PRÁCTICA III:

...

Implementación de un Sistema de Recuperación de Información utilizando Lucene

...

Parte C: Búsqueda

4 de noviembre de 2021

Índice

1. Objetivo

El objetivo de esta práctica es profundizar en el uso de Lucene como herramienta para construir un SRI. En este caso, asumiremos que ya tenemos un índice creado, sobre el que un usuario desea encontrar los documentos relevantes a una necesidad de información. Lucene nos permite realizar una gran variedad de consultas (query), la mayoría de las cuales las tenemos implementadas en el paquete org.apache.lucene.search (ver documentación del paquete en `http://lucene.apache.org/core/7_1_0/core/org/apache/lucene/search/package-summary.html#package_description`). Los distintos tipos de consultas se pueden combinar para permitir un amplio abanico de posibilidades.

Para realizar una búsqueda, primero tenemos que crearnos una `Query` y dejamos al `IndexSearcher` que se encargue de computar la similitud entre los documentos y la consulta. Actualmente, Lucene utiliza por defecto como medida de similitud **BM25Similarity** con parámetros por defecto $k1 = 1{,}2$ y $b = 0{,}75$. Internamente Lucene asigna un único identificador a cada documento $docID$ cuando son añadidos al índice y permite asociarlo con los datos almacenados (Store.YES). Por tanto, como resultado de la búsqueda Lucene devolverá un conjunto de DocIDs ordenados por el valor de similitud (aunque es posible ordenar por otro criterio) denominado `TopDocs`.

```
1   IndexSearcher.search(Query,int);
```

2. IndexSearch

Antes de empezar la búsqueda debemos de obtener un IndexSearcher que es la clase encargada de realizar la búsqueda sobre el índice (ver documentación de la clase en `https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/search/IndexSearcher.html`. Dispone de distintos métodos para realizar la búsqueda y el conjunto de documentos más relevante los devuelve en un objeto de la clase TopDocs. Para realizar la búsqueda, el IndexSearch necesita un objeto de la clase IndexReader que se encarga de ver el

contenido del índice. Una vez abierto el índice, cualquier cambio que se realice en el mismo por un IndexWriter no será visible hasta que un nuevo IndexReader sea abierto.

Los pasos para obtener un IndexSearcher son los siguientes:

```
1  Directory dir = FSDirectory.open(Paths.get(INDEX_DIR));
2  IndexReader reader = DirectoryReader.open(dir);
3  IndexSearcher searcher = new IndexSearcher(reader);
```

En la primera línea obtenemos el directorio donde se encuentra el índice, que será abierto para lectura en la segunda línea. Cuando tenemos instanciado el IndexReader, podemos crearnos el objeto IndexSearcher.

Con el IndexSearcher podremos, entre otros:

- Consultar la medida de similitud (modelo de recuperación) `searcher.getSimilarity( )`. Por defecto es BM25Similarity.
- Modificar la medida de similitud `searcher.setSimilarity( Similarity sim)`
- Obtener un Document `searcher.doc(int docID)` del índice.
- Realizar una consulta `searcher.search(Query Q,int N)` devolviendo los top N documentos relevantes a Q considerando la medida de similitud.
- Explicar cómo se ha obtenido el escore de un documento ante la consulta `searcher.explain(Query q, int doc)`

3. Creación de Consultas Lucene con QueryParser

Pero el proceso esencial para poder responder a las necesidades de un usuario es la creación de una consulta (objeto del tipo `Query`), que se puede ver como un conjunto de clausulas. Quizás la forma más simple para construir una consulta es escribirla en un String, y dejar al QueryParser `http://lucene.apache.org/core/7_1_0/queryparser/org/apache/lucene/queryparser/classic/QueryParser.html` que se encargue de construirla. Es importante

indicar que debemos de incluir el JAR en las bibliotecas (lucene-queryparser-....) ya que QueryPaser no forma parte del core Lucene.

Puede ser difícil para un usuario conocer todo el lenguaje de consulta Lucene, por lo que en nuestras aplicaciones será conveniente tener más control sobre cómo se realizan las búsquedas (por ejemplo a través de distintos formularios que tiene que rellenar el usuario). Para ello, deberemos de crear varios objetos consulta sin utilizar el QueryParser. Además debemos de considerar que los términos que no se tokenizan (como por ejemplo IDs, keywords, fechas, etc.) es mejor añadirlos de forma explícita a la consulta y no a través del QueryParser.

El método principal es `parse(String)` que se puede ver como un conjunto de clausulas formadas a través de un conjunto de Terms (pares campo-valor). Ejemplos de como construir consultas los podemos encontrar en `http://lucene.apache.org/core/7_1_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package.description`. Esto nos permitirá realizar las consultas por campos, de entre los que podemos destacar uno por defecto. Podemos buscar por cualquier campo indicando field:termino. Así, asumamos que el índice Lucene contiene dos campos, title y text, siendo text el campo por defecto. Entonces podemos realizar la siguiente consulta,

`title:"Don Quijote" text:Dulcinea`

que buscaría libros con título "Don Quijote" (deben aparecer consecutivas) en los que en el contenido del libro apareciese la palabra Dulcinea. Si no se indica el campo se utiliza el campo por defecto, por lo que la siguiente consulta es equivalente. `title:"Don Quijote" Dulcinea` .

Aquí solo indicaremos que podemos construir consultas que consideren expresiones regulares, caracteres comodín ($*$, ?), consultas difusas (distancia entre términos), búsqueda por proximidad, por rango, consultas lógicas (AND, OR, NOT), por rango, etc. En la documentación de Lucene podemos encontrar más ejemplos. `http://lucene.apache.org/core/7_1_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package.description`.

Para construir un QueryParser, lo más usual es considerar el constructor `QueryParser(String f, Analyzer a)` donde f es el nombre del campo por defecto y a el Analyzer que se va a utilizar para procesar la consulta (puede ser un wrapper analizer).

Como resultado, devolvemos un objeto de tipo Query (no realizamos la búsqueda)

```
1 QueryParser parser = new QueryParser("text", new
    WhitespaceAnalyzer());
2 Query q1,q2,q3;
3 q1 = parser.parse("title:\"Don Quijote\" text:Dulcinea");
4 q2 = parser.parse("title:\"Don Quijote\" and text:Dulcinea");
5 q3 = parser.parse("title:\"Don Quijote\" AND text:Dulcinea");
```

En este ejemplo, q1 es una consulta que busca documentos que tengan en el título "Don Quijote" o en el campo text a Dulcinea; q2 es una consulta que contenga en el título "Don Quijote" o **and** o Dulcinea en text, la consulta que se genera es "text:and text:Dulcinea". Finalmente, q2 es una consulta que contenga "Don Quijote" en el título junto con Dulcinea en el texto.

4. Realizando la búsqueda

Una vez que tenemos la consulta, podemos indicar a IndexSearcher que realice la búsqueda, y como resultado tendremos los documentos ordenados por relevancia.

```
1 int cuantos = 20;
2 TopDocs docs = indexSearcher.search(q,cuantos);
3
4 System.out.println("Documentos encontrados: "+ docs.totalHits);
5 for (ScoreDoc sd : docs.scoreDocs) {
6       Document d = searcher.doc(sd.doc);
7       System.out.println(sd.score +"Libro: "+ d.get("title"));
8     }
```

En la línea 2 le indicamos al indexSearcher que busque los documentos relevantes a la consulta y devuelva sólo los 20 primeros. TopDocs es una estructura que recoge los hits (emparejamientos) entre documentos y consulta. Tiene dos campos, totalHits que contiene el número total de emparejamientos encontrados y scoreDocs que es un vector de scoreDoc con tamaño máximo `cuantos` (20), cada uno conteniendo el atributo doc, que representa el docID del documento en el índice y el atributo score, que representa el grado de similitud entre documento

y consulta. Una vez que tenemos el docID del documento, podemos recuperarlo a través del docID (línea 6 del código).

Dado que el usuario no consulta normalmente todos los documentos relevantes a una consulta (que podrían ser miles o cientos de miles), por cuestiones de eficiencia se recomienda utilizar valores relativamente bajos para el número de documentos a devolver. Si necesitamos mas, es posible utilizar el método `searchAfter` de la clase indexSearch.

4.1. Ejemplo básico de lo que sería una clase búsqueda

```
1  package BusquedasLucene
2
3  import org.apache.lucene.analysis.Analyzer;
4  import org.apache.lucene.analysis.standard.StandardAnalyzer;
5  import org.apache.lucene.document.*;
6
7  import org.apache.lucene.queryparser.classic.QueryParser;
8  import org.apache.lucene.search.IndexSearcher;
9  import org.apache.lucene.search.Query;
10
11 ....
12
13
14 public class busqueda {
15   // Ubicacion del indice
16   String indexPath = "./index";
17
18
19
20   public static void main(String[] args) {
21
22     Analyzer analyzer = new StandardAnalyzer();
23
24     Similarity similarity = new ClassicSimilarity();
25     // ================================
26     // Otras alternativas implementadas en Lucene
27     // Similarity similarity = new BM25Similarity(); // Valor
       por defecto en Lucene
```

```
28      // Similarity similarity = new LMDirichletSimilarity();
29
30      // Busqueda en el indice
31      indexSearch(analyzer, similarity);
32    }
33
34
35    // ================================================
36    // Asumimos que en el indice hay dos campos
37    // Cuerpo de tipo TextField, almacenado
38    // ID de tipo entero
39
40    public void indexSearch(Analyzer analyzer, Similarity
          similarity) {
41
42      IndexReader reader = null;
43      try {
44        reader = DirectoryReader.open(FSDirectory.open(Paths.get(
              indexPath)));
45        IndexSearcher searcher = new IndexSearcher(reader);
46        // Asignamos como se calcula la similitud entre documentos
47        searcher.setSimilarity(similarity);
48
49        BufferedReader in = null;
50        in = new BufferedReader(new InputStreamReader(System.in,
              StandardCharsets.UTF_8));
51
52        // El campo cuerpo sera analizado utilizando el analyzer
53        QueryParser parser = new QueryParser("Cuerpo", analyzer);
54        while (true) {
55          System.out.println("Consulta?: ");
56
57          String line = in.readLine();
58
59          if (line == null || line.length() == -1) {
60            break;
61          }
62          // Eliminamos caracteres blancos al inicio y al final
63          line = line.trim();
```

```
64          if (line.length() == 0) {
65            break;
66          }
67
68          Query query;
69          try {
70            query = parser.parse(line);
71
72          } catch (org.apache.lucene.queryparser.classic.
                ParseException e) {
73            System.out.println("Error en cadena consulta.");
74            continue;
75          }
76
77          TopDocs results = searcher.search(query, 100);
78          ScoreDoc[] hits = results.scoreDocs;
79
80          int numTotalHits = results.totalHits;
81          System.out.println(numTotalHits + " documentos
                encontrados");
82
83          for (int j = 0; j < hits.length; j++) {
84            Document doc = searcher.doc(hits[j].doc);
85            String cuerpo = doc.get("Cuerpo");
86            Integer id = doc.getField("ID").numericValue().
                  intValue();
87            System.out.println("
                  ----------------------------------------");
88            System.out.println("ID: " + id);
89            System.out.println("Cuerpo: " + cuerpo);
90            System.out.println();
91          }
92
93          if (line.equals("")) {
94            break;
95          }
96        }
97        reader.close();
98      } catch (IOException e) {
```

```
99          try {
100           reader.close();
101         } catch (IOException e1) {
102           e1.printStackTrace();
103         }
104         e.printStackTrace();
105       }
106     }
107
108
109
110 }
```

5. Clase Query: Como crear consultas

Como hemos visto, hay distintas formas de realizar consultas Lucene, la primera utilizando un QueryParser que obtiene la consulta a partir de un String. En esta sección veremos como podemos utilizar nuestro programa para construir consultas, para ello utilizaremos las distintas subclases de la clase abstracta Query `https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/search/Query.html`

- TermQuery
- BooleanQuery
- PhraseQuery
- WildcardQuery
- PrefixQuery
- MultiPhraseQuery
- FuzzyQuery
- RegexpQuery

- TermRangeQuery
- PointRangeQuery
- ConstantScoreQuery
- DisjunctionMaxQuery
- MatchAllDocsQuery

Pasaremos a discutir algunas de ellas, el resto lo debéis ver a partir de la documentación de Lucene.

5.1. TermQuery

TermQuery es la consulta más simple que podemos hacer utilizando únicamente un Term Lucene, esto es un par compuesto por al nombre del campo sobre el que queremos buscar el texto (un string) que queremos encontrar en el documento, que puede representar algo más que una palabra.

```
1 Term t = new Term("title","Quijote");
2 Query q = new TermQuery( t );
```

Es importante indicar que este string no se tokeniza, sino que pasa directamente a la búsqueda tal y como viene, considerado como un único token. Así, si buscamos `new Term("title","Don Quijote")`, estaremos buscando el token **Don_Quijote**, y no lo encontraremos en el caso de que el documento haya sido tokenizado.

Distintas termQuery pueden combinarse utilizando BooleanQuery.

5.2. BooleanQuery

Nos permite construir consultas que combinen distintas subconsultas, `BooleanClause`, obtenidas a partir de TermQuery, PhraseQuery o otras BooleanQuery. Estas consultas se construyen a partir de BooleanClause (subconsultas), utilizando un BooleanQuery.Builder. Por defecto, el número de BooleanClause que se pueden utilizar es 1024, aunque se puede modificar si es necesario (como cuando queremos realizar consultas muy largas con, por ejemplo, 3000 términos distintos).

Una BoolenClause esta formada por un par (Query,restricción), este último indica como esperamos que la clausula ocurra en los documentos que emparejan con la clausula, BooleanClause.Occur. Lo aclararemos con el siguiente ejemplo donde pretendemos recuperar los libros que contengan ”Quijote“ AND ”Mancha“ en el título:

```
1  Query q1 = new TermQuery(new Term("title","Quijote"));
2  Query q2 = new TermQuery(new Term("title","Mancha"));
3
4
5  BooleanClause bc1 = new BooleanClause(q1,BooleanClause.Occur.
       MUST);
6  BooleanClause bc2 = new BooleanClause(q2,BooleanClause.Occur.
       MUST);
7
8  BooleanQuery.Builder bqbuilder = new BooleanQuery.Builder();
9  bqbuilder.add(bc1);
10 bqbuilder.add(bc2);
11
12 BooleanQuery bq = bqbuilder.build();
13
14 TopDocs tdocs = searcher.search(bq, 20);
15 System.out.println("Hay "+tdocs.totalHits+" docs");
```

En esta consulta pedimos que sea obligatorio que ocurra en el título tanto el término Quijote como el término Mancha, por eso creamos las dos clausulas booleanas (líneas 5 y 6) con la restricción MUST. Una vez construidas las dos clausulas se las añadimos a un objeto de la clase BooleanQuery.Builder que es finalmente el encargado, cuando contenga todas las clausulas necesarias, de construir la Query (línea 12).

Con respecto a los distintos valores que toman las restricciones de una BooleanClause.Occur encontramos:

- MUST se utiliza para clausulas que deben aparecer forzosamente en los documentos que emparejan con la consulta.
- SHOULD se utiliza para clausulas que pueden aparecer en los documentos que emparejen.

- MUST_NOT se utiliza cuando no queremos que la clausula aparezca en documentos relevantes a la consulta
- FILTER actúa como MUST, pero esta clausula no se utiliza a la hora de calcular el score.

5.3. PhraseQuery

Se utiliza cuando buscamos documentos por frases. En este caso todos los términos en la frase deben emparejar en el documento y en la misma posición. Veamos el proceso de creación para la consulta ”Don Quijote de la Mancha“ en título.

```
1 PhraseQuery frase = new PhraseQuery("title","Don","Quijote","de
    ","la","Mancha");
```

Hay que tener cuidado, porque si construimos `PhraseQuery("title","Don Quijote de la Mancha")` estamos creando la consulta con un único token, ”Don_Quijote_de_la_Mancha“, que puede no ser lo deseado.

Las consultas por frases también se pueden utilizar cuando queremos que los términos estén a una determinada distancia, como muestra el siguiente ejemplo donde permitimos que Quijote y Mancha estén en el documento a una distancia máxima de 3 (este parámetro se denomina slop).

```
1 PhraseQuery frase = new PhraseQuery(3,"title",,"Quijote","
    Mancha");
```

5.4. Consultado IntPoint

Aunque existe PointRangeQuery como clase abstracta para consultar sobre los points indexados (IntPoint, FloatPoint, ...), la documentación de Lucene nos aconseja utilizar los métodos existentes en las distintas clases (IntPoint, FloatPoint, ...) para realizar consultas. Veremos el ejemplo con IntPoint `https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/document/IntPoint.html`, que proporciona entre otros los siguientes métodos:

- newExactQuery(String field, int value): Busca el valor exacto en el campo

- newRangeQuery(String field, int lowerValue, int upperValue): Busca en el campo documentos con valores en el intervalo cerrado [lowerValue,upperValue]
- newSetQuery(String field, int... values): Busca documentos cuyos valores estén en el conjunto de parámetros dado.

El siguiente ejemplo nos muestra como construir las consultas sobre IntPoint

```
1 Query qe = IntPoint.newExactQuery("curso", 4); //asignaturas
    del curso 4
2 Query qr = IntPoint.newRangeQuery("curso", 2, 4); //
    Asignaturas de los curso segundo, tercero y cuarto
3 Query qs = IntPoint.newSetQuery("curso", 1,4); //Asignaturas de
    los cursos primero y cuarto, pero no segundo ni tercero
```

6. MatchAllDocsQuery

Crea una consulta que empareja con todos los documentos. Se suele utilizar en la etapa de prueba del sistema y también cuando queremos computar todas los elementos que hay bajo una determinada categoría en la búsqueda por facetas.

```
1 Query q = new MatchAllDocsQuery();
```

7. Modificando el criterio de ordenación de resultados

Como hemos comentado, Lucene nos devuelve los resultados ordenador por el valor de score (similitud entre documento y consulta), asegurándonos que los documentos más relevantes a la consulta ocupen las primeras posiciones. Este es el correcto funcionamiento en muchas aplicaciones, sin embargo encontramos ejemplos donde podemos estar interesados en mostrar los documentos que emparejan considerando un orden distinto, como por ejemplo en un sitio de e-comercio puede ser el precio del producto o el número de existencias del mismo. O un sitio de noticias, donde podemos estar interesados en situar en las primeras posiciones las noticias mas recientes.

Sin embargo, no podemos ordenar por todos los campos almacenados. Si queremos hacer esto lo tenemos que indicar de forma explícita en el momento de indexación pues para ello Lucene debe almacenar los DocValues. Los campos utilizados para ordenar deben ser seleccionados con cuidado. Los documentos deben contener un único termino en ese campo y el valor del término debe permitir determinar la posición relativa del documento en el orden. El campo debe indexarser, pero no puede tokenizarse, no necesitando estar almacenado (salvo que queramos devolverlo en la salida)

7.1. En Indexación

Según la documentación de Lucene DocValues son una forma de almacenar los valores de los campos que permite realizar de forma mas eficiente algunas tareas como por ejemplo la ordenación y el uso de facetas. DocValue se puede considerar como una columna en el índice que permite relacionar un documento con sus valores. Esto también se hace en el momento de indexación, aliviando algunos de los requisitos de memoria cuando trabajamos con facetas, ordenamos o realizamos agrupamientos. Así, supongamos que queremos ordenar una lista de documentos por un determinado campo. En este caso, tendremos que recorrer la lista de documentos y recuperar para cada uno sus valores. Considerando DocValues es posible para un valor obtener los documentos que le corresponde, lo cual es útil si partimos de una lista de documentos.

Para ello, debemos indexar los campos como NumericDocValuesField SortedNumericDocValuesField, SortedDocValuesField o SortedSetDocValuesField (`https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/document/Field.html`)

Si queremos que los campos almacenados en los DocValuesField también sean mostrados al usuario, debemos almacenarlos aparte como StoredField con el mismo nombre

```
1 doc.add(new NumericDocValuesField("curso",9L));
2 doc.add(new StoredField("curso",9));
3
4 // Otros ejemplos de DocValuesField son
5
```

```
6 doc.add(new DoubleDocValuesField("Nota",7.98));
7 doc.add(new BinaryDocValuesField("bin", new BytesRef(Integer.
      toString(10).getBytes())));
8 doc.add(new SortedDocValuesField("sorted", new BytesRef(Integer
      .toString(10).getBytes())));
```

Por ejemplo, SortedDocValuesField puede ser utilizado cuando queremos devolver la salida ordenada por fecha

```
1 String fecha = "2017-11-10 17:47:38"
2 doc.add(new SortedDocValuesField ("Fecha", new BytesRef(fecha)
      ));
3 doc.add(new StoredField("Fecha", date));
```

7.2. En Consulta

Para poder ordenar, primero debemos declarar un objeto de la clase SortField, que almecena información de cómo ordenar los documentos por un determinado campo (obviamente esta campo no puede haber sido analizado, lo que implica que sólo contiene un término - no se tokeniza).

```
1 SortField sf =new SortField("curso",SortField.Type.INT,true);
      // Campo de ordenacion curso, entero, decreciente
2 sf.setMissingValue(0); // Valor por defecto cuando los docs no
      tiene dicho campo
3 Sort orden  = new Sort(sf);
4
5
6 Sort sortFecha = new Sort(  new SortField("Fecha", Type.STRING),
      SortField.FIELD_SCORE);
```

En este caso, hemos definido el campo `curso` como campo para ordenar, que es de tipo entero (INT). Otros tipos pueden ser LONG, DOUBLE, FLOAT, STRING y también DOC (por docID) o SCORE (por score) o cualquier otro que podamos definir. El último parámetro indica que vamos a ordenar considerando un criterio decreciente sobre los elementos.

En el caso de ordenación por Fecha, se introducen dos parámetros de tipo SortField, y como consecuencia ordena por el primer parámetro (Fecha) y en caso

de empate considera el Score.

En la línea 2 se define el valor que tomará en campo ”curso“ para aquellos documentos que no contengan dicha información (missing values). En nuestro caso es 0.

Una vez que tenemos definido el orden, podemos realizar la búsqueda utilizando el método search de IndexSearcher, pero hay dos pequeñas diferencias:

- Recibe un nuevo parámetro que es orden que se utiliza
- Devuelve un TopFieldDocs, una especialización de la clase TopDocs, que además de scoreDocs (vector de ScoreDoc) y totalHits, también almacena información sobre el SortField utilizado para la ordenación.

```
1  TopFieldDocs tfds;
2  tfds = searcher.search(q, 10, orden);
3  ....
4
5  System.out.println("Orden segun campo: "+
6          tfds.fields[0].getField());
7  for (ScoreDoc hit : tfds.scoreDocs){
8      Document d = searcher.doc(hit.doc);
9      System.out.println( d.get("asignatura") + " curso" + d.get
           ("Nota")+ "score="+hit.score);
10 }
```

Otra alternativa que podemos utilizar para la búsqueda es

```
1  TopFieldDocs  search(Query query, int n, Sort sort, boolean
       doDocScores, boolean doMaxScore)
```

Donde doDocScores representa si queremos calcular o no la similitud de cada documento recuperado. Si finalmente consideramos otro criterio de ordenación, no necesitamos los scores y por tanto podemos agilizar el cálculo. Debemos habilitarlo si consideramos un criterio de ordenación mixto, como por ejemplo ordenar primero por curso y en caso de empate ordenar por score.

doMaxScore nos indica si debemos de calcular el score máximo para los documentos que son relevantes a la consulta. Este cálculo representa un mayor costo.

8. Uso de Colectores

Para trabajar con la lista de resultados de búsqueda se desarrolló la clase Collector con el objetivo de facilitar tareas que impliquen la manipulación de los mismos como pueden ser la ordenación, el filtrado o agrupamiento de resultados `http://lucene.apache.org/core/7_1_0/core/org/apache/lucene/search/Collector.html`. Por tanto, los resultados de búsqueda podrán ser enviados a un colector para procesarlos. Así, al llamar al método `IndexSearch.search` se utiliza internamente el colector por defecto, `TopScoreDocCollector` que ordena los resultados considerando el score, para finalmente devolver la lista ordenada en un objeto de la clase TopDocs.

Lucene tiene implementados distintos colectores por lo que pasamos a ver cómo se pueden utilizar con algunos ejemplos.

- TopScoreDocCollector es una clase que ordena los top N documentos considerando score+docID (primero por score y en caso de empate docID). Es el collector más utilizado. Al crearlo (ver línea 1) tenemos que indicarle el tamaño máximo del mismo.

```
1 TopScoreDocCollector collector = TopScoreDocCollector.create
    (10); // Considera un maximo de 10 resultados
2
3 searcher.search(q, collector); // Los resultados de la
    busqueda los pasa a colector de resultados,
4
5 System.out.println("Hay "+ collector.getTotalHits() +"
    resultados.");
6 for (ScoreDoc hit : collector.topDocs().scoreDocs){
7       Document d = searcher.doc(hit.doc);
8       System.out.println( d.get("asignatura") + " curso" + d
            .get("Nota")+ "score="+hit.score);
9 }
```

- TopFieldCollector es un colector que recoge los resultados y los ordena por campos, por lo que necesita un objeto de tipo Sort. Es el utilizado por IndexSearch cuando se le pasa Sort como parámetro.

```
1 TopFieldCollector tfcollector =   TopFieldCollector.create(
    orden,10,false,true,false);
2 searcher.search(q, tfcollector);
3
4 System.out.println("Hay "+ tfcollector.getTotalHits() +"
    resultados.");
5 TopFieldDocs tfd = tfcollector.topDocs();
6 for (ScoreDoc hit : tfd.scoreDocs){
7       Document d = searcher.doc(hit.doc);
8       System.out.println( d.get("asignatura") + " curso" +
          d.get("Nota")+ "score="+hit.score);
9 }
```

Como vemos en el ejemplo, su uso es similar. Pasamos a ver los parámetros del método create.

```
1 TopFieldCollector create(Sort sort, int numHits, boolean
    fillFields, boolean trackDocScores, boolean
    trackMaxScore)
```

El primero es el criterio de ordenación, numHits indica el tamaño máximo a devolver, indica si los FieldDoc serán devueltos con el resultado, trackDocScores indica si los resultados de los scores serán devueltos con los resultados (si no es así, devolvera un NaN y trackMaxScore indica si recoge el valor de score máximo (TopDocs.getMaxScore())

- TopDocsCollector, es la clase base para aquellos colectores que devuelven una lista de TopDocs como salida. Este colector nos permite modificar el comportamiento al pasarle al constructor una PriorityQueue con el criterio de ordenación que deseamos. Tanto TopScoreDocCollector como TopFieldCollector son subclases de esta. Esta sería la clase que tendremos que extender si queremos buscar un comportamiento específico sobreescribiendo algunos de sus métodos.

- PositiveScoresOnlyCollector, envuelve cualquier otro colector y evita que se recojan resultados de búsqueda con $score <= 0$.

```
1 TopScoreDocCollector collector = TopScoreDocCollector.create
    (10);
2 searcher.search(q, new PositiveScoresOnlyCollector(collector
    ));
3 TopDocs topDocs = collector.topDocs();
```

- TimeLimitingCollector, envuelve cualquier otro colector, abortando la búsqueda si esta tarda demasiado tiempo.

Como hemos visto, normalmente no tenemos que utilizar un colector, pues disponemos de alternativas para realizar el proceso, pero si queremos personalizar como tratamos los resultados de búsqueda es una buena idea crear uno propio extendiendo la clase Collector

9. Fecha de entrega y defensa de la práctica

El tiempo previsto para realizar esta parte relativa a consultas, junto con su documentación asociada será de tres semanas. En esta parte, debemos empezar a pensar en la interfaz de usuario que propondremos para nuestra aplicación.