Sistema de Recuperación de Información

Jose Manuel Rodríguez Calvo

Juan Miguel Vilchez Rodríguez

portada: Der Bücherwurm Carl Spitzweg, 1850

Índice

Indexación

Esta parte es una de las más importantes de nuestro proyecto y en la que más tiempo hemos invertido debido a que una buena indexación dará lugar a unos buenos resultados en las búsquedas. Los tres documentos con los que empezamos son **Questions**, **Answers** y **tags**, todos los utilizamos para ofrecer unos mejores resultados y entre ellos tienen un formato diferente que hace que se aplique diferentes métodos para almacenar la información, sin embargo usamos una indexación lucene para los tres. Los tratamientos para los diferentes archivos son los siguientes:

77434 ,14008 ,2008−09−16T21:40:29Z ,171,How to access the last value in a vector? , " Suppose I have a vector that is nestedinadata frameone or two levels . Is there a quick and dirty way to access the last value , without using the <code> length ( ) </code> function? Something a la PERL's <code >$ # </code> specialvar? So I would like something like : <pre><code> dat $vec1 $vec2 [$ #] </code> </pre> instead of < pre><code> dat $vec1 $vec2 [length(dat $vec1 $vec2)] </code> </pre> "	All: StoredField Id_Q: StringField UserId: StringField DateLong: LongPoint→ para ser consultado Date: StoredField ScoreInt: IntPoint→para ser consultado Score: StoredField Title: TextField Body: TextField Code: TextField

Además de indexarlos, hemos usado un **PerFieldAnalyzerWrapper** para analizar según el campo ya que no tiene que recibir el mismo tratamiento el título que el código. Tanto para el campo **Title** como para el **body** hemos usado **EnglishAnalyzer** ya que todos están en Inglés, mientras que para el campo **Code** hemos desarrollado un analizador propio para que sea más exacto con la necesidad de información.

Para la indexación de las respuestas, usamos un método algo diferente aunque el trasfondo del funcionamiento es muy similar, llegando así a reutilizar código implementado en el apartado anterior. A continuación muestro un ejemplo del tratamiento de los datos

9336125,1216960,2012-02-17T22:07:41Z,9336071,2,FALSE," Don't know if this exactly answers what you are asking, but:
<pre><code>vector <- res$par
> vector
[1] 0.9771973 1.0218072
> vector[1]
[1] 0.9771973
</code></pre>
 There may be an accessor method that is more 'proper.'
	All: StoredField
	Id_A: StringField
	UserId: StringField
	DateLong: LongPoint→ para ser consultado
	Date: StoredField
	Id_parent: StringField
	ScoreInt: IntPoint→para ser consultado
	Score: StoredField
	Aproved: StringField
	Body: TextField
	Code: TextField

El tratamiento de los datos, en concreto sobre el analizador que le aplicamos a cada campo es el mismo que preguntas. la única diferencia son los campos: **Aproved** e **Id_parent** ; a los que no le hemos aplicado ningún analizador debido a que son tipos de datos unitarios donde solo son datos simples que son buscados de forma booleana.

Para el tratamiento de los **tags**, son considerados como parejas de id de la pregunta y el tag, por tanto el tratamiento de este documento es sencillo, donde incluso solo aplicamos a tag el analizador **WhiteSpaceAnalyzer** porque el otro campo será buscado de manera unitaria y sin alterar ese valor:

77434,vector	Id_T : StringField tag: StringField

A parte de estos campos hemos tenido que añadir otro campo más para poder conseguir un mejor tratamiento de los datos que es el campo Type que es un *StringField* en el que almacenamos el tipo del documento sobre el que estamos tratando, así cuando estamos ante una pregunta el valor de **Type** será "**Question**", el de Answer con su correspondiente "**Answer**" y tag con "**tag**".

Con todo este proceso funcional, el problema que intentamos solucionar se vuelve más sencillo y cómodo de implementar posteriores pasos. Sobre el rendimiento de este proceso que se realiza antes de mostrarle los resultados a los usuarios, por tanto el tiempo de esta parte no es relevante para nuestro problema. Para una mejor monitorización de nuestro sistema, con un intel core i7 de 2.80 GHz y una memoria RAM de 32GB este proceso tarda alrededor de tres minutos y ocupa un total de 815 MB.

Para la ejecución de esta parte '**core.java**' será muy importante la carpeta **index**, que se creará en el mismo repositorio y si no está creada esa carpeta la creará el programa, en cambio si ya está creada, el programa interpreta que ya no hay que hacer el índice y no procederá a ejecutarlo.

Búsqueda

Este es otro de los puntos muy importantes para obtener buenos resultados a la necesidad de información, en este proceso podemos distinguir dos funciones importantes, la **búsqueda normal** (Junto con la búsqueda avanzada) y la **búsqueda por facetas**.

La búsqueda normal hace acopio de una función especializada que busca el *query* deseado en todos los posibles campos de los documentos almacenados devolviéndolos ordenados por **Score**. Dado que *Stack Overflow* tiene su propio **Score** reflejado en cada documento, hemos extraído manualmente este para mezclarlo con el **Score** de lucene y así reflejar lo que creemos que es un orden por puntuación más realista.

La búsqueda avanzada hace una consulta más específica, a la que se le pasa una lista con las querys deseadas por cada campo, por ejemplo, podríamos buscar documentos que tengan el query deseado en el código, o en el título, o podríamos buscar fácilmente por usuarios concretos. Esto se hace mediante una combinación de BooleanClauses.

Aquí tenemos ejemplos de funciones que usamos y resultados de las búsquedas.

```
public static Map searchin_process(String consultation, String Facets, String FacetType, String [] listaFacetas)throws Exception{
  Set<Document> solutions_All = new HashSet<Document>();
  Set<String> solutions_Tags = new HashSet<String>();
  //test that consultation is not a file
  File af = new File(consultation);
  String document="";
  String line="";
  if (af.isFile()){
    BufferedReader br = new BufferedReader(new FileReader(consultation));
      while ((line = br.readLine()) != null) {
        document=document+" "+line;
      }
      body_titlequery(document,solutions_All);
      codequery(document,solutions_All);
  }
  else if(Integer.parseInt(Facets) == 1){
        facetsQuery(consultation,solutions_All, Integer.parseInt(FacetType));
  }
  else if(Integer.parseInt(Facets) == 2){
        facetsQuerySearch(consultation, listaFacetas,solutions_All, Integer.parseInt(FacetType));
  }
  else{
    //test if the query is an ID of user, an ID of question/answer or a tag
    if(consultation.indexOf(" ")==-1){
      unitariquery(consultation,solutions_All,solutions_Tags);
    }
    else{
      body_titlequery(consultation,solutions_All);
      codequery(consultation,solutions_All);
    }
  }
```

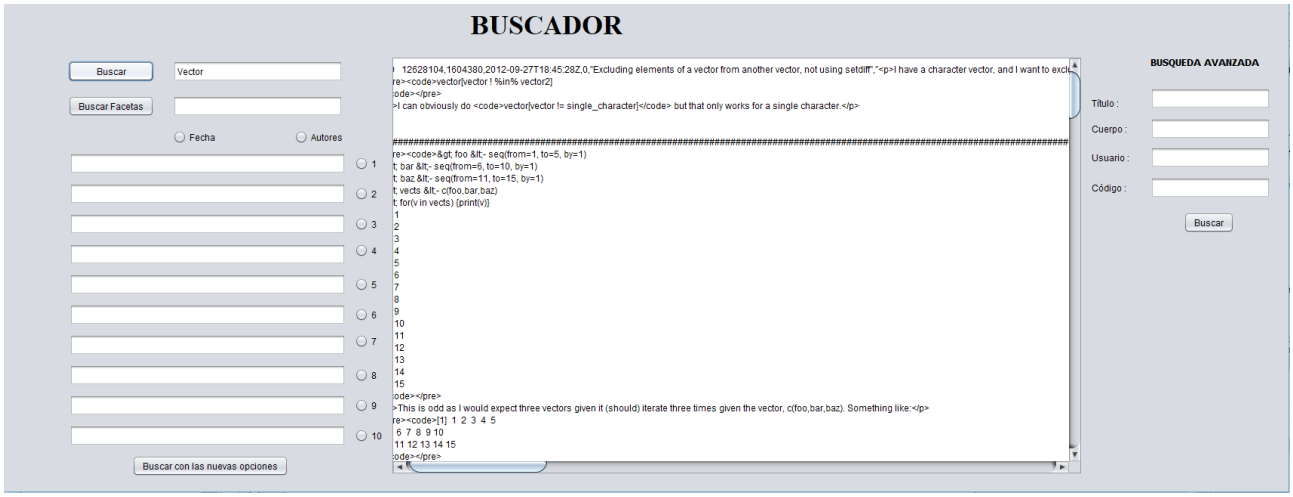
Este es el método principal de las búsquedas, al que se le pasa la consulta, un string (“1” o “2”) que diferencia una búsqueda normal de una por facetas, String FacetType, que nos dice el tipo de faceta que estamos buscando, y una lista de facetas para una vez elegida poder buscar con esos nuevos filtros. Como podemos ver, tenemos una comprobación para ver si la consulta es un único término (en cuyo caso le pasamos un filtro para comprobar si se está buscando

un usuario o una ID) y luego otro tipo de consulta para si tenemos más de un término (Que busca en el título, en el cuerpo y en el código)

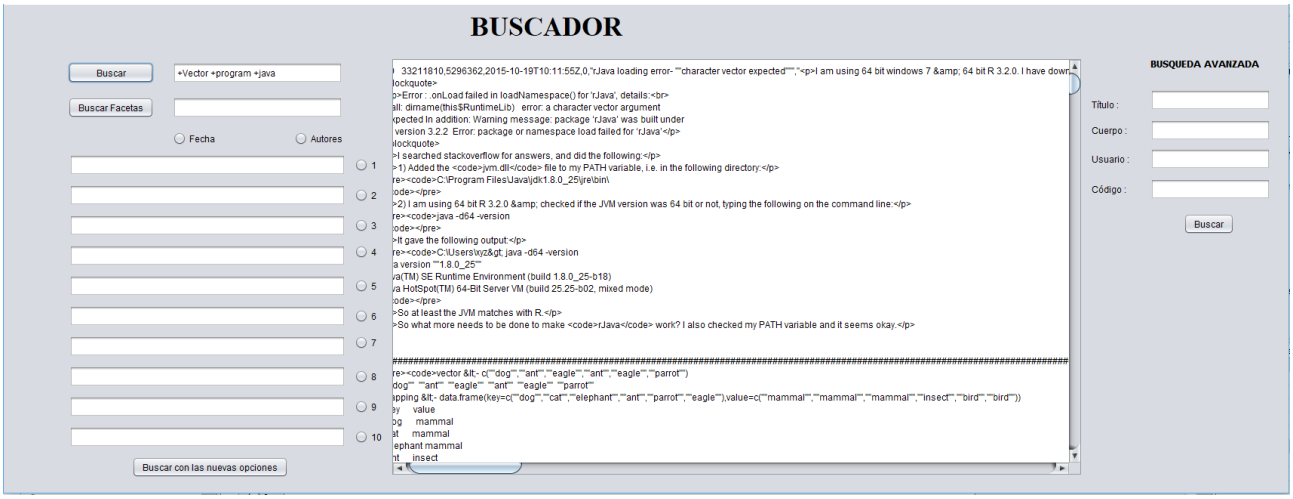
En este ejemplo vemos como se construyen las clausulas booleanas para la búsqueda :

```
QueryParser parserID_Q =new QueryParser("Id_Q",new EnglishAnalyzer());
Query       q1= parserID_Q.parse(query);
QueryParser parserID_A =new QueryParser("Id_A",new EnglishAnalyzer());
Query       q2= parserID_A.parse(query);
QueryParser parserID_T =new QueryParser("Id_T",new EnglishAnalyzer());
Query       q3= parserID_T.parse(query);
QueryParser parserUser =new QueryParser("User",new EnglishAnalyzer());
Query       q4= parserUser.parse(query);
QueryParser parsertag =new QueryParser("tag",new EnglishAnalyzer());
Query       q5= parsertag.parse(query);

BooleanClause bc1 = new BooleanClause(q1, BooleanClause.Occur.SHOULD);
BooleanClause bc2 = new BooleanClause(q2, BooleanClause.Occur.SHOULD);
BooleanClause bc3 = new BooleanClause(q3, BooleanClause.Occur.SHOULD);
BooleanClause bc4 = new BooleanClause(q4, BooleanClause.Occur.SHOULD);
BooleanClause bc5 = new BooleanClause(q5, BooleanClause.Occur.SHOULD);

BooleanQuery.Builder bqbuilder = new BooleanQuery.Builder();
bqbuilder.add(bc1);
bqbuilder.add(bc2);
bqbuilder.add(bc3);
bqbuilder.add(bc4);
bqbuilder.add(bc5);


BooleanQuery bq = bqbuilder.build();

TopDocs docs=searcher.search(bq,20);
```

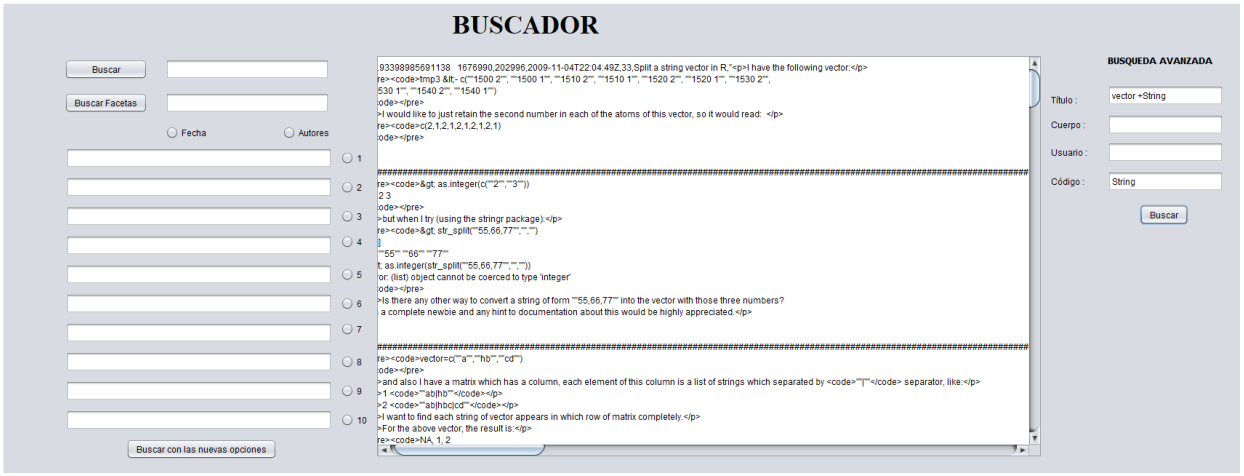
Aquí unos pantallazos de algunos resultados obtenidos en las búsquedas con la aplicación.



Aquí probamos las búsquedas lucene, solicitando documentos que tengan la query vecto, program y java obligatoriamente.



Podemos ver en este pantallazo un ejemplo de búsqueda avanzada, donde buscamos documentos que tengan puedan contener vector y obligatoriamente String en el título y que puedan tener String en la parte del código.



Facetas

La inclusión de facetas en nuestro buscador nos permite hacer búsquedas más específicas y mejorar la interacción del usuario con el proyecto. Nosotros hemos implementado dos facetas diferentes, que son, la **fecha** de los documentos y los **autores** de los mismos, permitiendo al usuario sesgar la búsqueda con ellos.

Primero debemos indexar las facetas deseadas, lo hacemos junto con la indexación del resto de campos de la siguiente manera :

```
FSDirectory taxoDir = FSDirectory.open(Paths.get(FacetsIndexPath));
FacetsConfig fconfig = new FacetsConfig();

DirectoryTaxonomyWriter taxoWriter = new DirectoryTaxonomyWriter(taxoDir);
fconfig.setHierarchical("Publish_Date", true);
fconfig.setMultiValued("Author", true);
```

Aquí abrimos el índice de facetas y generamos su configuración, creando la faceta de carácter jerárquico ***Publish_Date*** y el campo multivalued ***Author***.

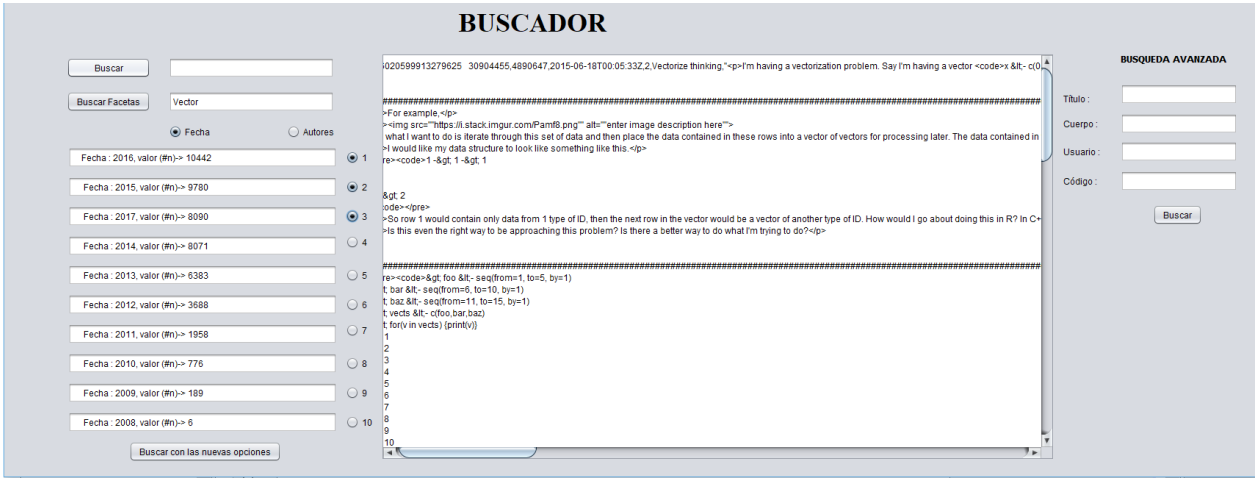
```
doc.add(new FacetField("Author", separate[1]));
doc.add(new FacetField("Publish_Date", String.valueOf(year)));

writer.addDocument(fconfig.build(taxoWriter, doc));
```

Una vez creados, para cada documento indexamos su valor de faceta, para el autor la ID del creador del documento y para la fecha, nos quedamos con el año.

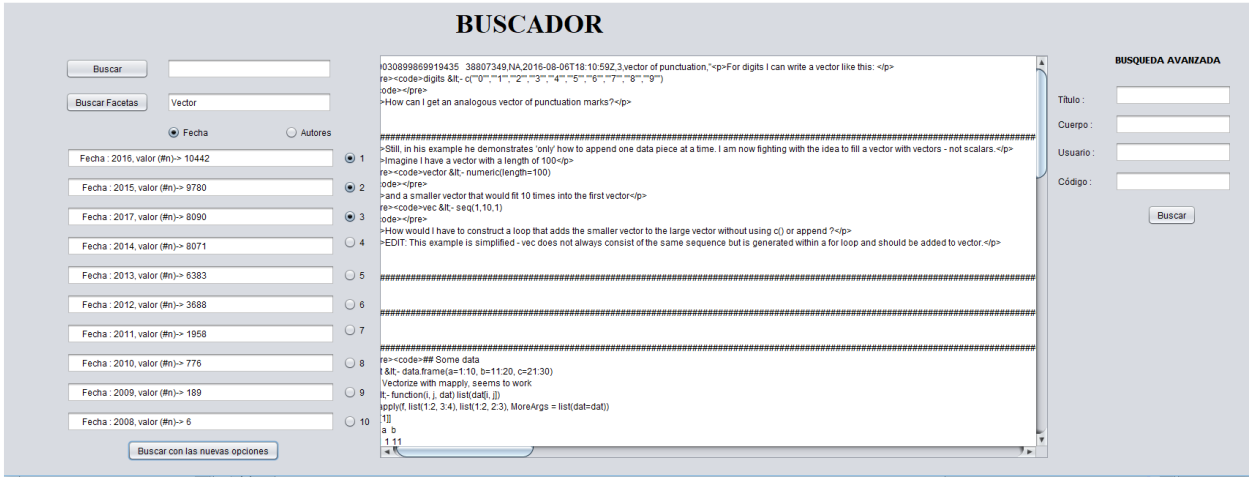
En la parte de búsqueda de nuestro programa, creamos un apartado específico para las facetas. Que son buscadas con la siguiente función :

```
DrillDownQuery ddq = new DrillDownQuery(fconfig, bq);
for (String prueba : Facet){
    System.out.println("Autor : " + prueba);
}

if(faceta==1){
    for (String facetaAutor : Facet){
        if(facetaAutor!=null){
            ddq.add("Author", facetaAutor);
        }
    }
}
else{
    for (String facetaFecha : Facet){
        if(facetaFecha!=null){
            ddq.add("Publish_Date", facetaFecha);
        }

    }
}

FacetsCollector fc = new FacetsCollector();
TopDocs tdc = FacetsCollector.search(searcher, ddq, 10, fc);

Facets facetas = new FastTaxonomyFacetCounts(taxoReader, fconfig, fc);
List<FacetResult> TodasDims = facetas.getAllDims(20);

for (ScoreDoc sd : tdc.scoreDocs){
  Document d=searcher.doc(sd.doc);
  solutions_All.add(d);
}
```

Usaremos a continuación un pantallazo de la búsqueda por facetas para explicar a nivel de usuario como funcionan.



Primero buscamos en la pestaña "**Buscar facetas**" el query deseado, en nuestro caso es "*vector*", nos saldrán los resultados para dicho query usando una búsqueda normal, pero a la izquierda, tendremos un recuento (Gracias a las facetas) de en este caso, las fechas en las que más documentos se han creado. Por ejemplo queremos los documentos que contengan vector, de los tres años en los que más documentos fueron creados (2016, 2015, 2017), y ahora relanzamos la búsqueda con los nuevos parámetros.



Ahora los resultados obtenidos son de estas tres fechas. En el caso de los usuarios es la misma funcionalidad, nos aparecen los usuarios referentes a ese tema que más documentos han publicado, y podemos filtrar las búsquedas para ver sus respuestas o preguntas.

Bibliografía

Material e información esencial para la indexación, búsqueda y las facetas :
https://decsai.ugr.es/index.php?p=material_alumno

Página principal de Lucene 7.10 API, donde hemos encontrado las bibliotecas necesarias para el funcionamiento de la práctica y métodos ya desarrollados en lucene que han facilitado la misma :
https://lucene.apache.org/core/7_1_0/core/overview-summary.html

Stack Overflow ha sido usado para resolver dudas sobre sintaxis o problemas concretos :
https://es.stackoverflow.com/

Video tutoriales para aprender a manejar Jframe y poder desarrollar la interfaz gráfica de la aplicación :
https://www.youtube.com/watch?v=HC6jIjJ1484
https://www.youtube.com/watch?v=xCpkFcZ6Yfw
https://www.youtube.com/watch?v=C2zP_UN43x4
https://www.youtube.com/watch?v=2dAWfs1NZns
https://netbeans.org/kb/docs/java/gui-functionality.html
https://www.youtube.com/watch?v=5ynxP-P-nG0

Instalación de NeatBeans y puesta a punto para la práctica :
https://netbeans.org/community/releases/82/install.html

Páginas de información adicional sobre lucene :
http://www.lucenetutorial.com/lucene-in-5-minutes.html
https://lucene.apache.org/core/7_1_0/index.html