

PRÁCTICA III:

...

**Implementación de un Sistema de
Recuperación de Información utilizando
Lucene**

...

Indexación

16 de octubre de 2025

| | |
|---------------|---|
| <i>ÍNDICE</i> | 2 |
|---------------|---|

Índice

| | |
|---|-----------|
| 1. Objetivo | 3 |
| 2. Conjunto de Datos AirBnB Los Angeles | 3 |
| 3. Diseñando el Sistema de Recuperación de Información | 5 |
| 3.1. Índice en Lucene | 6 |
| 3.2. Determinando los campos de indexación de nuestros documentos . | 7 |
| 3.2.1. Atributos de un campo | 7 |
| 3.3. Búsqueda por Facetas | 9 |
| 4. Indexación de documentos | 10 |
| 4.1. Selección del analizador | 14 |
| 5. Creación de un Document Lucene | 16 |
| 6. Segmentos en Lucene | 19 |
| 7. Ejercicios | 22 |
| 8. Indexación de la base de datos XMarket | 25 |

1. Objetivo

El objetivo final de esta práctica es que el alumno comprenda todos los procesos que intervienen en el diseño de un Sistema de Recuperación de Información y cómo puede ser implementado utilizando la biblioteca Lucene. En concreto nos centraremos en la fase de construcción del índice. Para ello, en este curso utilizaremos el conjunto de datos **Los Angeles Airbnb Data (June 2025)**, visto en la práctica anterior. Nuestro objetivo será construir un programa que se pueda ejecutar en línea de comandos y que sea el encargado de generar/actualizar nuestro índice.

Recordemos que para construir una aplicación de búsqueda podemos distinguir los siguientes pasos, que detallaremos a continuación:

1. Análisis de requisitos y adquisición de datos.
2. Procesamiento de los datos, que ya hemos considerado en parte en prácticas anteriores.
3. Indexación y almacenamiento de los mismos. (esta práctica)
4. Búsqueda sobre el índice y presentación de los resultados.

2. Conjunto de Datos AirBnB Los Angeles

Este conjunto de datos contiene información pública del proyecto Inside Airbnb para Los Ángeles, California. Los datos se extrajeron el 17 de junio de 2025 y ofrecen un panorama detallado de los alojamientos de Airbnb en la ciudad en ese momento.

Inside Airbnb¹ es un proyecto independiente y sin fines de lucro que proporciona datos y promueve el impacto de Airbnb en comunidades de todo el mundo.

¹Este conjunto de datos proviene del proyecto Inside Airbnb. Los datos se utilizan bajo la Licencia Creative Commons Atribución 4.0 Internacional. Si utiliza estos datos, asegúrese de citar a Inside Airbnb.

Para encontrar la fuente original de los datos y más información podemos visitar <http://insideairbnb.com/get-the-data/>

| Tipo | Columnas |
|-----------|--|
| Propiedad | <code>id, listing_url, name, description,</code> <code>neighborhood_overview,</code> <code>neighbourhood_cleansed, latitude,</code> <code>longitude, property_type, bathrooms,</code> <code>bathrooms_text , bedrooms,</code> <code>amenities, price , number_of_reviews,</code> <code>review_scores_rating</code> |
| Anfitrión | <code>host_url, host_name, host_since,</code> <code>host_location, host_about,</code> <code>host_response_time, host_is_superhost,</code> <code>host_neighbourhood</code> |

Cuadro 1: Lista de columnas seleccionadas por tipología

El conjunto de datos se incluye en el archivo `listings.csv` que tiene aproximadamente 79 columnas para más de 45 000 alojamientos activos/inactivos. En nuestro caso no trabajaremos con todas las columnas sino que haremos una selección de las mismas, agrupándolas en dos tipologías, las relativas a la *Propiedad* y las relativas al *Anfitrión*, que consideraremos las básicas para realizar nuestra primera aplicación de búsqueda.

En concreto, las columnas seleccionadas las encontramos en el Cuadro 1.

Este conjunto de datos tiene muchas características que lo hacen atractivo para este curso:

- es lo suficientemente grande como para proporcionar algunos datos interesantes, pero lo suficientemente pequeño como para poder ejecutarlo en nuestro equipo de trabajo.
- los datos tienen relaciones que se pueden modelar en nuestro sistema
- no es demasiado complejo, lo que no distrae la atención del curso
- también hay valores incompletos o que se podrían considerar basura, lo cual es extremadamente valioso ya que será necesario identificarlos y decidir qué hacer con ellos.

Por lo tanto, será necesario hacer un análisis de los valores almacenados en los distintos campos para ver el formato con el que vienen almacenados los mismos, así como su integridad. Por ejemplo, nos podemos encontrar casos donde en algún campo aparezca información del tipo Run Runyon Canyon

Gym & Sauna
..., casos como la propiedad con *id* = 11709303 donde la fila parece estar corrupta en el CSV o valores incompletos.

3. Diseñando el Sistema de Recuperación de Información

A la hora de diseñar cualquier aplicación de búsqueda, el primer paso es analizar qué tipo de información se va a buscar y cómo se realizan las búsquedas por parte de un usuario. Para ello, uno de los primeros pasos es analizar los conjuntos de datos de que se dispone.

En la práctica consideraremos tres tipos de información, no necesariamente excluyente:

- La susceptible de ser tratada como categorías, como el género
- La numérica, como valoraciones, etc.
- La eminentemente textual, que contiene descripción del atributo en lenguaje natural

Una vez que conocemos los datos de nuestra práctica, podremos imaginar qué tipos de consultas se podrían realizar por un usuario así como la/las respuesta que daría el sistema, en la cual se devuelven los elementos ordenados por relevancia, facilitando las labores del usuario. Lucene permite hacer consultas por campos, por lo que será necesario identificar los mismos. No todos los campos en un conjunto de datos son importantes para las tareas de búsqueda y recuperación. A su vez, un mismo campo podrá utilizarse para dos funciones distintas, por ejemplo como texto sin tokenizar y texto tokenizado. Los campos concretos que utilicemos dependerán de la aplicación concreta sobre la que estemos trabajando.

3 DISEÑANDO EL SISTEMA DE RECUPERACIÓN DE INFORMACIÓN 6

Por ejemplo, si estamos interesados en una apartamento cerca de Rodeo Drive podemos realizar la consulta `apt Rodeo Drive`. En este caso, podemos realizar la consulta sobre el campo `description` o quizás también sobre el campo `neighborhood` pero si queremos tener información sobre los anfitriones podemos realizar la consulta sobre el campo `host_about`. Una vez recuperado los documentos podremos mostrar los resultados al usuario.

Como podemos esperar, no todos los documentos recuperados tienen que estar relacionados con la temática de la consulta. Esto es normal cuando consideramos un sistema de recuperación de información.

3.1. Índice en Lucene

EL objetivo de esta práctica es construir un índice, pero antes veamos que es un índice Lucene. Conceptualmente es similar a una tabla en una base de datos. La tabla en una base de datos relacional tradicional o en una base de datos NoSQL necesita al menos tener su esquema definido cuando se crea. Hay algunas restricciones claras en la definición de la clave primaria, las columnas y similares. Sin embargo, no existen tales restricciones en un índice Lucene.

Un índice Lucene puede entenderse como un conjunto de documentos. Se puede añadir un nuevo documento al conjunto o se puede sacar uno, pero si se quiere modificar uno de los documentos, primero hay que sacarlo, modificarlo y volver a meterlo en el conjunto. En nuestro conjunto podemos encontrar todo tipo de documentos (pueden tener distinto formato), y Lucene puede indexar el documento sea cual sea su contenido. Esto nos permitirá incluir dentro del mismo índice documentos de distinta tipología, y realizar la búsqueda sobre los mismos.

Pero, qué es un documento Lucene. Se prodría asimilar a una fila en una base de datos relacional. Cuando un documento se inserta en el índice se le asigna un único identificador (`DocId`). Los documentos se componen de uno o varios campos. Un campo es la unidad más pequeña definible de un índice de datos en Lucene. A cada campo se le asocia un conjunto de valores así como la forma en que los mismos se almacenará en el índice, que dependerá del tipo de campo (`FieldType`).

Lucene proporciona muchos tipos diferentes de campos ya creados, aunque

podemos personalizarlos y crear nuestros propios tipos de campos.

3.2. Determinando los campos de indexación de nuestros documentos

En gran medida, los campos a considerar depende del uso (tipos de consultas) que un usuario pueda realizar sobre nuestra colección así como la/las respuesta que daría el sistema.

Los campos concretos dependerá de la aplicación concreta sobre la que estamos trabajando, pero como hemos visto si es necesario un mismo campo podrá utilizarse para dos funciones distintas, por ejemplo como texto sin tokenizar y texto tokenizado.

En cualquier caso, en nuestra aplicación deberemos identificar al menos los siguientes tipos de campos sobre los documentos de entrada:

- **StringField:** Texto simple que se considera literalmente (no se tokeniza), útil para la búsqueda por facetas, filtrado de consultas y también para la presentación de resultados en la interfaz de búsqueda, por ejemplo ID, tags, direcciones webs, nombres de fichero, etc.
- **TextField:** Secuencia de términos que es procesada para la indexación, esto es, pasa por un analyzer pudiendo ser convertida a minúscula, tokenizada, estemizada, etc. Como podría ser el título, resumen, etc de un artículo científico.
- **Numérico,** datos que se expresan mediante este tipo de información, bien sean enteros o reales.
- **Facetas (Categorías)** que permiten una agrupación lógica de los documentos con la misma faceta o categoría, como por ejemplo podrían ser el vecindario.

3.2.1. Atributos de un campo

Adicionalmente, para cada campo podemos establecer un conjunto de atributos que determinan el comportamiento del mismo, entre ellos podemos destacar:

3 DISEÑANDO EL SISTEMA DE RECUPERACIÓN DE INFORMACIÓN 8

- stored: indica si se guarda el campo. Si es true se indexa y se almacena el campo. Si es falso, Lucene no almacena el valor del campo y como consecuencia los documentos devueltos en los resultados de la consulta sólo contendrán campos guardados.
- tokenized: representa si se tokeniza o no. En Lucene, sólo es necesario tokenizar el campo TextField.
- termVector: un vector de términos es una estructura de datos que guarda toda la información relacionada con un término, incluyendo el valor del término, las frecuencias y las posiciones del mismo en el documento. No se recomienda habilitar el vector de términos para los campos cortos porque toda la información de los términos puede obtenerse tokenizando de nuevo el campo. Sin embargo, se recomienda habilitar el vector de términos para campos más largos o con un alto coste de tokenización.

Cuando se activan, los term vectors se almacenan junto con el documento en el índice de Lucene y pueden consultarse durante la búsqueda para mejorar la precisión de los resultados. Sin embargo, el uso de term vectors aumenta el tamaño del índice y el uso de memoria,

Hay dos usos principales del vector de términos. El primero es el resaltado de palabras clave (highlighting) y el otro es la comparación de similitudes entre documentos (more-like-this).

- omitirNorms: Norms representa la normalización por la longitud del documento. Lucene permite que cada campo de cada documento tenga guardado un factor de normalización, que es un coeficiente que puede afectar al score del documento. Sólo se necesita un byte para guardar Norms, pero se guarda un Norms distinto en cada campo de cada documento y cada dato de Norms se carga en memoria. Así que habilitar las Normas consume espacio de almacenamiento y memoria adicionales. Pero si se desactiva, no podrá utilizarlo en tiempo de consulta.
- indexOptions: Lucene proporciona cinco parámetros opcionales para los índices invertidos (NONE, DOCS, DOCS_AND_FREQS,

DOCS_AND_FREQS_AND_POSITIONS,
DOC_AND_FREQS_AND_POSITIONS_AND_OFFSETS), que se utilizan para seleccionar si el campo necesita ser indexado, y qué contenido indexar.

- docValueType: DocValue es una característica introducida en Lucene 4.0 (asocia los docids a un campo), que mejora enormemente la eficiencia de la clasificación, las facetas y la agregación. DocValues es una estructura de almacenamiento con un esquema fuerte, por lo que todos los campos con DocValues activado deben tener exactamente el mismo tipo. Actualmente Lucene sólo proporciona los cinco tipos de NUMERIC, BINARY, SORTED, SORTED_NUMERIC y SORTED_SET.
- dimensión: Lucene soporta la indexación de datos multidimensionales, empleando una indexación especial para optimizar las consultas de datos multidimensionales. El escenario de uso más típico de este tipo de datos es un índice de ubicaciones geográficas. Este es el método de indexación que se utiliza generalmente para los datos de latitud y longitud.

3.3. Búsqueda por Facetas

La búsqueda por facetas, también conocida como "faceted search.^{en} inglés, es una técnica de búsqueda avanzada que permite a los usuarios refinar y filtrar los resultados de una consulta de búsqueda de manera más específica y detallada. En Lucene, al igual que en otros sistemas de búsqueda, las búsquedas por facetas son una característica poderosa que permite a los usuarios explorar y organizar grandes conjuntos de datos de manera efectiva. Son características específicas de los documentos que se utilizan para categorizar y organizar los resultados. Estas características pueden ser atributos de los documentos, como categorías, etiquetas, fechas, precios, autores, etc. Las facetas se extraen de los documentos durante el proceso de indexación. El proceso de búsqueda por facetas es el siguiente

- Resultados iniciales: Cuando un usuario realiza una consulta de búsqueda en Lucene, obtiene un conjunto inicial de resultados que coinciden con los términos de búsqueda.

- Recuento de facetas: Para cada faceta, Lucene calcula cuántos documentos en el conjunto de resultados iniciales tienen valores específicos para esa faceta. Por ejemplo, cuántos documentos tienen una categoría "Deportes", cuántos tienen una categoría "Tecnología", etc.
- Presentación de facetas: Los valores de las facetas se presentan al usuario como opciones de filtro. Estos pueden mostrarse en una barra lateral, una lista desplegable u otra interfaz de usuario. Los usuarios pueden hacer clic en una faceta para refinar los resultados.
- Refinamiento de la búsqueda: Cuando un usuario hace clic en una faceta específica, los resultados se ajustan automáticamente para mostrar solo los documentos que tienen ese valor de faceta. Esto permite a los usuarios reducir el conjunto de resultados y explorar de manera más efectiva.
- Facetas anidadas: A menudo, las facetas pueden ser anidadas o tener múltiples niveles. Por ejemplo, la categoría "Deportes" puede tener subcategorías como "Fútbol", "Baloncesto" "Tenis". Los usuarios pueden navegar por estas subcategorías para refinar aún más los resultados.

Las búsquedas por facetas son especialmente útiles en casos en los que se manejan grandes conjuntos de datos, como sitios de comercio electrónico, bibliotecas en línea, bases de datos de productos, etc. Los usuarios pueden explorar y refinar los resultados de manera intuitiva y rápida, lo que mejora la experiencia de búsqueda y la precisión de los resultados. En Lucene, las bibliotecas como Apache Solr y Elasticsearch ofrecen funcionalidades avanzadas para implementar búsquedas por facetas de manera eficiente. Aunque el uso de facetas lo consideraremos en la última práctica, desde ya podemos ir pensando en qué tipo de facetas o categorías vamos a considerar.

4. Indexación de documentos

Indexar es una de las principales tareas que podemos encontrar en Lucene. La clase que se encarga de la indexación de documentos es IndexWriter. Esta clase

Cuadro 2: Clases involucradas en la indexación

| Clase | Descripción |
|-------------|--|
| IndexWriter | Clase esencial que crea/modifica un índice. |
| Directory | Representa la ubicación del índice. |
| Analyzer | Es responsable de analizar un texto y obtener los tokens de indexación. |
| Document | Representa un documento Lucene, esto es, un conjunto de campos (fields) asociados al documento. |
| Field | La unidad más básica, representa un par clave-valor, donde la clave es el nombre que identifica al campo y el valor es el contenido del documento a indexar. |

se encuentra en `lucene-core` y permite añadir, borrar y actualizar documentos Lucene. Un documento Lucene esta compuesto por un conjunto de campos: par nombre del campo (string)- contenido del campo (string), como por ejemplo (“autor”, “Miguel de Cervantes”) o (“Título”, “Don Quijote de la Mancha”) o (“Cuerpo”, “En un lugar de la Mancha de cuyo....”). Cada uno de estos campos será indexado como parte de un documento, después de pasar por un Analyzer. En el Cuadro 2 podemos ver resumido el conjunto de clases que son usadas frecuentemente en la indexación.

El IndexWriter http://lucene.apache.org/core/10_3_0/core/org/apache/lucene/index/IndexWriter.html toma como entrada dos argumentos:

- Directory: Representa el lugar donde se almacenará el índice http://lucene.apache.org/core/10_3_0/core/org/apache/lucene/store/Directory.html. Podemos encontrar distintas implementaciones, pero para un desarrollo rápido de un prototipo podemos considerar ByteBuffersDirectory (el índice se almacena en memoria) o FSDirectory (el índice se almacena en el sistema de ficheros)
- IndexWriterConfig: es una clase en Lucene que se utiliza para configurar y personalizar el comportamiento del IndexWriter, que es responsable de agregar, actualizar y eliminar documentos en un índice de búsqueda. IndexWriterConfig permite especificar varias configuraciones y parámetros para el proceso de indexación.

La documentación sobre IndexWriterConfig la podemos encontrar en el propio Lucene http://lucene.apache.org/core/10_3_0/core/org/apache/lucene/index/IndexWriterConfig.html

Entre los elementos que podemos fijar se encuentran:

- Analyzer (Analizador): Puedes especificar el Analyzer que se debe utilizar para dividir y indexar el texto en el índice. El analizador tiene un impacto significativo en cómo se indexa el texto, por lo que es una configuración importante (lo veremos después).
- OpenMode (Modo de Apertura): Puedes establecer el OpenMode, que determina cómo debe tratarse el índice al abrirlo. Las opciones comunes incluyen OpenMode.CREATE (crear un índice nuevo), OpenMode.APPEND (añadir a un índice existente) y OpenMode.CREATE_OR_APPEND (crear un índice nuevo si no existe o abrir el existente para agregar más documentos). Se configura con el método setOpenMode.
- Similarity (Similitud): Puedes especificar el modelo de similitud que se utilizará para calcular la relevancia de los documentos durante la búsqueda. Dependiendo del modelo considerado, en indexación se almacenará toda la información para garantizar que los datos estén disponibles en la búsqueda, por tanto Lucene necesita conocer el modelo en tiempo de indexación y de búsqueda (el modelo debe ser el mismo en ambos casos). Esto se configura mediante el método setSimilarity. Por defecto Lucene usa el BM25, pero podemos encontrar otras como ClassicSimilarity (TFIDF), LMDirichletSimilarity, LMJelinekMercerSimilarity, etc. La documentación de sobre las mismas y su implementación en Lucene la podemos encontrar en
https://lucene.apache.org/core/10_3_0/core/org/apache/lucene/search/similarities/package-summary.html
- Política de Fusión (Merge): Puedes configurar la política de fusión que controla cómo se combinan los segmentos durante el proceso de optimización del índice. Lucene proporciona varias políticas de fusión,

y puedes configurarlas mediante el método setMergePolicy

- ... otras muchas que nos garantizan obtener el mejor índice para nuestras necesidades (ver documentación).

Aunque se recomienda mirar la documentación de las distintas clases, ilustraremos su uso mediante el siguiente ejemplo,

```

1 FSDirectory dir = FSDirectory.open( Paths.get(INDEX_DIR) );
2 IndexWriterConfig config = new IndexWriterConfig(analyzer);
3 config.setOpenMode(IndexWriterConfig.OpenMode.CREATE);
4 // config.setSimilarity(new LMDirichletSimilarity());
5
6
7 // Crea un nuevo índice
8 IndexWriter writer = new IndexWriter(dir, config);
9
10 List<String> listaFicheros = ....
11
12 for (String nombre: listaFicheros)
13     Document doc = ObtenerDocDesdeFichero(nombre);
14     writer.addDocument(doc);
15
16 writer.commit();
17 // Ejecuta todos los cambios pendientes en el índice
18 writer.close();

```

Las líneas 1 a 4 son las encargadas de crear las estructuras necesarias para el índice. En concreto el índice se almacenará en disco en la dirección dada por el path. La línea 2 declara un IndexWriterConfig config con el analizador que se utilizará para todos los campos (si no se indica, utilizará por defecto el standardAnalyzer). Para ello, debemos de seleccionar, de entre los tipos que tiene Lucene implementados (o los que nosotros creemos), el que se considere adecuado para nuestra aplicación. Este es un criterio importante para poder alcanzar los resultados óptimos en la búsqueda. Puede que sea necesario el utilizar un analizador distinto para cada uno de los posibles campos a indexar, en cuyo caso utilizaremos un PerFieldAnalyzerWrapper (ver siguiente sección).

La línea 3 se indica que el índice se abre en modo CREATE (crea un nuevo índice o sobreescribe uno ya existente), otras posibilidades son APPEND (añade

documentos a un índice existente) o CREATE_OR_APPEND (si el índice existe añade documentos, si no lo crea para permitir la adición de nuevos documentos. Es posible modificar la configuración del índice considerando múltiples setter. Por ejemplo, podremos indicar la función de similitud que se utiliza (ver línea 4) o criterios para la mezcla de índices, etc.

Por defecto, la función de similitud es BM25 con parámetros $k1 = 1,2$, $b = 0,75$ y discountOverlaps= true².

Una vez definida la configuración tenemos el IndexWriter listo para añadir nuevos documentos. Los cambios se realizarán en memoria y periódicamente se volcarán al Directory, que cuando sea necesario realizará la mezcla de distintos segmentos.

Una vez añadidos los documentos, debemos asegurarnos de llamar a commit() para realizar todos los cambios pendientes, línea 16. Finalmente podemos cerrar el índice mediante el comando close (línea 18).

4.1. Selección del analizador

El proceso de análisis nos permite identificar qué elementos (términos) serán utilizados en la búsqueda y cuales no (por ejemplo mediante el uso de palabras vacías)

Las operaciones que ejecuta un analizador incluyen: extracción de tokens, supresión de signos de puntuación, acentos o palabras comunes, conversión a minúsculas (normalización), stemización. Para ello, debemos de seleccionar el que se considere adecuado para nuestra aplicación, justificando nuestra decisión. Este es un criterio importante para poder alcanzar los resultados óptimos en la búsqueda.

En esta práctica será necesario utilizar un analizador distinto al por defecto para algunos de los posibles campos a indexar, para ello podemos utilizar un PerFieldAnalyzerWrapper³.

²Que discountOverlaps sea True implica que los tokens que se solapan (overlap), esto es tokens con un incremento de posición igual a cero (como los añadidos por un analizador que considera sinónimos) no son tenidos en cuenta a la hora de calcular la longitud del documento.

³https://lucene.apache.org/core/10_3_0/analysis/common/org/apache/lucene/analysis/miscellaneous/PerFieldAnalyzerWrapper.html

PerFieldAnalyzerWrapper es una clase en Lucene que se utiliza para aplicar analizadores diferentes a campos específicos en un índice de búsqueda. Esto permite personalizar el proceso de indexación y búsqueda para cada campo en función de sus requisitos individuales. La principal ventaja de PerFieldAnalyzerWrapper es que puedes definir analizadores diferentes para cada campo, lo que es especialmente útil cuando los campos contienen tipos de datos diversos o cuando se necesita un análisis especializado.

Algunos de los usos comunes de PerFieldAnalyzerWrapper incluyen:

- Indexación de campos de texto: Puedes utilizar un analizador estándar como StandardAnalyzer para campos de texto que requieren un análisis textual completo.
- Campos de palabras clave: Para campos que contienen palabras clave o identificadores únicos, puedes utilizar un analizador de palabras clave o un analizador simple que no realice divisiones de palabras ni análisis.
- Campos de fechas: Para campos que almacenan fechas, puedes utilizar un analizador especializado que convierta y normalice las fechas en un formato específico.
- Campos personalizados: Si tienes campos con necesidades de análisis particulares, puedes utilizar analizadores personalizados para esos campos.

```
1 // map field -name to analyzer
2 Map<String , Analyzer> analyzerPerField = new HashMap<>();
3 analyzerPerField.put( "uncampo" , new StandardAnalyzer() );
4 analyzerPerField.put( "otrocampo" , new EnglishAnalyzer() );
5
6 // create a per-field analyzer wrapper using the Whitespace as
.. default analyzer ;
7 PerFieldAnalyzerWrapper analyzer = new PerFieldAnalyzerWrapper(
    new WhitespaceAnalyzer() , analyzerPerField );
```

5. Creación de un Document Lucene

Hemos visto que para indexar la información es necesario la creación de un documento, Document, Lucene. La documentación la podemos encontrar en http://lucene.apache.org/core/10_3_0/core/org/apache/lucene/document/Document.html.

Un Document es la unidad de indexación y búsqueda. Está compuesto por un conjunto de campos Fields, cada uno con su nombre y su valor textual. Los distintos campos de un documento se añaden utilizando el método add, por ejemplo

```
1 Document doc = new Document()
2 doc.add(field1)
3 doc.add(field2)
```

Un field puede estar asociado al nombre de un producto, su descripción, su ID, etc. Veremos brevemente cómo se gestionan los Fields ya que es la estructura básica de la que está compuesta un Document. Información sobre los Fields la podemos encontrar en http://lucene.apache.org/core/10_3_0/core/org/apache/lucene/document/Field.html.

Un Field tiene tres componentes, el nombre, el valor y el tipo. En nombre hace referencia la nombre del campo (equivaldría al nombre de una columna en una tabla). El valor hace referencia al contenido del campo (la celda de la tabla) y puede ser texto (String, Reader o un TokenStream ya analizado), binario o numérico. El tipo determina como el campo es tratado, por ejemplo indicar si se almacena (store) en el índice, lo que permitirá devolver la información asociada en tiempo de consulta, o si se tokeniza.

Para simplificar un poco la tarea, Lucene dispone de tipos predefinidos, entre los que podemos destacar

- **TextField:** Reader o String indexado y tokenizado, sin term-vector⁴. Sirve para almacenar el contenido de un documento, sobre el que normalmente

⁴En Lucene, un term-vector implica que para cada término conocemos: el id del documento, el nombre del campo, el texto en sí, la frecuencia, la posición y los offsets. Esta información podemos hacer cosas interesantes en la búsqueda como conocer dónde emparejan los términos de la consulta (por ejemplo para hacer un highlighting)

realizaremos las búsquedas. Suele pasar por un analizador antes de almacenar la información en el índice.

- **StringField:** Un campo String que se indexa como un único token. Por ejemplo, se puede utilizar para ID de un producto, el path donde se encuentra el archivo, etc. Si queremos que la salida de una búsqueda pueda ser ordenada según este campo, se tiene que añadir otro campo del tipo SortedDocValuesField.

Para profundizar mas sobre TextField y StringField se puede consultar <https://northcoder.com/post/lucene-fields-and-term-vectors/>

- **IntPoint:** Entero indexado para búsquedas exactas o por rango. Si queremos devolverlo en consultas, lo debemos de añadir como StoredField
- **LongPoint:** Long indexado para búsquedas exactas o por rango. Si queremos devolverlo en consultas, lo debemos de añadir como StoredField
- **FloatPoint:** float indexado para búsquedas exactas o por rango. Si queremos devolverlo en consultas, lo debemos de añadir como StoredField
- **DoublePoint:** double indexado para búsquedas exactas o por rango. Si queremos devolverlo en consultas, lo debemos de añadir como StoredField.
- **LatLonPoint:** par de valores que permiten almacenar datos geográficos como latitud-longitud y realizar consultas geográficas. Si quieres recuperar luego las coordenadas exactas lo debemos de añadir como StoredField.
- **SortedDocValuesField:** byte[] indexados con el objetivo de permitir la ordenación o el uso de facetas por el campo
- **SortedSetDocValuesField:** Permite añadir un conjunto de valores, SortedSet, al campo para su uso en facetas, agrupaciones o joinings.
- **NumericDocValuesField:** Field que almacena a valor long por documento con el fin de utilizarlo para ordenación, facetas o el propio cálculo de scores.

- **SortedNumericDocValuesField:** Añade un conjunto de valores numéricos, **SortedSet**, al campo
- **StoredField:** Valores almacenados que solo se utilizan para ser devueltos en las búsquedas.

Los distintos campos de un documento se añaden con el método add.

```

1 Document doc = new Document();
2
3 doc.add(new StringField("isbn", "978-0071809252", Field.Store.YES));
4 doc.add(new TextField("titulo", "Java: A Beginner's Guide, Sixth Edition", Field.Store.YES)); // por defecto no se
5 doc.add(new TextField("contenido", "Fully updated for Java Platform, Standard Edition 8 (Java SE 8), Java ....."));
6 doc.add(new IntPoint("size", 148));
7 doc.add(new StoredField("size", 148));
8 doc.add(new SortedSetDocValuesField("format", new BytesRef("paperback")));
9 doc.add(new SortedSetDocValuesField("format", new BytesRef("kindle")));
10
11 Date date = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss'Z'").parse(...);
12 doc.add(new LongPoint("Date", date.getTime()));

```

Como podemos imaginar, para cada tipo tenemos un comportamiento específico, aunque nosotros podremos crear nuestro propio tipo de campo.

```

1 FieldType authorType = new FieldType();
2 authorType.setIndexOptions(IndexOptions.DOCS_AND_FREQS);
3 // authorType.setIndexOptions(IndexOptions.
4     DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS);
5 authorType.setStored(true);
6 authorType.setOmitNorms(true);
7 authorType.setTokenized(false);
8 authorType.setStoreTermVectors(true);
9

```

```
10 doc.add(new Field("author", "Arnaud Cogoluegnes", authorType))  
11 ;  
11 doc.add(new Field("author", "Thierry Templier", authorType));  
12 doc.add(new Field("author", "Gary Gregory", authorType));
```

6. Segmentos en Lucene

Veamos con un poco mas de detalle como Lucene gestiona internamente un índice.

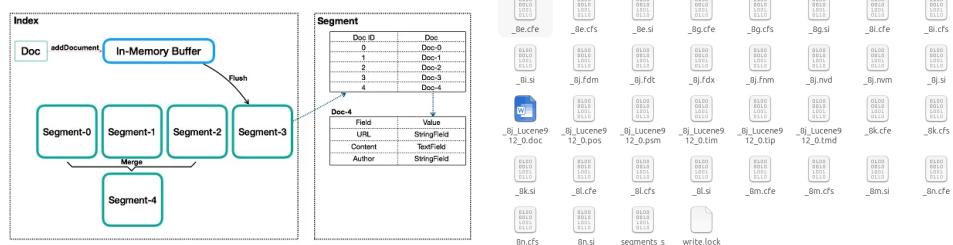
Un índice se compone de uno o varios subíndices. Un subíndice se denomina segmento y se refiere a una parte indivisible y autocontenido de un índice de búsqueda. Cada segmento, compuesto de múltiples ficheros, representa un índice invertido que relaciona términos con los documentos que los contienen. Estos índices permiten búsquedas rápidas y eficientes al identificar los documentos que coinciden con un término de búsqueda, pero hay que tener en cuenta que el número de segmentos de un índice Lucene juega un papel crucial en el rendimiento de las búsquedas y las operaciones de actualización y eliminación. Cuando se realiza una búsqueda, Lucene consulta cada segmento del índice, combinando luego los resultados. La cantidad de segmentos afecta directamente el tiempo de búsqueda, el uso de memoria y el rendimiento de las actualizaciones y eliminaciones. Por ejemplo, cuantos más segmentos tenga un índice, más consultas y fusiones de resultados debe realizar Lucene durante una búsqueda, lo que puede ralentizar la respuesta.

Para garantizar un tiempo de respuesta correcto, y conforme los segmentos crecen en tamaño, Lucene realiza fusiones (merge) de segmentos para reducir su número y optimizar el rendimiento del índice. Al realizar una mezcla, los documentos que estaban marcados como borrados en un índice son eliminados físicamente del mismo. Estas fusiones son intensivas en recursos (CPU y disco) y pueden afectar el rendimiento de la aplicación. Se puede reducir la frecuencia de fusiones a especificando las políticas de mezcla MergePolicy (por ejemplo, basada en el número de documentos del segmento o su tamaño)

A la izquierda de la figura 1 se muestra, a muy alto nivel, la estructura básica de un índice Lucene. A la derecha se muestra los ficheros que componen un índice

con varios segmentos.

Figura 1: Segmentos en Lucene, a la izquierda un índice con un único segmento, a la derecha uno con varios de ellos.



Cuando Lucene escribe datos, primero lo hace en un buffer en memoria (similar a MemTable en LSM, Log-Structured Merge). Cuando los datos del buffer alcanzan una determinada cantidad, se vacían para convertirse en un segmento. Cada segmento tiene su propio índice independiente y se puede buscar de forma independiente, pero los datos nunca pueden ser modificados. Este esquema evita las escrituras aleatorias. Los datos se escriben en lotes (Batch) y se añaden (Append) alcanzando un alto rendimiento. Los documentos escritos en el Segmento no pueden ser modificados, pero pueden ser borrados. El método de borrado no modifica el archivo en su ubicación interna original, pero el DocID del documento que se va a borrar se guarda en otro archivo para garantizar que el archivo de datos no pueda ser modificado.

El diseño conceptual de los segmentos en Lucene es similar al de LSM (log-Structured Merge) que es un estándar en bases de datos NoSQL y utilizado en Google BigTable, Apache Hbase, Apache Cassandra y muchos otros.

Los segmentos Lucene heredan las ventajas de escritura de datos de un LSM, pero sólo proporcionan consultas en tiempo casi real y no en tiempo real. Hasta que la información no está volcada al índice, los datos se almacenan en la memoria y no se pueden buscar. Esta es otra razón por la que se dice que Lucene proporciona consultas en tiempo casi real y no en tiempo real.

Algunas características clave de los segmentos en un índice Lucene:

- Independencia y Atómico: Cada segmento es independiente y atómico, lo que significa que puede considerarse como una unidad de indexación y bús-

queda por sí misma. Esto permite realizar operaciones de escritura y búsqueda en paralelo, ya que cada segmento es independiente de los demás.

- Optimización: Lucene optimiza los segmentos para mejorar el rendimiento y reducir el espacio en disco. Esto implica fusionar segmentos más pequeños en segmentos más grandes y eliminar documentos eliminados. La optimización es un proceso continuo en Lucene.
- Actualizaciones Incrementales: Cuando se actualiza un documento en un índice Lucene, se crea un nuevo segmento con la versión actualizada del documento. Los segmentos antiguos no se modifican, lo que garantiza la integridad de los datos históricos.
- Eliminación Lógica: La eliminación de documentos se realiza de manera lógica, lo que significa que los documentos no se eliminan físicamente de un segmento hasta que se optimiza el índice.
- Búsqueda Eficiente: Los segmentos permiten búsquedas eficientes, ya que los términos se almacenan en índices invertidos que apuntan a los documentos correspondientes en cada segmento. La búsqueda se realiza en paralelo en todos los segmentos.
- Estructura de Directorio: Los segmentos se almacenan en el sistema de archivos en un directorio separado. Un índice de Lucene generalmente contiene varios segmentos, y la estructura del directorio se utiliza para administrar y acceder a estos segmentos.
- Recuperación y Tolerancia de Fallos: La estructura de segmentos proporciona una forma de recuperación y tolerancia de fallos. Si un segmento se corrompe o se pierde debido a un error, los demás segmentos en el índice aún son utilizables.

Los segmentos son una parte fundamental de la arquitectura de Lucene y desempeñan un papel crucial en la eficiencia y la administración de índices de búsqueda. La segmentación facilita la administración de grandes cantidades de datos y permite la optimización gradual y la búsqueda eficiente en colecciones de documentos indexados.

7. Ejercicios

1. Basandonos en el código anterior, implementar un pequeño programa que nos permite añadir varios documentos a un índice Lucene, podemos seguir el siguiente esquema.

```
1 import org.apache.lucene.analysis.Analyzer;
2 import org.apache.lucene.analysis.standard.
3     StandardAnalyzer;
4 .....
5
6 public class IndiceSimple {
7
8     String indexPath = "./index";
9     String docPath = "./DataSet";
10
11    boolean create = true;
12
13    private IndexWriter writer;
14
15    public static void main(String[] args) {
16        // Analizador a utilizar
17        Analyzer analyzer = new StandardAnalyzer();
18        // Medida de Similitud (modelo de recuperacion) por
19        // defecto BM25
20        Similarity similarity = new ClassicSimilarity();
21        // Llamados al constructor con los parametros
22        IndiceSimple baseline = new IndiceSimple( ... );
23
24        // Creamos el indice
25        baseline.configurarIndice(analyzer, similarity);
26        // Insertar los documentos
27        baseline.indexarDocumentos();
28        // Cerramos el indice
29        baseline.close();
30
31    }
32 }
```

```
33 public void configurarIndice(Analyzer analyzer, Similarity
34     similarity) throws IOException {
35
36     IndexWriterConfig iwc = new IndexWriterConfig(
37         analyzer);
38     iwc.setSimilarity(similarity);
39     // Crear un nuevo indice cada vez que se ejecute
40     iwc.setOpenMode(IndexWriterConfig.OpenMode.CREATE);
41     // Para insertar documentos a un indice existente
42     // iwc.setOpenMode(IndexWriterConfig.OpenMode.
43     // CREATE_OR_APPEND);
44
45     // Localizacion del indice
46     Directory dir= FSDirectory.open(Paths.get(indexPath)
47         );
48
49     // Creamos el indice
50     writer = new IndexWriter(dir, iwc);
51
52 }
53
54 public void indexarDocumentos() {
55
56     // Para cada uno de los documentos a insertar
57     for (elementos d : docPath) {
58
59         // leemos el documento sobre un string
60         String cadena = leerDocumento( d );
61         // cadena <-- "134, hola esto es un ejemplo ....", 233
62
63         // creamos el documento Lucene
64         Document doc = new Document();
65
66         // parseamos la cadena (si es necesario)
67         Integer start,end; // Posiciones inicio ,fin
```

```
68
69      // Obtener campo Entero de cadena
70      Integer start = ... // Posicion de inicio del campo;
71      Integer end = ... // Posicion fin del campo;
72      String aux = cadena.substring(start, end);
73      Integer valor = Integer.decode(aux);
74
75      // Almacenamos en el campo en el documento Lucene
76      doc.add(new IntPoint("ID", valor));
77      doc.add(new StoredField("ID", valor));
78
79      // Obtener campo texto de cadena
80      start = ... // Posicion de inicio del campo;
81      end = ... // Posicion fin del campo;
82      String cuerpo = cadena.substring(start, end);
83      // Almacenamos en el campo en el documento Lucene
84      doc.add(new TextField("Body", cuerpo, Field.Store.YES));
85
86      // Obtenemos los siguientes campos
87
88      // .....
89
90      // Insertamos el documento Lucene en el indice
91      writer.addDocument(doc);
92      // Si lo que queremos es actualizar el documento
93      // writer.updateDocument(new Term("ID", valor.
94      // toString()), doc);
95
96  }
97
98
99  public void close() {
100    try {
101      writer.commit();
102      writer.close();
103    } catch (IOException e) {
104      System.out.println("Error closing the index.");
```

```
105     }
106     }
107     }
108 }
```

2. Utilizar Luke para ver el índice y realizar distintas consultas sobre el mismo.

8. Indexación de la base de datos AirBnB

Cada grupo deberá diseñar la estructura de los índices necesaria para almacenar la información asociada a nuestra base de datos, tanto a nivel de *propiedad* como de *anfitriones*. Los alumnos que trabajen solos podrán optar por trabajar solo con el fichero de *propiedad*. El resto del grupo deberá considerar el trabajar con los datos de los dos tipos de información

El primer paso será configurar cada uno de los índices, identificando los campos y en su caso el analizador que se deseé utilizar. Una vez configurado el índice, pasaremos al proceso de indexación propiamente dicho. Para ello, se deberá identificar la información que queremos almacenar como documento Lucene. Una vez extraída la información textual, debemos procesarla como sea necesario para crear el documento Lucene con sus campos correspondientes y añadirlo al índice.

Se debe generar un ejecutable java (.jar) con todas las dependencias que recibirá los parámetros que permitan crear o bien crear un índice desde el principio o añadir posibles nuevos capítulos a la colección. Para ello, un parámetro será el tipo del fichero que contiene los datos así como el directorio donde se encuentran estos.

La corrección del índice creado la podemos ver si lo abrimos con Luke.

La entrega de la práctica es el 5 de noviembre de 2025.