Universidad de Granada

RECUPERACIÓN DE INFORMACIÓN

PRÁCTICA 3: INDEXACIÓN Y BÚSQUEDA
PARTE II: BÚSQUEDA

SERGIO SAMANIEGO MARTÍNEZ

Índice

1. INTRODUCCIÓN 3

2. EXPLICACIÓN DE CÓDIGO 3

2.1 SAMPLE.FXML 3

2.2 CONTROLLER 5

2.2.1 BtnHandler 5

2.2.2 doSearch 6

2.2.3 doIntSearch 8

2.2.4 doBooleanSearch 9

2.2.5 addFacets 14

2.2.6 filterAction 15

3. MANUAL FINAL DE USUARIO 16

1. INTRODUCCIÓN

En esta segunda parte de la práctica de búsqueda con Lucene, se va a proceder a realizar el proceso de búsqueda sobre un índice ya creado de Lucene.
Para facilitar el uso de nuestro programa, se creará una interfaz de usuario que perimitirá al usuario realizar distintos tipos de búsquedas (libre, booleana y por rango), poder visualizar los resultados sobre una tabla y aplicar filtros de facetas sobre dichos resultados.

2. EXPLICACIÓN DE CÓDIGO

Antes de comenzar a ver la funcionalidad de la aplicación, se va a realizar una explicación del código de la misma que nos permitirá comprender mejor el funcionamiento de esta aplicación.

Para la realización de la interfaz gráfica se decide hacer uso del lenguaje JavaFX, ya que es un lenguaje muy actual y bastante fuerte.

Al crear la aplicación de JavaFX se nos crean 3 clases, que van a ser:

- Controller: En esta clase realizaremos todos los cálculos que necesitemos para nuestra aplicación.
- Main: Esta clase se va a encargar de iniciar el entorno gráfico.
- sample.fxml: Como podemos ver, por su extensión es un archivo parecido a XML que nos permitirá definir los objetos que contendrá la interfaz gráfica.

2.1 SAMPLE.FXML

Este archivo nos permitirá realizar la definición de los distintos objetos que va a contener nuestra interfaz gráfica.

Como ya hemos comentado, al usuario se le permitirá realizar 3 tipos de búsquedas, por lo que por cada tipo de búsqueda tendremos un Textfield para escribir la consulta, un Combobox que nos permitirá seleccionar sobre el campo que queremos realizar la consulta y un botón para realizar la búsqueda.

Por lo tanto, en sample.fxml tendremos el siguiente código:

```
<Label fx:id="searchlabel" layoutX="60" layoutY="30">Write the free search: </Label>
<TextField fx:id="textfield" layoutX="50.0" layoutY="50.0"  prefHeight="0" prefWidth="200" />
<Label fx:id="fieldlabel" layoutX="260" layoutY="30">Select the field search: </Label>
<ComboBox fx:id="fieldbox" layoutX="260.0" layoutY="50.0" prefWidth="150.0" promptText="choose">
    <items>
        <FXCollections fx:factory="observableArrayList" >
            <String fx:value="All" />
            <String fx:value="Author" />
            <String fx:value="Title" />
            <String fx:value="Year" />
            <String fx:value="Source Title" />
            <String fx:value="Link" />
            <String fx:value="Resume" />
            <String fx:value="AuthorKeywords" />
            <String fx:value="IndexKeywords" />
            <String fx:value="EID" />
        </FXCollections>
    </items>
</ComboBox>
<Button fx:id="btn" layoutX="420.0" layoutY="50.0" mnemonicParsing="false" onAction="#btn1handle" text="Button" />
```

En la imagen anterior podemos ver los tres elementos comentados anteriormente, y hacen referencia al apartado de la búsqueda libre.
En la siguiente imagen tendremos esos mismos elementos referentes a la búsqueda booleana, que como vemos lo que cambia son los identificadores y la posición de estos elementos que podemos verla en los atributos **layoutY layoutX.**

```
<Label fx:id="searchlabelbool" layoutX="550" layoutY="30">Write the boolean search: </Label>
<TextField fx:id="textfieldbool" layoutX="540" layoutY="50"  prefHeight="0" prefWidth="200" />
<Label fx:id="fieldlabelbool" layoutX="750" layoutY="30">Select the field search: </Label>
<ComboBox fx:id="fieldboxbool" layoutX="750.0" layoutY="50" prefWidth="150.0" promptText="choose">
    <items>
        <FXCollections fx:factory="observableArrayList" >
            <String fx:value="All" />
            <String fx:value="Author" />
            <String fx:value="Title" />
            <String fx:value="Year" />
            <String fx:value="Source Title" />
            <String fx:value="Link" />
            <String fx:value="Resume" />
            <String fx:value="AuthorKeywords" />
            <String fx:value="IndexKeywords" />
            <String fx:value="EID" />
        </FXCollections>
    </items>
</ComboBox>

<Button fx:id="btnbool" layoutX="910.0" layoutY="50.0" mnemonicParsing="false" onAction="#btn1handle" text="Button" />
```

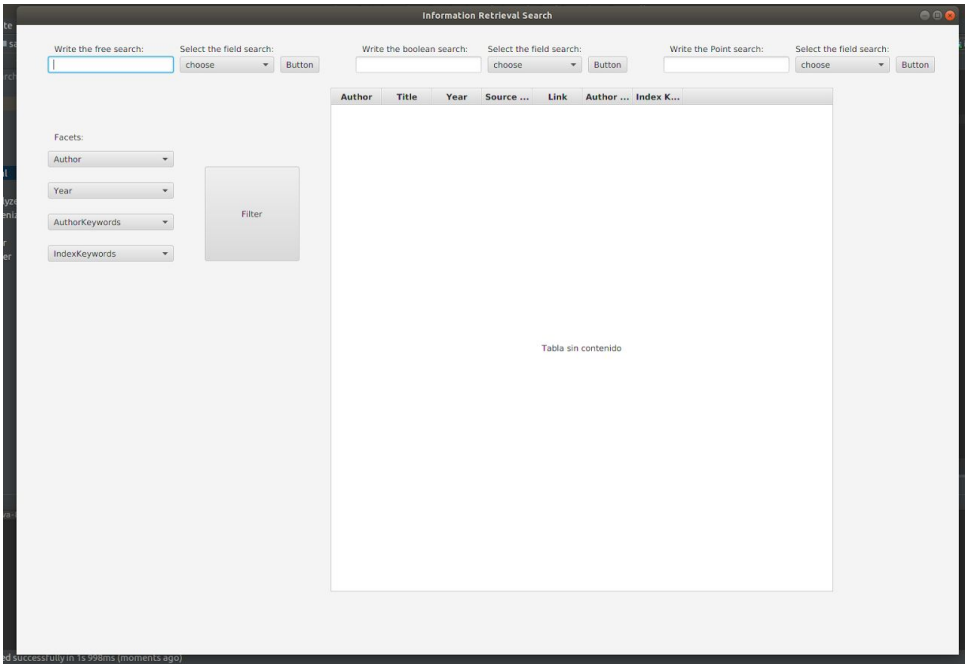
Por último tendremos los elementos referentes a la búsqueda por rango.

```
<Label fx:id="searchlabelint" layoutX="1040" layoutY="30">Write the Point search: </Label>
<TextField fx:id="textfieldint" layoutX="1030" layoutY="50"  prefHeight="0" prefWidth="200" />
<Label fx:id="fieldlabelint" layoutX="1240" layoutY="30">Select the field search: </Label>
<ComboBox fx:id="fieldboxint" layoutX="1240.0" layoutY="50" prefWidth="150.0" promptText="choose">
    <items>
        <FXCollections fx:factory="observableArrayList" >
            <String fx:value="Year" />
        </FXCollections>
    </items>
</ComboBox>

<Button fx:id="btnint" layoutX="1400.0" layoutY="50.0" mnemonicParsing="false" onAction="#btn1handle" text="Button" />
```

Finalmente, hechos los elementos para las búsquedas nos queda realizar el filtrado por facetas y la tabla para mostrar los resultados. Esto podemos verlo en la siguiente imagen.

```
<TableView fx:id="tabledocs" layoutX="500" layoutY="100" prefHeight="800" prefWidth="800">
    <columns>
        <TableColumn fx:id="author_col" text="Author"></TableColumn>
        <TableColumn fx:id="title_col" text="Title"></TableColumn>
        <TableColumn fx:id="year_col" text="Year"></TableColumn>
        <TableColumn fx:id="source_col" text="Source Title"></TableColumn>
        <TableColumn fx:id="link_col" text="Link"></TableColumn>
        <TableColumn fx:id="authorkey_col" text="Author Keywords"></TableColumn>
        <TableColumn fx:id="indexkey_col" text="Index Keywords"></TableColumn>
    </columns>
</TableView>

<Label fx:id="labelFacets" layoutX="60" layoutY="170"> Facets: </Label>
<ComboBox fx:id="AuthorComboBox" layoutX="50" layoutY="200" prefWidth="200" promptText="Author"></ComboBox>
<ComboBox fx:id="YearComboBox" layoutX="50" layoutY="250" prefWidth="200" promptText="Year"></ComboBox>
<ComboBox fx:id="AuKeyComboBox" layoutX="50" layoutY="300" prefWidth="200" promptText="AuthorKeywords"></ComboBox>
<ComboBox fx:id="InKeyComboBox" layoutX="50" layoutY="350" prefWidth="200" promptText="IndexKeywords"></ComboBox>
<Button fx:id="filterbutton" layoutX="300" layoutY="225" prefWidth="150" prefHeight="150" onAction="#filterAction" text="Filter"></Button>
```

Vemos que tendremos una tabla con las distintas columnas de los documentos, y por último una serie de **Combobox** que nos permitirán realizar los filtrados.



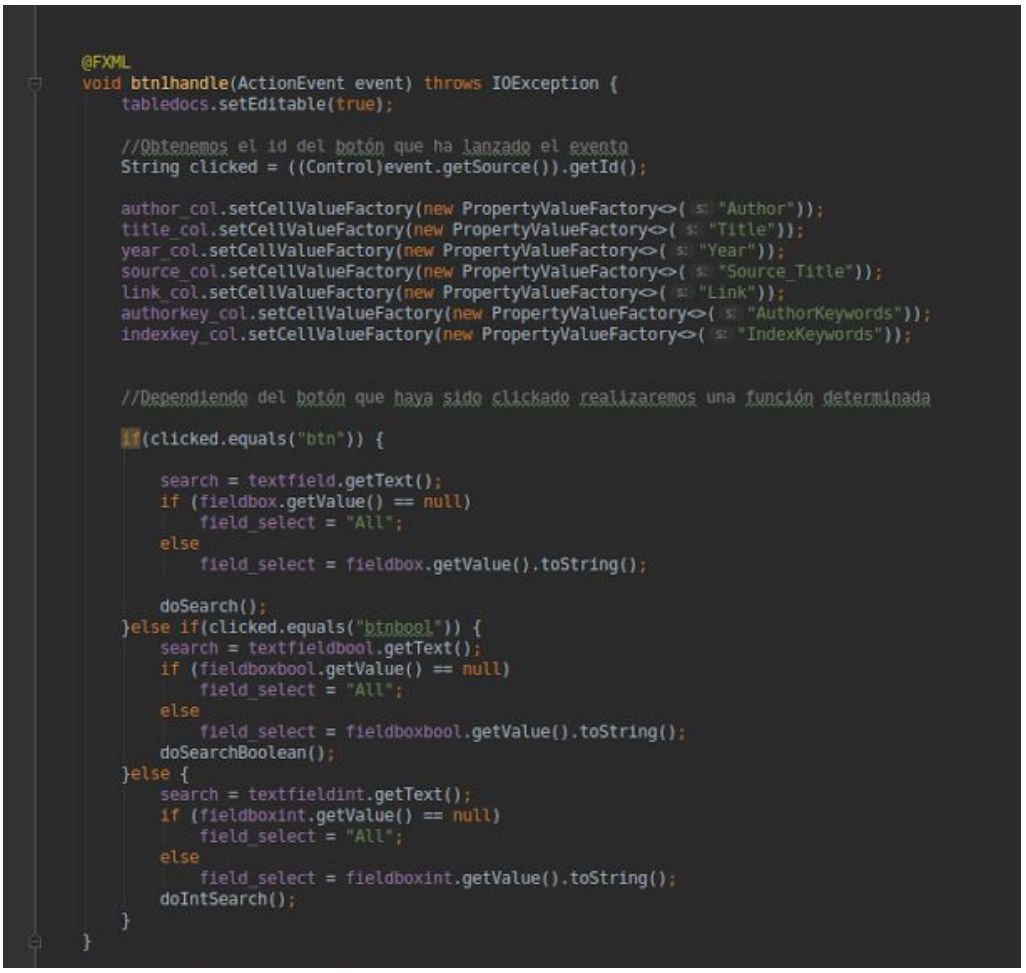
2.2 CONTROLLER

Ahora pasaremos a ver los tareas a realizar cada vez que el usuario pulse alguno de los botones de la interfaz.

2.2.1 BtnHandler

Esta función será la encargada de tratar las acciones a realizar cuando el usuario pulse alguno de los tres botones de búsqueda.

El código de la función es el siguiente:



```
@FXML
void btn1handle(ActionEvent event) throws IOException {
    tabledocs.setEditable(true);

    //Obtenemos el id del botón que ha lanzado el evento
    String clicked = ((Control)event.getSource()).getId();

    author_col.setCellValueFactory(new PropertyValueFactory<>( s: "Author"));
    title_col.setCellValueFactory(new PropertyValueFactory<>( s: "Title"));
    year_col.setCellValueFactory(new PropertyValueFactory<>( s: "Year"));
    source_col.setCellValueFactory(new PropertyValueFactory<>( s: "Source_Title"));
    link_col.setCellValueFactory(new PropertyValueFactory<>( s: "Link"));
    authorkey_col.setCellValueFactory(new PropertyValueFactory<>( s: "AuthorKeywords"));
    indexkey_col.setCellValueFactory(new PropertyValueFactory<>( s: "IndexKeywords"));


    //Dependiendo del botón que haya sido clickado realizaremos una función determinada

    if(clicked.equals("btn")) {

        search = textfield.getText();
        if (fieldbox.getValue() == null)
            field_select = "All";
        else
            field_select = fieldbox.getValue().toString();

        doSearch();
    }else if(clicked.equals("btnbool")) {
        search = textfieldbool.getText();
        if (fieldboxbool.getValue() == null)
            field_select = "All";
        else
            field_select = fieldboxbool.getValue().toString();
        doSearchBoolean();
    }else {
        search = textfieldint.getText();
        if (fieldboxint.getValue() == null)
            field_select = "All";
        else
            field_select = fieldboxint.getValue().toString();
        doIntSearch();
    }
}
```

En esta función obtendremos el ID del botón que ha sido pulsado, seguidamente situaremos cada una de las columnas de la tabla, indicando los datos que van a almacenar y por último, comprobaremos el ID para concoer qué debemos realizar.
Como vemos, al final de cada if o else, tenemos la llamada a una función que será la encargada de realizar todas las operaciones dependiendo del modo de búsqueda elegido.

2.2.2 doSearch

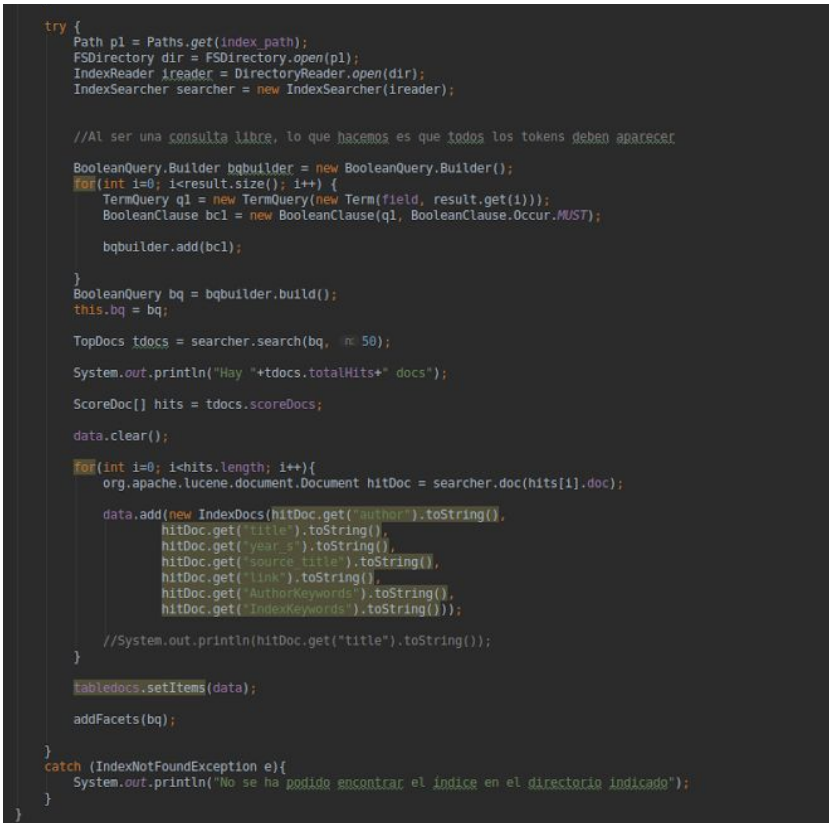
Lo primero que haremos en esta función será comprobar sobre qué campo quiere buscar el usuario para utilizar los distintos tipos de analizadores.

```
private void doSearch() throws IOException {
    Analyzer analyzer = null;
    List<String> result = new ArrayList<>();
    ArrayList<IndexDocs> array = new ArrayList<>();

    switch (field_select){
        case "Author":
            analyzer = new AuthorAnalyzer();
            field = "author";
            break;
        case "Title":
            analyzer = new SimpleAnalyzer();
            field = "title";
            break;
        case "Year":
            analyzer = new WhitespaceAnalyzer();
            field = "year";
            break;
        case "Source Title":
            analyzer = new SimpleAnalyzer();
            field = "source_title";
            break;
        case "Link":
            analyzer = new WhitespaceAnalyzer();
            field = "link";
            break;
        case "Resume":
            analyzer = new EnglishAnalyzer();
            field = "resume";
            break;
        case "AuthorKeywords":
            analyzer = new KeyAnalyzer();
            field = "AuthorKeywords";
            break;
        case "IndexKeywords":
            analyzer = new KeyAnalyzer();
            field = "IndexKeywords";
            break;
        case "EID":
            analyzer = new WhitespaceAnalyzer();
            field = "eid";
            break;
        case "All":
            analyzer = new EnglishAnalyzer();
            field = "content";
            break;
    }
```

Una vez conocido el campo, la consulta y el analizador a utilizar, pasamos a construir los tokens que vamos a buscar.

```
try {
    TokenStream stream  = analyzer.tokenStream( fieldName: null,  new StringReader(search));
    OffsetAttribute offsetAtt = stream.addAttribute(OffsetAttribute.class);
    CharTermAttribute cAtt= stream.addAttribute(CharTermAttribute.class);
    stream.reset();

    while (stream.incrementToken()) {

        //cad = stream.getAttribute(CharTermAttribute.class).toString();
        result.add( cAtt.toString()); // +" : ("+ offsetAtt.startOffset()+"," + offsetAtt.endOffset()+")");
    }
    stream.end();
} catch (IOException e) {
    // not thrown b/c we're using a string reader...
    throw new RuntimeException(e);
}
```

Como vemos en la imagen anterior, creamos un TokenStream con la consulta realizada y el analizador que vayamos a usar dependiendo del campo.
Esto nos devolverá una serie de stream que nos van a permitir crear un Array llamado result, con cada uno de los tokens que vamos a buscar.



```
    try {
        Path p1 = Paths.get(index_path);
        FSDirectory dir = FSDirectory.open(p1);
        IndexReader ireader = DirectoryReader.open(dir);
        IndexSearcher searcher = new IndexSearcher(ireader);


        //Al ser una consulta libre, lo que hacemos es que todos los tokens deben aparecer

        BooleanQuery.Builder bqbuilder = new BooleanQuery.Builder();
        for(int i=0; i<result.size(); i++) {
            TermQuery q1 = new TermQuery(new Term(field, result.get(i)));
            BooleanClause bc1 = new BooleanClause(q1, BooleanClause.Occur.MUST);

            bqbuilder.add(bc1);

        }
        BooleanQuery bq = bqbuilder.build();
        this.bq = bq;

        TopDocs tdocs = searcher.search(bq,  n: 50);

        System.out.println("Hay "+tdocs.totalHits+" docs");

        ScoreDoc[] hits = tdocs.scoreDocs;

        data.clear();

        for(int i=0; i<hits.length; i++){
            org.apache.lucene.document.Document hitDoc = searcher.doc(hits[i].doc);

            data.add(new IndexDocs(hitDoc.get("author").toString(),
                    hitDoc.get("title").toString(),
                    hitDoc.get("year_s").toString(),
                    hitDoc.get("source_title").toString(),
                    hitDoc.get("link").toString(),
                    hitDoc.get("AuthorKeywords").toString(),
                    hitDoc.get("IndexKeywords").toString()));

            //System.out.println(hitDoc.get("title").toString());
        }

        tabledocs.setItems(data);

        addFacets(bq);

    }
    catch (IndexNotFoundException e){
        System.out.println("No se ha podido encontrar el índice en el directorio indicado");
    }
}
```

Finalmente, una vez que ya tenemos nuestro array con los tokens, pasamos a leer el directorio del índice, y comenzamos la creación de la consulta.
Lo que haremos por lo tanto será crear una BooleanQuery que debe contener cada uno de los tokens almacenados en result.
Hecho esto, haremos la consulta que nos devolverá los 50 mejores ScoreDocs y que mostraremos en la tabla de resultados con el último bucle que podemos ver.

2.2.3 doIntSearch

El funcionamiento de esta función es exactamente el mismo y lo único que cambia con respecto a la anterior es la construcción de la consulta.

```
try {
    Path p1 = Paths.get(index_path);
    FSDirectory dir = FSDirectory.open(p1);
    IndexReader ireader = DirectoryReader.open(dir);
    IndexSearcher searcher = new IndexSearcher(ireader);
    Integer startpoint = Integer.valueOf(result.get(0));
    Integer endpoint = Integer.valueOf(result.get(result.size()-1));

    if(startpoint > endpoint){
        Integer aux = startpoint;
        startpoint = endpoint;
        endpoint = aux;
    }

    Query bq = IntPoint.newRangeQuery(field, startpoint, endpoint);


    TopDocs tdocs = searcher.search(bq,  n: 50);

    System.out.println("Hay "+tdocs.totalHits+" docs");

    ScoreDoc[] hits = tdocs.scoreDocs;

    data.clear();

    for(int i=0; i<hits.length; i++){
        org.apache.lucene.document.Document hitDoc = searcher.doc(hits[i].doc);

        data.add(new IndexDocs(hitDoc.get("author").toString(),
                hitDoc.get("title").toString(),
                hitDoc.get("year_s").toString(),
                hitDoc.get("source_title").toString(),
                hitDoc.get("link").toString(),
                hitDoc.get("AuthorKeywords").toString(),
                hitDoc.get("IndexKeywords").toString()));

        //System.out.println(hitDoc.get("title").toString());
    }

    tabledocs.setItems(data);


}
catch (IndexNotFoundException e){
    System.out.println("No se ha pogido encontrar el índice en el directorio indicado");
}
```

Como podemos ver, la consulta recibirá dos valores, que serán los dos extremos del rango sobre el que queremos buscar.
Lo primero que hacemos es comprobar que están introducidos en el orden correcto, y si no es así, ponerlos en el orden bueno.
Y seguidamente, pasamos a construir la consuta.
Ahora, en lugar de usar BooleanQuerys y BooleanClauses, usamos una Query IntPoint. Esta query, çunicamente necesita el campo sobre el que buscar el intervalo y los dos extremos, y ella ya se encarga de recolectar todos los documentos que se encuentre dentro de este intervalo, para posteriormente mostrarlos en la tabla como ya hemos comentado anteriormente.

2.2.4 doBooleanSearch

Esta consulta es la consulta más complicada de los 3 tipos, puesto que debemos tratar nosotros las consultas realizadas por el usuario para pasarlas a Lucene.

Una consulta booleana es del tipo:

information or retrieval

Nosotros debemos ser capaces de identificar ese operador “or” para decirle a lucen que no es necesario que retrieval aparezca en el documento.
Por ello, para cada tipo de analizador debemos realizar un tratado de la consulta diferente por los motivos que veremos a continuación.

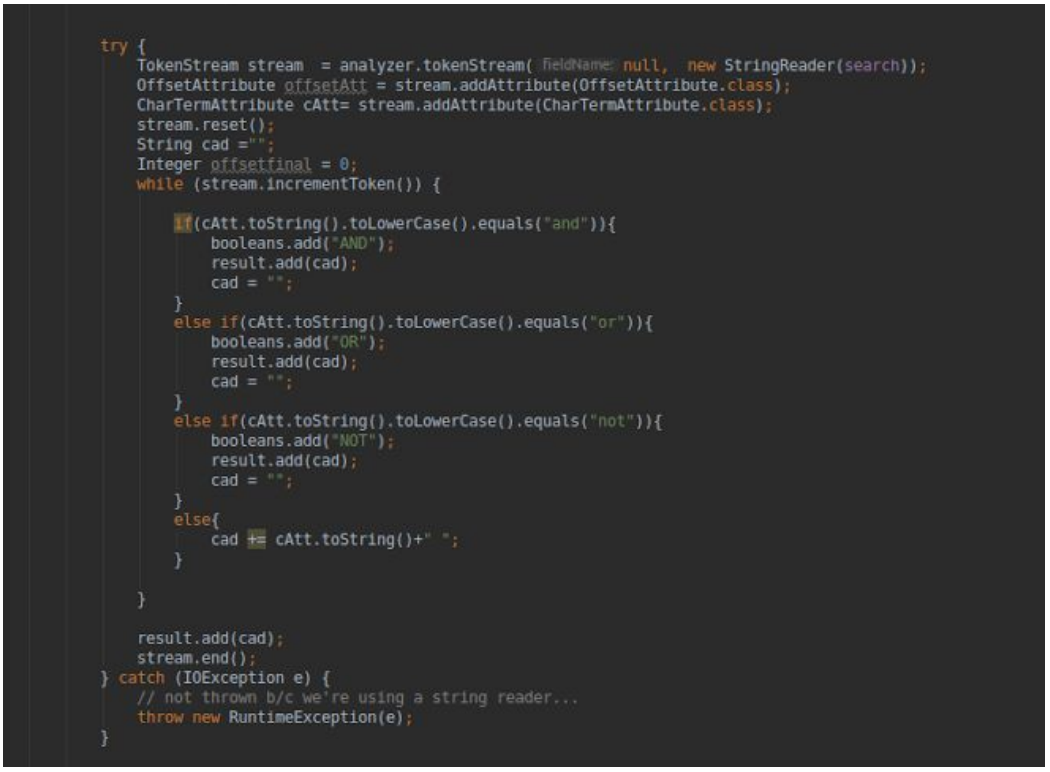
Comenzamos por ejemplo con el cas de EnglishAnalyzer().
Este analizador, cuando lo pasamos sobre la consulta recibida, elimina palabras vacías, y por lo tanto, las palabras OR, AND o NOT que nos van a indicar los operadores que vamos a realizar, las va a eliminar, por lo que nos tendremos que fijar en el offset de cada atributo para saber si entre cada par de tokens había alguna palabra vacía en medio que comprobaremos que sea uno de esos tres operadores.

```
        break;
    case "All":
        analyzer = new EnglishAnalyzer();
        field = "content";


        try {
            TokenStream stream  = analyzer.tokenStream( fieldName: null,  new StringReader(search));
            OffsetAttribute offsetAtt = stream.addAttribute(OffsetAttribute.class);
            CharTermAttribute cAtt= stream.addAttribute(CharTermAttribute.class);
            stream.reset();
            String cad ="";
            Integer offsetfinal = 0;
            while (stream.incrementToken()) {
                if(offsetAtt.startOffset() != 0){
                    cad = search.substring(offsetfinal, offsetAtt.startOffset());

                    cad = cad.trim();

                    if(cad.trim().contains(" ")){
                        Integer i1 = cad.indexOf(" ");
                        Integer i2 = cad.length();
                        cad = cad.substring(cad.indexOf(" "), cad.length());
                        cad = cad.trim();
                    }

                }
                else{
                    booleans.add("AND");
                }

                if(cad.equals("and") || cad.equals("AND")){
                    booleans.add("AND");
                }
                else if(cad.equals("or") || cad.equals("OR")){
                    booleans.add("OR");
                }
                else if(cad.equals("not") || cad.equals("NOT")){
                    booleans.add("NOT");
                }

                //cad = stream.getAttribute(CharTermAttribute.class).toString();!+
                result.add( cAtt.toString()); // +" : ("+ offsetAtt.startOffset()+"," + offsetAtt.endOffset()+")");
                offsetfinal = offsetAtt.endOffset();
            }
            stream.end();
        } catch (IOException e) {
            // not thrown b/c we're using a string reader...
            throw new RuntimeException(e);
        }


        break;
}
```

Como podemos observar, por cada token, miramos si existe una palabra entre medias, y comprobamos si es alguno de los 3 operadores.
En el caso de que así sea, almacenaremos en un array el operador que se ha utilizado, para posteriormente leerlo y conocer de qué tipo será la consulta.

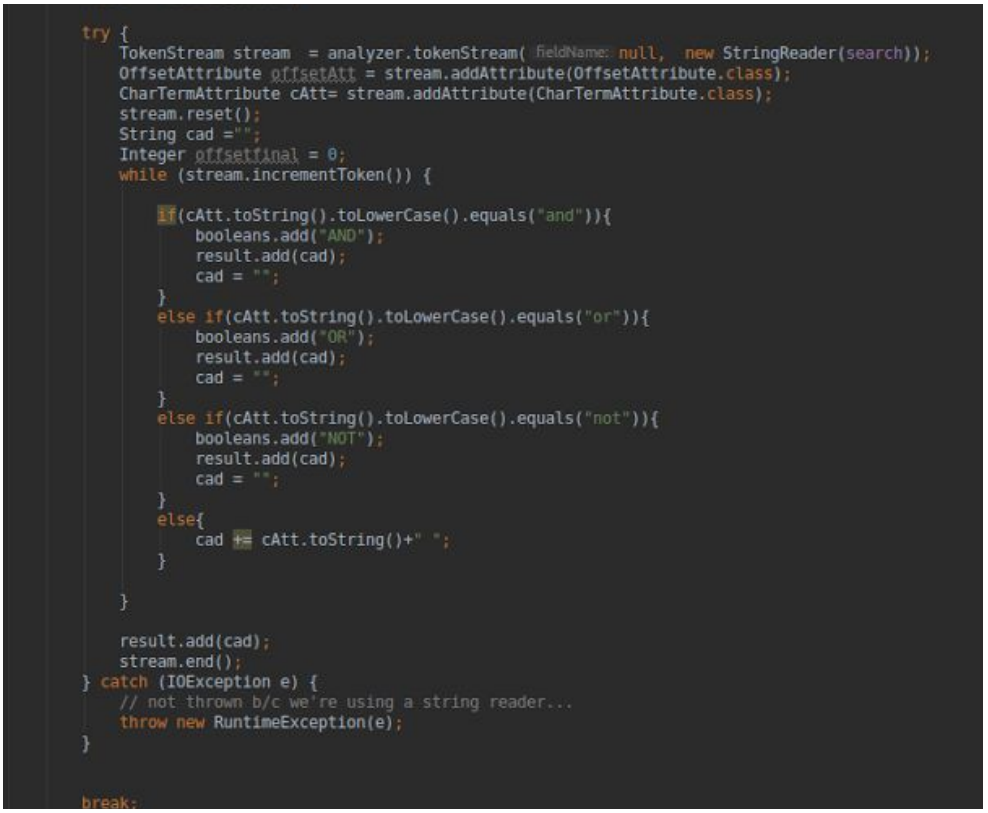
Otro caso es el de WhiteSpaceAnalyzer, que este almacena como tokens toda la consulta, por cada espacio en blanco genera un token.

Lo que hacemos por lo tanto es, unir los tokens que aparezcan antes de cada operador en uno y almacenarlo cuando se encuentre un operador.
Es decir, si tenemos la consulta “INFORMATION RETRIEVAL AND MD5” almacenaremos en un token, information retrieval y en otro md5.



```
try {
    TokenStream stream  = analyzer.tokenStream( fieldName: null,  new StringReader(search));
    OffsetAttribute offsetAtt = stream.addAttribute(OffsetAttribute.class);
    CharTermAttribute cAtt= stream.addAttribute(CharTermAttribute.class);
    stream.reset();
    String cad ="";
    Integer offsetfinal = 0;
    while (stream.incrementToken()) {

        if(cAtt.toString().toLowerCase().equals("and")){
            booleans.add("AND");
            result.add(cad);
            cad = "";
        }
        else if(cAtt.toString().toLowerCase().equals("or")){
            booleans.add("OR");
            result.add(cad);
            cad = "";
        }
        else if(cAtt.toString().toLowerCase().equals("not")){
            booleans.add("NOT");
            result.add(cad);
            cad = "";
        }
        else{
            cad += cAtt.toString()+" ";
        }

    }

    result.add(cad);
    stream.end();
} catch (IOException e) {
    // not thrown b/c we're using a string reader...
    throw new RuntimeException(e);
}
```

El siguiente analizador es el SimpleAnalyzer(). El comportamiento de este Analyzer con respecto a los operadores que nosotros queremos es el mismo que el comportamiento que tiene el WhiteSpaceAnalyzer, y es que las trata como otro token más, por lo que el código que veremos a continuación, podremos ver que es prácticamente igual que el códugo utilizado con el WhiteSpaceAnalyzer, puesto que el tratamiento es el mismo.



```
try {
    TokenStream stream  = analyzer.tokenStream( fieldName: null,  new StringReader(search));
    OffsetAttribute offsetAtt = stream.addAttribute(OffsetAttribute.class);
    CharTermAttribute cAtt= stream.addAttribute(CharTermAttribute.class);
    stream.reset();
    String cad ="";
    Integer offsetfinal = 0;
    while (stream.incrementToken()) {

        if(cAtt.toString().toLowerCase().equals("and")){
            booleans.add("AND");
            result.add(cad);
            cad = "";
        }
        else if(cAtt.toString().toLowerCase().equals("or")){
            booleans.add("OR");
            result.add(cad);
            cad = "";
        }
        else if(cAtt.toString().toLowerCase().equals("not")){
            booleans.add("NOT");
            result.add(cad);
            cad = "";
        }
        else{
            cad += cAtt.toString()+" ";
        }

    }

    result.add(cad);
    stream.end();
} catch (IOException e) {
    // not thrown b/c we're using a string reader...
    throw new RuntimeException(e);
}


break;
```

Por último nos quedan los campos que hemos tratado con los analizadores creados por mí mismo, como son el caso de los campos con palabras clave y el campo de autor.

Estos analizadores, dividían los tokens por cada “,” que se encontrara en el caso del camp de autor o bien por cada “;” en el caso de los campos de Keywords.

Por lo tanto, lo que hacemos es transformar la consulta con las funciones creadas correspondientes.

```java
// Función que nos permitirá transformar la consulta que escriba el usuario, de forma que cada vez que encuentre
// un operador booleano significará quetodo lo anterior era un token y por lo tanto introducirá una "," para separar
// dichos tokens
private String trasformsearch(String search){
    String aux = "";
    String[] tokens = search.split( s: " ");
    ArrayList<String> search_good = new ArrayList();
    int ultimo = 0;

    for(int i = ultimo; i<tokens.length; i++){
        String word = tokens[i];
        if(word.toLowerCase().equals("or") || word.toLowerCase().equals("and") || word.toLowerCase().equals("not")){
            String token = "";
            for(int j=ultimo; j<i; j++){
                token += " " + tokens[j];
            }

            token += ",";
            search_good.add(token);
            ultimo = i+1;
        }
    }

    String token = "";
    for (int i=ultimo; i<tokens.length; i++){
        token += " " + tokens[i];

    }
    token += ",";
    search_good.add(token);

    for(int i=0; i<search_good.size(); i++){
        aux += search_good.get(i);
    }

    return aux;

}
```

Esta función lo que nos hará será que cada vez que encuentre uno de los tres operadores, introduzca una coma (en el caso de los autores) para separar los distintos tokens.
Es decir, si tenemos la consulta:

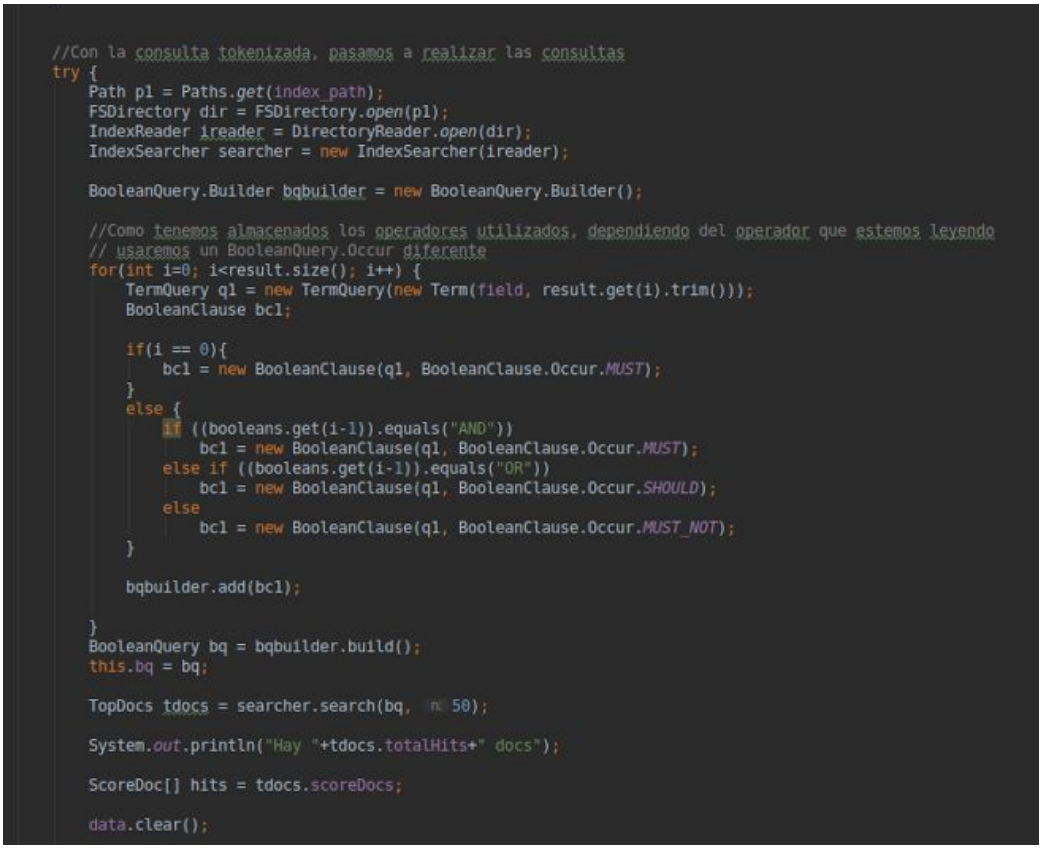
Miguel de Cervantes OR Valle-Inclán

Al pasarle esta función, esta nos devolverá la siguiente consulta:

Miguel de Cervantes, Valle-Inclán

Además de esto, usaremos otra función para recorrer la consulta y obtener los operadores utilizados.

Con la consulta transformada y los operadores obtenidos, únicamente queda crear la consulta.

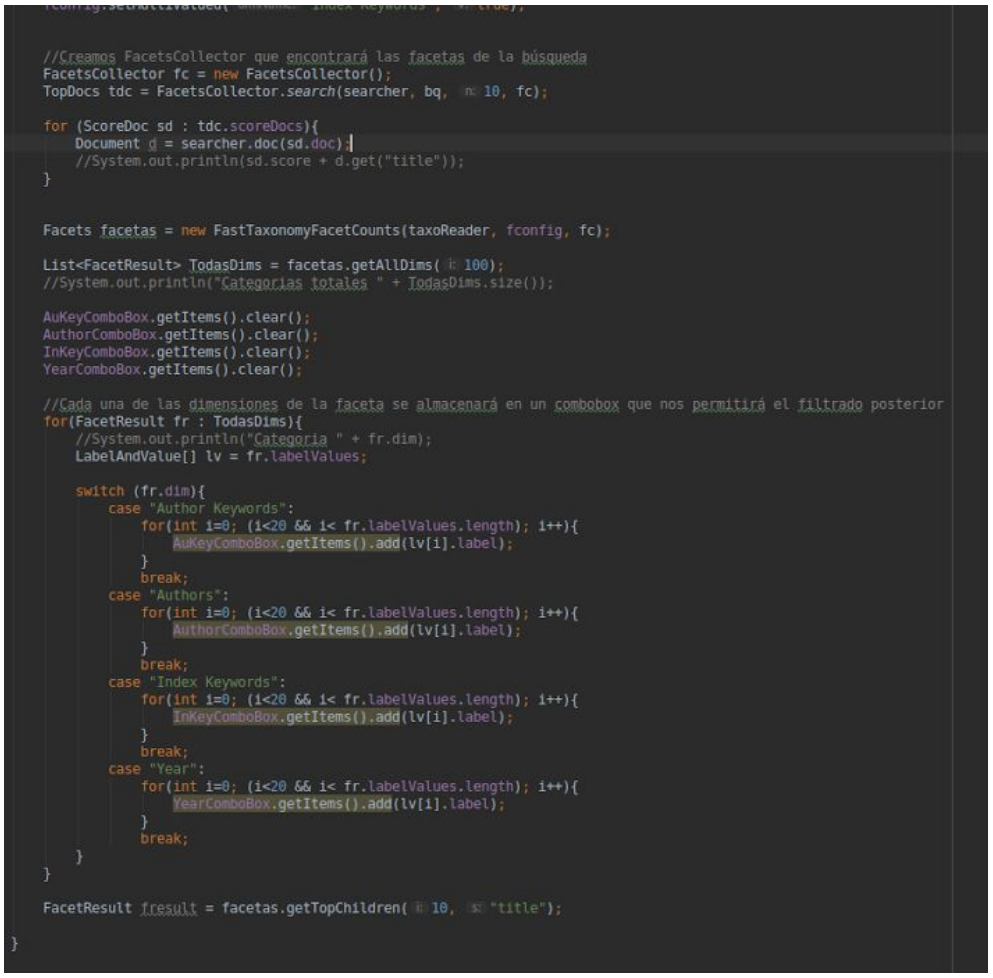


```
//Con la consulta tokenizada, pasamos a realizar las consultas
try {
    Path p1 = Paths.get(index_path);
    FSDirectory dir = FSDirectory.open(p1);
    IndexReader ireader = DirectoryReader.open(dir);
    IndexSearcher searcher = new IndexSearcher(ireader);

    BooleanQuery.Builder bqbuilder = new BooleanQuery.Builder();

    //Como tenemos almacenados los operadores utilizados, dependiendo del operador que estemos leyendo
    // usaremos un BooleanQuery.Occur diferente
    for(int i=0; i<result.size(); i++) {
        TermQuery q1 = new TermQuery(new Term(field, result.get(i).trim()));
        BooleanClause bc1;

        if(i == 0){
            bc1 = new BooleanClause(q1, BooleanClause.Occur.MUST);
        }
        else {
            if ((booleans.get(i-1)).equals("AND"))
                bc1 = new BooleanClause(q1, BooleanClause.Occur.MUST);
            else if ((booleans.get(i-1)).equals("OR"))
                bc1 = new BooleanClause(q1, BooleanClause.Occur.SHOULD);
            else
                bc1 = new BooleanClause(q1, BooleanClause.Occur.MUST_NOT);
        }

        bqbuilder.add(bc1);

    }
    BooleanQuery bq = bqbuilder.build();
    this.bq = bq;

    TopDocs tdocs = searcher.search(bq,  n: 50);

    System.out.println("Hay "+tdocs.totalHits+" docs");

    ScoreDoc[] hits = tdocs.scoreDocs;

    data.clear();
```

Únicamente tendremos que ir emparejando cada uno de los tokens con los operadores que hemos ido almacenando en el array de operadores para construir la consulta, como podemos ver en la anterior imagen.

2.2.5 addFacets

Ahora, con la consulta realizada y mostrada por pantalla, pasamos a obtener las facetas para esa consulta.



```
    //Creamos FacetsCollector que encontrará las facetas de la búsqueda
    FacetsCollector fc = new FacetsCollector();
    TopDocs tdc = FacetsCollector.search(searcher, bq,  n: 10, fc);

    for (ScoreDoc sd : tdc.scoreDocs){
        Document d = searcher.doc(sd.doc);
        //System.out.println(sd.score + d.get("title"));
    }


    Facets facetas = new FastTaxonomyFacetCounts(taxoReader, fconfig, fc);

    List<FacetResult> TodasDims = facetas.getAllDims( i: 100);
    //System.out.println("Categorias totales " + TodasDims.size());

    AuKeyComboBox.getItems().clear();
    AuthorComboBox.getItems().clear();
    InKeyComboBox.getItems().clear();
    YearComboBox.getItems().clear();

    //Cada una de las dimensiones de la faceta se almacenará en un combobox que nos permitirá el filtrado posterior
    for(FacetResult fr : TodasDims){
        //System.out.println("Categoria " + fr.dim);
        LabelAndValue[] lv = fr.labelValues;

        switch (fr.dim){
            case "Author Keywords":
                for(int i=0; (i<20 && i< fr.labelValues.length); i++){
                    AuKeyComboBox.getItems().add(lv[i].label);
                }
                break;
            case "Authors":
                for(int i=0; (i<20 && i< fr.labelValues.length); i++){
                    AuthorComboBox.getItems().add(lv[i].label);
                }
                break;
            case "Index Keywords":
                for(int i=0; (i<20 && i< fr.labelValues.length); i++){
                    InKeyComboBox.getItems().add(lv[i].label);
                }
                break;
            case "Year":
                for(int i=0; (i<20 && i< fr.labelValues.length); i++){
                    YearComboBox.getItems().add(lv[i].label);
                }
                break;
        }
    }

    FacetResult fresult = facetas.getTopChildren( i: 10,  s: "title");

}
```

Para ello creamos un FacetsCollector con la consulta y el número de facetas a coger, así como la configuración de ellas.
Con este FacetsCollector creado pasaremos a crear un FastTaxonomyFacetCounts, que nos permitirá obtener las distintas dimensiones existentes en las facetas y de cada una de ellas obtener los elementos que representen a los documentos de la consulta.
Todo este proceso puede verse en la imagen anterior.

2.2.6 filterAction

Por último queda aplicar el filtro de las facetas seleccionadas sobre los documentos de la consulta.

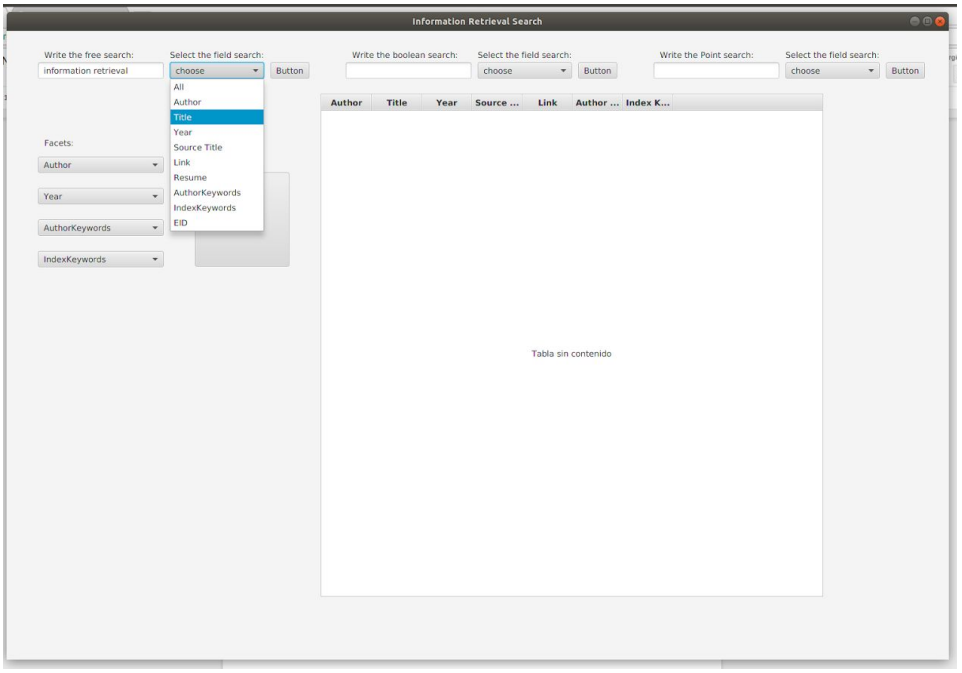
```
public void filterAction() throws IOException {
    Path p = Paths.get(index_path);
    FSDirectory indexDir = FSDirectory.open(p);
    DirectoryReader indexReader = DirectoryReader.open(indexDir);
    IndexSearcher searcher = new IndexSearcher(indexReader);
    Path p1 = Paths.get(facets_path);
    FSDirectory taxoDir = FSDirectory.open(p1);
    TaxonomyReader taxoReader = new DirectoryTaxonomyReader(taxoDir);

    DrillDownQuery dq = new DrillDownQuery(fconfig, bq);

    //Obtenemos las facetas seleccionadas de cada categoría

    if(AuthorComboBox.getValue() != null){
        dq.add( dim: "Authors", AuthorComboBox.getValue().toString());
    }
    if(AuKeyComboBox.getValue() != null){
        dq.add( dim: "Author Keywords", AuKeyComboBox.getValue().toString());
    }
    if(InKeyComboBox.getValue() != null){
        dq.add( dim: "Index Keywords", InKeyComboBox.getValue().toString());
    }
    if(YearComboBox.getValue() != null){
        dq.add( dim: "Year", YearComboBox.getValue().toString());
    }


    // Con las facetas seleccionadas y almacenadas en DrillDownQuery filtramos los resultados
    DrillSideways ds = new DrillSideways(searcher, fconfig, taxoReader);
    DrillSideways.DrillSidewaysResult dsresult = ds.search(dq,  topN: 100);
    //System.out.println("dsw hits "+dsresult.hits.totalHits);
    //System.out.println(dsresult.facets.getAllDims(100).toString());

    data.clear();
    for(ScoreDoc scoreDoc:dsresult.hits.scoreDocs){
        //Document doc = searcher.doc(scoreDoc.doc);
        //System.out.println("    Docs Score -> " + scoreDoc.score + "::" + doc.get("title"));

        org.apache.lucene.document.Document hitDoc = searcher.doc(scoreDoc.doc);
        data.add(new IndexDocs(hitDoc.get("author").toString(),
                hitDoc.get("title").toString(),
                hitDoc.get("year_s").toString(),
                hitDoc.get("source_title").toString(),
                hitDoc.get("link").toString(),
                hitDoc.get("AuthorKeywords").toString(),
                hitDoc.get("IndexKeywords").toString()));

    }

    tabledocs.setItems(data);
    addFacets(bq);

}
```

Lo que haremos para ello es crear un DrillDownQuery y un DrillSideways que sobre la consulta realizada aplique las facetas que el usuario haya seleccionado sobre las listas de estas.

3. MANUAL FINAL DE USUARIO

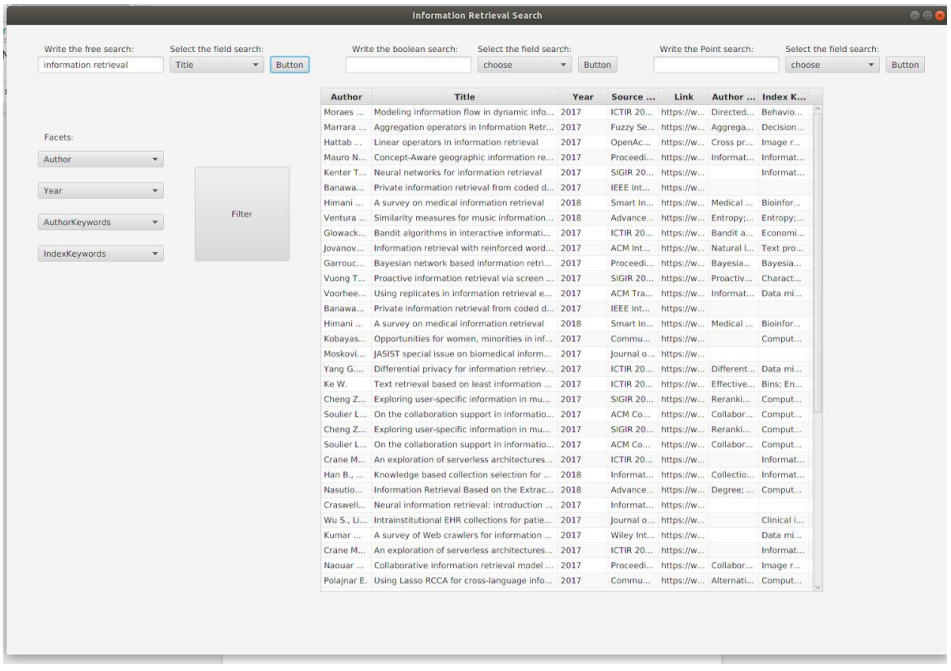
Por último vamos a ver cómo poder utilizar la aplicación.
Vamos a comenzar primero por la consulta libre.

En esta consulta el usuario escribirá la consulta como él desee y seleccionará el campo sobre el que quiere que se ejecute la consulta, pudiendo no seleccionar ninguno, y la consulta se ejecutará sobre todo el contenido del documento.

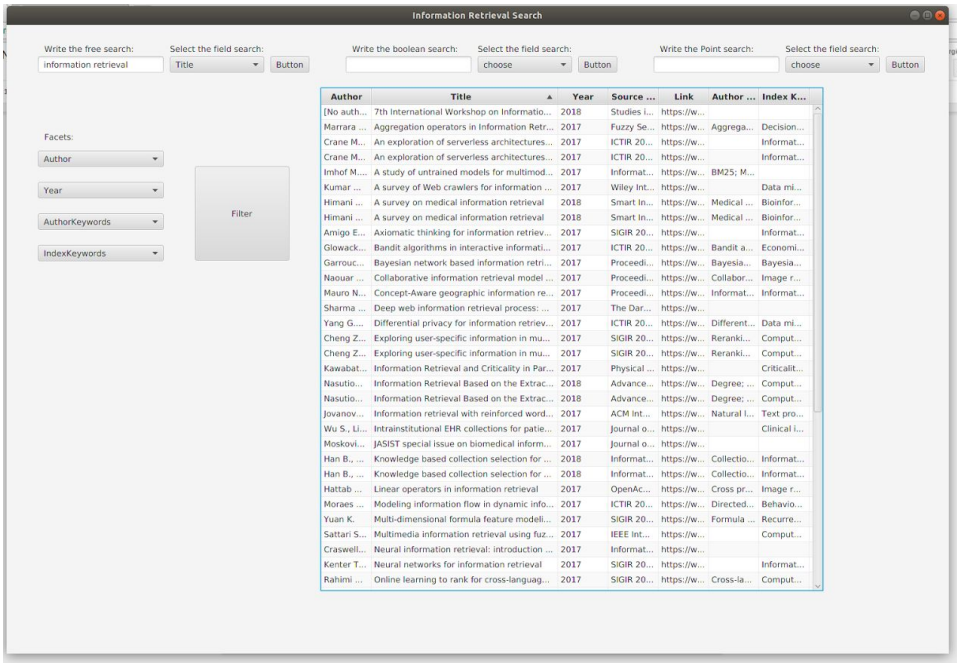


Como vemos, escribimos la consulta y seleccionamos el campo, y hecho esto daremos sobre el botón de **Buscar**.

Al hacer esto, se mostrarán los resultados por la tabla, ordenados por Score.

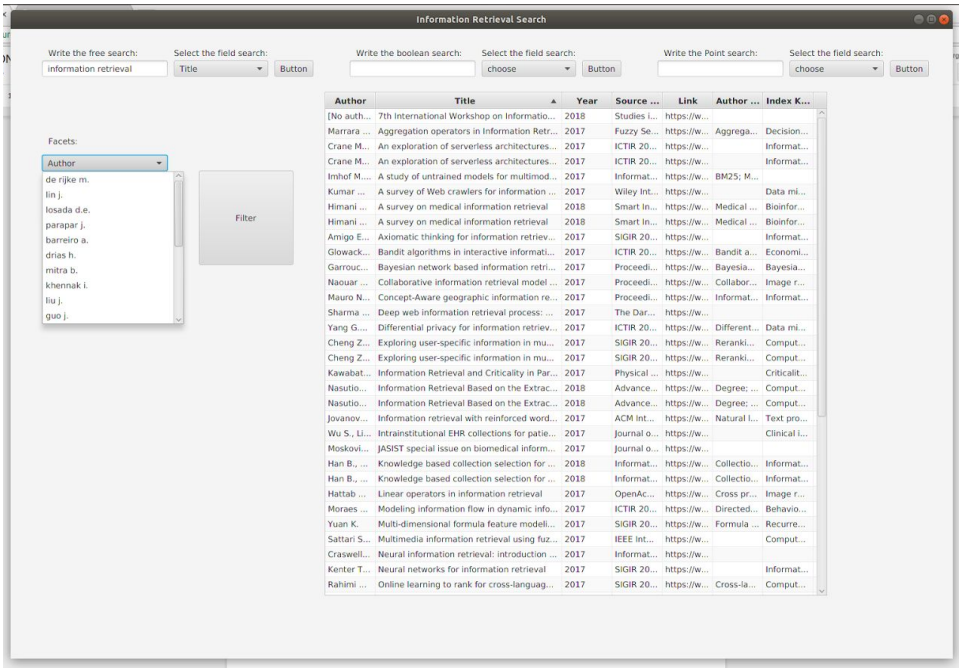


En el caso de que queramos ordenarlos por ejemplo, por orden alfabético por título, únicamente debemos hacer doble clic sobre la cabecera de la columna.

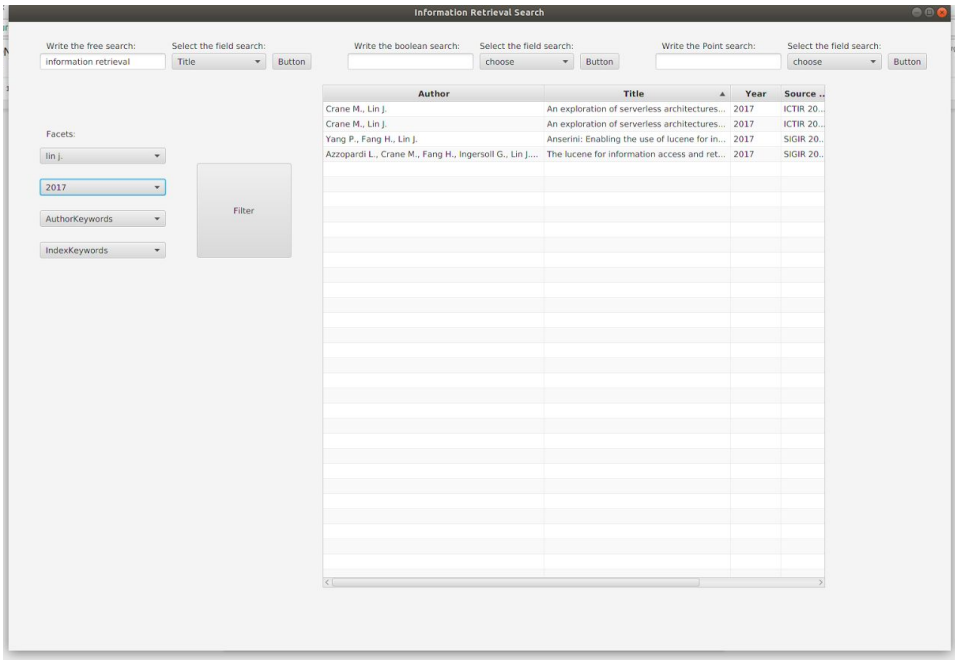


Ahora pasaremos a aplicar filtros en las facetas.

Al realizar la consulta, vemos que ya nos aparecen listas de facetas como vemos a continuación.

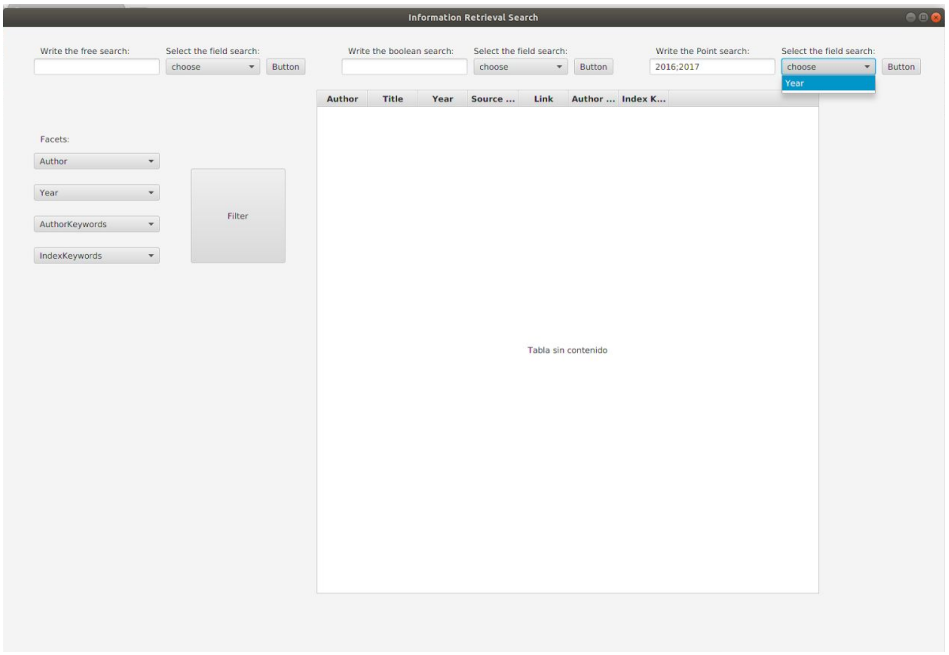


Ahora podremos seleccionar las facetas que queramos y pulsar sobre el botón **Filter** para filtrar la búsqueda.
Para mostar el ejemplo se va a aplicar un filtro en el que uno de los autores sea **Lin J.** y el artículo sea de 2017.



De esta forma, podemos ver claramente cómo se ha aplicado el filtro de facetas sobre la búsqueda.

Ahora vamos a pasar a mostrar las consultas por rango.
Puesto que únicamente en el índice hay un campo IntPoint, sólamente se podrá buscar por rango en el Año.
Así que para poner el rango separaremos los dos números por **“;”.**



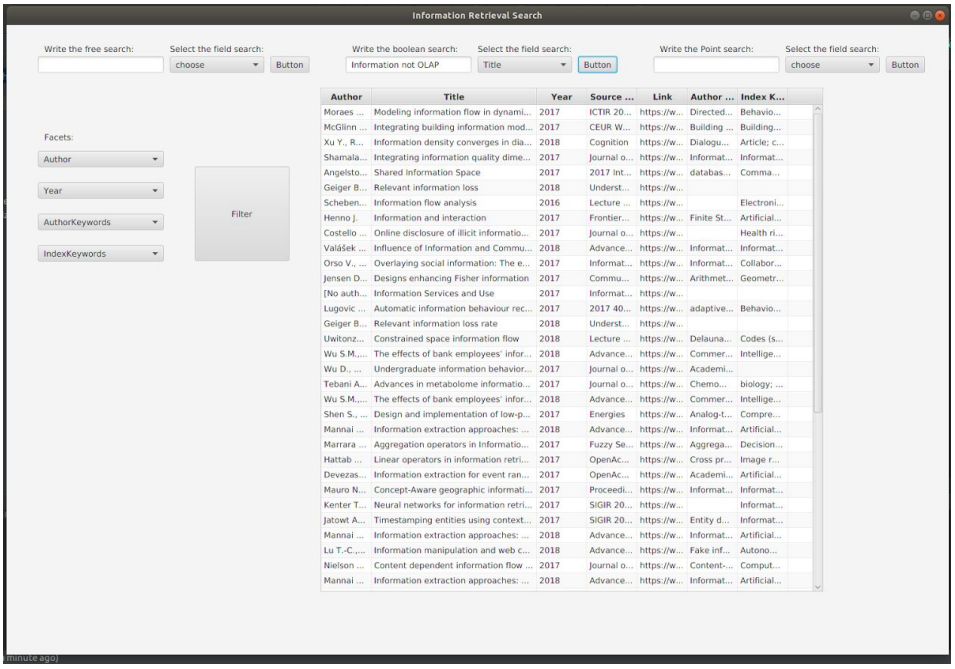
De esta forma obtendremos aquellos artículos que sean o bien del año 2016 o bien del año 2017.

Y por último, queda la consulta booleana.

La consulta booleana hace uso de tres operadores:

- AND : El término debe aparecer
- OR : El término puede aparecer
- NOT : El término no debe aparecer

Con esto, el usuario podrá construir consultas, de forma que pueda especificar bastante en su consulta.
El usuario podrá indicar consultas del siguiente tipo:



Y de esta forma se mostrarán aquellos artículos que contengan en el título Information pero que no contengan OLAP.