3-4-2018

Practica 3

Búsqueda y minería de información

Oscar García de Lara Parreño
Jorge Gómez Conde

CONTENIDO

TRABAJO REALIZADO

Hemos realizado completamente los ejercicios 1, 2, 3, 4 y 5. Mas adelanta se detallan las implementaciones utilizadas en la práctica.

INDICE POSICIONAL

En la implementación del índice posicional, hemos decidido extender la clase SerializedRAM en ambas partes, Index e IndexBuilder. A continuación, contamos los detalles en cada implementación.

POSITIONALINDEX

Esta clase es muy parecida a la clase SerializedRAMIndex que extiende. Simplemente nos preocupamos de que cargue los directorios y las normas de los documentos. También nos preocupamos de que el diccionario generado sea del tipo PositionalDictionary, que se explicara más adelante.

POSITIONALINDEXBUILDER

Es muy parecido al índice del modelo vectorial. Las diferencias residen en el método indextText() y el uso de un diccionario especifico, llamado PositionalDictionary.

POSITIONALDICTINARY

Esta nueva clase nos facilita la gestión de la creación y manejo de la información de cada documento. Aquí es donde guardamos la posición de cada termino en cada documento. Implementa la interfaz EditableDictionary implementando todos sus métodos.

Para apoyarse y guardar la información, utiliza un Map con un par de claves, valor. En este caso la clave es el termino y el valor es una nueva clase creada por nosotros mismos, PositionalPostingListImpl

POSITIONALPOSTINGLISTIMPL

Esta clase implementa la interfaz Serializable y la interfaz PostingList. Se encarga de gestionar la lista de documentos y las posiciones que mantiene el termino en el mismo documento. Para gestionar la información de cada documento creamos la clase PositionalPostingImpl.

POSITIONALPOSTINGIMPL

Esta clase extiende la clase PositioanlPosting. Esta sencilla clase es la encargada de gestionar la lista de posiciones dentro de un documento.

PAGERANK

En la implementación de un motor PageRank hemos creado 3 nuevas clases nuevas.

PAGERANK

Su función es dar acceso al índice (parte de Index) y generar a su vez las puntuaciones de los documentos y el ranking (parte de Engine).

La clase tienen en cuenta los sumideros que pudiera haber en el grafo, la función encargada de esto es sink(List<Double> scores).

PAGERANKBUILDER

Encargada de generar el índice. No extiende ninguna de las clases ni implementa ninguna interfaz de las dadas por el enunciado.

Esta clase simplemente lee el fichero que es indicado por parámetro en el constructor y comienza a añadir los pares de documentos al índice como corresponde. Si ambos documentos son nuevos, debemos crear su propio PageRankPosting antes de añadirlos al diccionario.

PAGERANKBUILDER

Guarda la información necesaria para evaluar la puntuación de un documento. Utiliza un entero para guardar el ID del documento y un String para guardar su nombre. También contiene dos listas de enteros. En estas listas se guardarán los ID de los documentos que apuntan y a los que apunta el documento en cuestión.

CRAWLER

Para ejecutar el Crawler basta con ejecutar TestCrawler que está en el paquete de test, en él se puede elegir dónde está la semilla y el número de documentos.

Si se quiere usar aparte la única limitación es que el indexBuilder se le tiene que pasar el método init antes y si se quiere guardar el contenido el método close, el motivo que no se hace dentro es por la limitación de parámetros del enunciado que no permite guardar.

El fichero de semillas lo incluimos y contiene:

```
https://www.ascodevida.com/
http://www.marca.com/
```

CONSIDERACIONES DE DISEÑO

COLA DE PRIORIDAD

Hemos creado una clase interna que nos permite establecer a cada ruta una prioridad, esta prioridad simula los segundos para que la pagina sea procesada, obviamente no son segundos, pero a menor sean antes se saca de la cola de prioridad.

Las páginas de la semilla se insertan con el valor 1 para que sean siempre las primeras, después cuando se ha sacado y se vuelve a meter, el tiempo es al azar entre 1 a 60, que simula el tiempo recomendado en teoría. También cuando leemos los enlaces de la página para insertarlos simulamos entre 1 y la prioridad de la página, para así priorizar el nuevo contenido.

ROBOTS.TXT

Hemos tenido en cuenta el robots.txt, para ello intentamos conectarnos y si tiene lo procesamos, en el caso contrario decidimos continuar la ejecución. Vamos almacenando las URLs base en forma de "cache" para evitar leerlo más de una vez y de paso evitar pedirlo en páginas internas.

De él tenemos en cuanta los Disallow para evitar entrarlos en la cola de prioridad y los Allow que añadimos. No procesamos Crawl-delay ya que muy pocos lo tienen y los sitemaps.

DETECCIÓN DE DUPLICADOS

Al pedir una arquitectura sencilla, hemos descartado incluir esta parte.

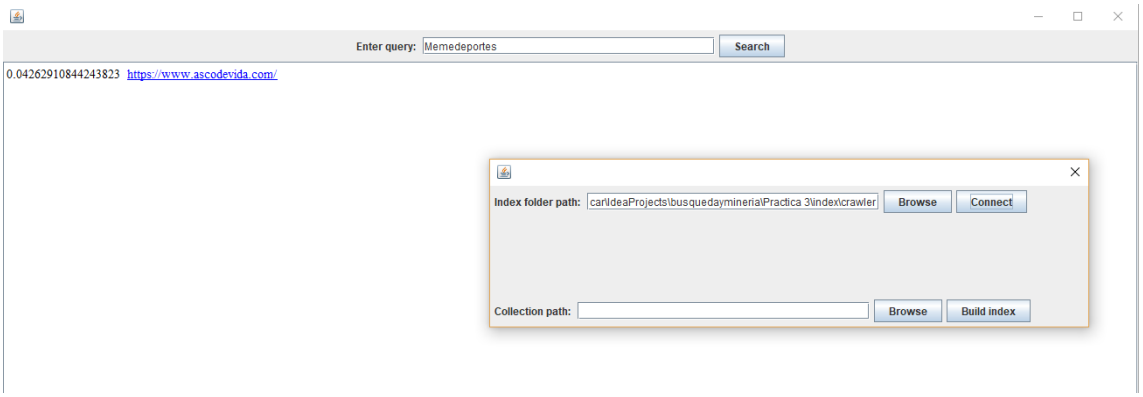
PRUEBAS

Hemos medido el tiempo:

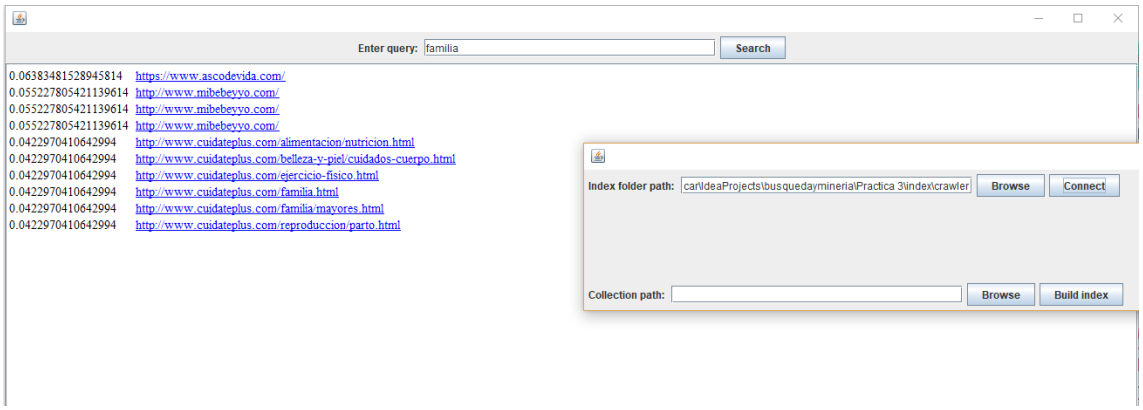
- 10 documentos: ha tardado 7s 71ms
- 100 documentos: ha tardado 49s 368ms
- 1000 documentos: ha tardado 8min 40s 200ms

Este tiempo es indicativo, ya que hasta que no se procesan las maxDoc distintas no se termina, ya que como tenemos aleatoriedad a la hora de la prioridad puede darse el caso que se procesen varias veces alguna antes que se lleguen al maxDoc.

Para ejecutar las pruebas sobre el índice hemos usado la interfaz gráfica que se proporciona con el cambio de que el índice sea SerializedRAMIndex.



Se puede apreciar dando al link que esa página lleva a otra con ese nombre.



Esta se puede ver que hemos terminado en páginas muy diversas.

BÚSQUEDA HIBRIDA

La clase encargada de la combinación de resultados es CombinedEngine.java. Almacena un array con la ponderación y los motores de búsqueda a combinar.

En el método search() realizada la llamada a cada uno de los motores contenidos en el array en SearchEngine[] searchArray, para guardar los resultados normalizados con Min-Max y combinándolos linealmente en el Map ranking.

Cabe añadir que utilizamos una la clase interna DocumentMapImpl, que implementa DocumentMap, para normalizar el docId entre disantos índices.

ANALISIS DE RESULTADOS

En el resultado mostrado por TestEngine no encontramos diferencias en los apartados de creación y búsqueda no difieren con los resultados del fichero test_output.txt.

A excepción de la creación de los índices en URLCollection encontramos ligereas diferencias en las listas de postings devueltas. En el LucenePosting de test_output aparecen menos frecuencia de la palabra “channel” que nuestro test (concretamente 2 unidades más). Puede que el motivo de esto es que Lucene realice steamming internamente.

Vemos que el resultado de PositionalIndex no es igual al de test_engine.txt, la frecuencia es adecuada pero no las posiciones, no somos capaces de concluir sobre este fenómeno, pues en la búsqueda obtenemos los datos correctos.