# Búsqueda y Minería de Información 2018-2019

**Universidad Autónoma de Madrid, Escuela Politécnica Superior**

**Grado en Ingeniería Informática 4º curso**

# Práctica 2 – Implementación de índices y funciones de ránking

## Fechas

- Comienzo: miércoles 20 de febrero
- Entrega: martes 12 de marzo (23:55h)

## Objetivos

Los objetivos de esta práctica son:

- La implementación eficiente de funciones de ránking, particularizada en el modelo vectorial.
- La implementación de índices eficientes para motores de búsqueda.

Se desarrollarán implementaciones de índices utilizando un diccionario y listas de postings. Y se implementará el modelo vectorial utilizando estas estructuras más eficientes para la ejecución de consultas.

Se podrá comparar el rendimiento de las diferentes versiones de índices y buscadores, contrastando la coherencia con los planteamientos estudiados a nivel teórico.

Mediante el nivel de abstracción seguido, se conseguirán versiones intercambiables de índices y buscadores. El único buscador que no será intercambiable es el de Lucene, que sólo funcionará con sus propios índices.

## Material proporcionado

Se proporcionan:

- Varias clases e interfaces Java en un comprimido src.zip, con las que el estudiante integrará las suyas propias.

  Las clases parten del código de la práctica anterior con algún ligero retoque y renombrado en el diseño (se adjunta explicación de los mismos en el fichero "cambios.txt").

  Igual que en la práctica 1, la clase `es.uam.eps.bmi.search.test.TestEngine` incluye un programa main que deberá funcionar con las clases a implementar por el estudiante.
- Las colecciones de prueba de la práctica 1: docs1k.zip con 1.000 documentos HTML y un pequeño fichero urls.txt.
- Dos colecciones más grandes: docs10k.zip y docs100k.zip, con 10.000 y 100.000 documentos HTML, respectivamente.
- Un documento de texto "test-output.txt" con la salida estándar que deberá producir la ejecución del programa "java TestEngine" (obtenido con un procesador de 2.5GHz, 2 cores, 16GB de RAM).

Además, el estudiante utilizará los mismos .jar que en la práctica anterior: como mínimo se necesitarán lucene-core-7.7.0.jar, lucene-queryparser-7.7.0.jar, y la librería JSoup.

## Ejercicios

### 1. Implementación de un modelo vectorial eficiente

Se mejorará la implementación de la práctica anterior aplicando algoritmos estudiados en las clases de teoría. En particular, se utilizarán listas de postings en lugar de un índice forward.

La reimplementación seguirá haciendo uso de la interfaz `Index`, y se podrá probar con cualquier implementación de esta interfaz (tanto la implementación de índice sobre Lucene como las propias).

#### 1.1 Método orientado a términos (1.5pt)

Escribir una clase `es.uam.eps.bmi.search.vsm.TermBasedVSMEngine` que implemente el modelo vectorial por el método orientado a téminos. Se sugiere definir esta clase como subclase (directa o indirecta) de `es.uam.eps.bmi.search.AbstractEngine`.

### 1.2 Método orientado a documentos (2pt)

Implementar el método orientado a documentos (con heap de coseno) en una clase `es.uam.eps.bmi.search.vsm.DocBasedVSMEngine`. Se sugiere definir esta clase como subclase (directa o indirecta) de `es.uam.eps.bmi.search.AbstractEngine`.

### 1.3 Heap de ránking (0.5pt)

Reimplementar las clases necesarias del paquete `es.uam.eps.bmi.search.ranking.impl` para utilizar un heap de ránking. Nótese que esta opción se aprovecha mejor con la implementación orientada a documentos, aunque es compatible con la orientada a términos.

## 2. Índice en RAM (3pt)

Implementar un índice propio que pueda hacer las mismas funciones que la implementación basada en Lucene definida en la práctica 1. Como primera fase más sencilla, los índices se crearán completamente en RAM. Se guardarán a disco y leerán de disco en modo serializado.

Para guardar el índice se utilizarán los nombres de fichero definidos por las variables estáticas de la clase `es.uam.eps.bmi.search.index.Config`.

Antes de guardar el índice, se borrarán todos los ficheros que pueda haber creados en el directorio del índice. Asimismo, el directorio se creará si no estuviera creado, de forma que no haga falta crearlo a mano. Este detalle se hará igual en los siguientes ejercicios.

### 2.1 Estructura de índice

Implementar la clase `es.uam.eps.bmi.search.index.impl.SerializedRAMIndex` como subclase de `es.uam.eps.bmi.search.index.AbstractIndex` con las estructuras necesarias: diccionario, listas de postings, más la información que se necesite.

Para esta práctica en las listas de postings sólo será necesario guardar los docIDs y las frecuencias; no es necesario almacenar las posiciones de los términos.

### 2.2 Construcción del índice

Implementar la clase `es.uam.eps.bmi.search.index.impl.SerializedRAMIndexBuilder` como implementación (directa o indirecta) de `es.uam.eps.bmi.search.index.IndexBuilder`, que cree todo el índice en RAM a partir de una colección de documentos.

## 3. Índice en disco (2pt)

Reimplementar los índices definiendo las clases `es.uam.eps.bmi.search.index.impl.DiskIndex` y `es.uam.eps.bmi.search.index.impl.DiskIndexBuilder` de forma que:

- El índice se siga creando entero en RAM (por ejemplo, usando estructuras similares a las del ejercicio 2).
- Pero el índice se guarde en disco dato a dato (docIDs, frecuencias, etc.).
- Al cargar el índice, sólo el diccionario se lee a RAM, y se accede a las listas de postings en disco cuando son necesarias (p.e. en tiempo de consulta).

Se sugiere guardar el diccionario en un fichero y las listas de postings en otro, utilizando los nombres de fichero definidos como variables estáticas en la clase `es.uam.eps.bmi.search.index.Config`.

Se valorará positivamente la separación de dos versiones de índice (en clases diferenciadas): estructuras para la construcción, y estructuras para el uso del índice. Las primeras (p.e. tablas hash, arrays dinámicos) favorecerán la velocidad de construcción, y facilitarán la programación; las segundas (p.e. arrays ordenados, arrays de longitud fija) ajustarán mejor el gasto de memoria.

## 4. Creación fraccionada del índice (1.5pt)

Reimplementar la construcción del índice en `es.uam.eps.bmi.search.index.impl.EfficientIndexBuilder` limitando el gasto de RAM, de forma que el índice se construya en varias fases volcando a disco segmentos parciales que se van fusionando. Se sugiere como límite utilizar no más de 1GB de memoria.

Esta versión de builder deberá ser compatible con la clase `DiskIndex` implementada en el ejercicio 3.

## 5. Ley de Heap (0.5pt)

Comprobar la ley de Heap en la construcción de alguno de los índices. Incluir en la memoria una gráfica con los datos resultantes.

## Indicaciones

Se sugiere trabajar en la práctica de manera incremental, asegurando la implementación de soluciones sencillas y mejorándolas de forma modular (la propia estructura de ejercicios plantea ya esta forma de trabajar).

En esta misma línea, se sugiere inicialmente guardar en disco las estructuras de índice en modo texto para poder depurar los programas. Una vez asegurada la corrección de los programas, puede ser más fácil pasar a modo binario.

Se podrán definir clases adicionales a las que se indican en el enunciado, por ejemplo, para reutilizar código. Y el estudiante podrá utilizar o no el software que se le proporciona, con la siguiente limitación:

- No deberá editarse el software proporcionado.
- El programa **TestEngine deberá compilar y ejecutar** correctamente.

Para la corrección de la práctica, el profesor unirá los fuentes entregados por el estudiante más los fuentes proporcionados por el profesor, comprobando que no haya archivos .java repetidos ni errores de compilación.

Por otra parte **en la memoria se reportarán los datos del coste y rendimiento** de cada tipo de índice implementado en una tabla como la siguiente (indicando las características del procesador del ordenador utilizado):

| | Construcción del índice | | | Carga del índice | |
|---|---|---|---|---|---|
| | Tiempo de indexado | Consumo máx. RAM | Espacio en disco | Tiempo de carga | Consumo máx. RAM |
| 1K | | | | | |
| 10K | | | | | |
| 100K | | | | | |

## Entrega

La entrega consistirá en un único fichero zip con el nombre **bmi-p2-XX.zip**, donde XX debe sustituirse por el número de pareja (01, 02, …, 10, …). Este fichero contendrá:

- Una carpeta **src/** con todos (**y sólo**) los ficheros .java con las implementaciones solicitadas en los ejercicios. Los .java se ubicarán en la ruta apropiada de subcarpetas según sus packages.
- En su caso, una carpeta **lib/** con los .jar adicionales que fuesen necesarios (no se incluirán los de Lucene ni JSoup). Se recomienda no obstante consultar con el profesor antes de hacer uso de librerías adicionales.
- Una **memoria bmi-p2-XX.pdf** donde se documentará:
  - En una primera sección, un listado sucinto de a) qué ejercicios y opciones se han realizado exactamente y b) en los correspondientes apartados, qué estructuras de datos se han utilizado para el diccionario y las listas de postings. Se enfatizarán en su caso los aspectos que puedan evaluarse favorablemente.
  - Un diagrama de clases.
  - Una sección donde se analicen resumidamente las diferencias de rendimiento observadas entre las diferentes implementaciones que se han creado y probado para cada componente.
  - Y cualquier otro aspecto que el estudiante considere oportuno destacar.

  La calidad de la memoria representará orientativamente el 10% de la puntuación de los ejercicios que se documentan en la misma.

No se deberán incluir en el .zip ninguno de los materiales proporcionados por el profesor (ni tan siquiera las clases Java) ni los .jar de Lucene ni JSoup, ni los archivos de proyecto Netbeans o Eclipse –ni por supuesto ningún .class!

El fichero de entrega se enviará por el enlace habilitado al efecto en el curso **Moodle** de la asignatura.

## Calificación

Esta práctica se calificará con una puntuación de 0 a 10 atendiendo a las puntuaciones individuales de ejercicios y apartados dadas en el enunciado. El peso de la nota de esta práctica en la calificación final de prácticas es del **30%**.

La calificación se basará en a) el **número** de ejercicios realizados y b) la **calidad** de los mismos. La calidad se valorará por los **resultados** conseguidos (economía de consumo de RAM, disco y tiempo; tamaño de las colecciones que se consigan indexar) pero también del **mérito** en términos del interés de las técnicas aplicadas y la buena programación.

La puntuación que se indica en cada apartado es orientativa, en principio se aplicará tal cual se refleja pero podrá matizarse por criterios de buen sentido si se da el caso.

Para dar por válida la realización de un ejercicio, el código deberá funcionar (a la primera) integrado con las clases que se facilitan. El profesor comprobará este aspecto añadiendo las clases entregadas por el estudiante a las clases facilitadas en la práctica, ejecutando el programa `TestEngine` así como otros main de prueba adicionales.