RI: Práctica 3

UNIVERSIDAD DE GRANADA

Juan Manuel Consigliere Picco
Óscar Pérez Tobarra

Índice

1. Creación del índice con Lucene:

Partiendo del ejemplo proporcionado en el guión de prácticas, hemos creado el programa IndiceSimple.java, que permite crear un índice Lucene y rellenarlo con los documentos que se encuentren en la carpeta docs, para posteriormente almacenarlo en la carpeta index. También con ello hemos creado las facetas. El comando para su ejecución es el siguiente:

```
java -cp
.:./lib/commons-beanutils-1.9.4.jar:./lib/commons-collections4-4.4
.jar:./lib/commons-lang3-3.12.0.jar:./lib/commons-text-1.9.jar:./l
ib/opencsv-5.5.2.jar:./lucene-8.10.1/core/lucene-core-8.10.1.jar:.
/lucene-8.10.1/analysis/common/lucene-analyzers-common-8.10.1.jar
indiceSimple.java
```

Veamos el código. Se ha definido una clase IndiceSimple que contiene unas variables de clase, un constructor y dos métodos. Empezaremos primero por las variables de clase:

```java
public class indiceSimple {
    String indexPath = "./index";
    String taxoPath = "./taxo";
    String docPath = "./docs";
    boolean create = true;
    private IndexWriter indexWriter;
    private DirectoryTaxonomyWriter taxoWriter;
    private FacetsConfig fconfig;
    Map<String, Analyzer> analyzerPerField = new HashMap<String, Analyzer>();
```

Tenemos tres variables que sirven para especificar la ruta relativa donde se encuentran los documentos a indexar (docPath), donde se alojarán los índices (indexPath), y donde las facetas (taxoPath). También tenemos una variable IndexWriter, que necesitaremos posteriormente para poder crear los índices, otra variable DirectoryTaxonomyWriter para crear las facetas, y un map de Strings y Analyzers, que nos servirá para indicar los analizadores que se usarán en cada campo indexado.

```java
indiceSimple(){
    analyzerPerField.put("author", new SimpleAnalyzer());
    analyzerPerField.put("title", new SimpleAnalyzer());
    analyzerPerField.put("source_title", new SimpleAnalyzer());
    analyzerPerField.put("affiliations", new SimpleAnalyzer());
    analyzerPerField.put("abstract", new EnglishAnalyzer());
    analyzerPerField.put("author_keywords", new SimpleAnalyzer());
    analyzerPerField.put("index_keywords", new SimpleAnalyzer());
}
```

Aquí observamos para los distintos campos que vamos a usar qué analyzer usaremos. Todos excepto abstract los hemos puesto con SimpleAnalyzer, y abstract con EnglishAnalyzer, que posteriormente servirá para que en la búsqueda podamos buscar palabras parecidas que coincidan en su raíz.

Pasamos a ver primero los métodos antes que el constructor, pues no podemos explicar la funcionalidad del constructor sin explicar primero la del método para crear y añadir documentos Lucene al índice.

Tenemos primero el método configurarIndice:

```
public void configurarIndice(Analyzer analyzer, Similarity similarity) throws IOException {
    PerFieldAnalyzerWrapper analyzerWrapper = new PerFieldAnalyzerWrapper(analyzer, analyzerPerField);

    IndexWriterConfig iwc = new IndexWriterConfig(analyzerWrapper);

    iwc.setSimilarity(similarity);
    iwc.setOpenMode(IndexWriterConfig.OpenMode.CREATE);

    Directory indexDir = FSDirectory.open(Paths.get(indexPath));
    Directory taxoDir = FSDirectory.open(Paths.get(taxoPath));

    fconfig = new FacetsConfig();
    taxoWriter = new DirectoryTaxonomyWriter(taxoDir);
    indexWriter = new IndexWriter(indexDir, iwc);

    fconfig.setMultiValued("year",true);
    fconfig.setMultiValued("category",true);
    fconfig.setMultiValued("doc_type",true);
}
```

No varía del ejemplo provisto por el guión de prácticas, salvo por el analizador que le pasamos como parámetro al IndexWriterConfig, que es el map que hemos visto antes. También añadimos las líneas de código para configurar el directorio de las facetas, inicializamos el taxoWriter, y finalmente escogemos las facetas y se las pasamos a fconfig, especificando si son multivaluadas o no. Estas serán year, category y doctype. Este método se ejecutará antes de procesar y crear los documentos Lucene.

El segundo método es donde se encuentra lo más importante: la creación de documentos Lucene.

```
public void indexarDocumentos(File archivo) throws IOException, CsvException {
    CSVReader reader = new CSVReader(new FileReader(docPath+"/"+archivo.getName()));
    reader.skip(1);
    String[] r = reader.readNext();
    System.out.println("Archivo "+archivo.getName());

    do{
        Document doc = new Document();
        doc.add(new TextField(  "category",
                                archivo.getName().substring(0, archivo.getName().lastIndexOf(".")),
                                Field.Store.YES));
        doc.add(new TextField("author", r[0], Field.Store.YES));
        doc.add(new StoredField("author_id", r[1]));
        doc.add(new TextField("title", r[2], Field.Store.YES));
        if(!r[3].equals("")){
            doc.add(new IntPoint("year", Integer.parseInt(r[3])));
            doc.add(new StoredField("year", Integer.parseInt(r[3])));
        }
        doc.add(new TextField("source_title", r[4], Field.Store.YES));
        doc.add(new StoredField("volume", r[5]));
        doc.add(new StoredField("issue", r[6]));
        doc.add(new StoredField("article_number", r[7]));
        doc.add(new StoredField("page_start", r[8]));
        doc.add(new StoredField("page_end", r[9]));
        doc.add(new StoredField("page_count", r[10]));
        if(!r[11].equals("")){
            doc.add(new NumericDocValuesField("cited_by", Integer.parseInt(r[11])));
            doc.add(new StoredField("cited_by", Long.parseLong(r[11])));
        }
```

El método recibe como parámetro un archivo, que es un archivo CSV obtenido de Scopus y que contiene 2000 documentos, el cual se le pasa a un lector de archivos CSV para poder procesarlo. Una vez cargado el lector, avanzamos una línea (la que incluye el nombre de cada columna) y empezamos a leer el archivo. Por cada fila leída, se creará un documento Lucene y se añadirá al IndexWriter. A continuación explicaremos los tipos que hemos escogido para campo:

- category: el nombre del archivo CSV (podríamos considerarlo la temática). Se ha definido como un TextField para poder permitir que se indexe mediante los tokens que contiene el campo (por ejemplo, natural/language/processing). Se especifica que el campo se almacene para devolverlo con “Field.Store.YES”. Es una faceta también.
- author: los creadores del documento a procesar. Se ha definido como un TextField porque el campo puede incluir múltiples autores, y se ha de permitir la búsqueda por cada uno de ellos. También se especifica que se almacene el campo para poder devolverlo en consultas.
- author_id: el identificador de los creadores del documento a procesar. Se ha definido como un StoredField, ya que no es común realizar búsquedas por identificadores.
- title: el título del documento. Se ha definido como un TextField para permitir la búsqueda por cada uno de los términos que componen el título, además de que es un atributo que suele formar parte de las búsquedas. También se especifica que se almacene el campo para poder devolverlo en consultas.
- year: el año en el que se publicó el documento. Se ha definido tanto como un IntPoint, como un StoredField. Esto será una faceta.
- source_title: el título de la fuente del documento (una revista científica, por ejemplo). Es un TextField, pues puede tener sentido realizar búsquedas por publicaciones científicas de confianza, y por lo tanto hay que permitir la búsqueda por los tokens que componen el campo. También se especifica que se almacene el campo para poder devolverlo en consultas.
- volume, issue, article_number, page_start, page_end, page_count: atributos del documento con respecto a la revista científica que lo publica. No tiene sentido indexar por ellos, así que se almacenan como StoredFields para poder devolverlos como atributos del documento.
- cited_by: el número de citas o referencias que ha tenido el documento. Se ha definido tanto como un SortedNumericDocValuesField como un StoredField. La razón es poder utilizarlo más adelante como un medidor de la relevancia del documento cuando implementemos la recuperación de documentos.

```
doc.add(new StoredField("doi", r[12]));
doc.add(new StoredField("link", r[13]));
doc.add(new TextField("affiliations", r[14], Field.Store.YES));
doc.add(new TextField("abstract", r[16], Field.Store.YES));
doc.add(new TextField("author_keywords", r[17], Field.Store.NO));
doc.add(new TextField("index_keywords", r[18], Field.Store.NO));
doc.add(new StoredField("doc_type", r[19]));
doc.add(new StoredField("public_status", r[20]));
doc.add(new StoredField("eid", r[23]));
```

- doi: el Identificador de Objeto Digital (DOI) del documento. Se ha definido como un StoredField, ya que no vamos a buscar por DOI.
- link: enlace que lleva al documento completo. No tiene sentido indexar por un enlace, así que se define como un StoredField y poder devolverlo en caso de que alguien quiera acceder al documento original.
- affiliation: entidades a las que están asociadas los autores del documento (por ejemplo, una universidad). Se ha definido como un TextField porque el campo incluye la afiliación de todos los autores, así que debería permitirse buscar por cada una de las entidades a las que están vinculados los autores. También se especifica que se almacene el campo para poder devolverlo en consultas.
- abstract: resumen del documento. Se define como un TextField para poder indexar por los tokens del abstract, lo cual es muy común cuando se están buscando artículos científicos. También se especifica que se almacene el campo para poder devolverlo en consultas.
- author_keywords: palabras clave relacionadas con los autores. Se ha definido como un TextField para permitir una búsqueda más rápida por términos, aunque en este caso no lo almacenaremos para devolverlo en las consultas porque consideramos que no incluye nada que no podamos mostrar con el campo “abstract”.
- index_keywords: palabras clave usadas por el índice (no el nuestro). Nos puede servir para nuestro propio índice, por lo que lo definimos como un TextField sin almacenarlo.
- doc_type: tipo del documento (artículo, libro, etc). Lo definimos como StoredField. Va a ser otra de las facetas.
- public_status: estado de publicación del documento. No se suele buscar por el estado de publicación, por lo que lo definimos como un StoredField para poder devolverlo en las consultas.
- eid: identificador asociado al documento en la base de datos de Scopus. No tiene sentido indexar por este campo teniendo el DOI como un identificador más importante, por lo que lo definimos como un StoredField.

Ahora vemos las tres facetas:

```
    doc.add(new FacetField("category",archivo.getName().substring
    doc.add(new FacetField("year",r[3]));
    doc.add(new FacetField("doc_type", r[19]));
    indexWriter.addDocument(fconfig.build(taxoWriter,doc));
    r = reader.readNext();
}while(r!=null);
```

- category: Hemos decidido incluirla, ya que tiene mucho sentido filtrar documentos por categoría. Puede verse también como cuando entramos en una web de ropa online y vemos la sección de pantalones, camisetas, zapatos, etc.
- year: Si queremos recuperar artículos de los años disponibles es la opción más sencilla.
- doc_type: Muchas veces podemos querer sacar artículos, revistas, reportajes, cartas, etc, por lo que filtrar por esto también es relevante.

Una vez explicada la funcionalidad de la clase IndiceSimple, podemos ver cómo realiza la ejecución el programa:

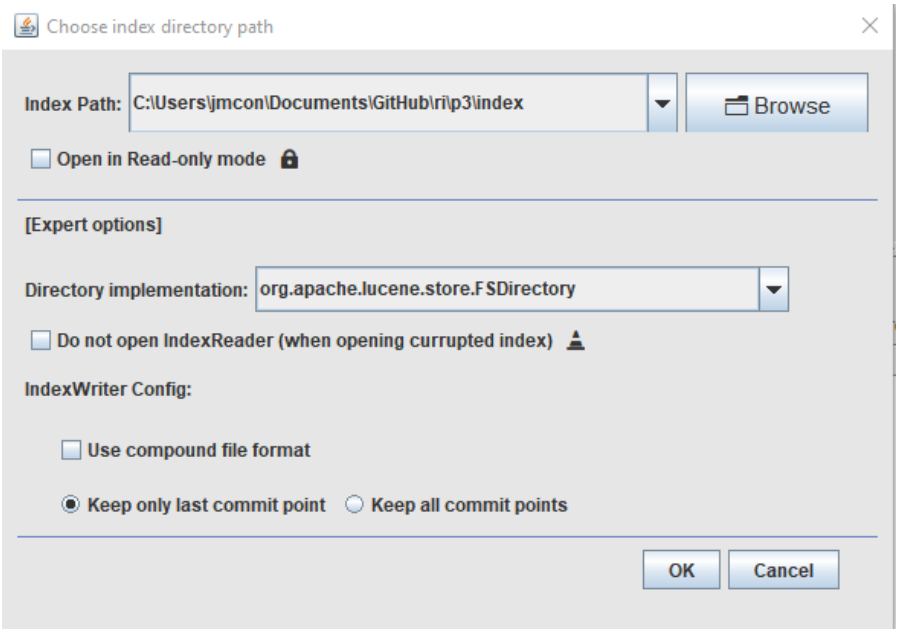
```
    public static void main(String[] args) throws IOException, CsvException {
        File dir = new File("./docs");
        File[] archivos = dir.listFiles();
        Analyzer analyzer = new StandardAnalyzer();
        Similarity similarity = new ClassicSimilarity();
        indiceSimple baseline = new indiceSimple();

        baseline.configurarIndice(analyzer, similarity);
        for(File archivo : archivos){
            baseline.indexarDocumentos(archivo);
        }
        baseline.close();
    }
}
```

Se define un analizador por defecto que pasarle al IndexWriter, así como un objeto Similarity para que Lucene pueda medir el peso de cada término (no es relevante en este momento). También obtenemos los ficheros que contienen los documentos a indexar. El procedimiento es llamar al constructor de IndiceSimple para que cree el analyzerPerFieldWrapper, que es común a todos los documentos ya que comparten estructura, configurar el índice para indicarle el analizador por defecto, y una vez configurado el índice, procedemos a iterar los archivos incluidos en el directorio “docs” para poder obtener los documentos Lucene que se insertarán en el índice Lucene. Este es el resultado de la ejecución:

```
$java -cp .:./commons-beanutils-1.9.4.jar:./commons-collections4-4.4.jar:./commons-lang3-3.12.0.jar:./commons-text-1.9.j
ar:./opencsv-5.5.2.jar:./lucene-8.10.1/core/lucene-core-8.10.1.jar:./lucene-8.10.1/analysis/common/lucene-analyzers-comm
on-8.10.1.jar indiceSimple.java
Archivo cloud-computing.csv
Archivo data-science.csv
Archivo digital-library.csv
Archivo geographic-information-system.csv
Archivo marketing.csv
Archivo natural-language-processing.csv
Archivo software.csv
Archivo spain.csv
Archivo videogames.csv
Archivo wireless-connection.csv
```

2. Índice y consultas con Luke:

Tras compilar y ejecutar el código anterior procedemos a abrir Luke. Este programa nos va a mostrar el índice resultante, así como nos va a dar una interfaz para realizar consultas y probar analizadores. Procedemos a abrir el índice creado:

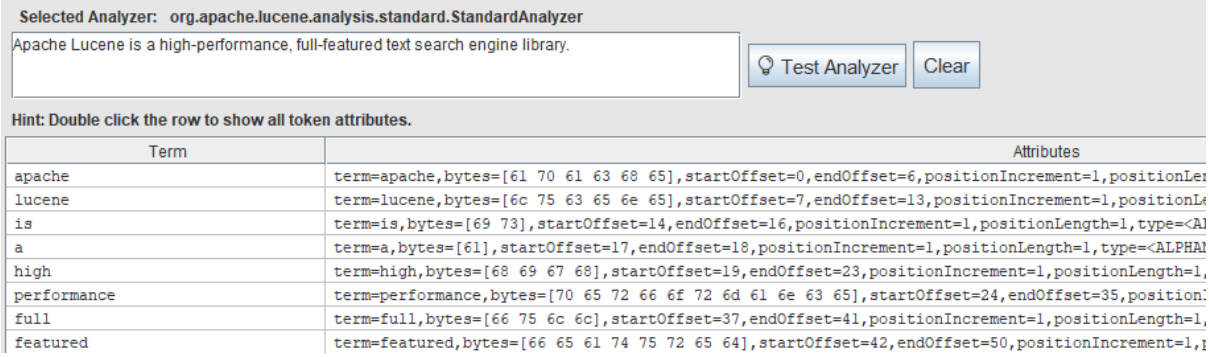


Aquí vamos a poder ver nuestro índice, el cual nos ha quedado de esta manera:

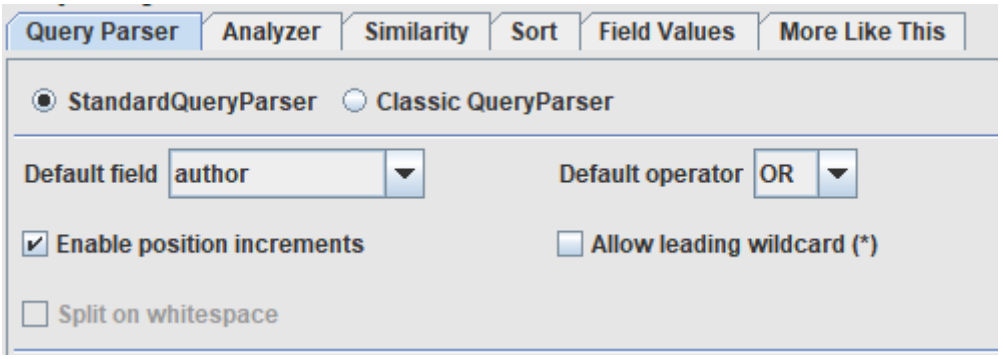
Name	Term count	%
abstract	65137	30.70 %
author	44024	20.75 %
affili...	32326	15.24 %
title	25128	11.84 %
author...	20956	9.88 %
index_...	20563	9.69 %
source...	3968	1.87 %
$facets	30	0.01 %
category	18	0.01 %
articl...	0	0.00 %
eid	0	0.00 %
issue	0	0.00 %
year	0	0.00 %
file_dir	0	0.00 %
public...	0	0.00 %
link	0	0.00 %
doc_type	0	0.00 %
volume	0	0.00 %
page_end	0	0.00 %
page_s...	0	0.00 %
cited_by	0	0.00 %
author_id	0	0.00 %
page_c...	0	0.00 %
doi	0	0.00 %

Se puede observar que hay algunos campos que no tienen conteo de términos. Estos son los StoredField que hemos especificado anteriormente, además de los valores Sorted especificados (year y cited_by). Ahora voy a mostrar algunos detalles, y finalmente haré unas consultas.

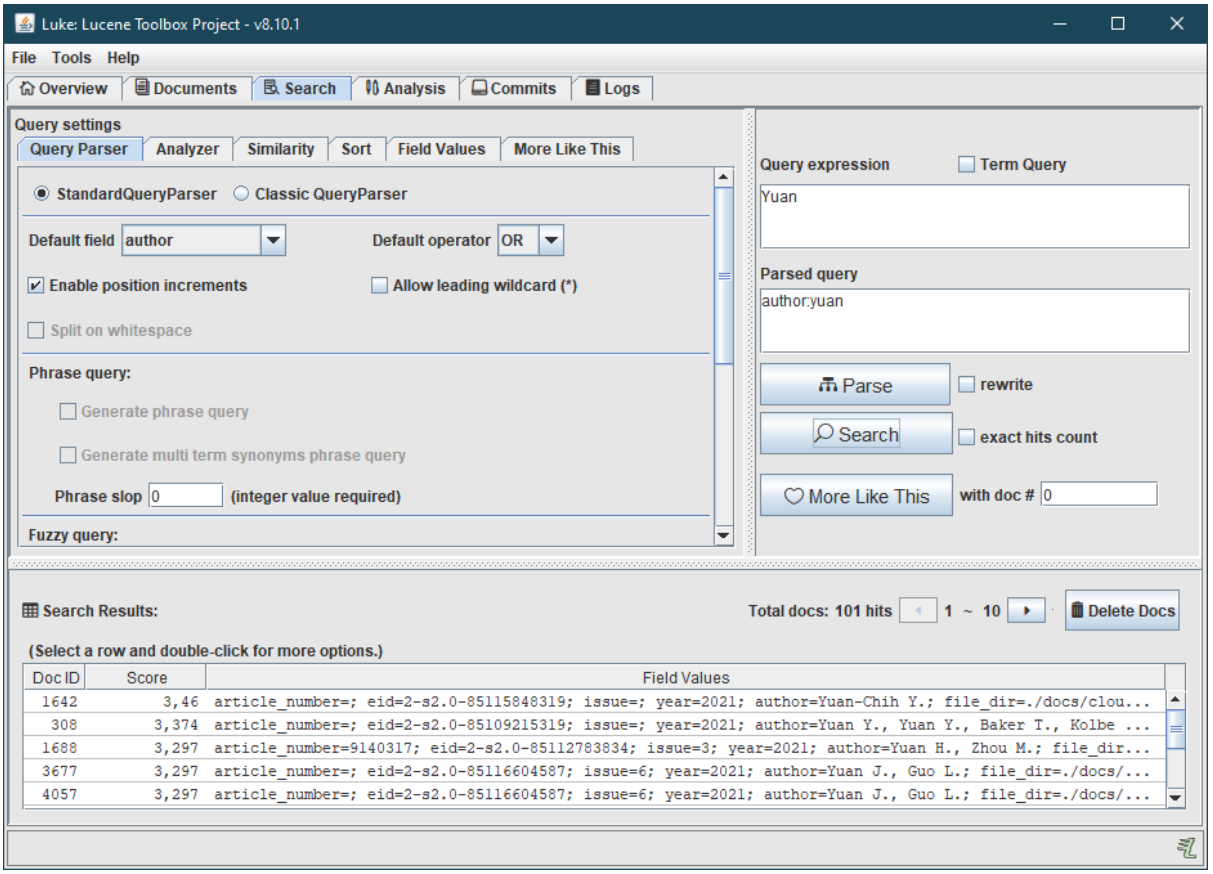
En la pestaña Analysis puedes probar los que necesites, además de crear analizadores personalizados. Con StandardAnalyzer() nos sale de esta manera:



Ahora pasaremos a realizar algunas búsquedas por campos, y ordenando. La primera búsqueda será con el campo author, la cual le pasaremos al autor Yuan.



La consulta queda de esta manera:

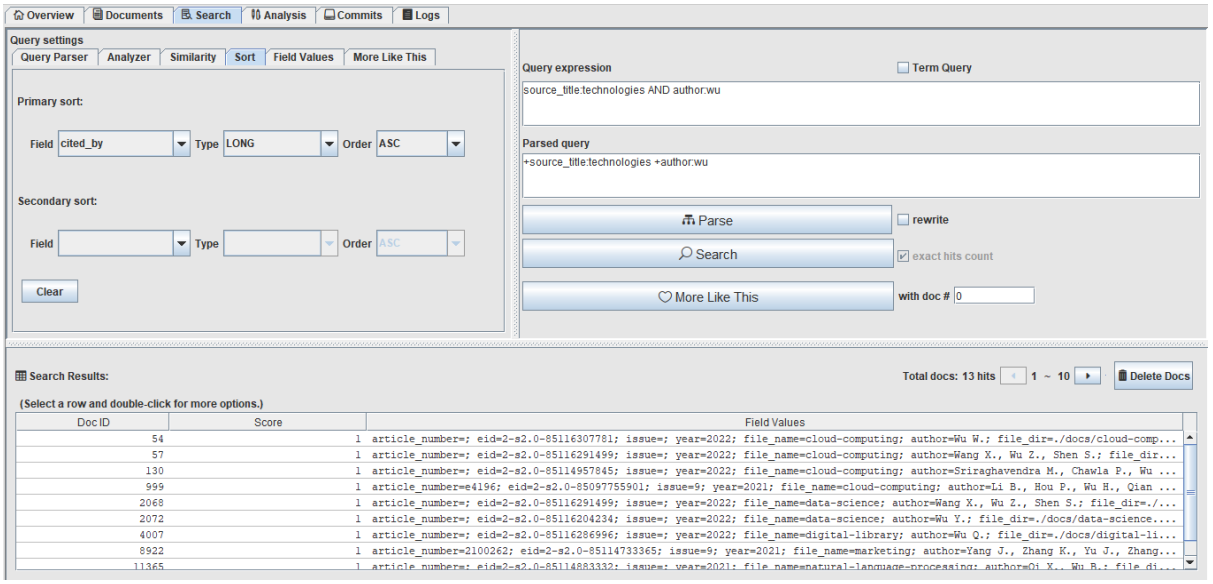


Hemos buscado por el autor yuan, y vemos cómo en la salida los documentos se ordenan por año. Si le damos a Show all fields nos muestra la salida. Los campos que figuran son los que hemos especificado como stored previamente en el código (aparece una S en la columna flags

(To copy all or arbitrary field value(s), unselect all rows or select row(s), and click 'Copy values' button.)

Field	Flags Help	Norm	Value
$facets	Id----------Dbinary----	0	
category	Idfp-N-S----------------	2	cloud-computing
file_dir	-------S----------------	0	./docs/cloud-computing.csv
author	Idfp-N-S----------------	3	Yuan-Chih Y.
author_id	-------S----------------	0	57275414400;
title	Idfp-N-S----------------	13	A dew computing architecture for smart parking system with
year	-------S-#i32--------T4/1	0	2021
source_title	Idfp-N-S----------------	10	Proceedings - 2021 IEEE 45th Annual Computers, Software, an
volume	-------S----------------	0	
issue	-------S----------------	0	
article_number	-------S----------------	0	

Ahora vamos a buscar de otra manera: vamos a la pestaña de Overview, clickeamos en cualquier campo (en este caso source_title) y clickeamos dos veces en alguno de los términos, dándole a Search docs by term. Lo que hace es generarnos una consulta con el término elegido. Nosotros lo hicimos con el término technologies y la salida fue correcta. Escogimos también ordenarlo de forma ascendente por número de citas y le añadimos la condición de que el autor sea Wu. La salida fue la siguiente:



Como conclusión vemos que Luke sirve para inspeccionar los índices ya creados y para realizar consultas y comprobar que todo lo hemos hecho correctamente. También es una herramienta muy útil para entender cómo funcionan los distintos analizadores que nos trae Lucene, y para probar algunos que nosotros creemos.

No obstante, esto es solamente el principio de la práctica y más adelante necesitaremos de nuestra propia interfaz para realizar las búsquedas.

3. Creación del buscador

Una vez tenemos la información indexada adecuadamente, el siguiente paso es permitir al usuario buscar a través de los documentos que tenemos almacenados en nuestro sistema de recuperación de información.

Se puede realizar búsquedas por un solo campo o por varios, y también se pueden excluir valores de cada campo. Una vez devolvemos los resultados de la búsqueda, se puede navegar a través de las facetas, las cuales serán **la categoría del documento, el tipo de documento y el año de publicación del documento**.

Una vez establecidas las premisas, vamos a empezar viendo el código que hemos creado a partir de los ejemplos proporcionados por el profesor en los guiones de esta práctica.

Empezaremos viendo el principio de la función del buscador, *indexSearch*:

```
public List<String> indexSearch(Analyzer analyzer, Similarity similarity, String[] campos, String[] facets) throws ParseException{
    IndexReader reader = null;
    List<String> output = new ArrayList<>();
    String resultados = "";

    try{
        reader = DirectoryReader.open(FSDirectory.open(Paths.get(indexPath)));
        IndexSearcher searcher = new IndexSearcher(reader);
        TaxonomyReader taxoReader = new DirectoryTaxonomyReader(FSDirectory.open(Paths.get(taxoPath)));

        searcher.setSimilarity(similarity);

        BufferedReader in = null;
        in = new BufferedReader(new InputStreamReader(System.in, StandardCharsets.UTF_8));

        Query query;
        String line = "";

        //Analyzer abstract_analyzer = new EnglishAnalyzer(ENGLISH_STOP_WORDS);

            int longitud = 0;
            String campo_actual = "";
            int id = -1;

            for(int i = 0; i < campos.length; i++){
                if(!campos[i].isEmpty()){
                    longitud++;
                    campo_actual = busquedaCampos[i];
                    id = i;
                }
            }
```

Como parámetros le pasamos el analizador por defecto, el tipo de similaridad a usar, todos los campos, estén rellenados o no, de la interfaz de usuario, y las facetas elegidas por el usuario (el array está vacío la primera vez que se llama a la función, esto sirve para llamadas futuras a la función después de haber realizado la búsqueda que permite realizar la navegación por las facetas)

Respecto a cargar y leer el índice es exactamente igual a los ejemplos de los guiones: abrimos la carpeta del índice, creamos un objeto *IndexSearcher* y un objeto *TaxonomyReader*, establecemos el tipo de similaridad y creamos la entrada de datos con un objeto *BufferedReader*.

A partir de aquí ya es diferente a los códigos de ejemplo. Primero de todo creamos una variable *longitud* que usaremos para determinar el número de campos que se han rellenado. También creamos una variable *campo_actual* y un identificador inicializado a -1. Estas variables las usamos en el caso de que se quiera realizar una búsqueda por un solo

campo, pues creamos en la clase un ArrayList que almacena el contenido de todos los campos para poder pasarlo por parámetro a las funciones correspondientes. Usando estas variables podemos afinar la ejecución de la función de la búsqueda por un solo campo para que no necesite iterar todos los campos vacíos.

Una vez hemos procesado los campos rellenados, llamamos a las funciones adecuadas según la cantidad de campos rellenados.

```
if(longitud > 1)
    query = multifieldSearch(campos);
else if(longitud == 1)
    query = singlefieldSearch(campo_actual, campos[id], id);
else
    query = new MatchAllDocsQuery();
```

Veamos cómo es la función *multifieldSearch*.

```
public Query multifieldSearch(String[] campos) throws ParseException{
    BooleanQuery.Builder bqbuilder = new BooleanQuery.Builder();
    BooleanClause bc;
    Query qaux;
    Query query = new MatchAllDocsQuery();

    for(int i = 0; i < campos.length; i++){
        if(i%2==0){
            if(!campos[i].isEmpty()){
                if(i == 8){
                    QueryParser parser = new QueryParser(busquedaCampos[i], new EnglishAnalyzer());
                    qaux = parser.parse(campos[i]);
                    bc = new BooleanClause(qaux, BooleanClause.Occur.MUST);
                    System.out.println(qaux.toString());
                    bqbuilder.add(bc);
                }
                else{
                    QueryParser parser = new QueryParser(busquedaCampos[i], new SimpleAnalyzer());
                    qaux = parser.parse(campos[i]);
                    bc = new BooleanClause(qaux, BooleanClause.Occur.MUST);
                    System.out.println(qaux.toString());
                    bqbuilder.add(bc);
                }

                query = bqbuilder.build();
            }
        } else{
            if(!campos[i].isEmpty()){
                if(i == 9){
                    QueryParser parser = new QueryParser(busquedaCampos[i], new EnglishAnalyzer());
                    qaux = parser.parse(campos[i]);
                    bc = new BooleanClause(qaux, BooleanClause.Occur.MUST_NOT);
                    System.out.println(qaux.toString());
                    bqbuilder.add(bc);
                }
                else{
                    QueryParser parser = new QueryParser(busquedaCampos[i], new SimpleAnalyzer());
                    qaux = parser.parse(campos[i]);
                    bc = new BooleanClause(qaux, BooleanClause.Occur.MUST_NOT);
                    System.out.println(qaux.toString());
                    bqbuilder.add(bc);
                }

                query = bqbuilder.build();
            }
        }
    }

    return query;
}
```

Al ser una búsqueda por varios campos, se tiene que generar una BooleanQuery que incluya una BooleanClause por cada campo. La función recorre el array de los campos, genera un objeto *QueryParser* que procesa el contenido del campo a consultar, genera una BooleanClause por cada uno de ellos y la añade al BooleanQuery. El funcionamiento del parser tiene en cuenta entradas múltiples y genera el tipo de Query correspondiente(por ejemplo: *cloud, software* generaría una BooleanQuery; *cloud computing* generaría una PhraseQuery).

Hay dos casos en particular a destacar: al incluir o excluir en el campo *abstract* (índices 8 y 9).. Esto se debe a que usan un analizador diferente al resto de campos (EnglishAnalyzer) y esto se debe tener en cuenta a la hora de pasarlo al parser de la consulta.

Veamos ahora cómo es la función *singlefieldSearch*.

```
public Query singlefieldSearch(String campo_actual, String busqueda, Integer id) throws ParseException{
    Query query;

    if(campo_actual.equals("abstract")){
        QueryParser parser = new QueryParser(campo_actual, new EnglishAnalyzer());
        query = parser.parse(busqueda);
        System.out.println(query.toString());
    }
    else{
        QueryParser parser = new QueryParser(campo_actual, new SimpleAnalyzer());
        query = parser.parse(busqueda);
        System.out.println(query.toString());
    }

    return query;
}
```

Es bastante más simple que la versión con múltiples campos. Simplemente le pasamos el campo rellenado al parser y genera el objeto Query correspondiente. Igual que en la búsqueda por múltiples campos, hay que tener en cuenta el analizador que utiliza el campo *abstract*.

Una vez tenemos la búsqueda generada podemos generar las facetas. Este es el código:

```
FacetsConfig fconfig = new FacetsConfig();
DrillDownQuery ddq = new DrillDownQuery(fconfig,query);

for(String aux : facets){
    System.out.println(aux);
    String faux[] = aux.split("=");

    ddq.add(faux[0],faux[1]);
}

FacetsCollector fc = new FacetsCollector(true);
TopDocs results = FacetsCollector.search(searcher, ddq, 100, fc);
long numTotalHits = results.totalHits.value;
ScoreDoc[] hits = results.scoreDocs;
Facets facetas = new FastTaxonomyFacetCounts(taxoReader, fconfig, fc);

List<FacetResult> lista = facetas.getAllDims(100);
```

Creamos un objeto FacetsConfig que le pasaremos al objeto DrillDownQuery que contendrá la búsqueda generada. Tras esto, recorremos el array de las facetas elegidas por el usuario y añadimos las diferentes dimensiones al objeto DrillDownQuery.

Una vez generadas las facetas ya podemos realizar la búsqueda. Se hace como en los ejemplos de los guiones: creamos un objeto FacetsCollector que usamos para obtener los resultados de la búsqueda y almacenarlos en un objeto TopDocs, obtenemos el número de hits, obtenemos los scores y obtenemos la lista de las facetas de los resultados obtenidos.

El código siguiente es para generar la salida que utilizaremos para la interfaz:

```
resultados += "Número de resultados: " + hits.length + "\n";
for(int i=0 ; i<hits.length ; i++){
    Document doc = searcher.doc(hits[i].doc);
    resultados += "Documento "+(i+1);
    resultados += "\n\tAuthor: " + doc.get("author");
    if(!doc.get("title").equals(""))
        resultados += "\n\tTitle: " + doc.get("title");
    if(!doc.get("category").equals(""))
        resultados += "\n\tCategory: " + doc.get("category");
    if(!doc.get("year").equals(""))
        resultados += "\n\tyear: " + doc.get("year");
    if(!doc.get("volume").equals(""))
        resultados += "\n\tVolume: " + doc.get("volume");
    if(!doc.get("issue").equals(""))
        resultados += "\n\tIssue: " + doc.get("issue");
    if(!doc.get("doc_type").equals(""))
        resultados += "\n\tDocument type: " + doc.get("doc_type");
    if(!doc.get("article_number").equals(""))
        resultados += "\n\tArticle number: " + doc.get("article_number");
    if(!doc.get("page_start").equals(""))
        resultados += "\n\tPage start: " + doc.get("page_start");
    if(!doc.get("page_end").equals(""))
        resultados += "\n\tPage end: " + doc.get("page_end");
    if(!doc.get("page_count").equals(""))
        resultados += "\n\tPage count: " + doc.get("page_count");
    if(!doc.get("doi").equals(""))
        resultados += "\n\tDOI: " + doc.get("doi");
    if(!doc.get("link").equals(""))
        resultados += "\n\tLink: " + doc.get("link");
    if(!doc.get("affiliations").equals(""))
        resultados += "\n\tAffiliations: " + doc.get("affiliations");
    if(!doc.get("abstract").equals(""))
        resultados += "\n\tAbstract: " + doc.get("abstract");
    if(!doc.get("public_status").equals(""))
        resultados += "\n\tPublic status: " + doc.get("public_status");
    if(!doc.get("eid").equals(""))
        resultados += "\n\tEID: " + doc.get("eid") + "\n\n";
}
output.add(resultados);
```

Simplemente añadimos todos los datos que almacenamos de cada documento a un String que formateamos y mostraremos al usuario a través de la interfaz (la posición 0). También añadimos las facetas al output:

```
for(FacetResult faux : lista){
    for(int i=0 ; i < faux.labelValues.length ; i++){
        output.add(faux.labelValues[i].label);
        output.add(faux.labelValues[i].value.toString());
        System.out.println(faux.labelValues[i].label+"|"+faux.labelValues[i].value.toString());
    }
}
```

De esta forma podemos procesar el String en la interfaz y generar las facetas de forma dinámica conforme el usuario va navegando a través de las facetas (se explicará con más precisión en el apartado de la interfaz).

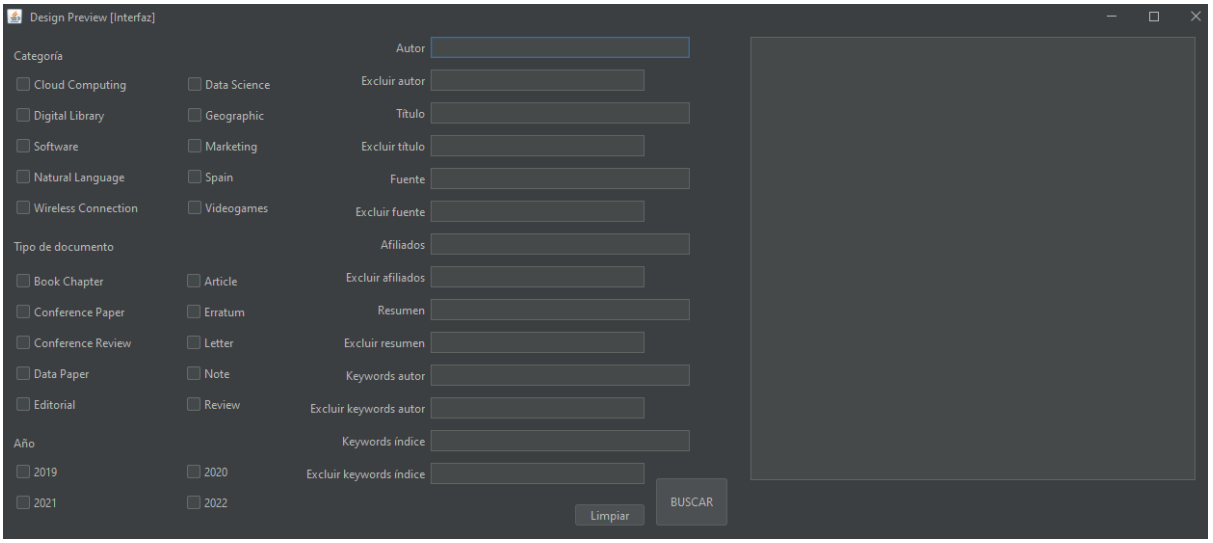
La función devuelve el String que hemos creado con los resultados y las facetas, y este String a su vez se devuelve en el main de la clase del buscador.

```
public static List<String> main(String[] args, String[] args2) throws ParseException{
    Analyzer analyzer = new SimpleAnalyzer();
    Similarity similarity = new ClassicSimilarity();
    // Similarity similarity = new LMDirichletSimilarity();
    // Similarity similarity = new BM25Similarity();

    scopusFinder busqueda = new scopusFinder();
    List<String> output = busqueda.indexSearch(analyzer, similarity, args, args2);

    return output;
}
```

Esto se hace para facilitar que la interfaz acceda a los resultados de la búsqueda simplemente llamando al main de la clase.

4. Creación de la interfaz

En esta sección veremos cómo hemos desarrollado la interfaz de usuario. La interfaz final es así:



Tenemos varios campos a incluir y a excluir, un botón de buscar abajo, a la izquierda las facetas y a la derecha una consola que nos mostrará el top 100 documentos que coincidan con nuestra búsqueda. Ahora voy a mostrar un poco el código y explicarlo.

```
public Interfaz() {
    initComponents();
    cloudcomputing.setEnabled(false);
    digitallibrary.setEnabled(false);
    software.setEnabled(false);
    naturallang.setEnabled(false);
    wireless.setEnabled(false);
    datascience.setEnabled(false);
    geographic.setEnabled(false);
    marketing.setEnabled(false);
    spain.setEnabled(false);
```

En el constructor de la interfaz inicializamos todos los hijos de las facetas en false para que aparezcan en gris y no puedan ser clickeadas, ya que aún no se ha hecho ninguna consulta. Son un total de 24 variables que realizan la función de los hijos de las facetas. No se ha visto en la captura pero antes del constructor declaramos un ArrayList global para las facetas llamado facetas y un ArrayList global llamado resultado.

Luego, las casillas de texto no tienen código alguno. Estas casillas sirven para enviar a la búsqueda los términos de cada dato escogido que queremos incluir o excluir, y esto se procesa en el código del botón de buscar, así que a continuación explicaré lo que ocurre cuando se pulsa este botón.

El código es el siguiente. Lo vamos a dividir en varias partes:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    output.setLineWrap(true);
    output.setWrapStyleWord(true);
    resultado.clear();
    String campos[] = new String[]  {   autor.getText(),
                                        eautor.getText(),
                                        titulo.getText(),
                                        etitulo.getText(),
                                        fuente.getText(),
                                        efuente.getText(),
                                        afiliados.getText(),
                                        eafiliados.getText(),
                                        resumen.getText(),
                                        eresumen.getText(),
                                        keyautor.getText(),
                                        ekeyautor.getText(),
                                        keyindice.getText(),
                                        ekeyindice.getText()
                                    };

    try{
        String[] aux = facetas.toArray(String[]::new);
        resultado = scopusFinder.main(campos, aux);
    } catch (ParseException pe){
        System.err.print("Campos no válido");
    }
```

Aquí vemos que llamamos a algunas funciones para output (la consola de salida de los documentos). Esas dos funciones hacen que cuando una línea se alargue más de los límites horizontales del output esta se pase a la siguiente línea de abajo.

Limpiamos la lista resultado, y declaramos un array de string campos, que contiene todos y cada uno de los campos que le hayamos especificado para incluir y excluir (los campos a rellenar).

Luego probamos a convertir el ArrayList facetas a un Array de string para poder pasarlo como argumento al main del archivo source scopusfinder.java, y a resultado le pasamos el resultado de llamar al main habiendo pasado los campos y las facetas. Internamente el main trata los dos arrays de manera que se genere la consulta y la salida, aunque eso ya lo hemos explicado anteriormente. Si por algún casual hay error en esas dos líneas de código se nos lanzará una ParseException.

```
autor.setEnabled(false);
eautor.setEnabled(false);
titulo.setEnabled(false);
etitulo.setEnabled(false);
fuente.setEnabled(false);
efuente.setEnabled(false);
afiliados.setEnabled(false);
eafiliados.setEnabled(false);
resumen.setEnabled(false);
eresumen.setEnabled(false);
keyautor.setEnabled(false);
ekeyautor.setEnabled(false);
keyindice.setEnabled(false);
ekeyindice.setEnabled(false);
```

Desmarcamos las casillas de texto mientras realizamos el filtro.

```
cloudcomputing.setText("Cloud Computing");
digitallibrary.setText("Digital Library");
software.setText("Software");
naturallang.setText("Natural Language");
datascience.setText("Data Science");
geographic.setText("Geographic");
marketing.setText("Marketing");
spain.setText("Spain");
videogames.setText("Videogames");
wireless.setText("Wireless Connection");
```

```
cloudcomputing.setEnabled(false);
digitallibrary.setEnabled(false);
software.setEnabled(false);
naturallang.setEnabled(false);
wireless.setEnabled(false);
datascience.setEnabled(false);
geographic.setEnabled(false);
marketing.setEnabled(false);
spain.setEnabled(false);
```

Ahora vamos a ver cómo tratamos las facetas. Primero inicializamos todas a su texto original y las ponemos en gris a todos los 24 hijos para que no puedan seleccionarse. Esto sirve para evitar que facetas que no se encuentran en nuestra búsqueda puedan seleccionarse, ya que si para x hijo no hay ningún documento que coincida no tiene sentido poder seleccionarlo. Ahora hecho esto podemos comprobar si hay hijos que estén:

```
for(int i=1 ; i<resultado.size() ; i+=2){
    if(resultado.get(i).equals("cloud-computing")){
        cloudcomputing.setText("Cloud Computing ("+resultado.get(i+1)+")");
        cloudcomputing.setEnabled(true);
    }

    if(resultado.get(i).equals("digital-library")){
        digitallibrary.setText("Digital Library ("+resultado.get(i+1)+")");
        digitallibrary.setEnabled(true);
    }

    if(resultado.get(i).equals("software")){
        software.setText("Software ("+resultado.get(i+1)+")");
        software.setEnabled(true);
```

En este bucle recorremos de una en una cada posición del array (excepto el 0, ya que en el 0 se aloja el output de documentos, y no las facetas que hemos encontrado). El nombre del hijo está en la posición i y el número de documentos en i+1, siendo i = 1 y sabiendo que itera sumando 2 posiciones.

Hay un if para cada hijo, en el que comprobamos si el nombre coincide, y si coincide al objeto correspondiente de la interfaz se le cambia el texto a nombre + número de documentos, y se activa para poder seleccionarlo.

```
    output.setText(resultado.get(0));
    output.setCaretPosition(0);
}
```

Finalmente ponemos como texto del output resultado[0], ya que es donde está la información del top 100 documentos, y la segunda función mueve el cursor hasta la primera posición del string.

Para rellenar el ArrayList facetas, que contiene las selecciones que hagamos para pasarlas al buscador es el mismo código para todos los hijos:

```
private void editorialActionPerformed(java.awt.event.ActionEvent evt) {
    if(editorial.isSelected()) facetas.add("doc_type=Editorial");
    else facetas.remove("doc_type=Editorial");

    this.jButton1ActionPerformed(evt);
}

private void year4ActionPerformed(java.awt.event.ActionEvent evt) {
    if(year4.isSelected()) facetas.add("year=2022");
    else facetas.remove("year=2022");

    this.jButton1ActionPerformed(evt);
}

private void wirelessActionPerformed(java.awt.event.ActionEvent evt) {
    if(wireless.isSelected()) facetas.add("category=wireless-connection");
```

Comprobamos si hemos seleccionado la casilla, y si la hemos seleccionado añadimos hijo=nombre del hijo. Si no lo hemos seleccionado borramos la faceta hijo+nombre del hijo. Esto último evita que tengamos restos de búsquedas anteriores, y eso estaría mal.

Ahora el botón limpiar, que limpia todas las búsquedas y prepara el programa para la siguiente:

```
private void limpiezaActionPerformed(java.awt.event.ActionEvent evt) {
    autor.setEnabled(true);
    eautor.setEnabled(true);
    titulo.setEnabled(true);
    etitulo.setEnabled(true);
    fuente.setEnabled(true);
    efuente.setEnabled(true);
    afiliados.setEnabled(true);
    eafiliados.setEnabled(true);
    resumen.setEnabled(true);
    eresumen.setEnabled(true);
```

Primero marcamos todas las casillas de texto para poder escribir.

```
facetas.clear();
resultado.clear();
output.setText("");

cloudcomputing.setText("Cloud Computing");
digitallibrary.setText("Digital Library");
software.setText("Software");
naturallang.setText("Natural Language");
datascience.setText("Data Science");
```

```
cloudcomputing.setEnabled(false);
digitallibrary.setEnabled(false);
software.setEnabled(false);
naturallang.setEnabled(false);
wireless.setEnabled(false);
datascience.setEnabled(false);
geographic.setEnabled(false);
```

Ahora limpiamos la salida, los array de resultado y facetas. Luego ponemos los textos de los hijos como al principio y desmarcamos las casillas de los hijos.

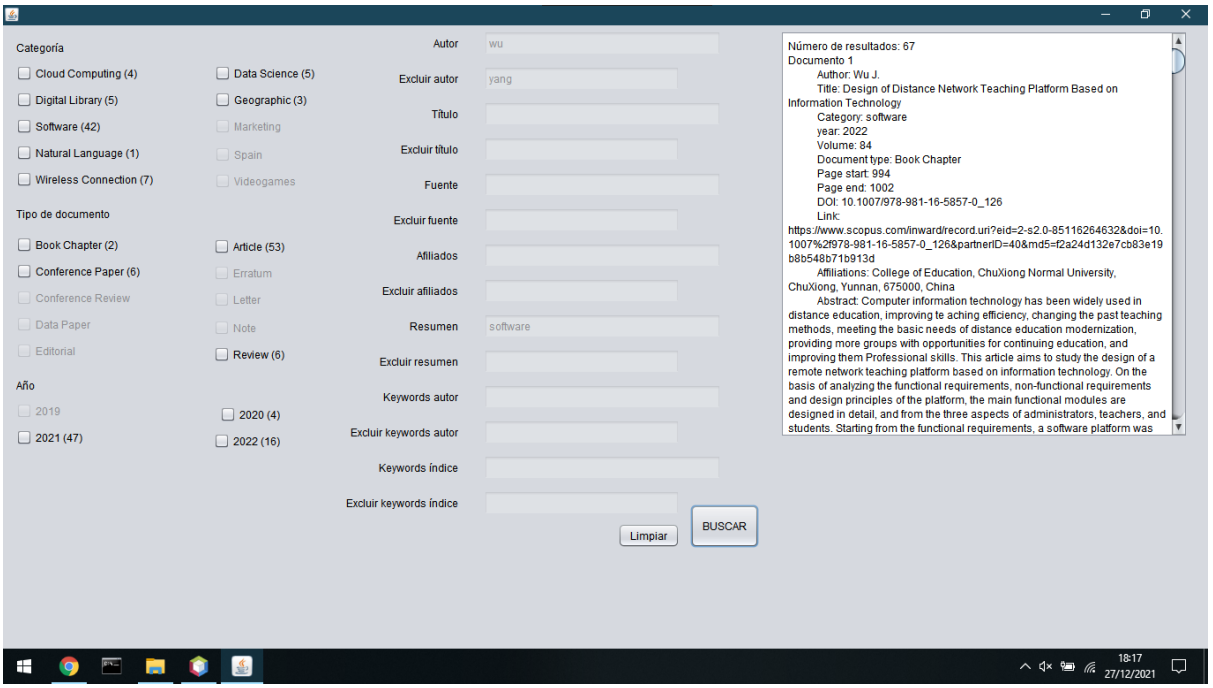
```
autor.setText("");
eautor.setText("");
titulo.setText("");
etitulo.setText("");
fuente.setText("");
efuente.setText("");
```

```
cloudcomputing.setSelected(false);
digitallibrary.setSelected(false);
software.setSelected(false);
naturallang.setSelected(false);
wireless.setSelected(false);
datascience.setSelected(false);
geographic.setSelected(false);
```

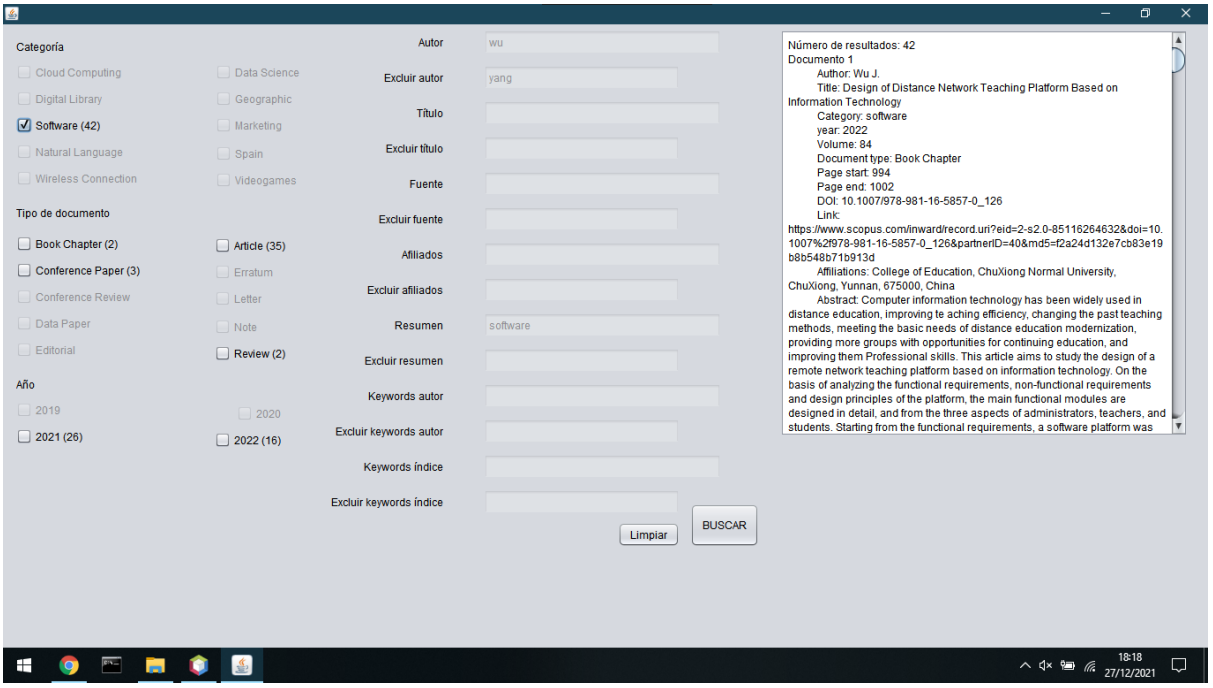
Finalmente quitamos el texto de las casillas de texto y deseleccionamos todas las checkboxes.

5. Ejemplo de la interfaz

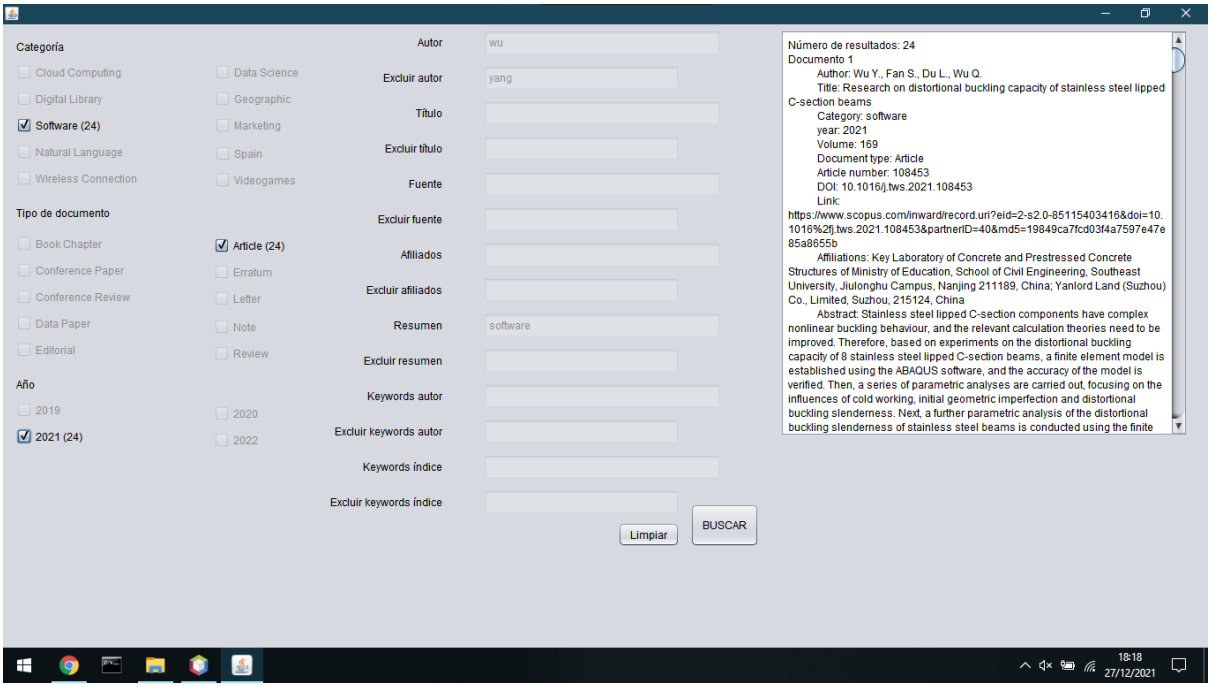
Vamos a mostrar un ejemplo de la interfaz en uso:



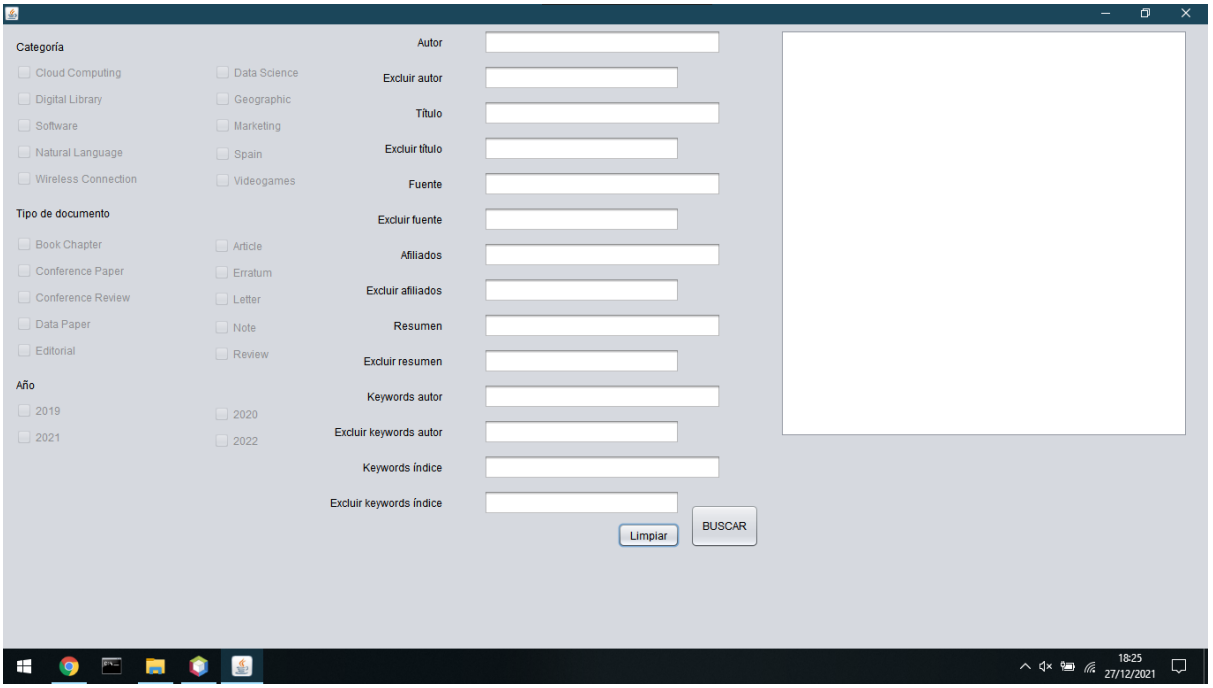
Aquí he realizado la búsqueda del autor que contenga wu, que excluya a yang y que su resumen tenga software. Ahora voy a usar los hijos de las facetas para filtrarlo. Se observa que correctamente aparecen el número de documentos que han coincidido.



En esta captura he filtrado por la categoría software. Vemos que ha cambiado el número de coincidencias con tipo de documento y año, así que esto funciona.



He usado todas las facetas y vemos que el número de resultados ha disminuido y que las coincidencias para los filtros se han igualado completamente.



Finalmente le he dado a limpiar y he vuelto a los parámetros iniciales. Estoy preparado para una nueva búsqueda.