13-3-2018

Practica 2

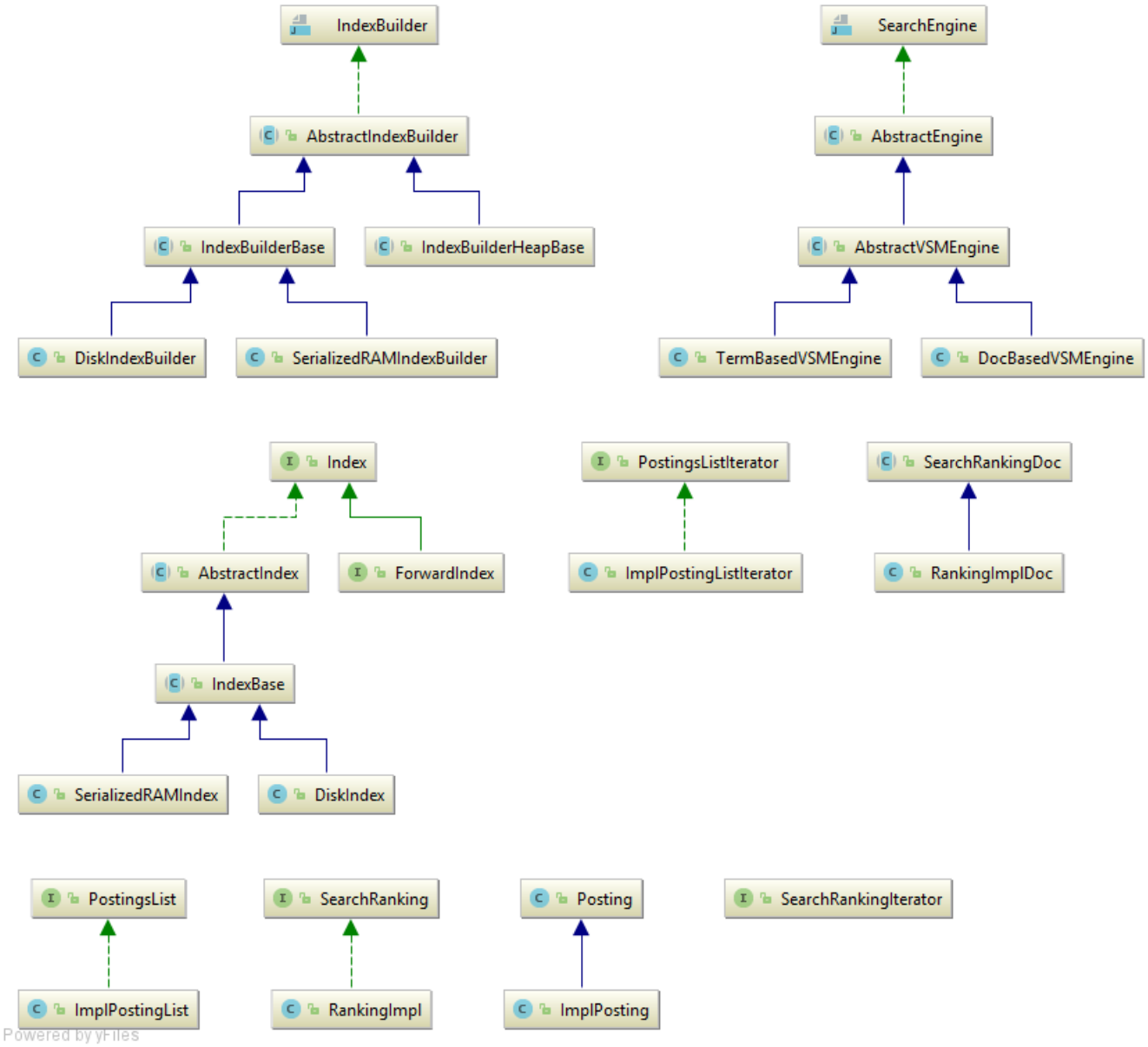
Búsqueda y minería de información

Oscar Garcia de Lara Parreño
Jorge Gómez Conde

CONTENIDO

TRABAJO REALIZADO

Hemos realizado completamente los ejercicios 1, 2, 3 y 5. Adjuntamos en la memoria un diagrama de clase compacto y otro en el zip con métodos y atributos, ya que consideramos que ponerlo como imagen en un pdf seria poco práctico.



INDICES

Las pruebas de rendimiento han sido realizadas sobre un ordenador del laboratorio 6B de la facultad EPS, con las siguientes características:

- CPU: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz de 8 núcleos.
- Memoria RAM: 16 GB
- SO: Ubuntu 16.04
- IDE: Netbeans 8.2

En las pruebas de cada apartado se han realizado ejecutando únicamente los test que corresponden en el apartado. De esta manera evitamos medir también valores residuales que pudieran haber aumentado el margen de error del uso de memoria RAM o mayor tiempo de ejecución.

BASE

Vamos a empezar la sección explicando nuestras dos clases abstractas que implementan nuestras versiones, esto lo hacemos para evitar duplicar código.

INDEXBUILDERBASE

En esta clase extendemos AbstractBuilderIndex. Como diccionario implentamos un Map<String,ImplListPosting>, esta lista básicamente es una List<Posting>.

También tenemos el método build y el indexText y uno abstracto saveIndex() que implementamos en los Builder correspondientes.

INDEXBASE<K>

Extendemos la clase AbstractIndex, y le ponemos un genérico, esto nos sirve para que el Map<String,K> nos sirva para las dos implementaciones, con ImplListPosting e Integer(offset).

Implementamos la parte genérica de cargar el indice que son las rutas, y las funciones de Index salvo getPostings() y una abstracta loadIndex(String ruta)

RAM

SERIALIZEDRAMINDEXBUILDER

En esta versión transmitimos el diccionario por parámetro a través del constructor del SerializedRAMIndex. También guarda el diccionario serializado en la ruta indicada por Config.

SERIALIZEDRAMINDEX

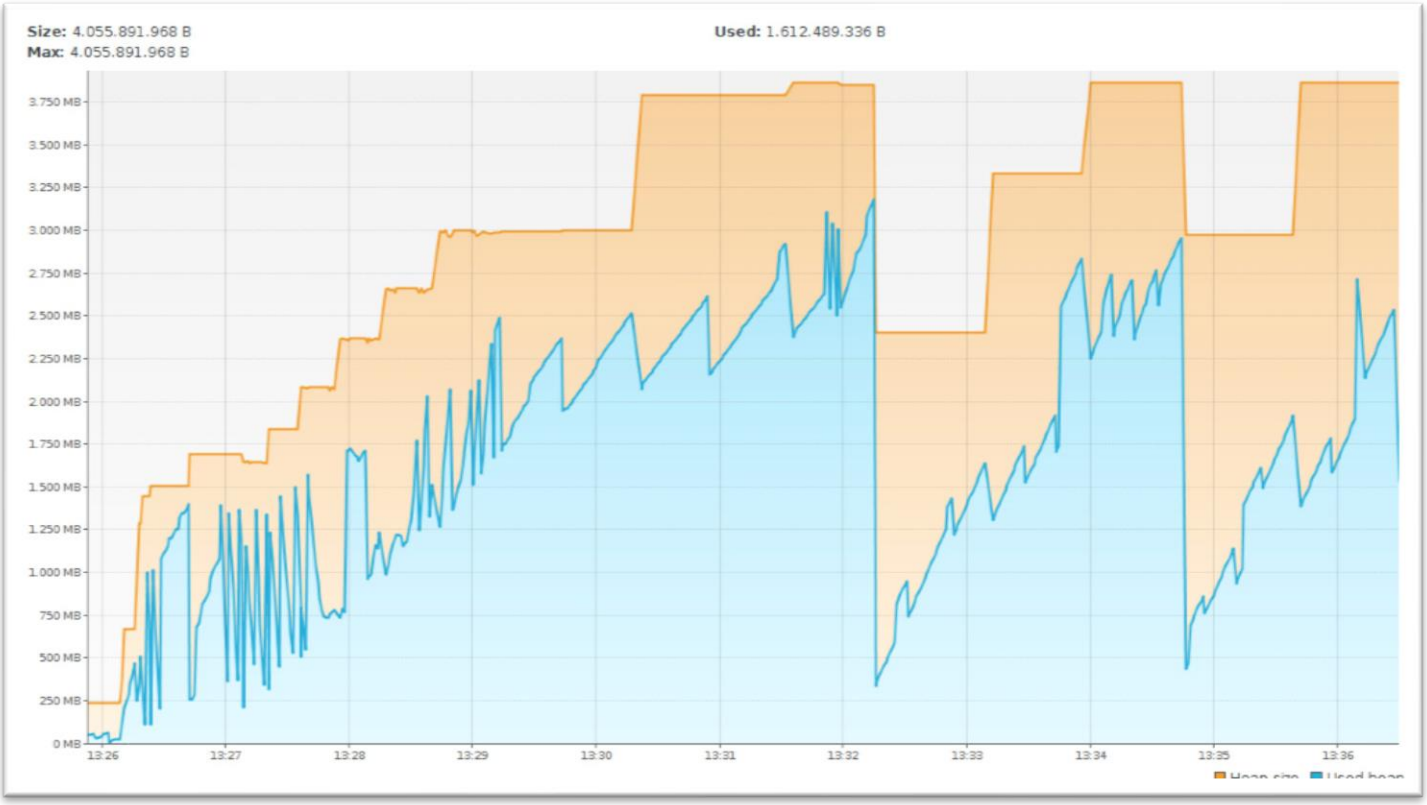
El índice permite inicializar su diccionario a través del constructor o bien desde loadIndex() introduciendo una ruta por parámetro. La función getPostings() accede directamente al diccionario.

RENDIMIENTO

El fichero que contiene la salida de TestEngine es log_RAM_Linux.log. El tiempo total de duración de la prueba fue de 10 minutos y 39 segundos. El rendimiento de SerializedRAM esta resumido en la siguiente tabla.

	Construcción del índice			Carga del índice	
	Tiempo de indexado	C. max de RAM	Espacio en disco	Tiempo de carga	C. max de RAM
1K	4s 746ms	501 MB	13973 KB	2s 55ms	542 MB
10K	29s 699ms	1470 MB	93205 KB	12s 48ms	1468 MB
100K	5min 984ms	3117 MB	942699 KB	2min 7s 326ms	2976 MB

Gráfica de uso de memoria RAM durante la ejecución de la prueba.



DISK

DISKINDEXBUILDER

Usamos el diccionario de IndexBuilderBase, en diferencia a la versión de RAM en getCoreIndex() no generamos el índice pasándole un map ya que obligaría a crearlo cuando lo guardamos en el fichero y en una prueba de rendimiento no hubo mejoría.

DISKINDEX

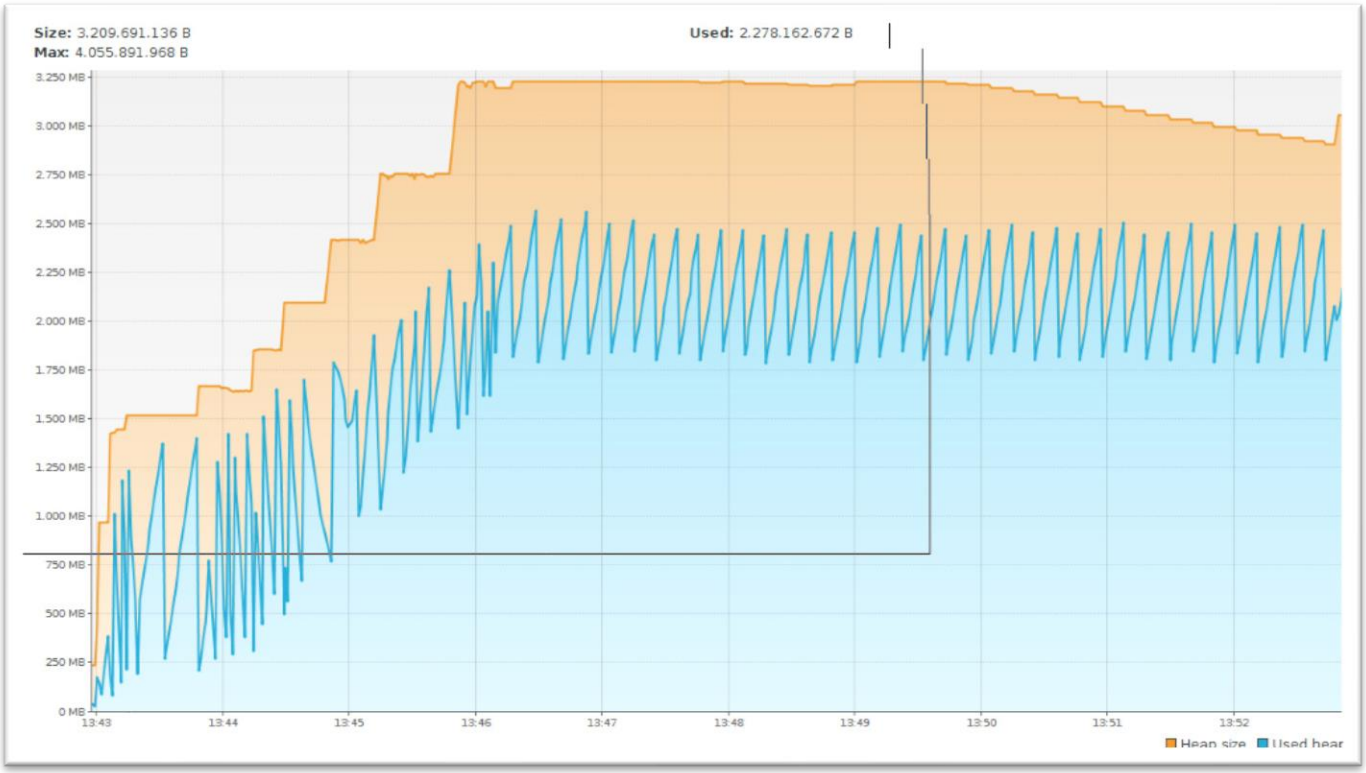
Usamos el diccionario de IndexBase<Integer>, lo único que mantenemos es el fichero para leer la lista de posting.

RENDIMIENTO

El fichero que contiene la salida de TestEngine es log_Disk_Linux.log. La duración total de la prueba fue de 9 minutos y 57 segundos. El rendimiento de Disk esta resumido en la siguiente tabla.

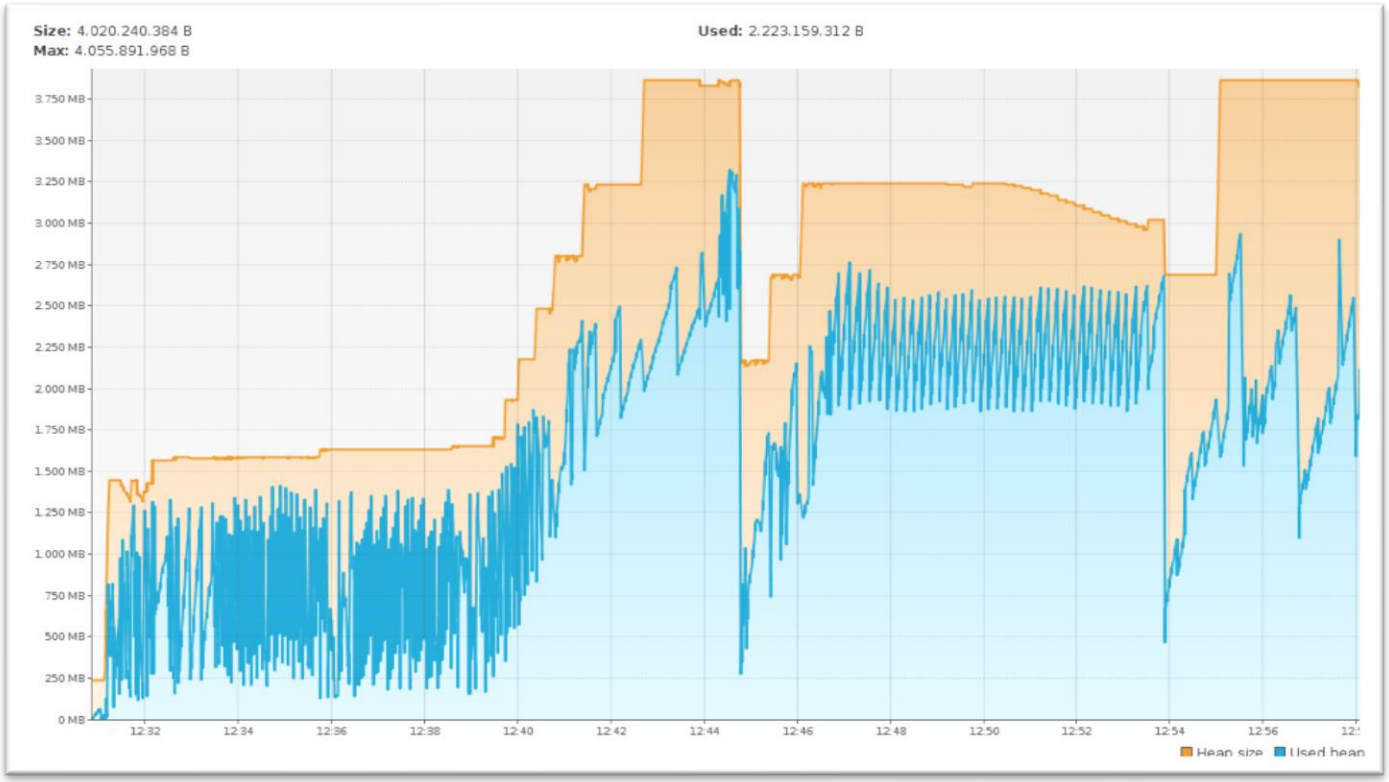
	Construcción del índice			Carga del índice	
	Tiempo de indexado	C. max de RAM	Espacio en disco	Tiempo de carga	C. max de RAM
1K	6s 415ms	1230 MB	5153 KB	42ms	1230 MB
10K	47s 456ms	1470 MB	37963 KB	78ms	816 MB
100K	8min 54s 862ms	2695 MB	427544 KB	450ms	2139 MB

Grafica de uso de memoria RAM durante la ejecución de la prueba.



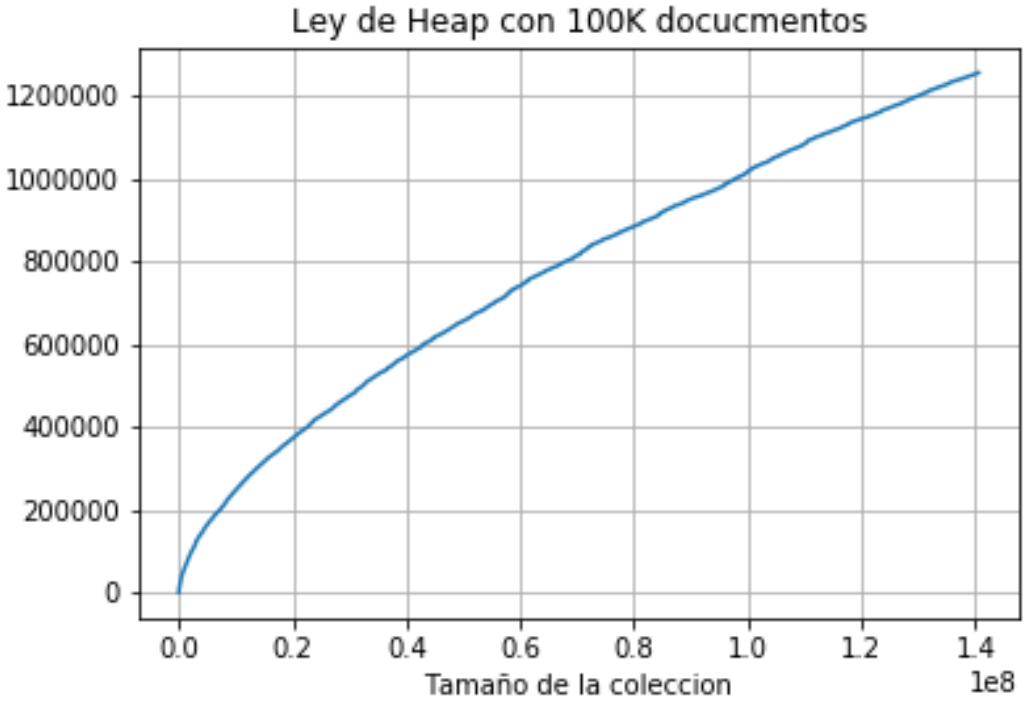
RENDIMIENTO TOTAL DE LA PRUEBA

Pese haber realizado las pruebas individualmente, incluimos también el resultado obtenido de realizar todas las pruebas de manera conjunta, en total 27 minutos y 19 segundos. Los datos obtenidos están log_Linux.log y la gráfica se muestra a continuación:



LEY DE HEAP

Para poder calcular la ley, hemos creado una clase llamada **IndexBuilderHeapBase,** para ir guardando el tamaño consecutivo de los ficheros y el tamaño del conjunto de términos, y así evitar que las pruebas de rendimientos se vean interferidas. Únicamente hay que cambiar la que extiende el índice actual por esta.



En la grafica se puede observar que con 100K documentos se cumple la Ley de Heap, ya que el crecimiento del número de palabras nuevas no crece linealmente al numero total de palabras en los documentos.

ANALISIS DE RESULTADOS

- Respecto al uso de RAM entre ambos índices, vemos que Disk utiliza menos RAM, la mantiene de manera más constante, mientras que SerializedRAM utiliza más RAM.

- En tiempo de construcción se nota que en RAM serializamos directamente el diccionario, y en Disk vamos construyendo dos ficheros para almacenarlos, pero al serializar con las propias funciones de java hace que el espacio usado sea mayor.

- Razón a la velocidad de carga del índice es mejor la de Disk, por el mismo motivo comentado anteriormente.

- Si entramos a medir el rendimiento en búsqueda, como es normal el índice sobre RAM da mejores resultados, pero esto es difícilmente evaluable ya que podríamos considerar otros factores como el coste de RAM sobre un SSD por ejemplo.