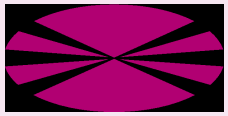


Apache Lucene

Alvaro Barreiro, Javier Parapar

Information Retrieval Lab
Departmento de Computación
Universidad de A Coruña

Apache Lucene 4.0.0 Working Notes

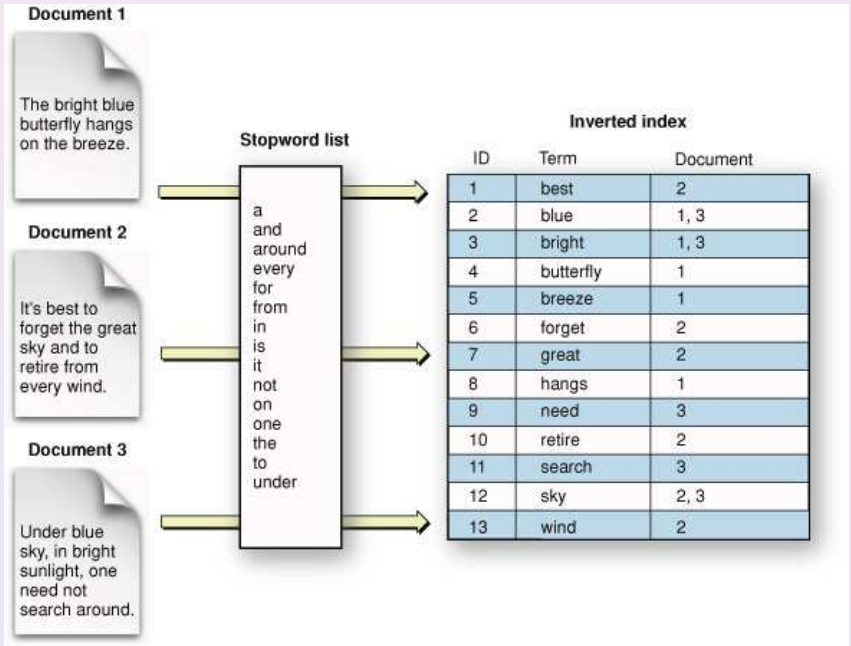
¿Qué es IR?

Information Retrieval (IR) a.k.a. Recuperación de Información

- *"El problema central en Recuperación de Información (IR) es encontrar documentos, generalmente texto, que satisfagan una necesidad de información por parte de un usuario en grandes colecciones de datos (repositorios, sistemas archivos, Web, etc.)"*
- Es un área de las ciencias de computación que se encarga de desarrollar técnicas y modelos para facilitar ante una necesidad de un usuario la búsqueda en una colección de muy gran tamaño de los documentos en los que está interesado.

Índices

- Dada la necesidad de proporcionar un método eficiente de responder a estas necesidades de información surgieron los "índices invertidos"



Búsqueda sobre Índices

- Una vez construidos los índices invertidos nos permiten obtener todos aquellos documentos que contienen los términos de la consulta del usuario simplemente operando sobre las "posting lists".
- Dependiendo del modelo de recuperación y esquemas de pesado que querramos usar a los documentos se les asignará una puntuación determinada por la fórmula de *scoring* del modelo: sim(Q,D).
- Los documentos serán posteriormente ordenados decrecientemente por dicha puntuación y la lista de documentos, ya en forma de ranking, es devuelta al usuario como respuesta a su consulta

Apache Lucene

- *"Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. Apache Lucene is an open source project available for free download."*
- Es decir, Lucene es un API que nos permite la construcción de índices invertidos y su consulta en procesos de búsqueda, i.e., Lucene es un sistema de recuperación de información.
- Lucene implementa como modelo de recuperación una versión del VSM (Vector Space Model), un modelo clásico en IR. Desde Lucene 4.0.0 incorpora otros modelos: probabilísticos (Okapi BM25 y DFR -Divergence From Randomness-) y Language Models.

Lucene, un poco de historia

- El proyecto fue iniciado por Doug Cutting en 1999.
- En 2001 se hizo cargo de él la Apache Software Foundation.
- Posteriormente se dividió en un proyecto principal, Lucene Java y varios subproyectos.
- Los ejemplos de estas notas están probados para la versión oficial 4.0.0, Lucene posee *ports* a múltiples lenguajes y es usado en muchas aplicaciones y sitios web.
- `http://lucene.apache.org/core/4_0_0/`

Objetivos

Objetivos de las prácticas

- Saber analizar documentos para su indexación.
- Saber indexar documentos con multiples campos.
- Saber postprocesar índices con borrado, modificación y añadido de documentos.
- Saber analizar consultas del usuario sobre multiples campos.
- Saber realizar consultas sobre los índices invertidos.
- Saber interpretar las lista de resultados y su información.
- Saber alterar los pesos de documentos, términos y campos.
- Estar capacitado para de forma autónomo abordar proyectos avanzados como la modificación del modelo de recuperación de información, la integración con un crawler (Nutch), o trabajar con proyectos relacionados (Solr, Tika, etc.).

Antes de empezar

Necesitaremos:

- Java - Oracle JDK 1.7
- Eclipse
- Colecciones de documentos.
- Binarios de Lucene (4.0.0)
- Fuentes de Lucene (en principio no)

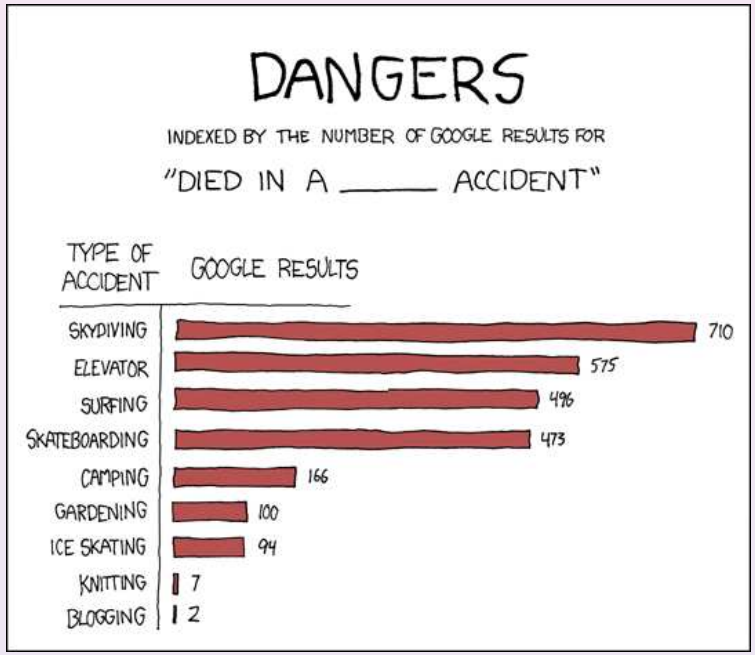
Getting started

- Bajar los binarios de Lucene (las versiones anteriores a la actual están en el Apache Arquive, http://archive.apache.org/dist/lucene/java/, los binarios están dentro de los *.tgz o los *.zip)
- Crear un proyecto TestLucene4.0.0
- Crear carpeta /lib para guardar los jar de lucene
  Al menos lucene-core-4.0.0.jar,
  lucene-analyzers-common-4.0.0.jar,
  lucene-queryparser-4.0.0.jar
- Añadir al build path del proyecto los jar de lucene

En este curso

Aprenderemos

- Indexar documentos.
- Manejo de índices.
- Búsqueda en índices.
- Construcción de *queries*.
- Explotación de resultados.
- ... pero primero un ejemplo sencillo.



```java
package simpledemo;

import java.io.IOException;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.TextField;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.queryparser.classic.ParseException;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.RAMDirectory;
import org.apache.lucene.util.Version;

public class SimpleDemo {

        /**
         * @param args
         * @throws IOException
         * @throws ParseException
         */
        public static void main(String[] args) throws IOException, ParseException {
                Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_40);

                // Store the index in memory:
                Directory directory = new RAMDirectory();
                // To store an index on disk, use this instead:
                // Directory directory = FSDirectory.open("/tmp/testindex");
                IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_40,
```

```
                        analyzer);
                IndexWriter iwriter = new IndexWriter(directory, config);
                Document doc = new Document();
                String text = "Hello world, how are you";
                doc.add(new Field("fieldname", text, TextField.TYPE_STORED));
                iwriter.addDocument(doc);
                iwriter.close();

                // Now search the index:
                DirectoryReader ireader = DirectoryReader.open(directory);
                IndexSearcher isearcher = new IndexSearcher(ireader);
                // Parse a simple query that searches for "text":
                QueryParser parser = new QueryParser(Version.LUCENE_40, "fieldname",
                        analyzer);
                Query query = parser.parse("world");
                ScoreDoc[] hits = isearcher.search(query, null, 1000).scoreDocs;
                // Iterate through the results:
                for (int i = 0; i < hits.length; i++) {
                        Document hitDoc = isearcher.doc(hits[i].doc);
                        System.out
                                        .println("This is the text of the document that
was indexed and there is a hit for the query 'world': "
                                                        + hitDoc.get("fieldname"));
                }
                ireader.close();
                directory.close();
        }

}
```

Creando índices

- Entendemos por indexación el proceso de construir los índices invertidos (y más cosas) a partir de la colección de documentos sobre la que queremos habilitar la búsqueda.
- El proceso general siempre es el siguiente:
  1. Partimos de la colección de documentos.
  2. Convertimos los documentos a formato textual.
  3. Creamos los "documentos Lucene" con los campos necesarios.
  4. Delegamos en Lucene la construcción del índice a partir de dichos documentos.

Clases clave

- A la hora de construir un índice necesitaremos tratar al menos con las siguientes clases
  - org.apache.lucene.analysis.Analyzer
  - org.apache.lucene.document.Field
  - org.apache.lucene.document.Document
  - org.apache.lucene.store.Directory
  - org.apache.lucene.index.IndexWriter
- La clase de Análisis (Analyzer) implementa una política de extracción de *index terms* a partir del texto. En principio no entraremos en detalle y trataremos con el analizador que se conoce como StandardAnalyzer (incluye un sencillo tokenizador y stopwords) org.apache.lucene.analysis.standard.StandardAnalyzer

Built-in analyzers

- WhitespaceAnalyzer: Splits tokens at whitespace.
- SimpleAnalyzer: Divides text at nonletter characters and lowercases.
- StopAnalyzer: Divides text at nonletter characters, lowercases, and removes stop words.
- KeywordAnalyzer: Treats entire text as a single token.
- StandardAnalyzer: Tokenizes based on a sophisticated grammar that recognizes email addresses, acronyms, Chinese-Japanese-Korean characters, alphanumerics, and more. It also lowercases and removes stop words.

Definiciones (I)

- The fundamental concepts in Lucene are index, document, field and term.
  An index contains a sequence of documents.
  A document is a sequence of fields.
  A field is a named sequence of terms.
  A term is a sequence of bytes.
- The same sequence of bytes in two different fields is considered a different term. Thus terms are represented as a pair: the string naming the field, and the bytes within the field.

Definiciones (II)

- Inverted Indexing. The index stores statistics about terms in order to make term-based search more efficient. Lucene's index falls into the family of indexes known as an inverted index. This is because it can list, for a term, the documents that contain it. This is the inverse of the natural relationship, in which documents list terms.
- Types of Fields. In Lucene, fields may be stored, in which case their text is stored in the index literally, in a non-inverted manner. Fields that are inverted are called indexed. A field may be both stored and indexed.
  The text of a field may be tokenized into terms to be indexed, or the text of a field may be used literally as a term to be indexed. Most fields are tokenized, but sometimes it is useful for certain identifier fields to be indexed literally.
  See the Field java docs for more information on Fields.

org.apache.lucene.document.Field (I)

- Cada una de las secciones de un documento (e.g. título, contenido, url, etc.). Un field consta de: nombre, valor y tipo. El valor puede ser texto o numérico. En Lucene 4.x los detalles de indexación del field se movieron al tipo.
- Hay ciertos tipos predefinidos por su uso común:
  - StringField, para indexar un string como un token único (i.e. lo indexa pero no lo analiza). This field turns off norms and indexes only doc IDS (does not index term frequency nor positions). This field does not store its value, but exposes TYPE_STORED as well.
  - TextField, para indexar y tokenizar un string (i.e. lo indexa y analiza). This field does not store its value, but exposes TYPE_STORED as well.
  - StoredField, campo que almacena un valor.
  - NumericField y DocValuesField (ver documentación con las subclases de field Numéricas y DocValues), básicamente se pueden usar para scoring de documentos, búsqueda por rangos y filtrado y ordenación de resultados.

org.apache.lucene.document.Field (II)

- new StringField("field", "value")
  If instead the value is stored:
  new Field("field", "value", StringField.TYPE_STORED)
- new TextField("field", value, Field.Store.NO)
  If instead the value is stored:
  new TextField("field", value, Field.Store.YES)
- You can of course also create your own FieldType from scratch:
  FieldType t = new FieldType();
  t.setIndexed(true);
  t.setStored(true);
  t.setOmitNorms(true);
  t.setIndexOptions(IndexOptions.DOCS_AND_FREQS);
  t.freeze();
  new Field("field", "value", t)

org.apache.lucene.document.Field (III)

- FieldType has a freeze() method to prevent further changes. In this example normalization were omitted for the field.
  Indexed Options:
  DOCS_AND_FREQS: Only documents and term frequencies are indexed: positions are omitted.
  DOCS_AND_FREQS_AND_POSITIONS: Indexes documents, frequencies and positions.
  DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS: Indexes documents, frequencies, positions and offsets.
  DOCS_ONLY: Only documents are indexed: term frequencies and positions are omitted.
  Más detalles en la documentación de FieldType
  (Las posiciones ordinales de un token en el texto son necesarios por ejemplo para resolver phrase queries; el character offset inicial y final de un token son necesarios por ejemplos para resaltarlos en el texto -keywords in context-).

org.apache.lucene.document.Document

- Cada una de las unidades de indexación que corresponden con uno de los documentos de la colección.
- Documents are the unit of indexing and search. A Document is a set of fields. Each field has a name and a textual value. A field may be stored with the document, in which case it is returned with search hits on the document. Thus each document should typically contain one or more stored fields which uniquely identify it.
  Note that fields which are not stored are not available in documents retrieved from the index, e.g. with ScoreDoc.doc or IndexReader.document(int).
- Pueden contener 1 ó n campos (Field)
- Operaciones más usadas:
  - add(IndexableField field): Adds a field to a document.
  - removeField(String name) Removes field with the specified name from the document.
  - get(String name): Returns the string value of the field with the given name if any exist in this document, or null.

org.apache.lucene.search.similarities (I)

- Vale la pena avanzar algo sobre los modelos de retrieval de Lucene
- Class TFIDFSimilarity : Implementa el modelo de espacio vectorial en Lucene, que es el modelo por defecto. Ver los detalles en la documentación de Lucene. org.apache.lucene.search.similarities.TFIDFSimilarity
- Lucene 4.x contiene implementaciones de otros modelos, en particular los muy usados BM25 y Language Models.

org.apache.lucene.search.similarities (II)

- Changing Similarity (No será necesario en este curso pero es una información que conviene tener presente)
  Chances are the available Similarities are sufficient for all your searching needs. However, in some applications it may be necessary to customize your Similarity implementation. For instance, some applications do not need to distinguish between shorter and longer documents (see a "fair"similarity).

org.apache.lucene.search.similarities (III)

- To change Similarity, one must do so for both indexing and searching, and the changes must happen before either of these actions take place. Although in theory there is nothing stopping you from changing mid-stream, it just isn't well-defined what is going to happen.
  To make this change, implement your own Similarity (likely you'll want to simply subclass an existing method, be it DefaultSimilarity or a descendant of SimilarityBase), and then register the new class by calling IndexWriterConfig.setSimilarity(Similarity) before indexing and IndexSearcher.setSimilarity(Similarity) before searching.

org.apache.lucene.store.Directory

- Representa una carpeta de un sistema de ficheros.
- No se usa el API estándar de Java para así poder crear distintas clases de directorios :
- Sobre el sistema de ficheros nativo
  - org.apache.lucene.store.FSDirectory
    Base class for Directory implementations that store index files in the file system. Para crear un directorio donde almacenar un índice:
    static FSDirectory open(File path)
    Creates an FSDirectory instance, trying to pick the best implementation given the current environment.
- En memoria
  - org.apache.lucene.store.RAMDirectory
    A memory-resident Directory implementation. This class is optimized for small memory-resident indexes. Everything beyond several hundred megabytes will waste resources.

org.apache.lucene.index.IndexWriter (I)

- El "indexador" de Lucene.
- A él se le pasarán las unidades de indexación (Document) e irá añadiendo sus campos a los índices según lo establecido en la construcción de dichos campos (Field).
- public IndexWriter(Directory d, IndexWriterConfig conf)
  Constructs a new IndexWriter per the settings given in conf.
- public final class IndexWriterConfig
  Holds all the configuration of IndexWriter.
  IndexWriterConfig(Version matchVersion, Analyzer analyzer)
  Creates a new config that with defaults that match the specified Version as well as the default Analyzer.

org.apache.lucene.index.IndexWriter (&II)

- Los métodos más usados:
  - addDocument(Document doc): Adds a document to this index.
  - addIndexes(IndexReader[] readers): Merges the provided indexes into this index.[1]
  - commit(): Commits all pending updates (added & deleted documents) to the index, and syncs all referenced index files, such that a reader will see the changes and the index updates will survive an OS or machine crash or power loss.
  - close(): Commits all changes to an index and closes all associated files.

[1] ya veremos que es un IndexReader

org.apache.lucene.index.IndexWriter (&III)

- deleteAll(): Delete all documents in the index.
- deleteDocuments(Query... queries): Deletes the document(s) matching any of the provided queries.
- deleteDocuments(Query query): Deletes the document(s) matching the provided query.
- deleteDocuments(Term... terms): Deletes the document(s) containing any of the terms.
- deleteDocuments(Term term): Deletes the document(s) containing term.

org.apache.lucene.index.IndexWriter (&IV)

- int maxDoc(): Returns total number of docs in this index, including docs not yet flushed (still in the RAM buffer), not counting deletions.
- int numDocs(): Returns total number of docs in this index, including docs not yet flushed (still in the RAM buffer), and including deletions.
- void updateDocument(Term term, doc): Updates a document by first deleting the document containing term and then adding the new document (it is really a delete followed by an add).

org.apache.lucene.index.IndexWriterConfig

- Métodos más usados
  - IndexWriterConfig.OpenMode getOpenMode(): Returns the IndexWriterConfig.OpenMode set by setOpenMode(OpenMode).
  - IndexWriterConfig setOpenMode(IndexWriterConfig.OpenMode openMode): Specifies IndexWriterConfig.OpenMode of the index:
    APPEND: Opens an existing index.
    CREATE: Creates a new index or overwrites an existing one.
    CREATE_OR_APPEND: Creates a new index if one does not exist, otherwise it opens the index and documents will be appended.

Segmentos y formatos de índices (I)

- Lucene indexes may be composed of multiple sub-indexes, or segments. Each segment is a fully independent index, which could be searched separately. Indexes evolve by:
  -Creating new segments for newly added documents.
  -Merging existing segments.
  Searches may involve multiple segments and/or multiple indexes, each index potentially composed of a set of segments.
- Each segment index maintains the following:
  -Segment info. This contains metadata about a segment, such as the number of documents, what files it uses.
  -Field names. This contains the set of field names used in the index.

Segmentos y formatos de índices (II)

-Stored Field values. This contains, for each document, a list of attribute-value pairs, where the attributes are field names. The set of stored fields are what is returned for each hit when searching. This is keyed by document number.
-Term dictionary. A dictionary containing all of the terms used in all of the indexed fields of all of the documents. The dictionary also contains the number of documents which contain the term, and pointers to the term's frequency and proximity data.
-Term Frequency data. For each term in the dictionary, the numbers of all the documents that contain that term, and the frequency of the term in that document, unless frequencies are omitted (IndexOptions.DOCS_ONLY)
-Term Proximity data. For each term in the dictionary, the positions that the term occurs in each document. Note that this will not exist if all fields in all documents omit position data.

Segmentos y formatos de índices (III)

-Normalization factors. For each field in each document, a value is stored that is multiplied into the score for hits on that field.
-Term Vectors. For each field in each document, the term vector (sometimes called document vector) may be stored. A term vector consists of term text and term frequency. To add Term Vectors to your index see the Field constructors
- Per-document values. Like stored values, these are also keyed by document number, but are generally intended to be loaded into main memory for fast access. Whereas stored values are generally intended for summary results from searches, per-document values are useful for things like scoring factors.
- Deleted documents. An optional file indicating which documents are deleted.

Segmentos y formatos de índices (IV)

- File Names. All files belonging to a segment have the same name with varying extensions.
- The extensions correspond to the different file formats and they are described in Apache Lucene 4.0.0. Documentation. Reference Documents. File Formats.
  `http://lucene.apache.org/core/4_0_0/core/org/apache/luc`
  segments.gen, segments_N write.lock .si .cfs, .cfe .fnm .fdx .fdt .tim .tip .doc .pos .pay .nvd, .nvm .dvd, .dvm .tvx .tvd .tvf .de

```java
package simpleindexing;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.IntField;
import org.apache.lucene.document.StringField;
import org.apache.lucene.document.TextField;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.store.LockObtainFailedException;
import org.apache.lucene.util.Version;

public class SimpleIndexing {

        /**
         * Project testlucene4.0.0 SimpleIndexing class write a lucene index with
         * some small documents. If the index already exists, the documents are
         * appended to the index
         */
        public static void main(String[] args) {

                if (args.length != 1) {
                        System.out.println("Usage: java SimpleIndexing indexFolder");
                        return;
                }

                String modelRef[] = new String[4];

                String modelDescription[] = new String[4];
                String modelAcronym[] = new String[4];
```

```
                int theoreticalContent = 10;
                int practicalContent = 10;

                modelRef[0] = "RM000";
                modelRef[1] = "RM001";
                modelRef[2] = "RM002";
                modelRef[3] = "RM003";

                modelAcronym[0] = "BM";
                modelAcronym[1] = "VSM";
                modelAcronym[2] = "CPM";
                modelAcronym[3] = "LM";

                modelDescription[0] = "The boolean model is a simple retrieval model
where queries are interpreted as boolean expressions and documents
are bag of words";
                modelDescription[1] = "The vector space model is a simple retrieval
model where queries and documents are vectors of terms and similarity
of queries and documents is computed with the cosine distance";
                modelDescription[2] = "In the classic probabilistic retrieval model the
probability of relevance of a document given a query is computed under
the binary and independence assumptions";
                modelDescription[3] = "The use of language models for retrieval
implies the estimation of the probability of generating a query given a document";

                String indexFolder = args[0];

                IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_40,
                                new StandardAnalyzer(Version.LUCENE_40));
                IndexWriter writer = null;

                try {
                        writer = new IndexWriter(FSDirectory.open(new File(indexFolder)),
                                        config);
                } catch (CorruptIndexException e1) {
                        System.out.println("Graceful message: exception " + e1);
```

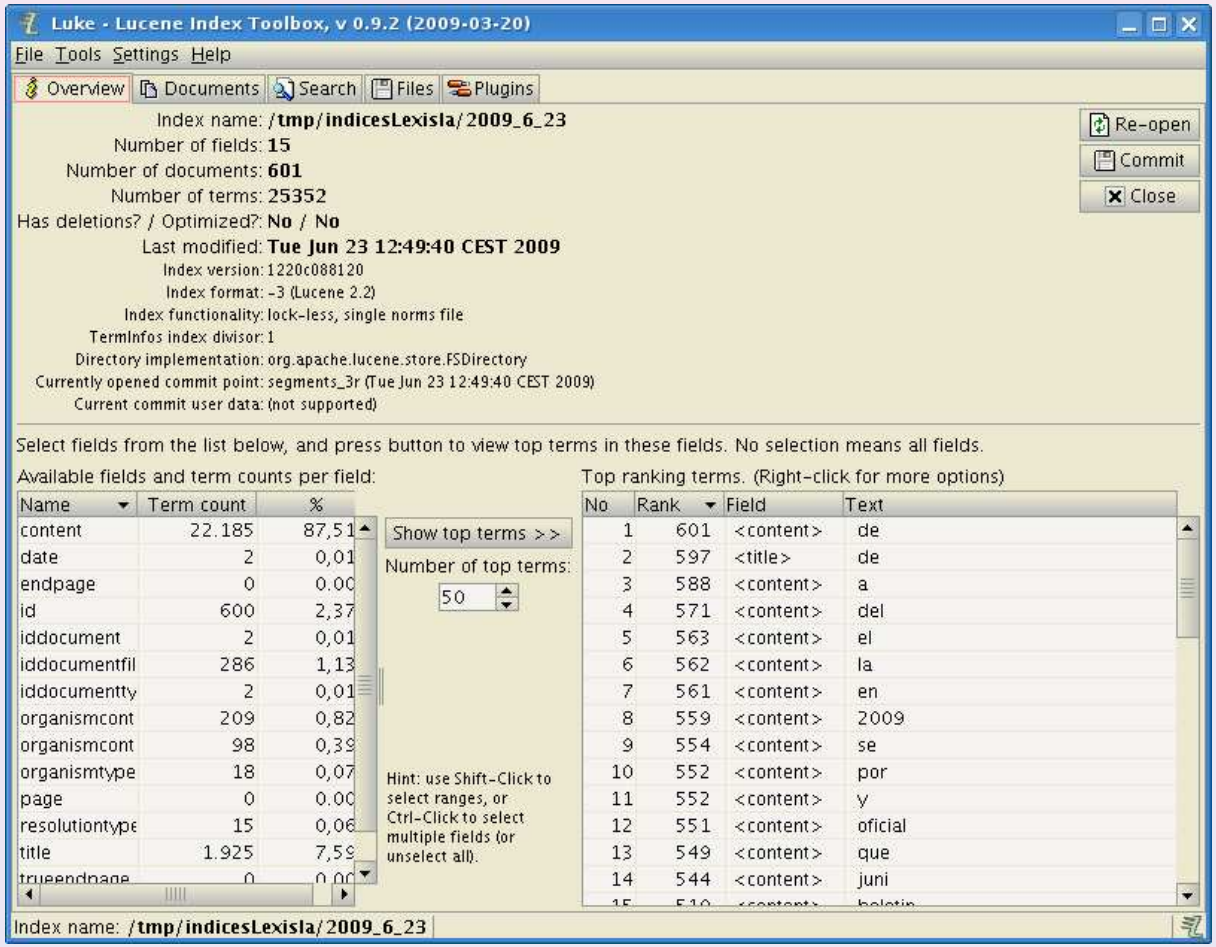
```
                e1.printStackTrace();
            } catch (LockObtainFailedException e1) {
                System.out.println("Graceful message: exception " + e1);
                e1.printStackTrace();
            } catch (IOException e1) {
                System.out.println("Graceful message: exception " + e1);
                e1.printStackTrace();
            }

            /*
             * With these calls to IndexWriterConfig and IndexWriter this program
             * creates a new index if one does not exist, otherwise it opens the
             * index and documents will be appended with writer.addDocument(doc).
             */

            for (int i = 0; i < modelRef.length; i++) {
                Document doc = new Document();
/*
 * Each document has five fields. modelRef is a StringField which is
 * indexed and not tokenized. modelAcronym is a StringField which is
 * indexed and not tokenized, addtionally it is stored.
 * modelDescription is a TextField which is indexed and tokenized,
* additionally it is stored. theoreticalContent is a NumericField
* that is indexed. practicalContent is a NumericField that is
* indexed, additionally it is stored.
*/
                doc.add(new StringField("modelRef", modelRef[i], null));
                doc.add(new Field("modelAcronym", modelAcronym[i],
                        StringField.TYPE_STORED));

                doc.add(new TextField("modelDescription", modelDescription[i],
                        Field.Store.YES));

                doc.add(new IntField("theoreticalContent",
                        theoreticalContent++,
                        Field.Store.NO));
```

```
                    doc.add(new IntField("practicalContent",
                                         practicalContent++,
                                         Field.Store.YES));

                    try {

                        writer.addDocument(doc);
                    } catch (CorruptIndexException e) {
                        System.out.println("Graceful message: exception " + e);
                        e.printStackTrace();
                    } catch (IOException e) {
                        System.out.println("Graceful message: exception " + e);
                        e.printStackTrace();
                    }

                    System.out.println("wrote document " + i + " in the index");

                }

                try {
                    writer.commit();
                    writer.close();
                } catch (CorruptIndexException e) {
                    System.out.println("Graceful message: exception " + e);
                    e.printStackTrace();
                } catch (IOException e) {
                    System.out.println("Graceful message: exception " + e);
                    e.printStackTrace();
                }

        }
}
```

Luke (I)

Bajar los binarios de la última versión de Luke, en el momento de escribir estas notas están en http://code.google.com/p/luke/
Los binarios contienen lucene y luke por lo que se pueden ejecutar con java -jar lukeall-4.x.x jar

Luke (II)


Luke - Lucene Index Toolbox, v 0.9.2 (2009-03-20)
File Tools Settings Help
Overview Documents Search Files Plugins
Index name: /tmp/indicesLexisla/2009_6_23
Number of fields: 15
Number of documents: 601
Number of terms: 25352
Has deletions? / Optimized?: No / No
Last modified: Tue Jun 23 12:49:40 CEST 2009
Index version: 1220c088120
Index format: -3 (Lucene 2.2)
Index functionality: lock-less, single norms file
TermInfos index divisor: 1
Directory implementation: org.apache.lucene.store.FSDirectory
Currently opened commit point: segments_3r (Tue Jun 23 12:49:40 CEST 2009)
Current commit user data: (not supported)
Re-open
Commit
Close
Select fields from the list below, and press button to view top terms in these fields. No selection means all fields.
Available fields and term counts per field:
Name Term count %
content 22.185 87,51
date 2 0,01
endpage 0 0.00
id 600 2,37
iddocument 2 0,01
iddocumentfil 286 1,13
iddocumentty 2 0,01
organismcont 209 0,82
organismcont 98 0,39
organismtype 18 0,07
page 0 0.00
resolutiontype 15 0,06
title 1.925 7,59
Show top terms >>
Number of top terms:
50
Hint: use Shift-Click to select ranges, or Ctrl-Click to select multiple fields (or unselect all).
Top ranking terms. (Right-click for more options)
No Rank Field Text
1 601 <content> de
2 597 <title> de
3 588 <content> a
4 571 <content> del
5 563 <content> el
6 562 <content> la
7 561 <content> en
8 559 <content> 2009
9 554 <content> se
10 552 <content> por
11 552 <content> y
12 551 <content> oficial
13 549 <content> que
14 544 <content> juni
Index name: /tmp/indicesLexisla/2009_6_23

org.apache.lucene.index.Term (I)

- A Term represents a word from text. This is the unit of search. It is composed of two elements, the text of the word, as a string, and the name of the field that the text occurred in, an interned string. Note that terms may represent more than words from text fields, but also things like dates, email addresses, urls, etc. Constructores:
  - Term(String fld): Constructs a Term with the given field and empty text.
  - Term(String fld, String txt): Constructs a Term with the given field and text.
  - Term(String fld, BytesRef bytes): Constructs a Term with the given field and bytes.

org.apache.lucene.index.Term (II)

- Métodos
  - BytesRef bytes(): Returns the bytes of this term.
  - int compareTo(Term other): Compares two terms, returning a negative integer if this term belongs before the argument, zero if this term is equal to the argument, and a positive integer if this term belongs after the argument.
  - boolean equals(Object obj)
  - String field(): Returns the field of this term.
  - String text(): Returns the text of this term.
  - String toString(): Returns a string representation of the object.

org.apache.lucene.index.DirectoryReader

- Con una llamado al método estático open de esta clase podemos obtener un index reader con el que leer un índice
  - open(Directory directory): Returns a IndexReader reading the index in the given Directory
  - indexExists(Directory directory): Returns true if an index exists at the specified directory.

org.apache.lucene.index.IndexReader(I)

- Una vez construido un índice, y obtenido un index reader, podremos acceder a su contenido con esta clase.
  - close(): Closes files associated with this index.
  - maxDoc(): Returns (int) one greater than the largest possible document number.
  - numDoc(): Returns the number of documents in this index.
  - docFreq(Term t): Returns (int) the number of documents containing the term t. This method returns 0 if the term or field does not exists. This method does not take into account deleted documents that have not yet been merged away.
  - public final Document document(int n): Returns the stored fields (a Document object is a set of fields) of the nth Document in this index.
  - public final Document document(int docID, Set<String> fieldsToLoad): Like document(int) but only loads the specified fields.

org.apache.lucene.index.AtomicReader(I)

- Fields fields(): Returns Fields for this reader.
- Terms terms(String field): This may return null if the field does not exist.
- DocsEnum termDocsEnum(Term term): Returns DocsEnum for the specified term.
- DocsAndPositionsEnum termPositionsEnum(Term term): Returns DocsAndPositionsEnum for the specified term.

org.apache.lucene.index.Document

- Métodos de la clase Document para acceder a los fields y su contenido. (Note that fields which are not stored are not available in documents retrieved from the index. Note that IndexableField is an experimental API)
  - String get(String name): Returns the string value of the field with the given name if any exist in this document, or null
  - IndexableField getField(String name): Returns a field with the given name if any exist in this document, or null
  - List<IndexableField> getFields(): Returns a List of all the fields in a document.

```java
package simpleindexing;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexableField;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;

public class SimpleReader1 {

        /**
         * Project testlucene4.0.0 SimpleReader class reads the index SimpleIndex
         * created with the SimpleIndexing class
         */
        public static void main(String[] args) {

                if (args.length != 1) {
                        System.out.println("Usage: java SimpleReader SimpleIndex");
                        return;
                }
                // SimpleIndex is the folder where the index SimpleIndex is stored

                File file = new File(args[0]);

                Directory dir = null;
                DirectoryReader indexReader = null;
                Document doc = null;

                List<IndexableField> fields = null;
```

```
try {
        dir = FSDirectory.open(file);
        indexReader = DirectoryReader.open(dir);
} catch (CorruptIndexException e1) {
        System.out.println("Graceful message: exception " + e1);
        e1.printStackTrace();
} catch (IOException e1) {
        System.out.println("Graceful message: exception " + e1);
        e1.printStackTrace();
}

for (int i = 0; i < indexReader.maxDoc(); i++) {

        try {
                doc = indexReader.document(i);
        } catch (CorruptIndexException e1) {
                System.out.println("Graceful message: exception " + e1);
                e1.printStackTrace();
        } catch (IOException e1) {
                System.out.println("Graceful message: exception " + e1);
                e1.printStackTrace();
        }
        System.out.println("Documento " + i);
        System.out.println("modelRef = " + doc.get("modelRef"));
        System.out.println("modelAcronym = " + doc.get("modelAcronym"));
        System.out.println("modelDescription = "
                        + doc.get("modelDescription"));
        System.out.println("theoreticalContent = "
                        + doc.get("theoreticalContent"));
        System.out.println("practicalContent = "
                        + doc.get("practicalContent"));
}

/**
 * Note doc.get() returns null for the fields that were not stored
 */
```

```
                for (int i = 0; i < indexReader.maxDoc(); i++) {

                        try {
                                doc = indexReader.document(i);
                        } catch (CorruptIndexException e1) {
                                System.out.println("Graceful message: exception " + e1);
                                e1.printStackTrace();
                        } catch (IOException e1) {
                                System.out.println("Graceful message: exception " + e1);
                                e1.printStackTrace();
                        }

                        System.out.println("Documento " + i);

                        fields = doc.getFields();

                        for (IndexableField field : fields) {
                          String fieldName = field.name();
                          System.out.println(fieldName + ": " + doc.get(fieldName));

                        }

                }

                try {
                        indexReader.close();
                } catch (IOException e) {
                        System.out.println("Graceful message: exception " + e);
                        e.printStackTrace();
                }

        }
}
```

org.apache.lucene.index.AtomicReader

DirectoryReader devuelve la visión más abstracta de un índice. Internamente Lucene utiliza atomic readers y multireaders. Básicamente un atomic reader se asocia a un segmento y un índice puede constar de varios segmentos. Un atomic reader proporciona métodos para acceder a los fields de un índice, a los terms de un field, y a los documentos para un term. Pero para acceder a ellos debemos emular un atomic reader de forma que el comportamiento sea fiable tanto teniendo uno o varios segmentos.

org.apache.lucene.index.SlowCompositeReaderWrapper

This class forces a composite reader (eg a MultiReader or DirectoryReader) to emulate an atomic reader. A partir de esta clase. Esta es la forma de acceder a los fields y términos de un Indice desde Lucene 4.x.

- Fields fields(): Returns Fields for this reader.
- Terms terms(String field): This may return null if the field does not exist.
- DocsEnum termDocsEnum(Term term): Returns DocsEnum for the specified term.
- DocsAndPositionsEnum termPositionsEnum(Term term): Returns DocsAndPositionsEnum for the specified term.

```java
package simpleindexing;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.index.CompositeReader;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.DocsEnum;
import org.apache.lucene.index.Fields;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.index.SlowCompositeReaderWrapper;
import org.apache.lucene.index.Term;
import org.apache.lucene.index.Terms;
import org.apache.lucene.index.TermsEnum;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;

public class SimpleReader2 {

        /**
         * Project testlucene4.0.0 SimpleReader class reads the index SimpleIndex
         * created with the SimpleIndexing class
         *
         * @throws IOException
         */
        public static void main(String[] args) throws IOException {

                if (args.length != 1) {
                        System.out.println("Usage: java SimpleReader SimpleIndex");
                        return;
                }
                // SimpleIndex is the folder where the index SimpleIndex is stored

                File file = new File(args[0]);
```

```
Directory dir = null;
IndexReader indexReader = null;
SlowCompositeReaderWrapper atomicReader = null;
/*
 * SlowCompositeReaderWrapper emulates an atomic reader allowing the
 * access to fields and terms
 */

Fields fields = null;
Terms terms = null;
TermsEnum termsEnum = null;

try {
        dir = FSDirectory.open(file);
        indexReader = DirectoryReader.open(dir);
        atomicReader = new SlowCompositeReaderWrapper(
                        (CompositeReader) indexReader);

} catch (CorruptIndexException e1) {
        System.out.println("Graceful message: exception " + e1);
        e1.printStackTrace();
} catch (IOException e1) {
        System.out.println("Graceful message: exception " + e1);
        e1.printStackTrace();
}

fields = atomicReader.fields();
for (String field : fields) {
        if (!(field.equals("practicalContent") || field
                        .equals("theoreticalContent"))) {
                System.out.println("Field = " + field);
                terms = fields.terms(field);
                termsEnum = terms.iterator(null);
                while (termsEnum.next() != null) {
                        String tt = termsEnum.term().utf8ToString();
```

```
                                        System.out.println(tt + " totalFreq()="
                                                + termsEnum.totalTermFreq()
                                                + " docFreq="
                                                + termsEnum.docFreq());

                                }
                        }
                }

                // totalFreq equals -1 if the value was not stored in the codification
                // of this index

                /*
                 * Terms are ByteRef in lucene 4.x (see Class ByteRef of lucene api),
                 * for this reason the above code is not the best for printing numeric
                 * fields
                 */

                Term t = new Term("modelDescription", "probability");
                Document d = new Document();
                DocsEnum docsEnum = atomicReader.termDocsEnum(t);
                int doc;
                while ((doc = docsEnum.nextDoc()) != DocsEnum.NO_MORE_DOCS) {
                System.out
.println("El termino (field=modelDescription text=probability) aparece en el doc num: "
                                                + doc);
                d = atomicReader.document(doc);
                System.out
                .println("modelDescription= " + d.get("modelDescription"));
                }

                try {
                        atomicReader.close();
                        indexReader.close();
                } catch (IOException e) {
                        System.out.println("Graceful message: exception " + e);
```

```
                e.printStackTrace();
            }

        }
```

org.apache.lucene.index.Terms

- Access to the terms in a specific field. Experimental API.
- Constructor: Terms(). Métodos más usados:
  - TermsEnum iterator(TermsEnum reuse): Returns an iterator that will step through all terms.
  - long size(): Returns the number of terms for this field, or -1 if this measure isn't stored by the codec.
  - int getDocCount(): Returns the number of documents that have at least one term for this field, or -1 if this measure isn't stored by the codec.

org.apache.lucene.index.TermsEnum

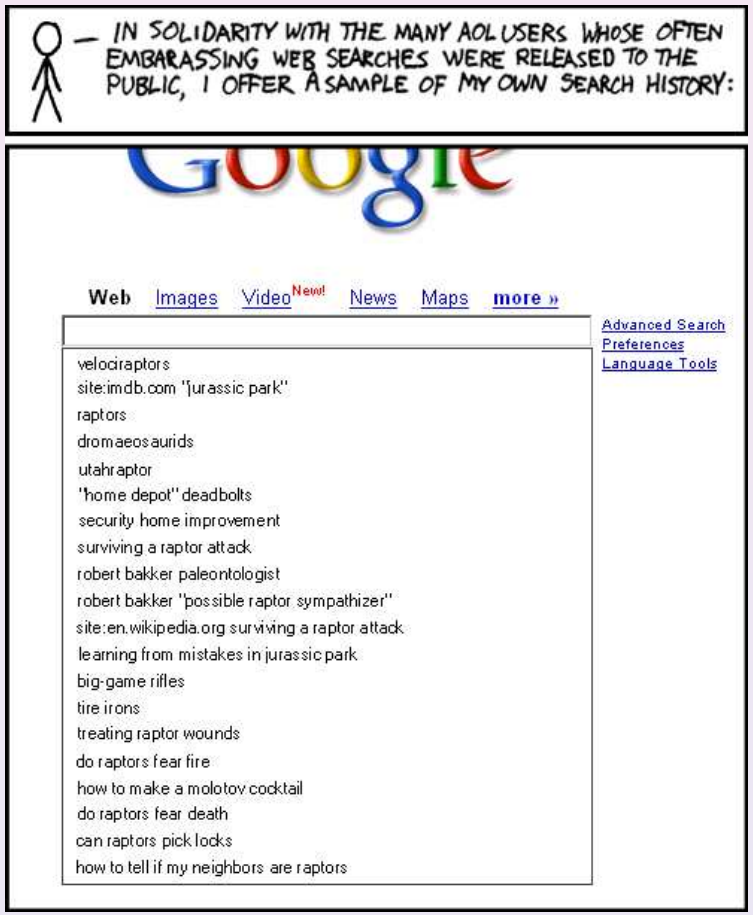
- Constructor TermsEnum(). (Experimental API). Métodos más usados
  - int docFreq(): Returns the number of documents containing the current term.
  - BytesRef term(): Returns current term.
  - long totalTermFreq() Returns the total number of occurrences of this term across all documents (the sum of the freq() for each doc that has this term).
  - DocsEnum docs(Bits liveDocs, DocsEnum reuse): Get DocsEnum for the current term.
  - DocsAndPositionsEnum docsAndPositions(Bits liveDocs, DocsAndPositionsEnum reuse): Get DocsAndPositionsEnum for the current term.
  - DocsAndPositionsEnum docsAndPositions(Bits liveDocs, DocsAndPositionsEnum reuse, int flags): Get DocsAndPositionsEnum for the current term, with control over whether offsets and payloads are required.

org.apache.lucene.index.DocsEnum

- public abstract class DocsEnum extends DocIdSetIterator:
  Iterates through the documents and term freqs. NOTE: you must first call DocIdSetIterator.nextDoc() before using any of the per-doc methods.
  (DocsAndPositionsEnum also iterates through positions.)
  Métodos:
  - int freq() Returns term frequency in the current document, or 1 if the field was indexed with FieldInfo.IndexOptions.DOCS_ONLY.
  - int nextDoc():
    Advances to the next document in the set and returns the doc it is currently on, or NO_MORE_DOCS if there are no more docs in the set.

Búsqueda

- Ya sabemos construir un índice, leerlo y modificarlo.
- Ahora aprendermos a usarlos para búsqueda.
- Veremos cómo buscar (IndexSearcher, Query).
- Cómo interprentar los resultados (TopDocs).
- La sintaxis de las consultas.
- Los distintos tipos de consultas admitidas.



org.apache.lucene.search.IndexSearcher

- Una vez construido un índice, podremos buscar en su contenido con esta clase.
  - search(Query query, Filter filter, int n) Finds the top n hits for query, applying filter if non-null.
  - search(Query query, Filter filter, int n, Sort sort) Search implementation with arbitrary sorting.
  - explain(Query query, int doc): Returns an Explanation that describes how doc scored against query.
  - setSimilarity(Similarity similarity): Expert: Set the Similarity implementation used by this Searcher.
  - getSimilarity(): Expert: Return the Similarity implementation used by this Searcher.
- Constructores: IndexSearcher(IndexReader r)
  Creates a searcher searching the provided index

Filter, Sort, Explanation, Similarity

- Filter: Abstract base class for restricting which documents may be returned during searching.
  Se pueden restringir con criterios de rangos de docID pero también de valores de campo y otros.
- Sort: Encapsulates sort criteria for returned hits.
  El orden por defecto es relevance pero se podría cambiar a otros como IndexOrder o por valores de campos.
- Explanation: Describes the score computation for document and query.
- Similarity: Similarity defines the components of Lucene scoring.

org.apache.lucene.search.Query

- Da soporte a las posibilidades de consulta de Lucene. Algunas subclases:
  - TermQuery: un término.
  - BooleanQuery: expresiones lógicas (must, must not, should) must: AND, must not: NOT, should: el término debiera aparecer pero aunque no aparezca no se excluye el documento.
  - WildcardQuery: uso de comodines.
  - PhraseQuery y MultiPhraseQuery: búsqueda por frases. Ejemplo de PhraseQuery: "New York". Una MultiPhraseQuery permitiría por ejemplo construir esa query pero donde el segundo término pudiera ser York, Orleans o Zealand.
  - PrefixQuery: búsqueda por prefijos.
  - FuzzyQuery: con Levenshtein (edit distance). Se puede indicar una distancia mínima entre los query terms y los matching terms
  - SpanQuery: SpanFirstQuery, SpanNearQuery, SpanNotQuery, SpanOrQuery, SpanTermQuery. Permite construir queries de proximidad con muchas más opciones que la simple PhraseQuery

org.apache.lucene.queryParser.classic.QueryParser (I)

- Constructor: QueryParser(Version matchVersion, String field, Analyzer a). Métodos:
  - parse(String query). Parses a query string, returning a Query.
  - setDateResolution(String fieldName, DateTools.Resolution dateResolution) Sets the date resolution used by RangeQueries for a specific field.
- org.apache.lucene.queryParser.classic.MultiFieldQueryParser. A QueryParser which constructs queries to search multiple fields. Constructors:
  MultiFieldQueryParser(Version matchVersion, String[] fields, Analyzer analyzer)
  MultiFieldQueryParser(Version matchVersion, String[] fields, Analyzer analyzer, Map$<$ *String*, *Float* $>$ boosts)
  boosts: boosting de términos, i.e., aumenta la relevancia de documentos que contienen esos términos

org.apache.lucene.queryParser.classic.QueryParser (II)

- Métodos para MultiFieldQueryParser.
  - public static Query parse(Version matchVersion, String[] queries, String[] fields, Analyzer analyzer)
    Parses a query which searches on the fields specified.
    If x fields are specified, this effectively constructs:
    (field1:query1) (field2:query2) (field3:query3)...(fieldx:queryx)
  - public static Query parse(Version matchVersion, String query, String[] fields, BooleanClause.Occur[] flags, Analyzer analyzer)
    Parses a query, searching on the fields specified. Use this if you need to specify certain fields as required, and others as prohibited.
    Ver ejemplo en la documentación
- Sintaxis de la query en http://lucene.apache.org/core/4_2_0/queryparsersyntax.html

org.apache.lucene.search.TopDocs

This class represents hits returned by IndexSearcher.search(Query,Filter,int) and IndexSearcher.search(Query,int).
Constructor: TopDocs(int totalHits, ScoreDoc[] scoreDocs, float maxScore).
Fields:

- scoreDocs Expert: The top hits for the query.
- totalHits Expert: The total number of hits for the query.
- org.apache.lucene.search.ScoreDoc
  - int doc Expert: A hit document's number.
  - float score Expert: The score of this document for the query.

```java
package simpleindexing;

import java.io.File;
import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryparser.classic.ParseException;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.BooleanClause.Occur;
import org.apache.lucene.search.BooleanQuery;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.Sort;
import org.apache.lucene.search.SortField;
import org.apache.lucene.search.TermQuery;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class SimpleSearch {

        /**
         * Project testlucene4.0.0 SimpleSearch class reads the index SimpleIndex
         * created with the SimpleIndexing class, creates and Index Searcher and
         * search for documents which contain the word "probability" in the field
         * "modelDescription" using the StandardAnalyzer
         *
         */
        public static void main(String[] args) {
```

```
if (args.length != 1) {
        System.out.println("Usage: java SimpleSearch SimpleIndex");
        return;
}
// SimpleIndex is the folder where the index SimpleIndex is stored

File file = new File(args[0]);

IndexReader reader = null;
Directory dir = null;
IndexSearcher searcher = null;
QueryParser parser;
Query query = null;

try {
        dir = FSDirectory.open(file);
        reader = DirectoryReader.open(dir);

} catch (CorruptIndexException e1) {
        System.out.println("Graceful message: exception " + e1);
        e1.printStackTrace();
} catch (IOException e1) {
        System.out.println("Graceful message: exception " + e1);
        e1.printStackTrace();
}

searcher = new IndexSearcher(reader);
parser = new QueryParser(Version.LUCENE_40, "modelDescription",
                new StandardAnalyzer(Version.LUCENE_40));

try {
        query = parser.parse("probability");
} catch (ParseException e) {

        e.printStackTrace();
```

```
}

TopDocs topDocs = null;

try {
        topDocs = searcher.search(query, 10);
} catch (IOException e1) {
        System.out.println("Graceful message: exception " + e1);
        e1.printStackTrace();
}
System.out
                .println("\n"
                                + topDocs.totalHits
                                + " results for query \""
                                + query.toString()
                                + "\" showing for the first "
                                + 10
                                + " documents the doc id , "
                                + "score and the content of"
                                + " the modelDescription field");

for (int i = 0; i < Math.min(10, topDocs.totalHits); i++) {
        try {
                System.out.println(topDocs.scoreDocs[i].doc
                + " --- score: "
                + topDocs.scoreDocs[i].score
                + " --- "
                + reader.document(topDocs.scoreDocs[i].doc).get(
                                "modelDescription"));
        } catch (CorruptIndexException e) {
                System.out.println("Graceful message: exception " + e);
                e.printStackTrace();
        } catch (IOException e) {
                System.out.println("Graceful message: exception " + e);
                e.printStackTrace();
        }
```

```
}

// follows and example using a numeric field for sorting the results
// by default this does not compute the scores of docs, since the
// ranking is imposed by the sorting

boolean reverse = true;
try {
        topDocs = searcher.search(query, 10, new Sort(new SortField(
                        "practicalContent", SortField.Type.INT, reverse)
} catch (IOException e1) {
        System.out.println("Graceful message: exception " + e1);
        e1.printStackTrace();
}
System.out
                .println("\n"
                                + topDocs.totalHits
                                + " results for query \""
                                + query.toString()
                                + "in the sort given by"
                                + " the field practicalContent , "
                                + "\" showing for the first "
                                + 10
                                + " documents the doc id , "
                                + "score and the content "
                                + "of the modelDescription field");

for (int i = 0; i < Math.min(10, topDocs.totalHits); i++) {
        try {
                System.out.println(topDocs.scoreDocs[i].doc
                + " --- score: "
                + topDocs.scoreDocs[i].score
                + " --- "
                + reader.document(topDocs.scoreDocs[i].doc).get(
                                "modelDescription"));
```

```
            } catch (CorruptIndexException e) {
                    System.out.println("Graceful message: exception " + e);
                    e.printStackTrace();
            } catch (IOException e) {
                    System.out.println("Graceful message: exception " + e);
                    e.printStackTrace();
            }

    }

    // follows an example of a simple programmatic query

    BooleanQuery booleanQuery = new BooleanQuery();
    Query vector = new TermQuery(new Term("modelDescription", "vector"));
    Query space = new TermQuery(new Term("modelDescription", "space"));

    booleanQuery.add(vector, Occur.MUST);
    booleanQuery.add(space, Occur.MUST);

    try {
            topDocs = searcher.search(booleanQuery, 10);
    } catch (IOException e1) {
            System.out.println("Graceful message: exception " + e1);
            e1.printStackTrace();
    }
    System.out
                    .println("\n"
                                    + topDocs.totalHits
                                    + " results for query \""
                                    + booleanQuery.toString()
                                    + "\" showing for the first "
                                    + 10
                                    + " documents the doc id , "
                                    + "score and the content of"
                                    + " the modelDescription field");
```

```
                for (int i = 0; i < Math.min(10, topDocs.totalHits); i++) {
                        try {
                                System.out.println(topDocs.scoreDocs[i].doc
                                + " - score: "
                                + topDocs.scoreDocs[i].score
                                + " - "
                                + reader.document(topDocs.scoreDocs[i].doc).get(
                                                "modelDescription"));
                        } catch (CorruptIndexException e) {
                                System.out.println("Graceful message: exception " + e);
                                e.printStackTrace();
                        } catch (IOException e) {
                                System.out.println("Graceful message: exception " + e);
                                e.printStackTrace();
                        }

                }

                try {
                        // searcher.close(); It was necessary in Lucene 3.x
                        reader.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

}
```