MEMORIA PRÁCTICA 2

Miguel Laseca Espiga, Pablo Marcos Manchón

1. Implementación de un modelo vectorial eficiente

1.1. Método orientado a términos

Para la implementación de la búsqueda en la clase ***TermBasedVSMEngine*** se ha empleado un HashMap, indexado por docID y cuyos valores son las puntuaciones parciales de nuestra búsqueda. Tras iterar sobre las listas de postings de todos los términos de la consulta,actualizando las puntuaciones a los documentos correspondientes obtenidos en los postings, se utiliza normaliza cada puntuación usando la norma de cada documento y se almacenan en un heap de ranking para obtener de forma eficiente los resultados ordenados por puntuación. Las puntuaciones obtenidas no son divididas por la norma de la consulta, por tanto no se utiliza el coseno directamente, sino una puntuación proporcional al coseno con los mismo resultados. Esta puntuación es utilizada en el método orientado a documentos también.

1.2. Método orientado a documentos

En la implementación de la clase ***DocBasedVSMEngine*** hemos utilizado un heap para realizar la iteración cosecuencial entre las listas de postings. Durante la búsqueda almacenamos en el heap unas estructuras que contienen la listas de postings de los diferentes términos de la consulta y que son ordenados por el docID del documento correspondiente a la iteración de cada lista. Esta estructura nos permite iterar fácilmente sobre todas las listas a la vez.

1.3. Heap de Ranking

Se han implementado en es.uam.eps.bmi.search.ranking.impl las clases ***RankingDocImpl***, ***RankingImpl*** y ***RankingIteratorImpl***, utilizando la implementación de minheap de la clase PriorityQueue de Java.

2. Índice en RAM

En la implementación del índice en RAM hemos seguido una estructura básica: hemos guardado en un ArrayList los paths de los documentos, mientras que los términos y sus correspondientes listas de postings son almacenados de forma organizada en un HashMap, indexado por términos y con las listas de postings correspondientes.

Las listas de postings son una clase que hemos creado que implementa las interfaces ***PostingsList*** y ***Serializable***, que almacena una ArrayList con los postings y añade a la estructura básica de la lista los métodos declarados en ***PostingsList***.
El índice es almacenado utilizando la serialización de Java, guardando el HashMap con los postings en disco directamente, así como el array con los paths.
Al cargar el índice tan solo se leen los objetos serializados.

A continuación se muestran dos tablas con los rendimientos del índice en Ram y el índice de Lucene como referencia para comparar. No se han podido monitorear las cantidades de RAM usadas debido a un problema con el IDE. Las ejecuciones han sido realizadas en un MacBook Air (2015) con un procesador Intel Core i5 (1,6 GHz) y 4Gb de RAM 1600MHz DDR3. El límite fijado (por defecto en eclipse) en la memoria de la JVM es de 256Mb, el medidor de RAM utilizado oscila siempre en valores entre 100Mb y 210MB, excepto en los casos en los que se obtuvo OutOfMemoryError, sin embargo estas mediciones no son fiables debido a que incluyen la memoria de la JVM completa junto con el colector de basura, por tanto no es un reflejo directo de la memoria utilizada por el índice y no se han incluido las mediciones en las tablas.

	LuceneIndex				
	Construcción del Índice			**Carga del Índice**	
	T. Indexado	**Máx. RAM**	**Espacio Disco**	**T. Carga**	**Máx. RAM**
1K	5s 594ms		1993K	28ms	
10K	31s 433ms		12403K	196ms	
100K	6min 35s 535ms				

	SerializedRAMIndex				
	Construcción del Índice			**Carga del Índice**	
	T. Indexado	**Máx. RAM**	**Espacio Disco**	**T. Carga**	**Máx. RAM**
1K	16s 63ms		14739K	9s 618ms	
10K	1min 43s 958ms		95299K	59s 479ms	
100K	OutOfMemoryError	-	-	-	-

3. Índice en disco

Para la implementación del índice en disco se cargan completamente en memoria las listas de postings a partir de la cual se construye un HashMap indexado por términos y cuyos valores corresponden a los offsets en un fichero de disco de las posiciones de las listas de postings de cada término. Los paths y los postings son almacenados en disco sin utilizar la serialización de Java.
Al cargar el índice los postings no son cargados en RAM, y son leídos de disco cada vez que son consultados.
A continuación se muestran los resultados para la ejecución en las mismas condiciones que en el apartado 2.

	DiskIndex				
	Construcción del Índice			**Carga del Índice**	
	T. Indexado	**Máx. RAM**	**Espacio Disco**	**T. Carga**	**Máx. RAM**
1K	56s 933ms	-	9153K	152ms	-
10K	8min 42s 858ms	-	62212K	676ms	-
100K	OutOfMemoryError	-	-	-	-

4. Creacción fraccionada del índice

Se ha implementado un índice con una creación más eficiente, ***EfficientIndexBuilder*****,** de forma fraccionada.
Para limitar la RAM por simplificación se usa como límite un máximo de documentos cuyos postings puedan estar a la vez en RAM, por defecto fijado en 500, de esta manera nos aseguramos que todos los postings con un mismo docID se almacenarán en el mismo índice parcial, lo que posteriormente facilita la fusión de índices.
Para comenzar realizamos una serie de optimizaciones respecto a la versión anterior del índice en disco, evitando ahora almacenar en RAM todos los paths, escribiendolos directamente en disco sin almacenarlos en un array. Además se evita el empleo de HashMaps duplicados para almacenar los offsets, escribiendo directamente los offsets y

términos en disco. También se evita almacenar los offsets en los diccionarios auxiliares, pues estos solo serán accedidos durante el merge de forma ordenada.
Durante la creación del índice en primer lugar se generan diccionarios de términos parciales ordenados alfabéticamente para posibilitar la fusión de índices posterior, así como ficheros de postings parciales auxiliares, llamados *diccionario_auxiliar_x.dat* y *postings_auxiliar_x.dat*.

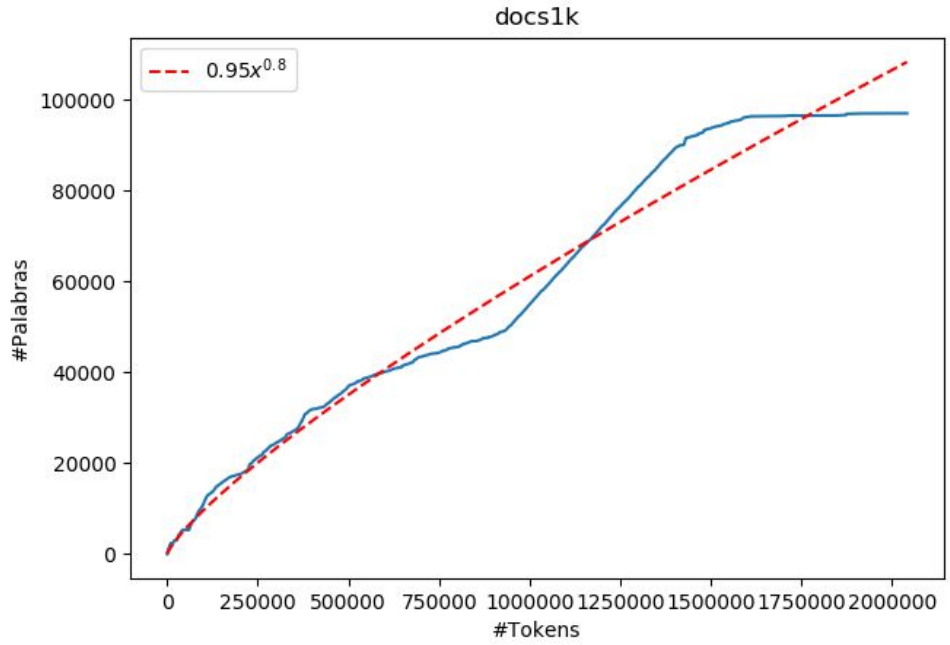
Una vez creados los índices parciales se procede a la fusión de índices parciales mediante un K-merge. Para realizar el K-merge se ha utilizado un heap, y utilizando una estructura auxiliar se realiza la iteración cosecuencial y la fusión de los diccionarios y postings en disco, sin tener en RAM más de una lista de postings por índice parcial a la vez.
Para abrir el índice se utiliza la clase ***DiskIndex*** implementada en el apartado anterior.
En la siguiente tabla no se han borrado los ficheros de disco con los índices parciales para reflejar el espacio utilizado durante la creación, pues si se eliminaran el espacio ocupado sería el mismo que en el caso de ***DiskIndex*** (el índice final).
En la ejecución se ha utilizado un tamaño máximo de 500 documentos en RAM, utilizándose en 2 índices parciales en 1K, 20 en 10K y 200 en 100K.

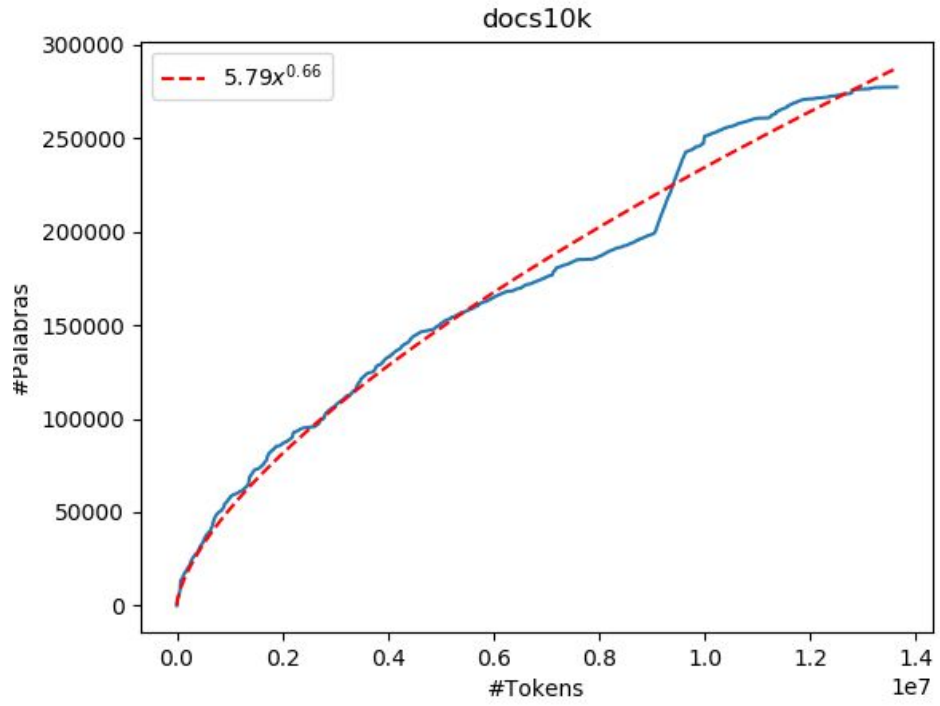
Para la ejecución con el conjunto 100k sería necesario

	EfficientIndex				
	Construcción del Índice			**Carga del Índice**	
	T. Indexado	**Máx. RAM**	**Espacio Disco**	**T. Carga**	**Máx. RAM**
1K	2min 11s 234ms	160MB	18067K	202ms	211MB
10K	19min 36s 873ms	180MB	135804K	1s 337ms	210MB
100K	> 2h				

5. Ley de Heap

En las siguientes figuras se muestra el resultado de representar el número de tokens de un documento frente al número de palabras descubiertas durante la construcción del índice. Se ha ajustado una ley de potencias para visualizar el comportamiento de la ley de heap.





La información ha sido extraída modificando el ***SerializedRAMIndexBuilder.java*** para ir calculando las nuevas apariciones, aunque en la versión entregada no se encuentra esta modificación. Las gráficas han sido generadas con el script *heap_law.py.*

Anexo: Diagrama de clases implementadas