Memoria de la práctica 2

Búsqueda y Minería de la Información

Jorge Cifuentes, Alejandro Martín, pareja 01

Ejercicios realizados

Ejercicio 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. 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. Heap de ranking: RankingIteratorImpl

Para el Heap de ranking, el cual es un MaxHeap, hemos usado la clase ya implementada de Lucene *priorityQueue.* Para que esta cola de prioridad funcione efectivamente como un MaxHeap, le pasamos a su constructor una implementación de *lessThan* acorde. Con eso, ya tenemos el funcionamiento de MaxHeap: en el next del iterador hacemos un pop, que devuelve el máximo y lo elimina del Heap.

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.

Ejercicio 2

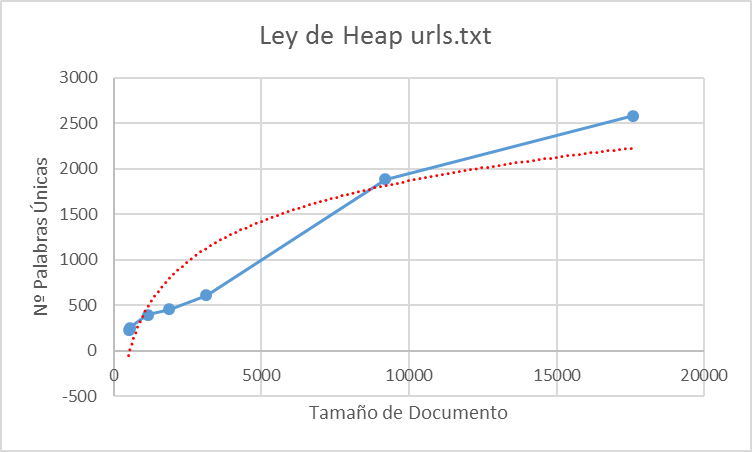
2.1) SerializedRAMIndex

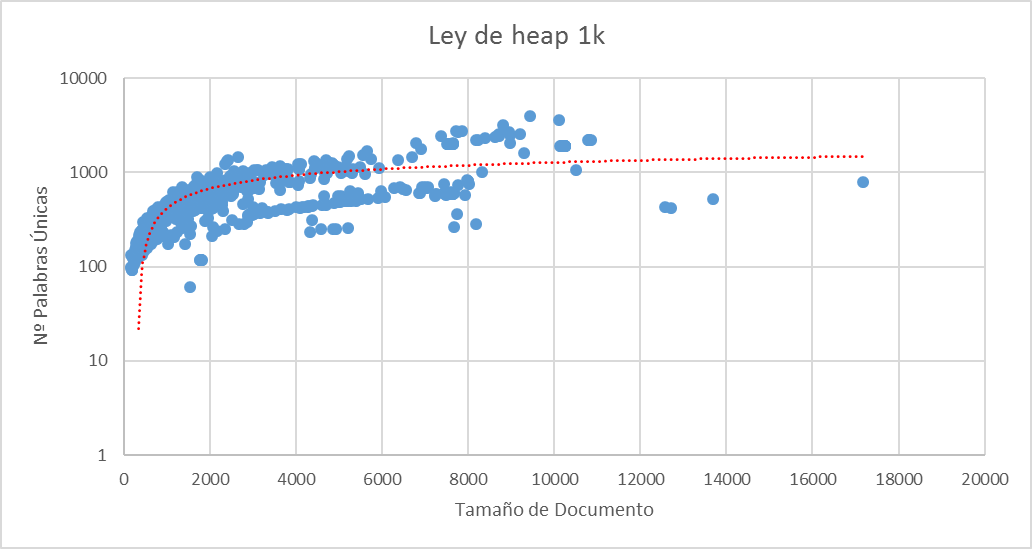
2.2) SerializeRAMIndexBuilder

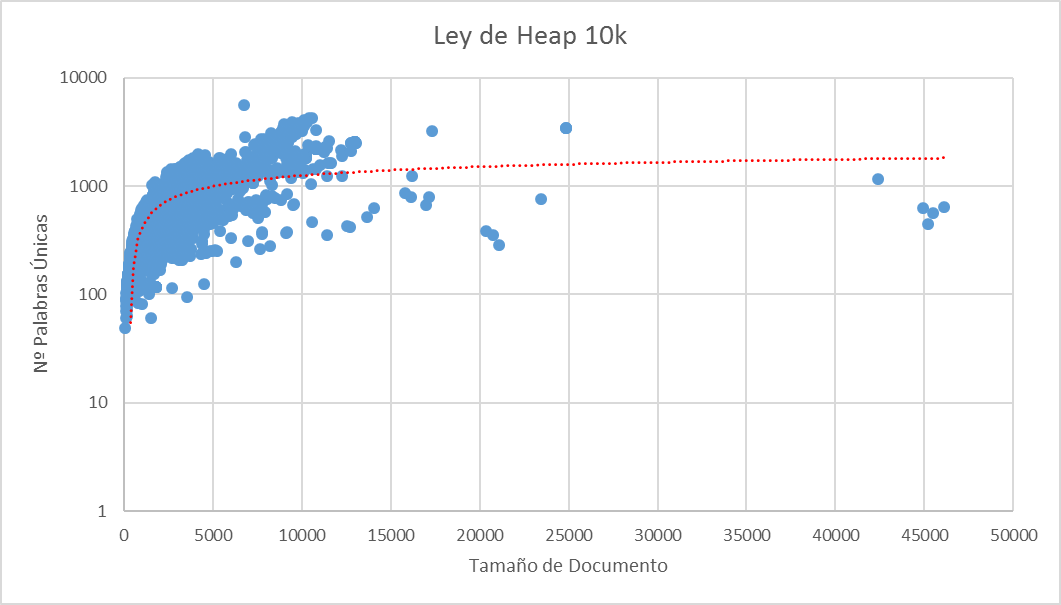
Esta es la clase importante.

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. 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.