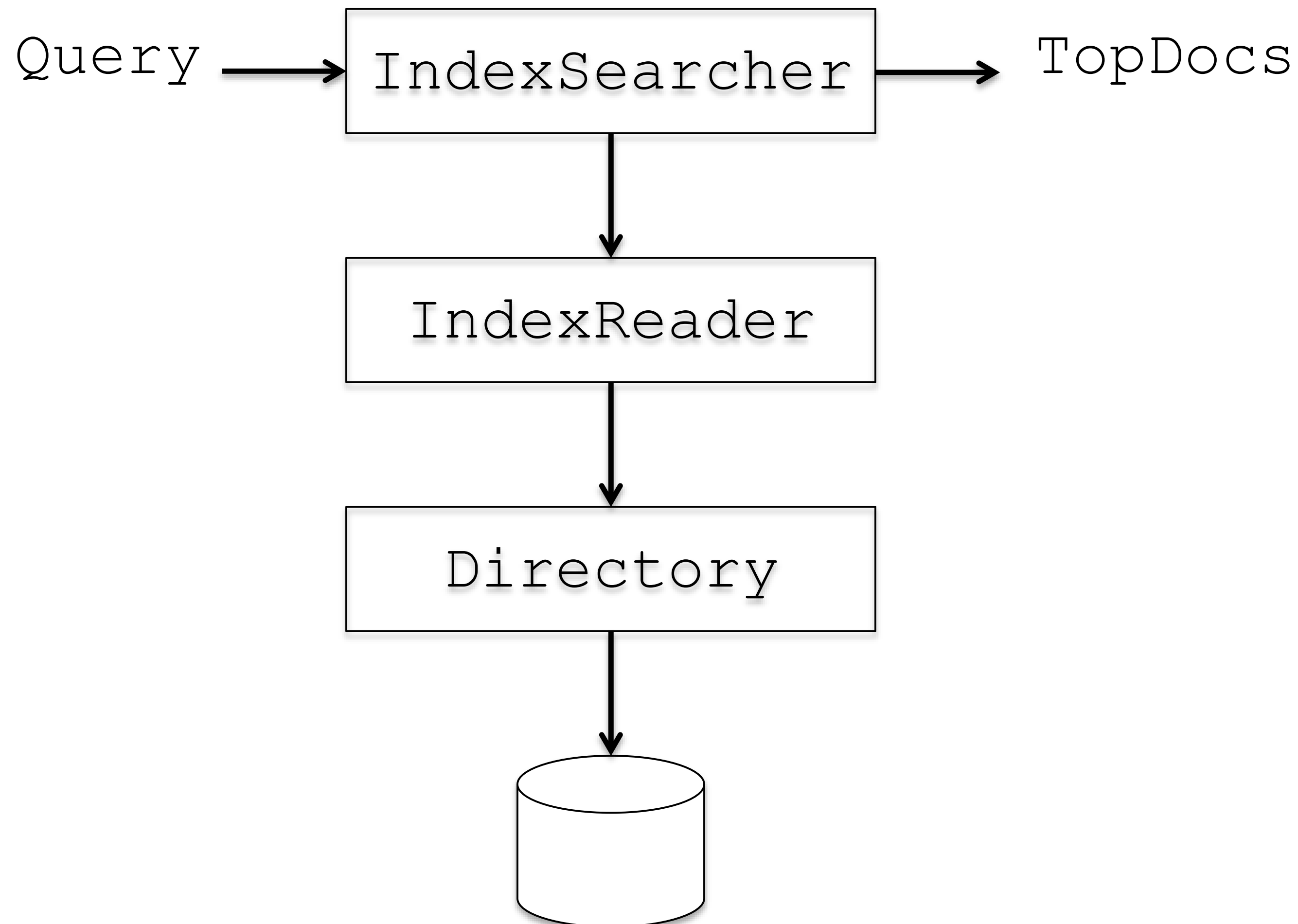
Closing `IndexSearcher`

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

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.

`Reopen`ing is more resource efficient than `open`ing 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 `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`

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

Πηγές

- Lucene can be downloaded from

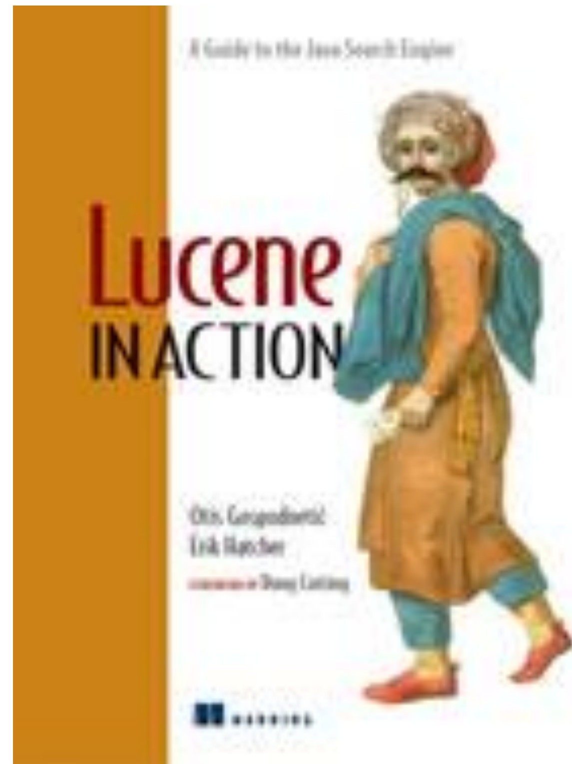
http://www.apache.org/dyn/closer.lua/lucene/java/6.0.0

- Solr can be downloaded from

http://www.apache.org/dyn/closer.lua/lucene/solr/6.0.0

Based on “Lucene in Action”

- By Michael McCandless, Erik Hatcher, Otis Gospodnetic



ΤΕΛΟΣ Μαθήματος

Ερωτήσεις?

Υλικό των:
✓ *Pandu Nayak and Prabhakar Raghavan, CS276:Information Retrieval and Web Search (Stanford)*