Memoria de la práctica 2

Búsqueda y Minería de la Información

Jorge Cifuentes, Alejandro Martín, pareja 01

Ejercicios realizados

Ejercicio 1

1.1) Motor de búsqueda orientado a términos: TermBasedVSMEngine

Hemos usado una Hashtable de pares (Integer – Double) como acumulador de los valores sumados al recorrer cada Posting List.

1.2) Motor de búsqueda orientado a documentos: DocBasedVSMEngine

Para implementar el motor de búsqueda orientado a documentos, se ha usado nuevamente una Hashtable de pares (Integer – Double) como acumulador de los valores sumados al recorrer cada Posting List, dos ArrayList (uno que contenga todos los postings de los términos y otro para guardar los iteradores de la posting list correspondiente) y otra Hashtable (Integer – Posting) para mantener los postings activos, simulando el comportamiento del Heap de postings.

Antes de comenzar a calcular el ranking obtenemos las listas de postings de todos los términos con sus iteradores, ya que en este motor de búsqueda, las listas de postings se recorren simultáneamente.

Para comenzar el algoritmo, se obtienen los primeros postings de cada término. A partir de aquí, se recorren todas las listas de postings hasta que todas queden vacías. En este proceso, se obtiene el tf-idf del posting con menor docID del Heap, para sumárselo al acumulador.

Una vez se ha terminado de recorrer la lista, se dividen todos los acumuladores por sus respectivos módulos y se introducen al ranking.

1.3) Heap de ranking: RankingIteratorImpl

Para el Heap de ranking, el cual es un MaxHeap, hemos usado la clase ya implementada de java TreeSet (coste del orden log(n) en operaciones básicas). Para que ésta clase funcione efectivamente como un MaxHeap, le pasamos a su constructor una implementación de *compareTo* acorde. Con eso, ya tenemos el funcionamiento de MaxHeap: en el next del iterador hacemos un pollFirst, que devuelve el máximo y lo elimina del Heap. Para mantener el tamaño máximo de cutoff elementos, si se sobrepasa diche tope se hace un pollLast.

Esta última funcion es la que nos ha hecho decantarnos por esta clase respecto a otras como PriorityQueue, ya que al ser esta última de tamaño incremental, eliminar el elemento más pequeño sería un coste O(n).

Cabe mencionar que esto hace que solo se pueda iterar una vez sobre él, dado que el heap se va vaciando, lo cual es el comportamiento esperado para el heap de ranking.

Tanto para el ejercicio 2 como para el 3, hemos creado dos clases baseIndex y baseIndexBuilder que agrupan las funciones básicas.

Ejercicio 2

2.1) SerializedRAMIndex

Hemos usado un HashMap de Strings y PostingsList, que se cargan enteramente en memoria.

2.2) SerializedRAMIndexBuilder

Después de indexar todos los términos añadiéndolos en un LinkedHashMap de String y PostingsList (para mantener el orden de inserción), serializa diccionario y postings a sendos archivos, que el SerializedRAMIndex carga de nuevo en memoria.

Ejercicio 3

3.1) DiskIndexBuilder

Funciona de manera muy similar al anterior, únicamente difieren en que el índice de disco guarda el offset de cada término respecto a su PostingList del archivo de Postings.

3.2) DiskIndex

Este índice solo carga el diccionario (término y offset) y cuando se necesita acceder a la posting list, se accede de manera directa a ella mediante el método seek() de la clase RandomAccessFile. Por lo que solo necesitamos en memoria un HashMap de pares Término-Offset.

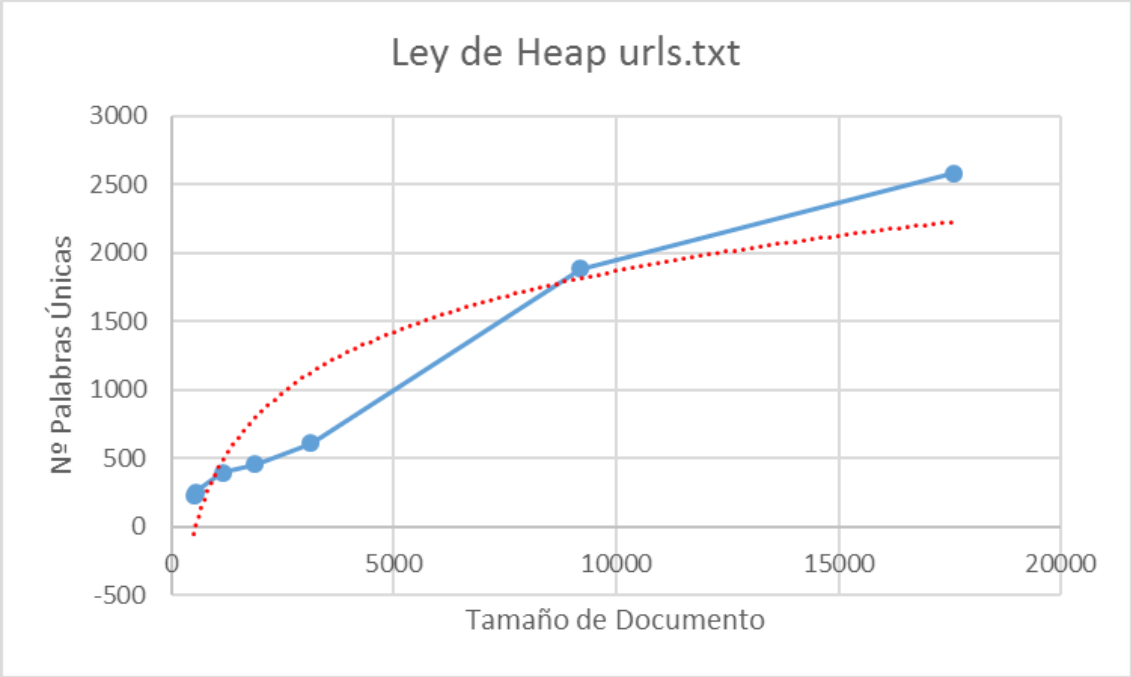
Esto es interesante ya que consigue una velocidad de consulta rápida, y a la vez una velocidad de carga y tamaño en memoria aceptables. También podría ser implementada con diversas soluciones híbridas, por ejemplo, una pequeña “caché” con las PostingsList más frecuentes almacenadas en memoria.

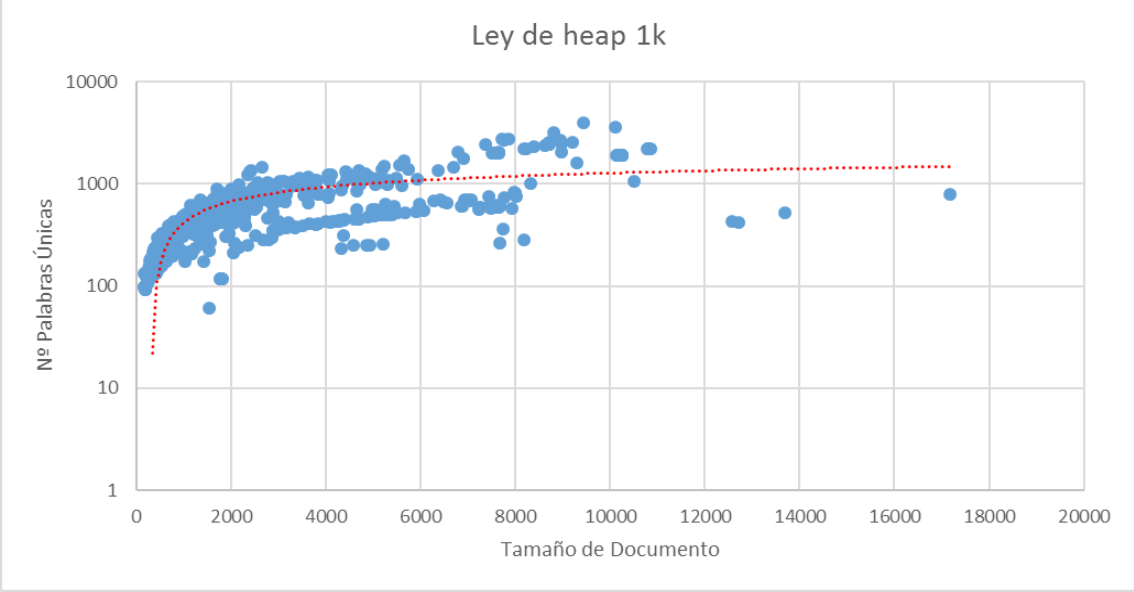
Ejercicio 5

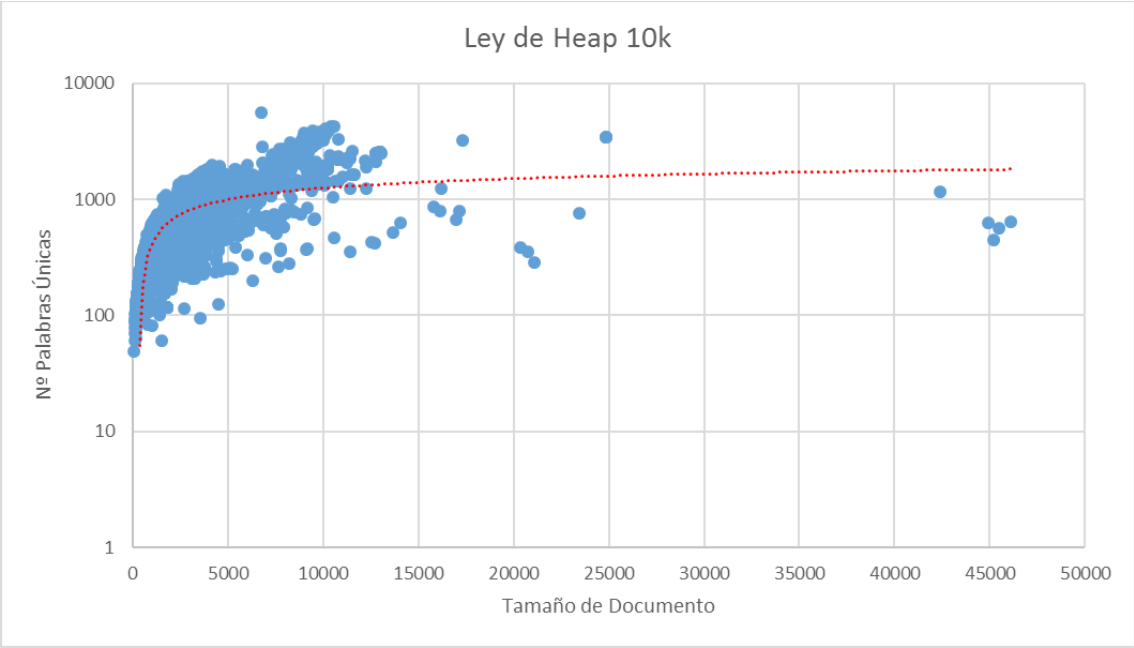
Para demostrar la ley de Heap, se han incluido algunas líneas de código en el método *indexText* de la clase *src/es/uam/eps/bmi/search/index/impl/BaseIndexBuilder.java*. En estas líneas se abre un fichero y se escribe sobre él.

Al igual que para ejecutar dos veces seguidas un determinado programa, antes de ejecutar el segundo, se deberá borrar el fichero, ya que si no todos los datos se escribirían a continuación de los datos correspondientes a la ejecución anterior.

Para comprobar la ley de de Heap, se ha ejecutado el test con el SerializedRAMIndexBuilder (línea 46 del *TestEngine.java*), y con tres ficheros diferentes: urls.txt, docs1k.zip y docs10k.zip. A continuación se muestran las gráficas correspondientes a las tres colecciones respectivamente:







Los puntos azules son los datos extraídos de los ficheros, y la línea roja es la tendencia de estos datos, que como bien se puede observar es logarítmica.

Como se puede comprobar todas ellas confirman la ley de Heap, es decir, que cuanto mayor es el tamaño de un documento, mayor es el número de palabras únicas que contiene, pero que para tamaños muy grandes, un mayor crecimiento solo implica unas pocas palabras diferentes más (el crecimiento se aplana).

Siempre hay alguna excepción, ya que puede haber algunos ficheros grandes que traten sobre un tema muy específico, y no tenga tantas palabras únicas, como otros algo más pequeños pero que traten sobre varios temas.

Diagrama de clases

Ver imagen *bmi-p2-01-ClassDiagram.gif* adjunta en la carpeta de la memoria.

Diferencias de rendimiento

Máquina: 2.5Ghz, dos cores, 4GB de Ram.

LuceneForwardIndex

Colección	Build time	Tamaño	Load time
1k	6s 730ms	4894K	18ms
10k	28s 129ms	37060K	21ms

LuceneIndex

Colección	Build time	Tamaño	Load time
1k	3s 893ms	1992K	10ms
10k	20s 408ms	13009K	19ms

SerializedRAMIndex

Colección	Build time	Tamaño	Máxima RAM build	Load time	Máxima RAM load
1k	27s 402ms	15164K	43,1%	1s 387ms	36%
10k	1m 34s 151ms	101705K	46,3%	9s 498ms	42,5%

DiskIndex

Colección	Build time	Tamaño	Máxima RAM build	Load time	Máxima RAM load
1k	23s 362ms	5618K	43,1%	57ms	34,5%
10k	2m 15s 325ms	41887K	48,2%	290ms	39%

En cuanto al tiempo de creación, tanto los índices de Ram y Disco son más lentos que los dos índices de Lucene, al tener que crear todas las listas de postings. Pero es interesante observar que el índice de Ram escala mejor ya que, mientras que los índices de Lucene tardan unoas 5 veces más en la colección de 10k respecto a la de 1k, el primero solo tarda unas 3 veces más. El de Disco no escala tan bien, al tener que añadir a la indexación el cálculo de todos los offsets.

Con respecto al tiempo de carga, el más lento con diferencias es el de Ram, ya que debe cargar desde dos archivos tanto el diccionario como todas las listas de postings. Los otros tres, mantienen unos tiempos bajos.

Finalmente, en lo que a búsquedas se refiere, es más rápido el índice de Ram que el de disco, al tener las listas de postings en memoria. Pero esto es con un coste muy alto tanto de carga, como de mantenimiento en memoria de un gran número de estructuras, lo cual es irrealizable para índices con millones de documentos.