

Lucene.

Sistema de RI de
código abierto:
Búsquedas

Conceptos fundamentales

- Los conceptos fundamentales en Lucene son index, document, field y term.
- Un índice (index) contiene una secuencia de documentos.
 - ♦ Un Document es una secuencia de Fields.
 - ♦ Un Field es una secuencia Term.
 - ♦ Un Term es una secuencia de bytes.
- La misma secuencia de bytes en dos campos diferentes se considera un término diferente. Así terms se representan como un par: la cadena con el nombre del campo y los bytes dentro del campo.

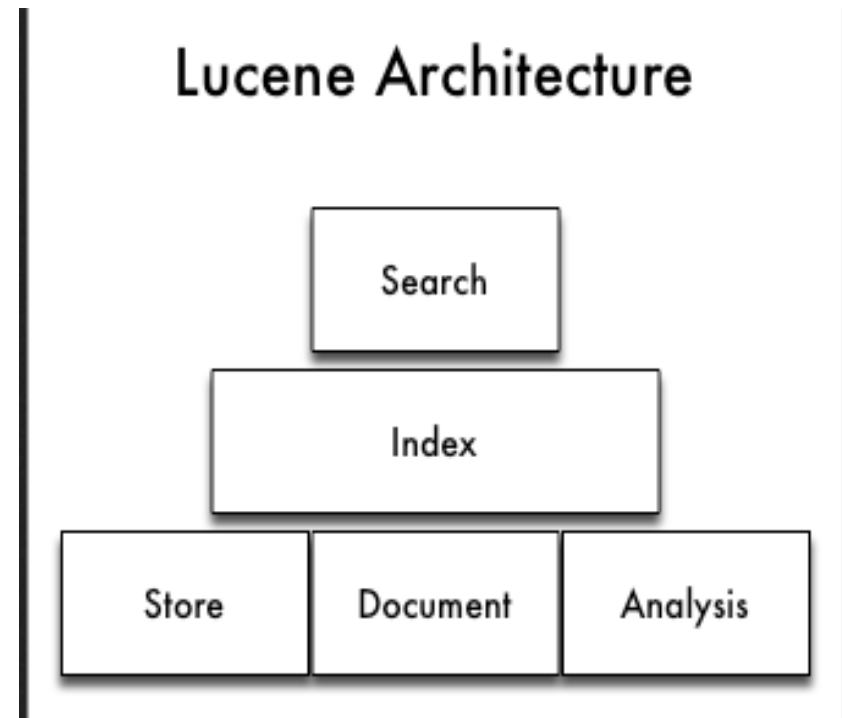
El índice Invertido

- El índice almacena estadísticas sobre los términos con el fin de hacer la búsqueda más eficiente. Dado un término se puede enumerar los documentos que lo contienen.
 - ♦ Tipos de campos: STORED, INDEXED, TOKENIZED
- Los índices de Lucene pueden estar compuestas de múltiples sub-índices, o **segmentos** . Cada segmento es un índice completamente independiente, puede ser buscado por separado.
- Los índices evolucionan a través de:
 - ♦ La creación de nuevos segmentos con nuevos docs
 - ♦ La fusión de segmentos existentes .
- Las búsquedas pueden incluir múltiples segmentos y/o varios índices, cada índice potencialmente compuesto por un conjunto de segmentos.

Recordemos

Para usar Lucene, una aplicación debería:

- Indexar:
 - Crear Documents añadiéndole Fields;
 - Crea un IndexWriter y añadir documentos con addDocument();
- Buscar:
 - Abrir el Índice
 - Llamar a QueryParser.parse() para construir una consulta desde un string
 - Buscar en el índice
IndexSearcher.search()



Apache Lucene: Indexación

```
Analyzer analyzer = new StandardAnalyzer();

// Store the index in memory:
Directory directory = new RAMDirectory();
// en disco ... //Directory directory = FSDirectory.open("/tmp/testindex");
IndexWriterConfig config = new
    IndexWriterConfig(analyzer);
IndexWriter iwriter = new IndexWriter(directory, config);
Document doc = new Document();
String texto = "En un lugar de la Mancha de cuyo ... .";
TextField cuerpo = new TextField("nCampo", texto, Field.Store.NO);
TextField autor = new TextField("nAutor", "Miguel de Cervantes",
Cervantes", Field.Store.YES);
doc.add(cuerpo);
doc.add(autor);
iwriter.addDocument(doc);
iwriter.close();
```

Apache Lucene: Búsqueda

.....

```
IndexReader ireader = DirectoryReader.open(directory);
IndexSearcher isearcher = new IndexSearcher(ireader);
// Parse a simple query that searches for "text":
QueryParser parser = new
    QueryParser("nCampo", analyzer);
Query query = parser.parse("lugar de la mancha"); //TEXTO A BUSCAE
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("salida "+hitDoc.get("nAutor").toString());
    System.out.println("salida "+hitDoc.toString());
}
ireader.close();
directory.close();
```

Profundizando en la búsqueda

Photoshop PSD file download · Resolution 1280x1024 px · www.psdgraphics.com



Esquema de búsqueda

- La búsqueda requiere un índice ya construido.
- Crear una consulta, [Query](#) Lucene implementa una gran variedad de alternativas para la consulta, que se pueden combinar para construir consultas complejas usando información posicional.
 - ♦ Por lo general, a través de [QueryParser](#), [MultiPhraseQuery](#), etc que analiza la entrada del usuario
 - ♦ Es importante que las consultas usen el mismo [Analyzer](#) que el que se utilizó cuando se creó el índice.
- Leer el Índice, [IndexReader](#)
- Buscar en el Índice, por ejemplo a través de [IndexSearcher](#)
 - ♦ [IndexSearcher.search \(Query, int\)](#)
 - ♦ [IndexSearcher.search \(Query, int, Filtro\)](#)
- Iterar a través de documentos devueltos, [Hits](#)
 - ♦ Extraer los elementos a mostrar al usuario, junto a los scores

Clases importantes en Búsqueda

- `IndexReader`
 - Lleva el índice a memoria
- `Searcher`
 - Proporciona los métodos de búsqueda
 - `IndexSearcher`, `MultiSearcher`, `ParallelMultiSearcher`
- `TopDocs`
 - Almacena los resultados de la búsqueda
- `QueryParser`
 - Gramática en JavaCC para crear las consultas
- `Query`
 - Representación lógica de las necesidades de inf.
- `Term`
 - Representa la unidad básica de búsqueda

Esquema del tema

- **Query:** Clases que manipulan la consulta Query
- **QueryParser:** Analiza la consulta del un usuario
- **IndexSearch:** Estudiaremos el proceso de búsqueda
- **IndexReader:** Analizaremos el Indice
- **Similarity:** Analizaremos la función de similitud de Lucene

Lucene Query

Clases de Consulta

- TermQuery
- BooleanQuery
- PhraseQuery
- TermRangeQuery
- NumericRangeQuery
- PrefixQuery
-



TermQuery

- Es el tipo de consulta más fácil de entender y el más utilizado en las aplicaciones. Una TermQuery devuelve todos los documentos que contienen el término (Term(“Ncampo”,”texto”)) en el Field Ncampo especificado
- La construcción de un TermQuery es tan simple como:

```
IndexSearcher searcher;
```

```
....
```

```
TermQuery tq = new TermQuery(new Term("fieldName", "term"));  
... searcher.search(tq, 5);
```


BooleanQuery

- Permite combinar multiples TermQuery en una consulta booleana, mediante el uso de cláusulas BooleanClauses, donde cada cláusula contiene una sub-consulta (instancia Query) y un operador (BooleanClause.Occur) que describe cómo ese sub-consulta se combina con las otras cláusulas:
 - SHOULD – Es un O-lógico.
 - MUST - Y-lógico.
 - MUST_NOT Negación
 - FILTER como MUST pero no se usan en el cálculo del score

```
BooleanQuery.Builder Constructor = new BooleanQuery.Builder();
```

```
Constructor.add(query1, BooleanClause.Occur.MUST);
```

```
Constructor.add(query2, BooleanClause.Occur.MUST)
```

```
BooleanQuery q = Constructor.build();
```

PhraseQuery

- Una consulta en la que los documentos relevantes deben contener una secuencia particular de términos.
- Esta consulta puede ser combinado con otros términos o consultas con un BooleanQuery.
- Una PhraseQuery se construye normalmente a través de QueryParser, tomando como entrada una cadena entre comillas como "Nueva York".
- Podemos controlar el número de palabras que se pueden permitir entre los términos en la frase, utilizando setSlop:
 - 0 indica búsqueda exacta (por defecto)

PhraseQuery(2, "Titulo", "Quijote", "Mancha");

Crea consulta con los términos "Quijote", "Mancha" en el campo Título con una distancia máxima de edición de 2.

```
PhraseQuery.Builder Constructor = new PhraseQuery.Builder();  
Constructor.add(term1); Constructor.add(term2); // "term1 term2"  
Constructor.setSlop(slop);  
PhraseQuery q = Constructor.build();
```

```
.... = indexSearcher.search(phraseQuery);
```

TermRangeQuery

- Una consulta cuyos documentos relevantes contienen los términos dentro de un rango (utilizan como criterio `Byte.compareTo (Byte)`)
- No vale para comparaciones numéricas!!!

Por ejemplo,

```
TermRangeQuery trq;
```

```
trq = trq.newStringRange("filename","cap100.txt","cap120.txt",true,false);
```

- `TermRangeQuery newStringRange(String field, String lowerTerm, String upperTerm, boolean includeLower, boolean includeUpper)`

Consultas Numéricas Por Rango

- Una consulta que coincide con los valores numéricos dentro de un rango especificado.
- Para utilizarla, es necesario que los campos hayan sido indexados considerando valores numéricos (utilizando IntPoint, LongPoint,...).
 - newExactQuery(String, int) for matching an exact 1D point.
 - newSetQuery(String, int...) for matching a set of 1D values.
 - newRangeQuery(String, int, int) for matching a 1D range.
- Ejemplo:

Query Q = IntPoint.newRangeQuery ("peso", 10, 20);

Devuelve los documentos con valor de “peso” dentro de los rangos, inclusive.

- El rendimiento de NumericRangeQuery es mucho mejor que el TermRangeQuery debido a que el número de elementos que deben ser buscados suele ser mucho menor

•

Importancia del campo Boost

- Lucene permite influir en los resultados de una búsqueda
 - ♦ En tiempo de indexación: llamando `Field.setBoost ()` antes de que un documento se añada al índice.
 - ♦ Impulso en tiempo de consulta: llamando a `Query.setBoost ()` para impulsar una cláusula
- Multiplica por el `getBoost()` todos los emparejamientos
- El boost también se tiene en cuenta a la hora de computar la norma del documento (para normalizar el score)

QueryParser

Parse::



QueryParser

- Aunque Lucene proporciona la capacidad de crear sus propias consultas, también proporciona un lenguaje de consulta a través de QueryParser (Analizador de consultas) , un analizador léxico transforma una cadena (string) en una consulta Lucene
- Está especialmente diseñado para consultas dadas por el usuario a través de una cadena, string. Si la consulta se genera “programándola” es mejor utilizar las API de consulta.
- Los campos NOTOKENIZADOS, como por ejemplo fechas, claves, campos con un conjunto limitado de valores (generados por un desplegable) , etc es mejor añadirlos directamente a la consulta. (No deben pasar por el analizador)
- La sintaxis analizador de consultas varía entre versiones de Lucene

QueryParser

- Métodos más importantes:
- Constructor:

`QueryParser(String f, Analyzer a)`

f – el Field por defecto sobre el que se busca
a – Analizador utilizado para indexar

- `Query parse(String cadena)`

Devuelve la consulta Lucene representada en cadena. Cadena debe estar escrita en el lenguaje de consulta Lucene

Expresiones válidas

Table 3.2 Expression examples that `QueryParser` handles

Query expression	Matches documents that...
<code>java</code>	Contain the term <i>java</i> in the default field
<code>java junit</code> <code>java or junit</code>	Contain the term <i>java</i> or <i>junit</i> , or both, in the default field ^a
<code>+java +junit</code> <code>java AND junit</code>	Contain both <i>java</i> and <i>junit</i> in the default field
<code>title:ant</code>	Contain the term <i>ant</i> in the <code>title</code> field
<code>title:extreme</code> <code>-subject:sports</code> <code>title:extreme</code> <code>AND NOT subject:sports</code>	Have <i>extreme</i> in the <code>title</code> field and don't have <i>sports</i> in the <code>subject</code> field
<code>(agile OR extreme) AND methodology</code>	Contain <i>methodology</i> and must also contain <i>agile</i> and/or <i>extreme</i> , all in the default field
<code>title:"junit in action"</code>	Contain the exact phrase " <i>junit in action</i> " in the <code>title</code> field
<code>title:"junit action"~5</code>	Contain the terms <i>junit</i> and <i>action</i> within five positions of one another
<code>java*</code>	Contain terms that begin with <i>java</i> , like <i>javaspaces</i> , <i>javaserver</i> , and <i>java.net</i>
<code>java~</code>	Contain terms that are close to the word <i>java</i> , such as <i>lava</i>
<code>lastmodified:</code> <code>[1/1/04 TO 12/31/04]</code>	Have <code>lastmodified</code> field values between the dates January 1, 2004 and December 31, 2004

^a The default operator is *OR*. It can be set to *AND* (see section 3.5.2).



Búsquedas simples:

<code>free AND "text search"</code>	Search for documents containing "free" and the phrase "text search"
<code>+text search</code>	Search for documents containing "text" and preferentially containing "search"
<code>giants -football</code>	Search for "giants" but omit documents containing "football"
<code>author:gosling java</code>	Search for documents containing "gosling" in the author field and "java" in the body

Lucene



Búsquedas con caracteres comodines:

- ? = Un único carácter (te?t = {text, test}).
- * = Varios caracteres (test* = {test, tests, tester}).
- Se pueden emplear combinados y como primer carácter del término.

Lucene



Búsquedas vagas:

- Se emplea el símbolo "~" al final de un término. Indica “parecido a”, “similar a” (usa la distancia Damerau-Levenshtein (emparejamiento optimal de strings))
- Ejemplo: `roam~` devolvería “foam” y “roams”.
- Se puede indicar un número real entre 0 y 1. Cuanto más próximo sea, más similar será con el término (`roam~0.8`). Por defecto, 0.5.

Lucene



Búsquedas por proximidad:

- Puede encontrar documentos en los que los términos especificados estén una distancia concreta unos de otros:
- Por ejemplo, "Jakarta apache"~10

Lucene



Consultas por rango:

- Permite hacer la correspondencia de documentos cuyos valores de los campos especificados estén entre un límite inferior y otro superior.
- Por ejemplo: `mod_date:[20020101 TO 20030101]` ó `title:{Aida TO Carmen}`
- Los límites exclusivos se nota con (y), mientras que los inclusivos con [y].

Lucene



Aumento de la importancia de un término en la consulta:

- Para incrementar la relevancia de un término para el usuario se utiliza el operador “^” que sigue al término y un número. Cuanto más alto sea, más importancia tendrá el término en la búsqueda.
- Por ejemplo, en lugar de la consulta: Jakarta apache
- Podemos indicar que le damos más importancia a “jakarta” mediante: Jakarta^4 apache.
- Los documentos donde aparezca “jakarta” serán más relevantes.
- También se pueden aumentar expresiones.



Para responder a cada una de estas consultas podemos utilizar la clase

IndexSearch

que es la que se encarga de computar el score para cada doc.

Realizar las consultas en Lucene es extraordinariamente rápido y esconde la casi totalidad de la complejidad al usuario.

En pocas palabras, funciona.

Clases importantes en Búsqueda

- Search
 - Clase abstracta que implementa los métodos principales de búsqueda.

IndexSearcher

- Implementa las búsquedas sobre un `IndexReader`.
El constructor recibe un `IndexReader`

```
IndexSearcher searcher = new IndexSearcher(reader);
```
- Una aplicación típica sólo necesita llamar a los métodos
 - `search(Query q, int n)`
 - `search(Query q, int n, Sort s)`
- Se recomienda abrir un único `IndexSearcher` y utilizarlo en todas las búsquedas

```
Import org.apache.lucene.search.IndexSearcher;
```

IndexSearch: Algunos métodos

Además de buscar, vía **search(...)**, la clase IndexSearch nos permite considerar componentes de una búsqueda

- **Document doc(int docID):** Nos devuelve del índice el documento docID
- **Explanation explain(Query q, int docId)** Nos explica cómo se computa el score de la consulta q para el docID
- **void setSimilarity(Similarity s):** Nos permite indicar la función de similaridad que se utiliza para calcular el score
 - TFIDFSimilarity (version del modelo de espacio vectorial)
 - BM25Similarity,
 - MultiSimilarity, (CombSum para combinar evidencias)
 - PerFieldSimilarityWrapper (usa una similaridad por campo)

Clases importantes en Búsqueda:

TopDocs

- Estructura que almacena los resultados de la búsqueda, representa una array ordenado de documentos, según relevancia a la consulta.
- Atributos de TopDocs:
 - ♦ **ScoreDocs** : Vector con los top hits
 - Un hits contiene un
 - Doc, numero de doc
 - Score el score de este doc
 - ♦ **TotalHits**: El numero total de hits

Ejemplo Search/TopDocs

```
TopDocs hits = searcher.search(query, 100);
```

```
System.out.println(hits.totalHits + " docs encontrados " );
```

```
for (ScoreDoc sd : hits.scoreDocs) {  
    Document d = searcher.doc(sd.doc);  
    System.out.println(d.get("titulo"), d.get("nfichero"));  
}
```

Imprime los docs con mejor score

```
System.out.println("Explicamos "+  
    searcher.explain(query, docId).toString());32
```


Personalizando las búsquedas....

- Podemos ordenar la salida considerando cualquier sobre los datos almacenados.

`TopFieldDocs search(Query query, int n, Sort sort)`

- Devuelve los top n docs, ordenados segun sort
 - ♦ Los campos utilizados para determinar el orden deben tener un único término, que determina la posición relativa en el orden
 - ♦ El campo debe indexarse, NO tokenizarse, y no necesita estar almacenado
- Atributos de TopFieldsDocs
 - ♦ `totalHits`
 - ♦ `ScoreDocs`
 - ♦ `SortField[] fields`; campos utilizados para ordenar

Ejemplo

```
SortField sf = new SortedNumericSortField("tama",SortField.Type.INT);  
sf.setMissingValue(0);  
Sort orden = new Sort(sf);  
TopDocs td = searcher.search(q, 5, orden);
```

```
ScoreDoc[] hits_tama = td.scoreDocs;  
System.out.println("Found " + hits_tama.length + " hits.");  
for(int i=0;i<hits_tama.length;++i) {  
    int docId = hits_tama[i].doc;  
    Document d = searcher.doc(docId);  
    System.out.println((i + 1) + ". " + d.get("path") + " score=" +  
        hits_tama[i].score+" sz " +d.get("tamaño"));  
}
```

Facetas (categorías) en Lucene

- Una faceta (categoría) puede ser utilizado por Lucene para clasificar documentos
 - ♦ Libros: autor, precio,
- En una búsqueda por facetas, además del conjunto estándar de resultados de búsqueda, también obtenemos listas por subcategorías.
 - ♦ Por ejemplo, para la faceta autor, se obtiene una lista de autores relevantes,
 - ♦ Cuando los usuarios hacen clic en estas subcategorías, se restringe la búsqueda
- En esencia, la búsqueda por facetas hace facilita la navegación a través de los resultados de búsqueda.
- Útil en e-comercio.

Doc: https://lucene.apache.org/core/6_2_1/facet/index.html

Shop by
Department ▾

Search All ▾ samsung

Go

Hello
You

Departments

Cell Phones & Accessories

- Unlocked Cell Phones
- Cell Phone Cases
- Smart Watches & Accessories

Electronics

- Televisions
- LED TVs
- LCD TVs
- Audio & Video Accessories
- Televisions & Video Products

Computers & Accessories

- Computer Tablets
- Laptop Computers
- Computer Components

+ See All 35 Departments

Eligible for Free Shipping

Free Shipping by Amazon

Brand

- ☐ Samsung
- ☐ Generic
- ☐ eForCity
- ☐ ULAK
- ☐ Importer520
- ☐ Empire
- ☐ Fosmon Technology
- ☐ MoKo
- ☐ Skinit
- ☐ Fintie
- ☐ MyBat

Gifts from \$25



"samsung"

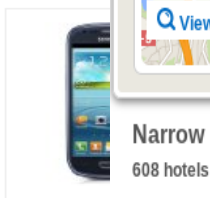
Related Searches: [samsung galaxy s4](#), [samsung s4](#), [samsung tv](#)

Showing 1 - 16 of 9,299,531 Results

Amazon's Samsung Store



Samsung

Unlocked Cell
PhonesCell Ph
Acces

Samsung Galaxy Tab 3 (7")

~~\$199.99~~ Click to see price Prime
In stock on December 20, 2013

More Buying Choices
\$149.99 used & new (57 offers)

Narrow results:

608 hotels

Name contains

Hotel name...

Price (total)

\$0 to \$500+



Star rating

- ☐ ★★★★★
- ☐ ★★★★
- ☐ ★★★
- ☐ ★★
- ☐ ★

Guest rating

0 to 5



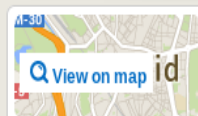
Neighbourhood

- ☐ Gran Via
- ☐ Puerta del Sol
- ☐ Barajas
- ☐ Salamanca

Hotels.com®
Wake Up Happy®

Home Deals Welcome Rewards®

Customize



Madrid, Spain Sat 14 - Sun 15, December 2013, 1 night 1 room, 2 adults,

Most popular

Star ratings

Distance

Guest ratings

Hotel Atlantico Madrid

Last booked 13 hours ago

Gran Via 38 Madrid, Madrid, 28013 Spain, 0044 203 564 5228



★★★★★

Gran Via

0.23 mi to city centre

Welcome Rewards

Outstanding
4.6 / 52,046
customer reviews

Regina

Last booked 7 hours ago

Alcala 19 Madrid, Madrid, 28014 Spain, 0044 203 564 5228



★★★★★

Gran Via

0.21 mi to city centre

Welcome Rewards

Excellent
4.3 / 51,293
customer reviews

Apartamentos Recoletos

Last booked 17 hours ago

Calle de Villanueva, 2 Madrid, Madrid, 28001 Spain, 0044 203 564 5228



★★★★★

Salamanca

0.79 mi to city centre

Welcome Rewards

Good
3.3 / 532
customer reviews

Indexar

- Para buscar por facetas es necesario indexar previamente
- Para cada documento de entrada:
 - ♦ Crear un nuevo Documento Lucene
 - ♦ Analizar el texto de entrada y agregar los campos de búsqueda de texto apropiados
 - ♦ Obtener las categorías asociadas al documento y crear un CategoryDocumentBuilder con la lista de categorías
 - ♦ Construir el documento - esto agrega las categorías al documento Lucene.
 - ♦ Añadir el documento al índice

Ejemplo (Indexacion):

.../lucene-6.2.1/docs/demo/src-
[html/org/apache/lucene/demo/facet/SimpleFacetsExample.html](http://org.apache.lucene/demo/facet/SimpleFacetsExample.html)

```
Directory indexDir = new RAMDirectory();  
Directory taxoDir = new RAMDirectory();  
FacetsConfig config = new FacetsConfig();
```

```
IndexWriter indexWriter = new IndexWriter(indexDir,  
    new IndexWriterConfig( new Analyzer(XX)).setOpenMode(XX ));
```

```
// Writes facet ords to a separate directory from the main index  
DirectoryTaxonomyWriter taxoWriter = new DirectoryTaxonomyWriter(taxoDir);
```

```
Document doc = new Document();  
doc.add(new FacetField("Author", "Bob"));  
doc.add(new FacetField("Publish Date", "2010", "10", "15"));  
indexWriter.addDocument(config.build(taxoWriter, doc));
```

```
doc = new Document();  
doc.add(new FacetField("Author", "Lisa"));  
doc.add(new FacetField("Publish Date", "2010", "10", "20"));  
indexWriter.addDocument(config.build(taxoWriter, doc));
```

```
indexWriter.close();  
taxoWriter.close();
```

Búsqueda ... (agrupando facetas)

- Usar Facetas permite ver los resultados de búsqueda como
 - Un conjunto de documentos - un subconjunto de los documentos del índice que coincidan con la consulta
 - Analizando las Facetas - Por ejemplo, contar con una cierta dimensión de la faceta.

```
DirectoryReader indexReader = DirectoryReader.open(indexDir);
```

```
IndexSearcher searcher = new IndexSearcher(indexReader);
```

```
TaxonomyReader taxoReader = new DirectoryTaxonomyReader(taxoDir);
```

```
FacetsCollector fc = new FacetsCollector();
```

```
FacetsCollector.search(searcher, query, 10, fc); // to search and collect all hits into the provided Collector
```

```
List<FacetResult> results = new ArrayList<>();
```

```
// Count both "Publish Date" and "Author" dimensions
```

```
Facets facets = new FastTaxonomyFacetCounts(taxoReader, config, fc);
```

```
results.add(facets.getTopChildren(10, "Author"));
```

```
results.add(facets.getTopChildren(10, "Publish Date"));
```

```
indexReader.close();
```

```
taxoReader.close();
```

Profundicemos en el Indice...

IndexReader



- Como hemos visto, Lucene nos permite desarrollar un sistema de recuperación de forma eficiente

En pocas palabras, funciona.

Por lo menos, hasta que no funciona, o no funciona como es de esperar que funcione. Entonces nos queda más remedio que ver lo que está pasando por dentro

La clase IndexReader

- Una clase que proporciona los métodos básicos para acceder al índice y a los Fields almacenados cuando se muestra la lista de resultados.
- Desde la versión 4.0 NO es posible recuperar términos o lista de ocurrencias a través del índice. Para acceder a ellos se debe hacer a través de las subclases
 - ♦ AtomicReader (atómico)
 - ♦ CompositeReader (múltiple readers)
- Para crear una instancia de un IndexReader sobre un índice en disco se construyen haciendo la llamada a método DirectoryReader.open(), se le puede pasar al IndexSearcher.
- Los documentos en la clase IndexReader se identifican por un entero, (docId)

```
Import org.apache.lucene.index.IndexReader
```

Ejemplo de uso

```
String dir = "Colecciones/index_ObrasCervantes";
```

```
...
```

```
IndexReader reader = DirectoryReader.open(FSDirectory.open(new  
File(dir)));
```

```
IndexSearcher searcher = new IndexSearcher(reader);
```

```
....
```

```
Document d = reader.document(docId);
```

```
reader.numDeletedDocs()
```

```
Int n = reader.numDocs()
```

```
reader.close();
```

Estructura del Índice en Lucene

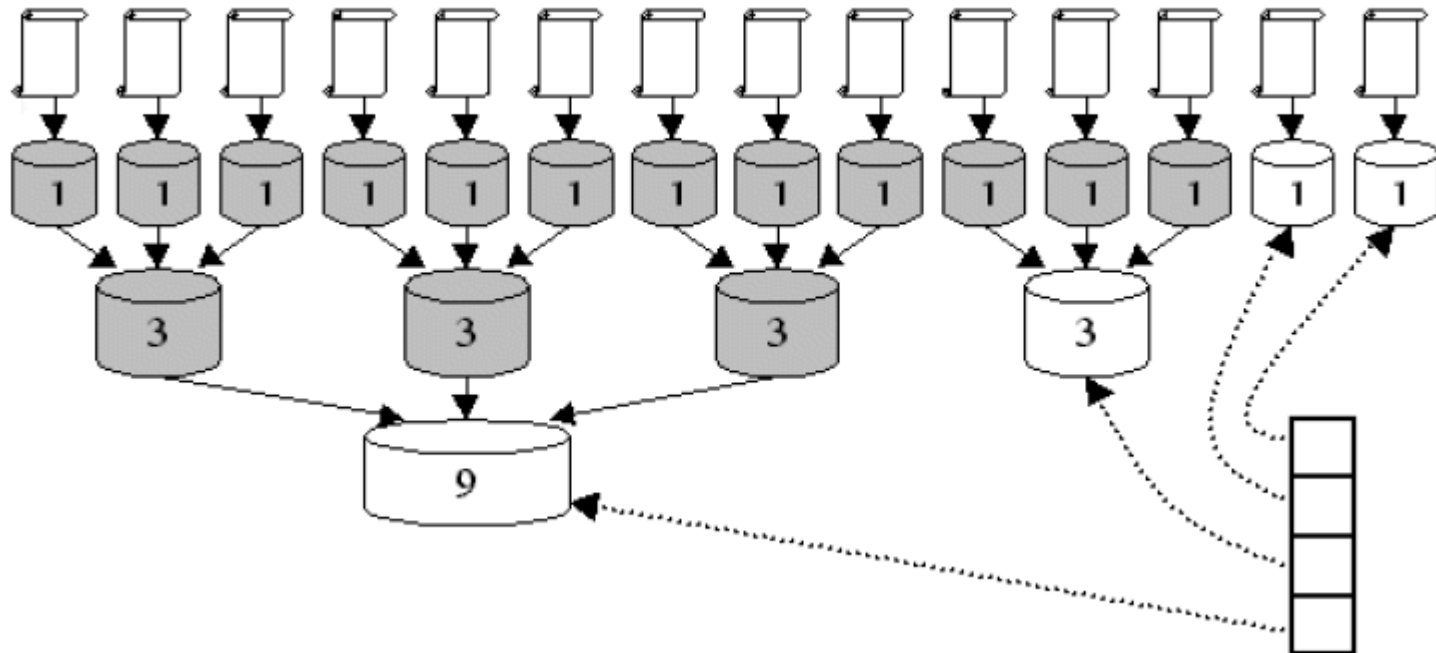
- Lucene fue el primer motor de búsqueda que soportó añadir y actualizar documentos, garantizando la coherencia del índice.
- Para ello, utiliza
 - una estructura de índices por segmentos
 - Al hacer un cambio en el índice se crean nuevos segmentos en Lucene.

Elementos de un Segmento:

- **Información de segmento.** metadatos como el número de documentos, archivos que utiliza, etc.
- **Nombres de campo**
- **Stored fields:** para cada documento , una lista de pares field – valor. Util para url, nombre fichero, etc. Clave DocId
- **Diccionario de términos.** Un diccionario que contiene todos los términos utilizados en todos los campos de todos los documentos. El diccionario también contiene el número de documentos que contienen el término, y los punteros a la frecuencia del término y los datos de proximidad.
- **Frecuencia de término.** Los documentos que contienen ese término, y la frecuencia del término en este documento
- **Datos de proximidad del término.** las posiciones que en las que aparece
- **Factores de normalización.** Para cada campo de cada documento, se guarda un valor de normalizacion (se multiplica por el score tras consulta)
- **Term Vector (opcional).** Para cada campo en cada documento, el término vector (a veces llamado vector documento)
- **Los documentos eliminados.**

Segmentos en Lucene

- Cada índice se compone de varios segmentos colocados en el directorio de índice.
- Todos los documentos son añadidos a nuevos segmentos, los cuales se mezclan (fusionan) con otros archivos en disco
- Se escriben los segmentos de forma incremental y se mezclan
- Optimizar un índice => un segmento









Visualización de mezcla de segmentos con Lucene.







- Indexando la Wikipedia en Inglés
- Video en: Mike McCandless' blog,
- <http://goo.gl/kI53f>

Los índices Lucene (hasta ver. 3.6)













- Cada segmento ("índice atómico") es un índice completamente funcional:
 - SegmentReader implementa la interfaz IndexReader para segmentos individuales
- Los índices compuestos:
 - DirectoryReader implementa la interfaz IndexReader sobre un conjunto de SegmentReaders
 - Multireader es una abstracción de múltiples IndexReaders combinado en un índice virtual
 - Term Dictionary: sobre la marcha mezcla y ordena los términos del índice
 - Postings: se añaden las listas de ocurrencias para cada término, haciendo los identificadores de doc globales

Mezcla de terminos y posting

Term 1	
Term 2	
Term 4	
Term 7	
Term 8	
Term 9	

Term 1	
Term 3	
Term 5	
Term 6	
Term 8	
Term 9	



Term 1		
Term 2		
Term 3		
Term 4		
Term 5		
Term 6		
Term 7		
Term 8		
Term 9		

Lucene 4.X: IndexReader

- Es la clase abstracta más general que proporciona la interfaz para acceder a un índice.
 - ♦ Superclase de tipos más específicos
 - ♦ No tiene constructor público
- NO sabe nada sobre el concepto de "índice invertido", NO tiene acceso a términos, ni posting, ... etc!
- Permite el acceso a los Fields almacenados por el docID: Útil para mostrar sólo los resultados de la búsqueda
- Permite conocer algunos metadatos muy limitados: Número de documentos, ...
- No permite modificaciones (borrados)
- Puede pasarse como argumento a IndexSearcher, como siempre
- Permite IndexReader.open () para la compatibilidad con versiones anteriores (en desuso)

Ejemplo ...

```
String index = “.....”;
```

```
IndexReader reader = IndexReader.open(FSDirectory.open(new  
File(index) )); // No es correcta actualmente (DEPRECATED)
```

```
Document d = reader.document(docID);
```

```
Int n = reader.numDocs();
```

```
If (reader.hasDeletions() ) {....}
```

```
IndexSearcher searcher = new IndexSearcher(reader);
```

```
Query q = ....
```

```
resultados = searcher.search(query, ...);
```

```
....
```

Tipos de IndexReaders

Hay dos tipos distintos de IndexReaders

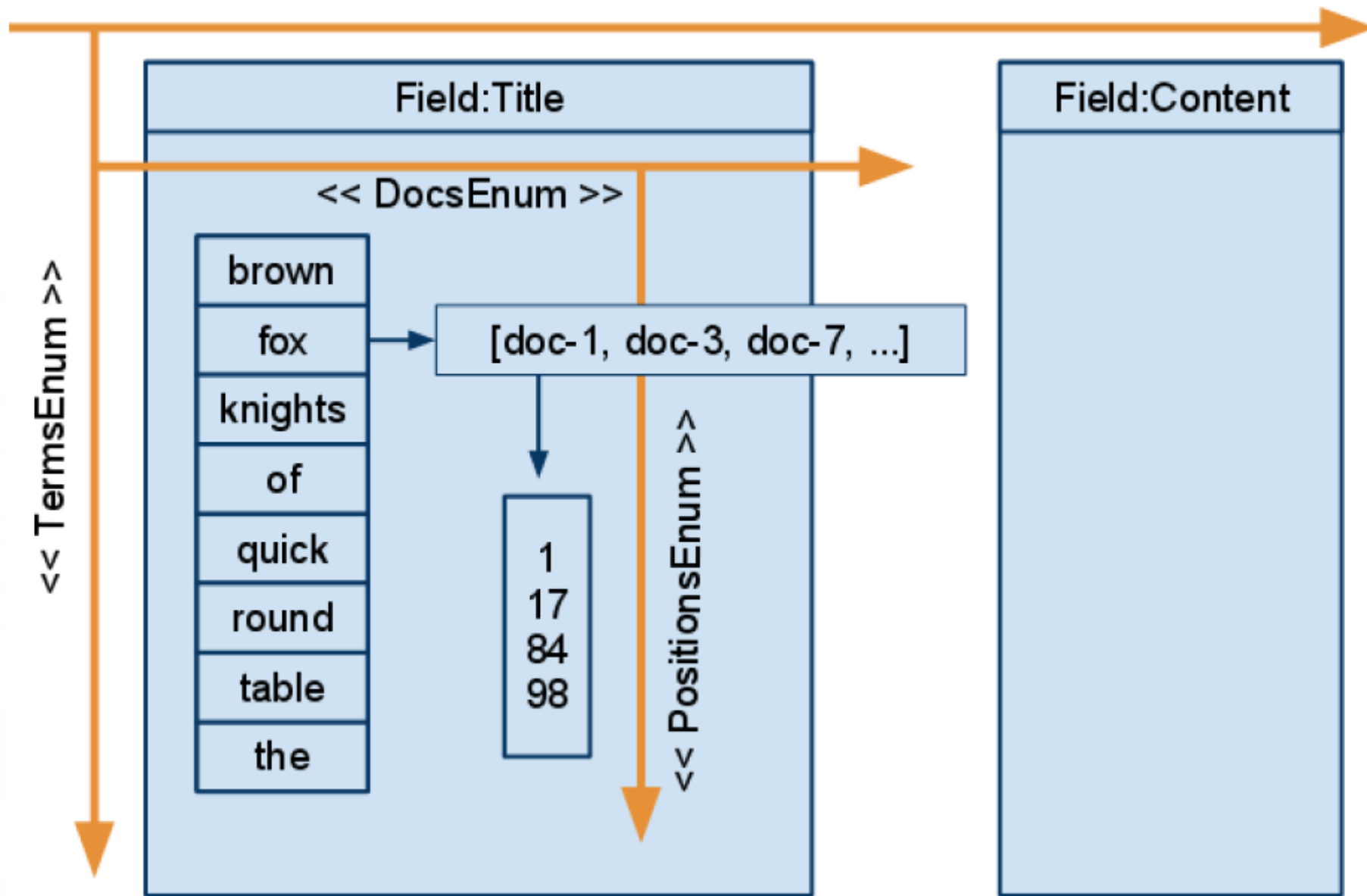
- **AtomicReader.** Estos índices son atómicos, permite el recuperar los Fields almacenados, los valores de doc, terminos y listans de ocurrencias.
- **CompositeReader:** Es un conjunto de AtomicReaders.
 - ♦ Una instancia de un CompositeReader (por ejemplo, DirectoryReader) sólo tiene acceso a los Fields almacenados en los distintos AtomicReaders.
 - ♦ Para construir un IndexReader de un índice en disco, usualmente se utiliza la llamada a **DirectoryReader.open()**

AtomicReader

- Hereda de IndexReader, permite el acceso al índice
- Controla el acceso a un índice atómico (segmentos individuales)
 - ♦ Permite recuperar los campos almacenados, docs, términos y listas de postings

AtomicReader

<< FieldsEnum >>



AtomicReader

- Mecanismos de acceso
 - ♦ **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.
 - ♦ **Long** **getSumDocFreq(String field)**
 - ♦

CompositeReader

- No tiene funcionalidades adicionales sobre IndexReader
- Proporciona getSequentialSubReaders () para recuperar todos los Readers hijos
- Esta clase se implementa como
 - ♦ [DirectoryReader](#)
 - ♦ [Multireader](#)

Alternativa + Costosa

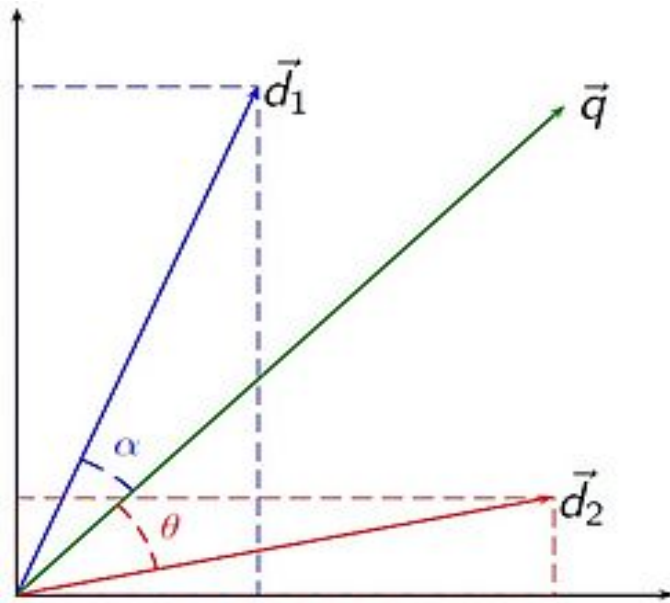
- **Empaquetar** los índices.
 - ♦ Considerar los IndexReaders de cualquier tipo como un AtomicReader, proporcionando acceso a términos, postings, borrado,, valores de doc

IndexReader r;

AtomicReader rd = SlowCompositeReaderWrapper.wrap(r);

- ♦ Hace que un reader compuesto (MultiReader o DirectoryReader) emule AtomicReader. Esto implica implementar la mezcla de postings, términos, etc. sobre la marcha. Internamente utiliza mismos algoritmos que los Readers de Lucene antiguos
- ♦ Fusiona los segmentos

Similarity: Función de Similitud



$$\text{sim}(d_j, q) = \frac{d_j \cdot q}{\|d_j\| \|q\|} = \frac{\sum_{i=1}^N w_{i,j} w_{i,q}}{\sqrt{\sum_{i=1}^N w_{i,j}^2} \sqrt{\sum_{i=1}^N w_{i,q}^2}}$$

Similaridad en Lucene

- Modelo conceptual (versión de la medida coseno)

$$\text{score}(q,d) = \text{coord-factor}(q,d) \cdot \text{query-boost}(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot \text{doc-len-norm}(d) \cdot \text{doc-boost}(d)$$

Cómo se calcula en la práctica:

$$\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \text{ in } q} (\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t,d))$$

$$\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \in q} (\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t,d))$$

- $\text{tf}(t \text{ in } d)$: por defecto $\text{frequency}^{1/2}$
- $\text{idf}(t)$: por defecto $\text{idf}(t) = 1 + \log (\text{numDocs} / (\text{docFreq} + 1))$
- $\text{coord}(q,d)$ Depende del cuantos términos de la consulta se encuentran en el documento, p.def: $\text{coord}(q,d) = \#(q \text{ and } d) / \#q$
- $\text{queryNorm}(q)$ factor de normalización para poder comparar entre distintas consultas., def: $\text{queryNorm}(q) = 1 / (q.\text{getBoost}() \cdot \text{sqrt}(\sum_t \text{wtq}^2))$

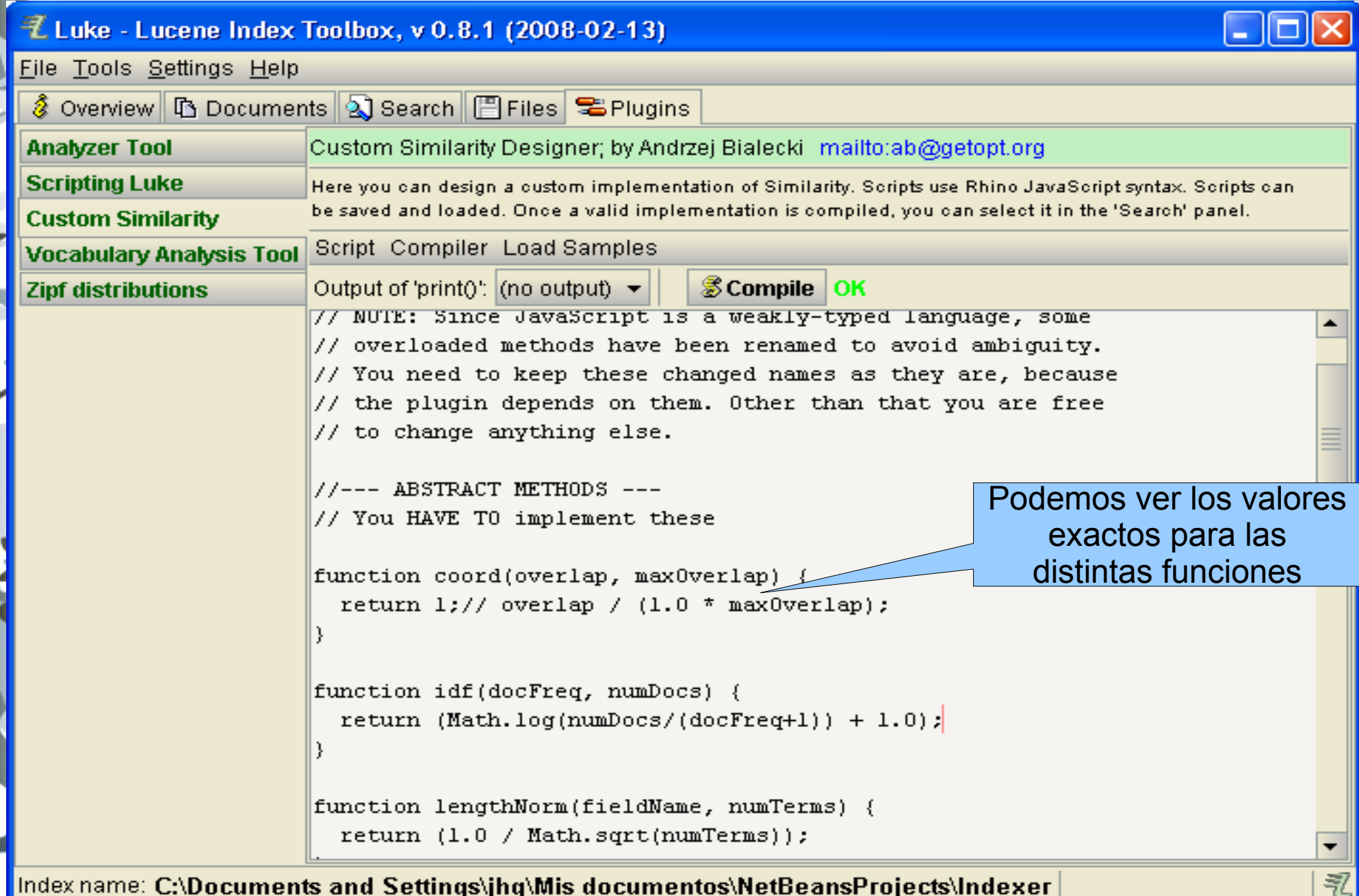
Por ejemplo en el caso de consulta booleana, se computa como

$$\text{wtq} = \text{idf}(t) \cdot t.\text{getBoost}()$$

- $t.\text{getBoost}()$ el boost del término en tiempo de búsqueda
- $\text{norm}(t,d)$ encapsula normalizacion en tiempo de indexación
 - Field boost – boost del campo
 - lengthNorm – se calcula cuando el documento se añade al índice, teniendo en cuenta el numero de tokens en cada campo.

$$\text{norm}(t,d) = \text{lengthNorm}(\text{field}(t) \text{ in } d) \cdot \prod (\text{field}(t) \text{ in } d) \text{ f.boost}()$$

Luke: consultar la función de similitud



Luke - Lucene Index Toolbox, v 0.8.1 (2008-02-13)

File Tools Settings Help

Overview Documents Search Files Plugins

Analyzer Tool

Scripting Luke

Custom Similarity


Vocabulary Analysis Tool

Zipf distributions

Custom Similarity Designer; by Andrzej Bialecki <mailto:ab@getopt.org>

Here you can design a custom implementation of Similarity. Scripts use Rhino JavaScript syntax. Scripts can be saved and loaded. Once a valid implementation is compiled, you can select it in the 'Search' panel.

Script Compiler Load Samples

Output of 'print()': (no output)  **Compile** **OK**

```
// NOTE: Since JavaScript is a weakly-typed language, some
// overloaded methods have been renamed to avoid ambiguity.
// You need to keep these changed names as they are, because
// the plugin depends on them. Other than that you are free
// to change anything else.

//--- ABSTRACT METHODS ---
// You HAVE TO implement these

function coord(overlap, maxOverlap) {
    return 1;// overlap / (1.0 * maxOverlap);
}

function idf(docFreq, numDocs) {
    return (Math.log(numDocs/(docFreq+1)) + 1.0);
}

function lengthNorm(fieldName, numTerms) {
    return (1.0 / Math.sqrt(numTerms));
}
```

Podemos ver los valores exactos para las distintas funciones

Index name: C:\Documents and Settings\jhq\Mis documentos\NetBeansProjects\Indexer

Nuevos Modelos de RI: modificar la funcion de similaridad

- En general `DefaultSimilarity` es suficiente. Pero en algunas aplicaciones podría ser necesario modificar el criterio de similaridad
 - Por ejemplo, no distinguir entre documentos cortos y largos
- La nueva definición debe tenerse en cuenta tanto a la hora de indexar como de buscar.
- Se debe implementar nuestra propia definición de similaridad (extender la clase (o subclases en `Similarity`) y utilizarla llamando a
 - `Searcher.setSimilarity (new Similarity..)`



Escalabilidad:

- Puede indexar unos 95GB/hora con un hardware moderno
- Requiere poca RAM -- sólo 1MB heap
- Indexación incremental es eficiente (tanto como hacerlo en batch)
- Tamaño del índice es aprox. 20-30% del tamaño del texto indexado.

<https://lucene.apache.org/core/6.2.1/benchmark/index.html>

Escalabilidad

- Query Rate
 - ♦ Lucene es rápido – Utiliza almacenamiento cache
 - ♦ Puede gestionar grandes cargas de trabajo (tiene un crecimiento casi lineal)
 - ♦ Podemos añadir más servidores de consultas
- Index size
 - ♦ Puede manejar fácilmente millones de documentos
 - ♦ Aunque el rendimiento de consulta puede degradarse, añadir documentos al índice tiene un lento factor de crecimiento.
 - ♦ Las principales limitaciones relacionadas con el tamaño del índice las encontramos con la capacidad del disco y los límites de E/S en disco.
 - ♦ Si necesitamos índices mayores,
 - ➔ Disponemos de métodos que permiten realizar consultas sobre varios índices remotos

Lucene



Concurrencia:

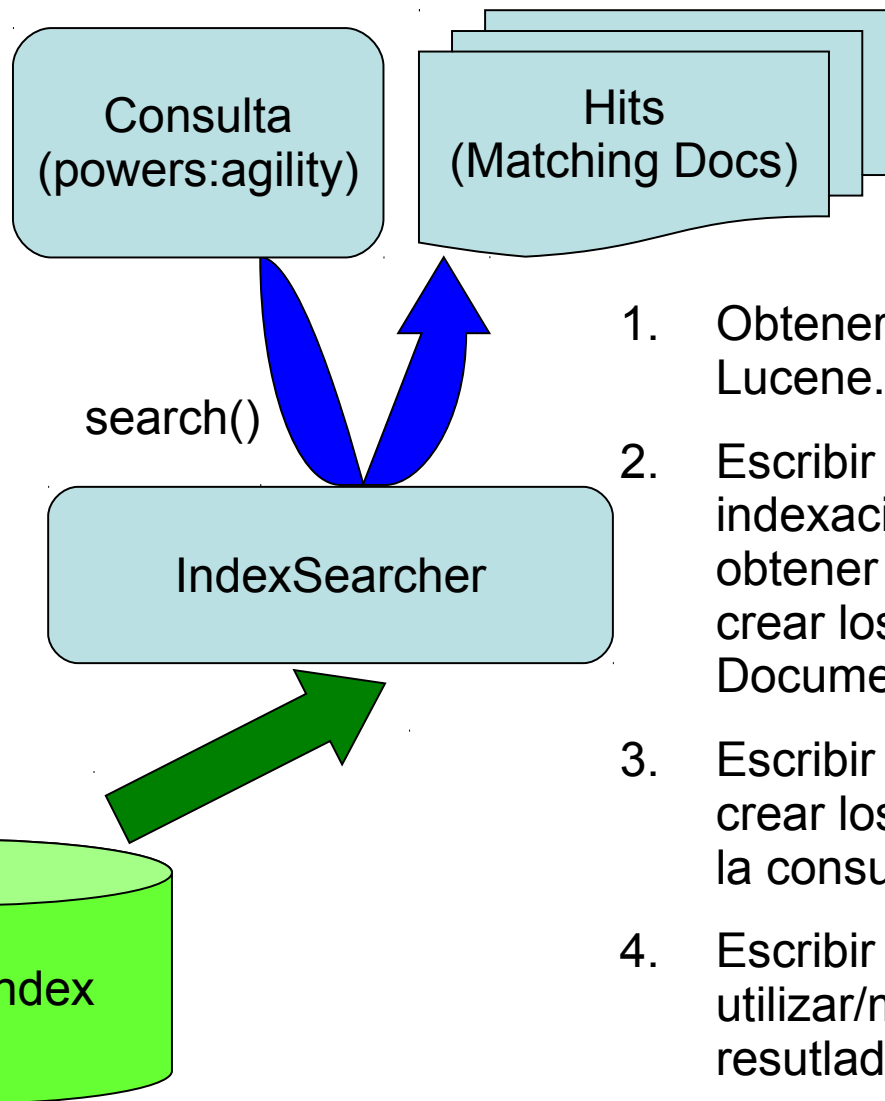
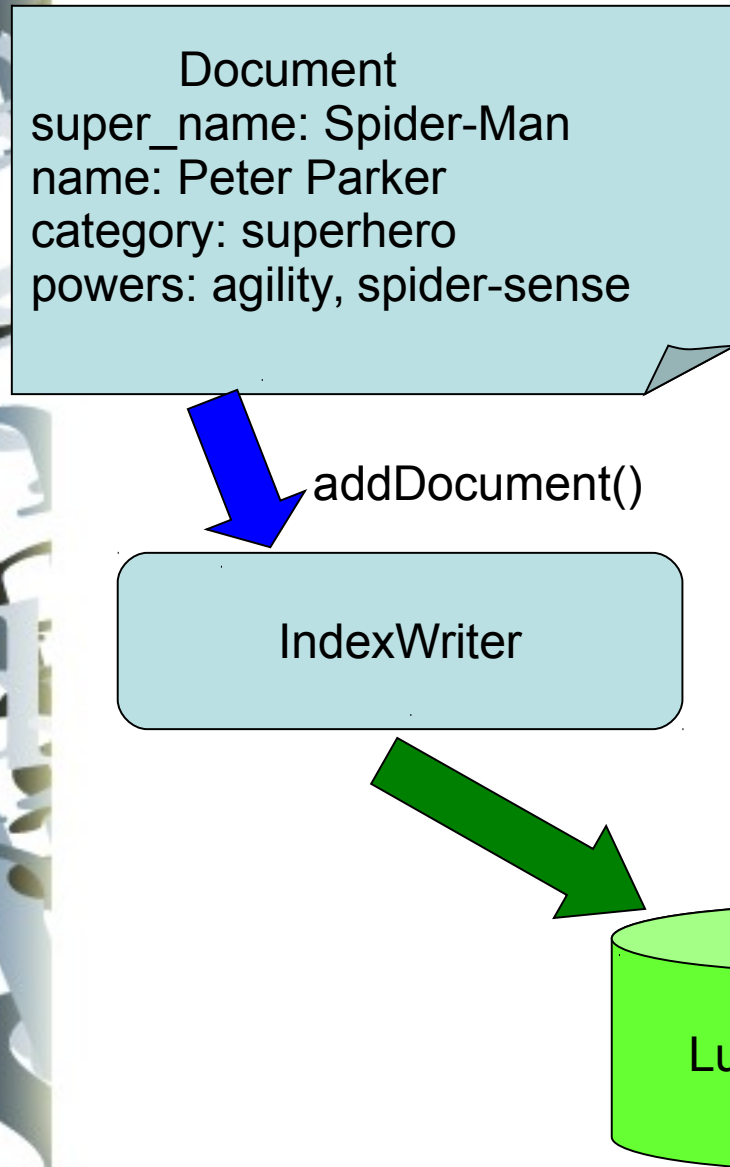
- Múltiples buscadores pueden acceder a los índices de Lucene a la vez.
- Los escritores o lectores de los índices de Lucene pueden editarlos mientras se realiza la búsqueda.
- Múltiples escritores o lectores de los índices pueden intentar editar a la vez.



Lucene

- Nutch es un software que implementa un buscador web.
- Está construido sobre Lucene y ofrece funcionalidades específicas como la araña, gestión de enlaces, parsers de HTML, etc.
- Realiza la gestión de temas complicados en los que no queremos programar y gastar tiempo.

Resumen



1. Obtener el .jar de Lucene.
2. Escribir el código de indexación para obtener los datos y crear los objetos Document.
3. Escribir código para crear los objetos de la consulta.
4. Escribir código para utilizar/mostrar resultados.

The Lucene Stack

