ugr | Universidad de Granada

BÚSQUEDA

Autores:
Raúl Rodríguez Rodríguez
Rafael García Andújar
Mario Casas Pérez

Curso: 4º
Nombre del grupo: Viernes-Engineers-RI

ÍNDICE

1. Introducción

Para la práctica final, hemos recopilado aquellas características más relevantes de las prácticas anteriores que hemos realizado (Tika, Analizadores, Datos, Creación de un índice, Búsqueda) de cara ha realizar esta práctica, además de tener en consideración otros aspectos relevantes que se tienen que tener en cuenta.

Para ello, hemos creado dos ficheros java los cuales uno lo usamos para indexar los datos, mientras que el otro lo usamos para mostrar dichos datos en una interfaz. En esta, el usuario podrá realizar búsquedas y aplicar filtros para mejorar dicha búsqueda y adaptarla a sus necesidades.

En este trabajo se presenta el desarrollo de un sistema de indexación y búsqueda avanzada usando Apache Lucene, acompañado de una interfaz gráfica diseñada para facilitar la interacción con los datos. Este sistema integra funcionalidades importantes para procesar, organizar y buscar información de forma eficiente haciendo uso de índices paralelos y soportes para facetas.

El componente de indexación nos permite procesar grandes cantidades de datos almacenados en formatos comprimidos (.gz), aplicando técnicas avanzadas de análisis de texto, detección automática de idiomas y clasificación jerárquica con facetas. Por otra parte, la interfaz gráfica ofrece al usuario una experiencia bastante intuitiva mediante la integración de filtros dinámicos, búsquedas personalizadas y paginación para explorar los resultados.

Este sistema está diseñado para ser flexible, escalable y modular, adaptándose a varios escenarios como el comercio electrónico, sistemas de recomendación y análisis de grandes cantidades de datos. La combinación de indexación robusta y una interfaz de usuario eficiente proporciona una solución bastante completa para gestionar y buscar información de manera óptima.

2. Indexación con las Facetas

Hemos creado un fichero llamado Indexar.java, el cual presenta una clase Indexar en Java para indexar documentos en función de su tipo (*reviews*, *metadata*, *reviewsEspecial*, *metadataEspecial*) y proporcionar capacidades avanzadas como lo son las facetas y análisis de texto. La creación, configuración e indexación de documentos usando la biblioteca Apache Lucene se encuentra en dicho fichero. Además, también hemos puesto algunas características adicionales tales como el análisis de idioma, categorización y el manejo de las facetas.

A continuación pasamos a indicar cuales han sido los principales componentes que hemos usado para hacer este código:

- **Dependencias usadas**
    - Apache Lucene: para la indexación, búsqueda y análisis de textos.
    - Tika: para la detección de idioma del texto que se está analizando.
    - Jackson: para el manejo y el procesamiento de archivos JSON (los dados de metadata y reviews los cuales están comprimidos en un .gz).
- **Propiedades clave**
    - *analyzer*: analizador para procesar los textos de entrada.
    - *writer*: escribe documentos en el índice.
    - *taxonomyWriter* y *configFacetas*: gestionan las facetas, categorización y jerarquías de datos.
- **Funcionalidades principales**
    - Configuración del índice: ajusta el modo de apertura, que es *create* (create) o *append* (añadir).
    - Indexación de documentos: procesa archivos JSON comprimidos (.gz), detecta el idioma del texto y lo analiza con el analizador correspondiente.
    - Soporte de facetas: permite agregar metadatos jerárquicos y multivalorados (nosotros lo hemos hecho con rango de precios, rango de calificación promedio y las categorías).
    - Finalización del índice: compacta los segmentos del índice para optimizar el rendimiento.
- **Gestión de idiomas**
    - Nuestro código selecciona un analizador adecuado (español, inglés, francés o estándar) basándose en la detección automática del idioma usando Apache Tika.
- **Jerarquización y facetas**
    - Los campos como *categorias*, *rangoPrecio* y *rangoAverageRating* están configuradas para soportar facetas. Esto nos ayuda a realizar búsquedas jerárquicas o con filtros.

Ahora, pasamos a detallar con detenimiento el código que hemos realizado:

1. **Inicialización**

La clase importa varias bibliotecas importantes, como Lucene para manejo de índices y análisis, Tika para detección de idiomas y Jackson para procesar JSON.

Esta clase tiene varios apartados a considerar, como lo son:

- **tipoDato:** define el tipo de datos a indexar.

- **directorio:** es la carpeta donde se encuentran los datos.

- **modo:** se indica si queremos crear el índice (create) o si queremos indexar datos a un índice en concreto (append).

- **analizador:** se especifica cómo analizar los textos.

- **writer:** controla el proceso de escritura en el índice.

- **taxonomyWriter:** permite manejar facetas jerárquicas.

- **configFacetas:** configura opciones de facetas como multivaloradas o jerárquicas.

2. **Constructor**

El constructor de indexar configura:

- **El analizador:** por defecto usamos un StandardAnalyzer.

- **Configuración del índice:** creamos un índice en un directorio especificado (INDEX_PATH). Además, configuramos la similitud como BM25Similarity y establecemos el modo de apertura del índice (create o append).

- **Facetas:** configura facetas jerárquicas y multivaloradas, como *categorias*, *rangoPrecio* y *rangoAverageRating.*

3. **Método main**

El método principal lo hemos realizado mediante un menú para que sea más cómodo al usuario final a la hora de interactuar con este programa. Para ello, hemos creado un índice en el cual muestra al usuario qué datos desea indexar (*reviews*, *metadata*, *metadataEspecial*, *reviewsEspecial*). Una vez se indique, aparecerá otro índice donde habrá dos opciones, crear el índice con los datos especificados (*create*) o insertar datos a un índice existente (*append*). Además, en el primer menú que le aparece, podrá indicar el usuario cuando quiere terminar de indexar datos (Cerrar programa), es decir, el usuario podrá seguir indexando los datos necesarios hasta que lo desee.

- **configurarIndice():** mensaje informativo.

```
public void configurarIndice() {
    System.out.println("Configuracion del indice para tipo de dato: " + tipoDato  );
}
```

- **indexarDocumentos():** es el proceso principal de indexación.

- **close():** cerramos el índice y guardamos los cambios.

```
public void close() {
    try {
        finalizarIndexado();
        writer.forceMerge(maxNumSegments:1);
        writer.close();
        taxonomyWriter.close();
    } catch (IOException e) {
        System.err.println("Error cerrando el índice o el taxonomyWriter: " + e.getMessage());
    }
}
```

4. **Método indexarDocumentos**

Esta función tiene la siguiente finalidad:

- Recorremos los subdirectorios del directorio que se haya especificado.

- Procesamos los archivos que tengan la extensión .gz (archivos comprimidos). Descomprimimos y leemos línea por línea y convertimos cada línea en un objeto JSON usando *ObjectMapper*.

- Procesamos los JSON y, si es un array JSON, procesamos cada objeto individualmente y si es un objeto JSON, lo pasamos al método *procesarDocumento*.

```
try {
    JsonNode nodo = mapper.readTree(linea);

    if (nodo.isArray()) {
        int cont=0;
        String aux="";
        for (JsonNode jsonObject : nodo) {
            String asin = jsonObject.path(fieldName:"asin").asText(defaultValue:"");
            if(!aux.equals(asin)){
                cont=0;
                aux = asin;
            }
            if(cont==0){
                procesarDocumento(jsonObject);
                cont++;
                finalizarIndexado();
            }

        }
    } else if (nodo.isObject()) {
        procesarDocumento(nodo);

    } else {
        System.err.println("Formato inesperado en línea: " + linea);
    }

} catch (Exception jsonException) {
    System.err.println("Error de formato JSON en línea: " + linea + " - " + jsonException.getMessage());
}
```

- Si hay errores (formato JSON incorrecto, líneas vacías), lo reportamos, pero sigue ejecutándose.

5. **Método procesarDocumento**

Este método varía según el tipo de dato (*reviews*, *metadata*, *reviewsEspecial* o *metadataEspecial* en nuestro caso):

- **Para reviews:**

    - Agregamos campos como *asin*, *reviewerID*, *reviewerName*, *reviewText*, *overall*, *summary*, *order* (el cual es para indexar los documentos de forma ordenada) y *todosReview* (es la unión de los campos *reviewText* y *summary*).

    - Almacenamos los documentos en el índice con el IndexWriter.

    - Detectamos el idioma de algunos campos usando Tika y seleccionamos en función de lo que este nos diga el analizador adecuado (español, inglés, francés).

- **Para metadata:**

    - Manejamos campos como *asin*, *title*, *averageRating*, *price*, *description*, *categories* y *ratingCount*.

    - Asignamos rangos categóricos a valores numéricos (esto lo hacemos pensando en la futura interfaz, donde queremos agrupar distintos rangos de precios y distintos rangos de calificaciones promedio para que sea más cómodo para el usuario filtrar por algunos de estos campos, además de que también queremos mostrar las categorías y las subcategorías de estas).

- **Para reviewsEspecial y metadataEspecial:**

    - Aquí es donde hacemos las operaciones necesarias como lo es concatenar múltiples reseñas asociadas al mismo asin.

6. **Método analizarTexto**

Esta función hace lo siguiente:

- Detectamos el idioma del texto usando Tika.

- Seleccionamos el analizador adecuado según el idioma que se detecte (*SpanishAnalyzer*, *EnglishAnalyzer*, etc).

- Tokenizamos el texto con Lucene y generamos una cadena de tokens procesados.

7. **Métodos auxiliares**

Entre ellos destacamos:

- **finalizarIndexado:** iteramos sobre los mapas reviewMap y summaryMap, teniendo las reseñas acumuladas antes de indexarlas.

```
public void finalizarIndexado() throws IOException {

    //iterar sobre los documentos y agregar al índice
    for (Map.Entry<String, StringBuilder> entry : reviewMap.entrySet()) {
        String asin = entry.getKey();
        StringBuilder review = entry.getValue();
        StringBuilder summary = summaryMap.get(asin);
        Document doc = new Document();
        doc.add(new TextField(name:"order", String.valueOf(doc_metadata_actual), Field.Store.YES));
        doc.add(new TextField(name:"reviewText", review.toString().trim(), Field.Store.YES));
        doc.add(new TextField(name:"summary", summary.toString().trim(), Field.Store.YES));
        doc.add(new StringField(name:"asin", asin, Field.Store.YES));
        doc_metadata_actual++;
        writer.addDocument(doc);
    }
    reviewMap.clear();
}
```

- **close:** mencionado previamente, cierra el IndexWriter y el taxonomyWriter asegurándonos de guardar cambios y optimizar el índice.

8. **Facetas**

Las facetas nos permite hacer consultas categorizadas, como lo son:

- **Rango de calificación promedio (rangoAverageRating):** clasificamos las calificaciones en rangos ([0-1], (1-2], (2-3], (3-4], (4-5]).

- **Rango de precios (rangoPrecio):** dependiendo del precio, clasificamos en rangos (si no tiene precio ponemos *Precio no estipulado*, (0-10], (10-50], (50-150], (150-300], (300-700], (700-1000], +1000).

- **Categorías (categorias):** manejamos facetas jerárquicas como *Libros -> Literatura y Ficción* o *Electrónica -> Accesorios*.

```
taxonomyWriter = new DirectoryTaxonomyWriter(dir2);
```

```
this.configFacetas = new FacetsConfig();
this.configFacetas.setMultiValued(dimName:"categorias", value:true);
this.configFacetas.setHierarchical(dimName:"categorias", v:true);
this.configFacetas.setMultiValued(dimName:"rangoPrecio", value:false);
this.configFacetas.setMultiValued(dimName:"rangoAverageRating", value:false);
```

```
doc.add(new FacetField(dim:"rangoAverageRating", rangoAvg));
```

```
doc.add(new FacetField(dim:"rangoPrecio", rangoPrecio));
```

```
doc.add(new FacetField(dim:"categorias", categoriaNiveles));
writer.addDocument(configFacetas.build(taxonomyWriter, doc));
```

Este código tiene varias ventajas que proporciona a la hora de la indexación de los datos de reviews y metadata, los cuales son:

- **Flexibilidad en el análisis de texto**
    - Usa analizadores concretos para diferentes idiomas detectados automáticamente, lo que mejora la precisión a la hora del procesamiento y el manejo de datos multilingües.
- **Soporte avanzado para facetas y búsqueda facetadas**
    - Configura categorías y rangos de datos (precios y calificaciones), lo cual permite búsquedas avanzadas, filtros dinámicos y agrupaciones de resultados, lo cual viene siendo ideal para aplicaciones como comercio electrónico o sistemas de recomendación.
- **Procesamiento eficiente de los datos**
    - Procesa grandes cantidades de datos en formato comprimido (.gz), con un manejo robusto de errores en JSON y líneas vacías, garantizando continuidad en el flujo de indexación.
- **Optimización del índice y búsquedas**
    - Configuración de *BM25Similarity* para manejar la importancia en las búsquedas.
    - Compactación de segmentos con *forceMerge* para mejorar el rendimiento del índice.
- **Escalabilidad y organización jerárquica**
    - Organiza los datos en subdirectorios y permite manejar múltiples tipos de datos (como reviews y metadata), facilitando el escalado del sistema.
- **Personalización del índice**
    - Configuración detallada de los campos, tanto el almacenamiento, su análisis y las facetas jerárquicas o multivaloradas.
- **Robustez y resiliencia en el procesamiento**
    - Manejo efectivo de errores en datos erróneos. Esto nos asegura la estabilidad del flujo de indexación

- **Adaptabilidad a múltiples tipos de datos**

    - Se permite indexar y manejar diferentes esquemas de documentos en un solo índice o índices separados, según sea necesario.

Ahora, también es importante resaltar aquellas ventajas adicionales que proporciona también nuestro código de indexación:

- **Compatibilidad con sistemas avanzados de búsqueda**

    - Nuestro código es compatible con sistemas de búsqueda avanzada que usan datos jerárquicos, como facetas y filtros, lo cual lo hace adecuado para aplicaciones complejas como marketplaces.

- **Modularidad**

    - La separación clara de métodos ayuda a extender el código para nuevos tipos de datos o funcionalidades.

- **Mantenimiento y claridad del código**

    - Aunque podemos apreciar que nuestro código es largo, este sigue patrones organizados y modulares, lo que ayuda a su mantenimiento y actualización.

3. Interfaz

Hemos creado un fichero Interfaz.java el cual implementa una aplicación gráfica en Java usando Lucene para realizar búsquedas avanzadas con soporte para facetas y paginación en un índice paralelo. Se estructura como una interfaz gráfica (GUI) usando Swing, con funcionalidades adicionales para análisis de texto y filtros dinámicos.

Los funcionamientos son los siguientes:

1. Propósito general

Este programa tiene como propósito general:

- Integrar múltiples índices (*metadata*, *reviewsEspecial*, *metadataEspecial*) en una búsqueda unificada usando un *ParallelCompositeReader*.

- Permite al usuario realizar búsquedas con términos, aplicar filtros por facetas y navegar entre resultados mediantes paginación.

- Detecta el idioma de las consultas y aplica analizadores concretos para mejorar la precisión.

2. **Inicialización y configuración**

Tenemos en el constructor Interfaz():

- **índices paralelos:**
    - Se abren 4 índices (*metadata*, *reviewsEspecial*, *facetas*, *metadataEspecial*) desde rutas específicas.
    - Se unen en un índice lógico único mediante *ParallelCompositeReader*.
    - *searcherParalelo* permite buscar en el índice combinado.

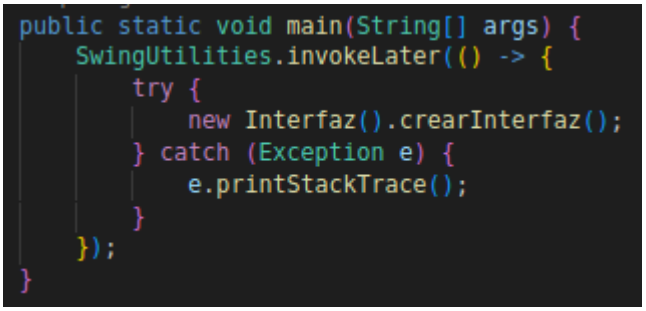
```
FSDirectory metadataDirectory = FSDirectory.open(Paths.get(metadataPath));
FSDirectory reviewsDirectory = FSDirectory.open(Paths.get(reviewsPath));
FSDirectory facetsDirectory = FSDirectory.open(Paths.get(facetsPath));
FSDirectory metadataEspecialDirectory = FSDirectory.open(Paths.get(metadataEspecialPath));

DirectoryReader metadataReader = DirectoryReader.open(metadataDirectory);
DirectoryReader reviewsReader = DirectoryReader.open(reviewsDirectory);
DirectoryReader metadataEspecialReader = DirectoryReader.open(metadataEspecialDirectory);

ParallelCompositeReader paralelo = new ParallelCompositeReader(metadataReader,reviewsReader,metadataEspecialReader);

taxoReader = new DirectoryTaxonomyReader(facetsDirectory);
searcherParalelo = new IndexSearcher(paralelo);
```

- **Facetas:**
    - Se configura *FacetsConfig* para habilitar facetas en campos como *rangoPrecio*, *rangoAverageRating* y *categorias*.
    - Se usan un *DirectoryTaxonomyReader* para manejar facetas jerárquicas.

```
facetsConfig = new FacetsConfig();
facetsConfig.setMultiValued(dimName:"rangoPrecio", value:false);
facetsConfig.setMultiValued(dimName:"rangoAverageRating", value:false);
facetsConfig.setMultiValued(dimName:"categorias", value:true);
```

3. **Método main:**

    - Llama al constructor e inicializa la interfaz gráfica mediante el método *crearInterfaz()*. Esto nos asegura que la GUI sea creada en el hilo correcto (*SwingUtilities.invokeLater*)



```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        try {
            new Interfaz().crearInterfaz();
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
}
```

4. **Creación de la interfaz gráfica**

El método *crearInterfaz()* construye la GUI con los siguientes elementos:

- **Árbol de categorías (JTree):**

    - Representa jerárquicamente las facetas de categorías obtenidas del índice.

    - Usamos el método *arbolCategorias()* para construir nodos basados en las facetas disponibles y sus correspondientes subcategorías.

- **Filtros de facetas:**

    - Se crean dinámicamente para *rangoPrecio* y *rangoAverageRating* con la función *anadirFacetas()*.

    - Las facetas disponibles se muestran con checkboxes (JCheckBox) con el formato “Etiqueta (Cantidad)”.

- **Campo de búsqueda y filtros adicionales:**

    - Un campo de texto (JTextField) permite al usuario introducir consultas.

    - El botón “Aplicar filtros” une la consulta con los filtros seleccionados para generar una búsqueda.

- **Tabla de resultados:**

    - Una tabla (JTable) enseña los documentos obtenidos con columnas como *Título*, *Descripción*, *Precio*, *Calificación promedio* y *Resumen*.

```
//Tabla para mostrar documentos
JTable table = new JTable(new DefaultTableModel(new Object[] { "Título","Descripción", "Precio","Calificación promedio", "Resumen" }, 0));
JScrollPane tableScrollPane = new JScrollPane(table);
```

    - Los resultados se actualizan de manera dinámica con paginación.

- **Botones de navegación:**

    - <-*Anterior* y *Siguiente*-> permiten moverse entre páginas de resultados (hay 30 resultados por página).

```
//Botones de paginación
JPanel paginationPanel = new JPanel(new FlowLayout());
JButton prevButton = new JButton("← Anterior");
JButton nextButton = new JButton("Siguiente →");
```

5. **Funcionalidades principales**

    a. **Creación de las consultas:**

El método *contruirConsulta()* crea una consulta compuesta:

- **Consulta base:** un *MatchAllDocsQuery*, que coincide con todos los documentos.

- **Consulta por texto:**

    - Detecta el idioma del término introducido con Tika y selecciona un analizador apropiado.

    - Aplica diferentes boosts a los términos buscados en campos concretos, como *tituloAnalizado*, *descripcionAnalizado* y *summary*.

    - Combina estas subconsultas en un *BooleanQuery*.

- **Filtros de categorías:** si el usuario selecciona una categoría en el árbol (JTree), se agrega como filtro.

- **Filtros de facetas:** usamos *DrillDownQuery* para añadir los filtros seleccionados (como lo son *rangoPrecio* o *rangoAverageRating*).

b. Búsqueda en el índice:

El método *buscarIndice()*:

- Ejecuta la consulta en el índice paralelo (*searcherParalelo.search(query)*).
- Extrae los documentos importantes (*TopDocs*) y los procesa para obtener campos como *title*, *description*, *price*, *averageRating*, etc.
- Devuelve una lista de documentos.

c. Filtros dinámicos:

El método *anadirFacetas()*:

- Obtiene los valores de facetas disponibles en el índice para un campo en concreto.
- Crea checkboxes para cada valor de faceta, mostrando en la interfaz el conteo correspondiente.

d. Actualización de resultados:

El método *actualizarTabla()*:

- Controla la paginación enseñando solo los documentos correspondientes a la página actual.
- Actualiza la tabla con los datos obtenidos de los documentos.

```
private void actualizarTabla(JTable table) {
    DefaultTableModel model = (DefaultTableModel) table.getModel();
    model.setRowCount(0);

    int start = paginaActual * PAGE_SIZE;
    int end = Math.min(start + PAGE_SIZE, documentos.size());

    for (int i = start; i < end; i++) {
        Document doc = documentos.get(i);
        model.addRow(new Object[] { doc.get(name:"title"), doc.get(name:"descripcion"), doc.get(name:"price"), doc.get(name:"averageRating"), doc.get(name:"summary")});

    }
}
```

e. Árbol de categorías:

El método *arbolCategorias()*:

- Construye un árbol jerárquico basado en las facetas del campo categorias.
- Cada nodo principal (categoría) puede tener subnodos (subcategorías) obtenidos mediante consultas adicionales.

6. **Análisis de texto**

En el programa hemos hecho dos funciones de análisis relacionadas con texto:

- **Detección de idioma:**
    - Usamos Tika para determinar el idioma de las consultas introducidas.
    - Seleccionamos un analizador adecuado para procesar los términos.
- **Normalización:**
    - Eliminamos las tildes en las consultas con el método *quitarTildes()* para mejorar las coincidencias.

La interfaz gráfica implementada en este código tiene varias ventajas importantes dado a su diseño y funcionalidades:

- **Búsqueda avanzada y personalizada**
    - **Soporte para consultas complejas:**

Combina diferentes tipos de búsqueda:

- Por texto (en múltiples campos como tituloAnalizado, descripcionAnalizado y summary).
- Por filtros de facetas (rangoPrecio, rangoAverageRating, categorias).
- Por boosts diferenciados para campos concretos, dando prioridad a algunas coincidencias.
    - **Búsqueda multilingüe:**

Detectamos de manera automática el idioma de la consulta con Tika y aplica un analizador adecuado, mejorando la precisión y relevancia de los resultados.

- **Soporte para facetas**
    - **Filtros dinámicos:**
        - Genera de manera automática checkboxes para facetas como rangos de precios, valoraciones promedio y categorías.
        - Muestra el número de documentos disponibles para cada valor de las facetas

- **Jerarquía de categorías:**

    - Presenta las categorías y subcategorías en un árbol (JTree), lo que permite al usuario explorar y filtrar datos intuitivamente.

- **Interfaz gráfica clara**

    - **Estructura clara y organizada:**

        - Incluye un árbol de categorías, filtros laterales, tabla central para resultados y botones de navegación.

        - Se permite realizar búsquedas desde un campo de texto y aplicar filtros adicionales fácilmente.

    - **Paginación eficiente:**

        - Muestramos los resultados en páginas, lo que mejora la experiencia del usuario al manejar grandes cantidades de datos.

        - Botones *anterior* y *siguiente* que ayudan a la navegación entre páginas.

- **Procesamiento eficiente de datos**

    - **Integración de índices paralelos:**

        - Usamos un ParallelCompositeReader para unir múltiples índices (metadata, reviewsEspecial y metadataEspecial), lo que permite hacer búsquedas unificadas sin duplicación de datos.

    - **Optimización de consultas:**

        - Las consultas son construidas eficientemente, integrando filtros, boosts y búsquedas textuales en un solo flujo.

- **Escalabilidad y modularidad**

    - **Escalabilidad con múltiples índices:**

        - Maneja múltiples tipos de datos (metadata, reviews) en un único sistema, lo que facilita la extensión para nuevos índices.

    - **Modularidad del código:**

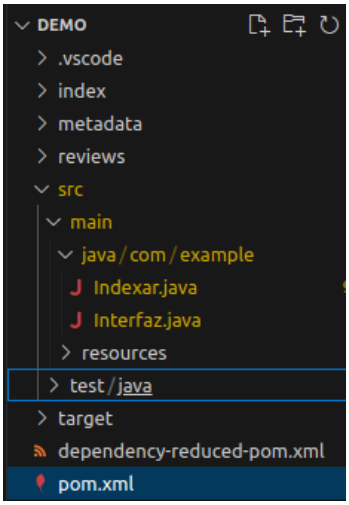
        - Métodos creados que nos ayudan en el mantenimiento y para añadir (en un futuro) nuevas funcionalidades.

- **Robustez y manejo de errores**
    - **Manejo de datos ausentes:**
        - Verificamos y procesamos documentos de manera que nos aseguramos que todos los campos requeridos estén disponibles antes de añadirlos a los resultados.
    - **Detección de idioma:**
        - Si no detecta un idioma en concreto, aplicamos un analizador predeterminado, asegurándonos que la búsqueda no falle.
- **Versatilidad en los resultados**
    - **Tabla de resultados:**
        - Presentamos la información que consideramos importante, como título, descripción, precio, calificación promedio y resumen.
        - La tabla se actualiza dinámicamente en base a los filtros y la paginación
- **Experiencia de usuario mejorada**
    - **Interfaz visual atractiva:**
        - Usamos componentes de Swing como JTable, JTree y JCheckBox, con paneles organizados.
    - **Actualización en tiempo real**
        - Cambios en filtros o categorías se mostrarán inmediatamente en la tabla de resultados.

4. Creación de programas con dependencias con Maven

Para preparar los programas en archivos .jar hemos utilizado Maven para aplicar nuestro código con las dependencias en unos archivos jar, en concreto uno para hacer la indexación y otro para la interfaz.

Primero creamos un proyecto de Maven, al cual le llamamos demo. En el habrá la siguiente estructura:



Donde se destacan dos elementos:

- src/main/java/com/example con los archivos con código con nuestras dos clases para hacer ambos objetivos.
- pom.xml que tiene las propiedades de compilación y empaquetado para aplicar todas las dependencias que necesita el código.

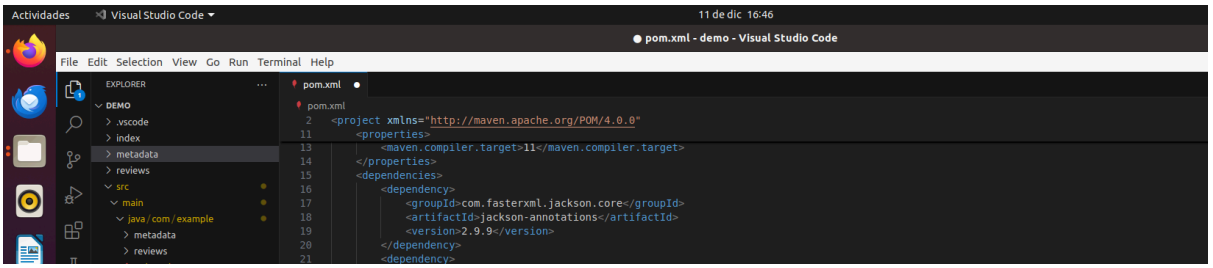
pom.xml tiene lo siguiente:

- La versión de xml y demás características de la compilación como la versión de java con la que se hace, y características de los jar creados.

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>
```

- Esta es una de todas las dependencias que se aplican al jar.



- Aquí el cambio de nombre para cada programa jar y uno de los plugins que utilizamos para la compilación.

```
<build>
    <finalName>Interfaz</finalName>
    <plugins>
        <!-- Plugin para compilar -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>11</source>
                <target>11</target>
                <encoding>UTF-8</encoding>
            </configuration>
        </plugin>
```

- Y por último el filtro para problemas relacionados con las firmas digitales y asignar la clase dada en cada objeto según el nombre del archivo java con nuestro código.

```
<configuration>
    <filters>
        <filter>
            <artifact>*:*</artifact>
            <excludes>
                <exclude>META-INF/*.SF</exclude>
                <exclude>META-INF/*.DSA</exclude>
                <exclude>META-INF/*.RSA</exclude>
            </excludes>
        </filter>
    </filters>
    <transformers>
        <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
            <mainClass>com.example.Indexar</mainClass> <!-- Cambia según tu clase principal -->
        </transformer>
    </transformers>
</configuration>
```

Tras tener esto listo se debe ejecutar lo siguiente por terminal:

- Para compilar de forma limpia poner: mvn clean compile
- Para empaquetar todo de forma limpia en un jar: mvn clean package

Y con estos pasos se obtiene una carpeta target con el jar. Obtenemos los jar para la indexación y la interfaz y los colocamos en una carpeta con las carpetas de metadata y reviews. Todo está especificado correctamente en la Guía del Usuario, pero se ejecutan con java -jar <nombre>.jar .

5. Conclusiones

A partir del desarrollo e implementación de los componentes que hemos realizado en esta práctica, hemos podido destacar las siguientes conclusiones:

1. **Integración efectiva de indexación y búsqueda avanzada:** la combinación de un proceso robusto de indexación (*Indexar.java*) y una interfaz gráfica intuitiva (*Interfaz.java*) nos permite construir un sistema capaz de gestionar grandes cantidades de información, dando resultados relevantes y personalizados al usuario final. Esto nos demuestra la capacidad de Apache Lucene para adaptarse a aplicaciones prácticas y complejas como sistemas de comercio electrónico o incluso bibliotecas digitales.

2. **Optimización del flujo de trabajo:** la modularidad y escalabilidad del sistema permite manejar múltiples tipos de datos (*metadata*, *reviews*, *metadataEspecial*, *reviewsEspecial*) en índices paralelos, garantizando eficiencia en el almacenamiento y recuperación. Además, las configuraciones avanzadas como el uso de similitud BM25 y la compactación de segmentos aseguran búsquedas rápidas y precisas.

3. **Soporte avanzado para facetas y filtros:** el uso de facetas jerárquicas en el índice y al meterlo en la interfaz gráfica proporciona una experiencia enriquecedora para el usuario ya que le permite realizar búsquedas basadas en categorías, rangos de precios o rango de valoraciones promedio. Esto destaca en la capacidad que tiene Lucene para trabajar con datos no estructurados y jerárquicos.

4. **Procesamiento multilingüe y personalizado:** la detección automática de idiomas y el uso de analizadores concretos mejoran la precisión en la indexación y recuperación de textos. Esto es muy útil en escenarios donde se aplican varios idiomas (como es en nuestro caso), asegurando que los términos de búsqueda se procesen adecuadamente según el idioma.

5. **Experiencia de usuario mejorada:** la interfaz gráfica que hemos hecho ofrece una navegación intuitiva gracias a elementos como el árbol de categorías, los filtros dinámicos, la tabla de resultados y la paginación. Hemos podido observar la posibilidad de integrar herramientas avanzadas de búsqueda en un entorno que sea accesible y fácil de usar.

6. **Robustez y manejo de errores:** en nuestro sistema hemos incluido controles para ir gestionando los errores en datos mal formateados, lo que garantiza un flujo continuo en el proceso de indexación y búsqueda. Este manejo ayuda a que el sistema sea más fiable en entornos reales.

7. **Escalabilidad y adaptabilidad:** la estructura modular del código permite ampliar fácilmente el sistema con nuevos índices, facetas o campos personalizados. Esto nos asegura su adaptabilidad a varios contextos y casos de uso, manteniendo un rendimiento eficiente incluso con colecciones de datos que sean crecientes.

8. **Aprendizaje práctico:** esta práctica nos ha permitido comprender y aplicar conceptos importantes de la indexación, búsqueda y diseño de interfaces, destacando la importancia de trabajar con herramientas como Lucene para implementar soluciones reales en el manejo de información.

En conclusión, esta práctica final nos ha permitido ver y aplicar varios aspectos que son fundamentales de indexación, búsqueda avanzada y diseño de interfaces, haciendo hincapié en los conocimientos que hemos conseguido de manera práctica. La combinación de un proceso robusto de indexación con una interfaz gráfica demuestra la versatilidad de Apache Lucene y la importancia de un diseño modular y escalable en sistemas reales.

Además, esta práctica nos ha hecho ver cómo las tecnologías actuales nos permiten enfrentar los problemas complejos en este ámbito, como la gestión de grandes cantidades de información multilingüe, ofreciendo soluciones precisas, personalizadas y accesibles al usuario final. Este trabajo ha sido aparte de un ejercicio técnico, una oportunidad para explorar el potencial de herramientas avanzadas en el desarrollo de aplicaciones prácticas, mostrándonos cómo tener una buena planificación y el uso de las tecnologías adecuadas pueden dar lugar a sistemas eficientes, adaptables y confiables.

6. Distribución del trabajo

Esta práctica ha sido desarrollada por los 3 integrantes del grupo. Tanto el código, la memoria, los ejecutables jar y el manual de usuario se han ideado y programado/escrito por Rafael García Andujar, Raúl Rodríguez Rodríguez y Mario Casas Pérez.