Práctica 2 BMI

Memoria

10/03/2020

Adrián Rubio Pintado
Jorge Muñoz Aguado

A continuación, se explicarán las tareas realizadas.

Tarea 1 - Implementación de un modelo vectorial eficiente.

Para las siguientes tareas se hará el uso del método *tf-idf*, calculando la puntuación final con el coseno.

Tarea 1.1 - Método orientado a términos.

Este método recorre término a término, de la *query*, cada lista de *postings*. En un *HashMap* va guardando las puntuaciones *tf-idf* acumuladas. Tras recorrer todos los términos se extraerán las puntuaciones de los documentos guardados y se calculará el coseno, incluyendo el resultado en el `RankingImpl`.

Tarea 1.2 - Método orientado a documentos.

Este método obtiene por cada termino de la query, la lista de postings. Crea un minheap de postings de tamaño numero de terminos de la query. Para ello utiliza PostingMinHeap para cada posting. Una extension de la clase Posting que guarda tambien el termino de origen, necesario para el algoritmo. Y va insertando y sacando del heap cada posting hasta que finalmente obtiene para cada documento un score. A medida que va obteniendo un score, lo va insertando en un MaxHeap de ranking, de tamaño cutoff, ordenando por el score del docID.

Tarea 1.3 - Heap de ránking.

El heap se ha imlementado en la clase rankingImpl como PriorityQueue<SearchRankingDoc> a nivel de implementacion, aprovechando las carateristicas de las colas de prioridad en Java. Dado que nativamente no se puede controlar el tamaño del las PriorityQueue, controlamos el tamaño del heao con la la funcion de add, que añade un elemento al heap.

Tarea 2 - Índice en RAM.

Tarea 2.1 - Estructura de índice.

Para el diccionario del índice hemos hecho uso de un *HashMap*, que permite acceder a el contenido de los términos de manera eficiente. Cada término tendrá asociado un *RAMPostingsList*, que extiende de *PostingsList*.

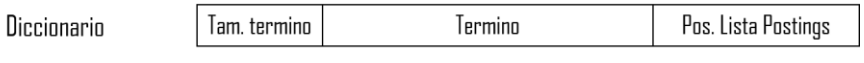
Tarea 2.2 - Construcción del índice.

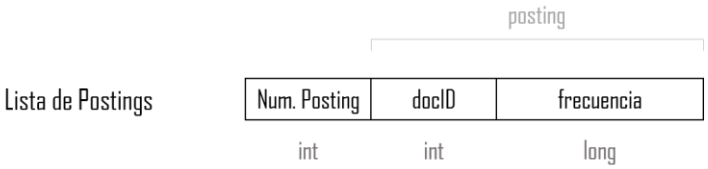
Para cada término obtenemos su *lista de postings* y recorremos esta lista, acumulando las frecuencias para cada documento en un *hashMap*. Una vez recorridos

Tarea 3 - Índice en Disco.

La construcción del índice en disco funciona igual que la construcción en RAM. Las diferencias que encontramos en este modelo son a la hora de guardar y leer los datos.

Por una parte, se guardará el diccionario en el correspondiente archivo. Con tal de minimizar el espacio que ocupe el mismo hemos decidido guardar los datos byte a byte. Para cada término en el diccionario guardamos el tamaño de término y el término, codificado en bytes. Por último, guardamos la posición de la *lista de postings* de ese término en el fichero de *diccionario*, del cual hablaremos más adelante.





Las *listas de postings* se almacenan en otro archivo. A cada lista se accederá a partir de la dirección en *bytes* almacenada en el diccionario. La codificación de las *listas de postings* comienza con el número de *postings* que tiene esa lista en concreto. Cada *posting* contendrá un *docID*, con el tamaño de un entero, y la frecuencia del término en el documento, codificado con un *long*. Podría hacerse una modificación sobre estos índices para en vez de guardar los bytes correspondientes a un entero, hacer uso de la codificación de *byte variable*.

Tarea 4 - Índice en Eficiente.

El índice eficiente debe generar un índice sin que sobrepase el límite de memoria *RAM* impuesto en el constructor de esta clase. Posteriormente el índice será leído por el *DiskIndex*. Debido a que el tamaño de memoria es una medida ambigua, pondrá un límite en el número de *postings* que puede almacenar en RAM.

Nota: para tener en cuenta el tamaño en RAM habría que ejecutar la siguiente función

```
runtime.totalMemory();
```

Para facilitar la implementación se ha creado la clase `es.uam.eps.bmi.search.index.structure.impl.PartPostingsFile` la cual se encargará de guardar y extraer los fragmentos del índice en un fichero.

Para el almacenamiento de los fragmentos en los ficheros, se hará uso de un mecanismo parecido al ya usado en disco, pero reduciendo el espacio a un solo fichero. Donde también los términos se repetirán como si de un diccionario se tratara.

Datos del coste y rendimiento.

A la hora de comparar los datos de rendimiento es importante tener en cuenta que depende del hardware de la computadora se obtendrán unos resultados u otros. En este caso hemos hecho uso de un ordenador portátil con 2 núcleos y 4 hilos de procesamiento, con una frecuencia de reloj 2.40GHz. Además, el ordenador contaba con 8 GB de memoria RAM y un disco duro de estado sólido (*SSD*).

Las cifras también son aproximadas ya que Java ejecuta el *garbage collector* en momentos determinados. Y el uso en RAM puede variar.

Índice construido en RAM (Tarea 2)

	Construcción del índice			Carga del índice	
	Tiempo de Indexado	Consumo máx. RAM	Espacio en disco	Tiempo de carga	Consumo máx. RAM
1K	24s 349ms	850MB	14014K	12s 640ms	1000MB
10K	2min 23s 407ms	2250MB	93599K	1min 18s 481ms	2800MB
100K	25min 33s 767ms	4800MB	946850K	12min 50s 200ms	6100MB

Índice construido en Disco (Tarea 3)

	Construcción del índice			Carga del índice	
	Tiempo de Indexado	Consumo máx. RAM	Espacio en disco	Tiempo de carga	Consumo máx. RAM
1K	25s 9ms	1100MB	9506K	1s 28ms	1100MB
10K	2min 5s 107ms	2800MB	63340K	2s 921ms	2800MB
100K	18min 44s 837ms	5100MB	638829K	14s 671ms	5800MB

Índice construido por fragmentos (Tarea 4) *

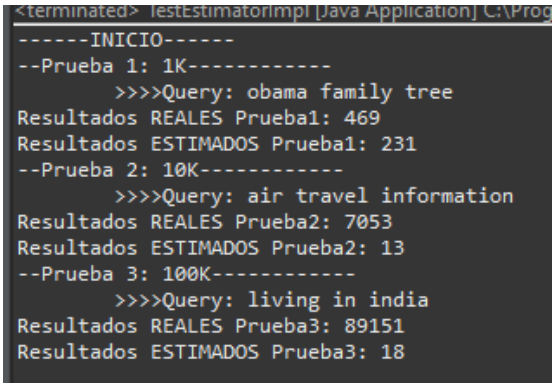
	Construcción del índice			Carga del índice	
	Tiempo de Indexado	Consumo máx. RAM	Espacio en disco	Tiempo de carga	Consumo máx. RAM
1K	1min 357ms	270MB	9506K	1s 28ms	1100MB
10K	5min 34s 318ms	300MB	63340K	2s 921ms	2800MB
100K	54min 15s 144ms	350MB	638829K	14s 671ms	5800MB

* El consumo en RAM se corresponde con el valor fijo de 10000 postings a la vez en la memoria.

Tarea 5 - Estimación del número de resultados.

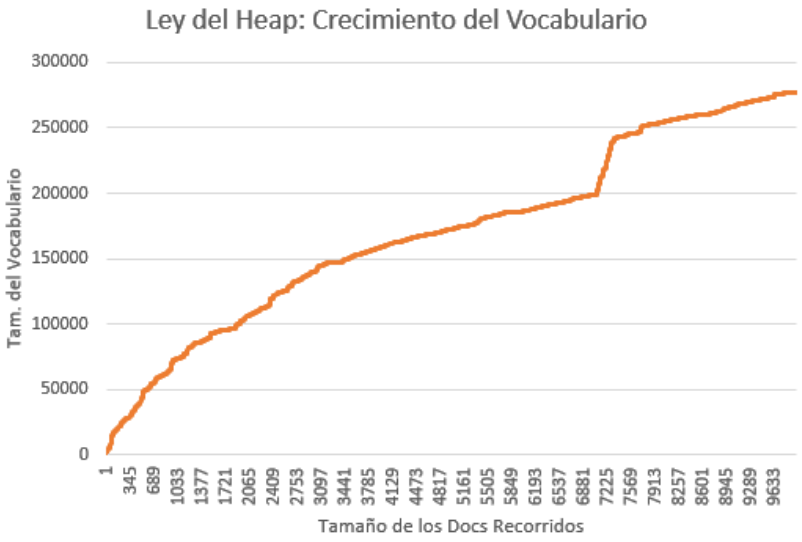
Para este apartado hemos implementado la clase TestEstimatorImpl, con la que podemos comprobar la implementación de la interfaz Estimator en la clase EstimatorImpl.

Las estimaciones de resultados se han hecho como la probabilidad de aparición de los terminos dentro de la colección. Suponiendo independencia de probabilidad de los terminos y estimando la probabilidad como le frecuencia del término dentro de la colección entre el tamaño de la colección.



Observamos en la imagen el resultado de la ejecución de la prueba. Observamos que para colecciones pequeñas, como la de 1k, el resultado no dista excesivamente. En cambio, para colecciones grandes, al ser el tamaño de la colección tan grande, las estimaciones resultan ineficaces.

Tarea 6 – Ley de Heap



En el ejemplo de la gráfica, hemos usado la colección de 10k y el IndexBuilder SerializedRAMIndexBuilder. Hemos modificado los métodos de build() e indexText() para que, a medida que va aumentando el texto procesado, se contabilice por cada documento cuantas terminos nuevos han sido añadidos a la colección.

De manera empírica observando la gráfica, observamos que el "descubrimiento" de nuevos terminos no es lineal respecto al a longitud de los documentos. Presenta un comportamiento, por el cual, cada vez le "cuesta" más obtener terminos diferentes.

ANEXO – Diagrama de clases.

En el archivo diagrama_clases.jpg adjunto en el zip se puede ver el diagrama de clases de esta práctica, con los métodos,atributos y relaciones entre clases más representativos, de modo que con u vistazo se pueda entender la lógica del diseño del proyecto.