Universidad Autónoma de Madrid

Búsqueda y minería de información.

Práctica 2 - Implementación de un motor de búsqueda

Guillermo Ruiz Álvarez
Enrique Cabrerizo Fernández

15 de marzo de 2016

Índice

1. Ejercicio 1: Creación de Índices.

En este apartado se expondrán las decisiones tomadas para tratar de construir un índice en disco que sea capaz de indexar la colección de 100.000 documentos en un tiempo y con un consumo de RAM aceptables.

1.1. Algoritmo empleado.

Para la creación del índice se han tenido en cuenta diversas implementaciones analizadas por la universidad de Stanford, concretamente las opciones Blocked sort-based indexing (BSBI [1]) y Single-pass in-memory indexing (SPIMI[2]). Finalmente se decidió adoptar el método SPIMI por ser más eficiente, $O(T)$ frente a $O(Tlog(T))$, donde T es el número total de tokens a procesar y por no necesitar una estructura para asociar términos a ids únicos.
En la figura 1 se puede ver el pseudo-código básico del algoritmo:

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Figura 1: Pseudo-código SPIMI

Nuestra implementación realiza una pequeña variación de este algoritmo. Cuando se quiere construir un índice, se ejecuta el método *build* de la clase *BasicIndex* con los parámetros apropiados. Dicha clase parsea cada documento a indexar y finalmente ejecuta el método *add*, de la clase *IndexWriter*. Éste método es el que ejecuta el pseudocódigo descrito arriba con una salvedad: solamente se comprueba si se ha excedido el tamaño de bloque al terminar de añadir todo el documento, es decir, un documento nunca se guardará en dos bloques distintos. Esto impide que un *posting* de un término sea escrito a disco cuando aún es posible que dicho término vuelva a aparecer en el documento, lo que implicaría crear un nuevo *posting* para el mismo término y mismo documento. Esto más adelante obligaría en la fase de *merge* a fusionar *postings*, además de las propias listas de *postings*.

[1] BSBI:http://nlp.stanford.edu/IR-book/html/htmledition/blocked-sort-based-indexing-1.html#4947

[2] SPIMI:http://nlp.stanford.edu/IR-book/html/htmledition/single-pass-in-memory-indexing-1.html#5079

Teniendo en cuenta que los archivos a indexar ocupan menos de 100kB (antes de parsear HTML) y que nuestros bloques por defecto (archivos escritos en disco), son de 10 MB, se ha considerado aceptable que el tamaño de bloque sea orientativo, pudiendo oscilar hasta aproximadamente los 10.10 MB a cambio de ahorrar tiempo en la fase de *merge*. De esta forma, solamente tendremos que realizar fusión de listas de *postings*, algo trivial con las estructuras de datos que utilizamos, descritas en la sección 1.2.

Al terminar de añadir documentos al índice se fusionan todos los índices parciales creados en cada bloque hasta obtener un único índice. Para ello se han usado hilos que hacen *merge* de los bloques 2 a 2, realizando un total de $\lceil log_2(k) \rceil$ *merges* en total, siendo k el número de bloques generados.

1.2. Estructuras de datos.

1.2.1. Creación del índice.

Durante la creación del índice mantenemos las siguientes estructuras de datos:

- Diccionario de bloque: Tabla hash con claves ordenadas (*TreeMap*), en la que se guardan los términos como clave y sus listas de *postings* como valor.
- Diccionario de Documentos: Tabla hash con asignaciones entre el nombre del documento (valor) a indexar y un id numérico único asignado al agregarlo al índice (clave). Esto nos permitirá guardar en las listas de *postings* los ids numéricos y no los nombres de documentos ahorrando espacio. La tabla nos permitirá recuperar posteriormente el nombre del documento en las búsquedas.

1.2.2. Merge.

Durante el último *merge* que realiza el algoritmo, se guarda en una tabla hash adicional la información de la posición de cada término (*offset*) dentro del índice para que la búsqueda posterior sea más eficiente. De forma que para buscar un término, consultaremos en la tabla hash su *offset* dentro del archivo en O(1) y accederemos directamente a esa posición para leer su lista de *postings*.

Como esto podría resultar demasiado costoso en RAM si el índice contuviera demasiados términos, se ha decidido implementar una versión que permite dividir el diccionario en bloques, cada uno de los cuales contiene *TERM_MAP_SIZE* (ver *IndexWriter*) términos, de forma que solamente el primer término de cada bloque se guarda en la tabla hash. De esta forma, para encontrar un término en el índice, consultamos el offset del mayor término menor que él (orden alfabético) en la tabla, y haremos busqueda lineal desde ahí. Por defecto se ha dado valor 100 al tamaño de bloque, por lo que si un término no es encontrado después de leer 100 términos, es que no se encuentra en el índice.

Adicionalmente, se aprovecha la lectura de cada término en el momento del volcado al archivo de índice definitivo para calcular los módulos de los documentos (implementación orientada a términos), con lo que se guarda una estructura adicional (array) de módulos para cada documento de la colección.

1.3. Índice en disco.

Él índice en disco se encuentra en el archivo *index* y consta de una sucesión de cadenas con el siguiente formato[3]:

$$T\_S_T\{D_1N_1P_{D_11}P_{D_12}\cdots P_{D_1N_1}\}\cdots\{D_mN_mP_{D_m1}\cdots P_{D_mN_m}\}$$

Donde cada letra representa lo siguiente:

- T: Término literal que se está leyendo, siempre va seguido de un espacio que indica el final del término, de forma que nuestro índice no acepta términos formados por varias palabras separadas por espacio.
- S_T: Tamaño en bytes de toda la lista de *postings* en disco del término T.
- D_x: Id del documento al cual pertenecen los *postings* que se leerán a continuación.
- N_x: Frecuencia del término en el documento D_x.
- P_{D_xy}: Valor de posición y del término T en el documento D_x.

Con esta estructura para cada término en los archivos el *merge* cuando encontramos el mismo T presente en dos bloques distintos es muy simple; simplemente se suman las S_T y se concatenan los bytes posteriores, poniendo primero la cadena correspondiente al menor bloque (el que fue creado antes) para conservar el orden de los postings.

Las tablas hash de ids de documentos y diccionario de términos con offsets, se guardan en disco en los archivos *docids* y *termsoffset*. El array de módulos de documentos se guarda en el archivo *modules* Para crear estos archivos se ha hecho uso del método *writeObject* de la clase *ObjectOutputStream*, ya que permite recuperar los objetos fácilmente desde archivo mediante el método *readObject*.

1.4. Parsing

La diferencia principal entre las tres clases de índice pedidas para este ejercicio se encuentra esencialmente en el *parser* que utilizan. Tanto el índice de *Stopwords* como el índice que realiza *Stemming* realizan exactamente el mismo indexado que el índice básico, con la diferencia de que el contenido que añaden para cada archivo ha sido previamente *parseado* de forma distinta. Por lo tanto, las tres clases de índice pedidas para este ejercicio, *BasicIndex*, *StopwordIndex* y *StemIndex* se diferencian únicamente los argumentos con los que ejecutan la

[3]Los corchetes no forman parte del archivo, se han puesto para añadir claridad

función *build* desde el método *main*.

A continuación vemos qué hace cada *parser*:

- Parser Básico (*BasicParser*): Además de eliminar las etiquetas HTML, elimina todos los símbolos que no sean caracteres [a-zA-Z] y posteriormente transforma todo el texto a lower case.
- Parser de *Stopwords* (*StopwordParser*): Realiza el filtrado del parser básico y a continuación elimina los *stopwords* y las palabras de longitud ≤ 2.
- Parser de *Stemming* (*StemParser*): Realiza el filtrado de *StopwordParser* y posteriormente utiliza el *snowball stemmer* con diccionario inglés realizando dos pasadas sobre cada término.

1.5. Algunos resultados

En la figura 1.5 podemos ver la gráfica extraída con el *profiler* de *NetBeans* para la ejecución de la clase *IndexBuilder* construyendo los tres índices sobre la colección de 100.000 documentos.

Con *output* auxiliar del programa se ha medido el tiempo de creación de cada índice, siendo éste aproximadamente de 25 minutos para los índices.

Como se puede comprobar en la gráfica, el procesamiento de documentos es más largo en los índices con filtrado de *stopwords* y *stemming*, pero la fase de *merge* se acorta, debido a la eliminación de más términos a consecuencia del filtrado.

El consumo de RAM se mantiene en valores razonables, alcanzando picos máximos de 900 MB durante las fases de procesamiento de documentos en los índices *StopwordIndex* y *StemIndex*. Esto se debe a que los parseos adicionales duplican el tamaño que ocupa el contenido de un archivo en memoria, ya que dicho contenido se parsea término a término y el resultado se añade a otra lista de contenido definitivo para el índice.

Por otro lado también se puede observar que la fase de merge es bastante ligera en cuanto a consumo de RAM y que se ha agregado al main una fase final de obtención de estadíisticas que son volcadas al archivo *indexstats* y del cual se pueden extraer gráficas similares a las pedidas en la práctica 1 ejecutándo el *script printstats.sh* adjunto en la práctica.

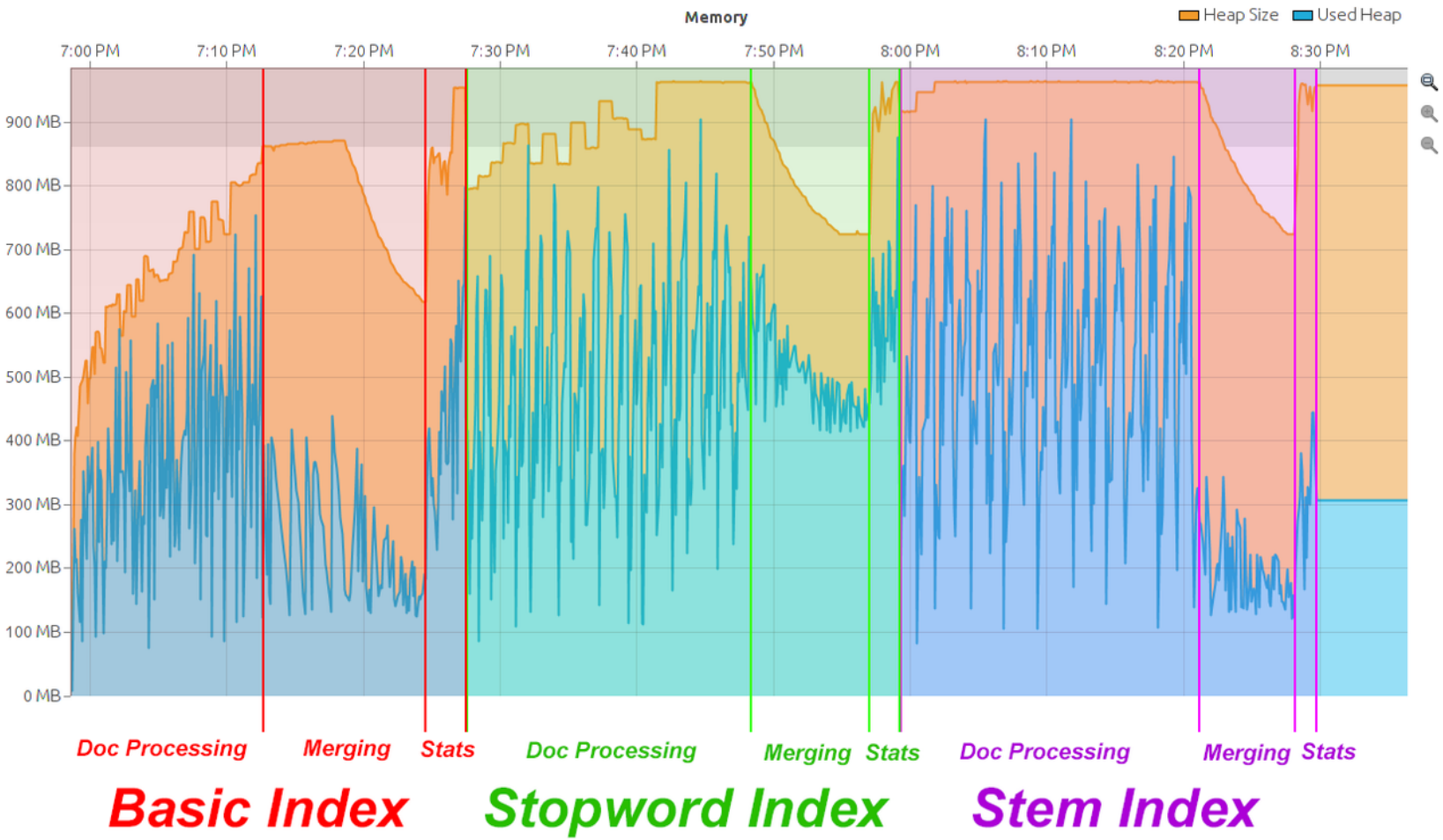

Figura 2: Profiler de NetBeans para la ejecución de *IndexBuilder*

2. Ejercicio 2: Recuperación de información.

En esta sección se describirá el diseño de los dos buscadores implementados (`TFIDFSearcher` y `LiteralSearcher`). Además, se detallarán las estructuras de datos utilizadas y se mostrarán ejemplos de resultados de búsqueda para algunas consultas.

2.1. Diseño e implementación de los buscadores

A continuación se describe por separado el diseño y la implementación de ambos buscadores.

2.1.1. Buscador TF-IDF

Para implementar este buscador se ha seguido el modelo orientado a documentos visto en teoría.

En primer lugar, cuando se realiza una búsqueda, se cargan los *postings* de los términos de la consulta en memoria. Tras ello, se toman los primeros *postings* de cada término y se extrae el identificador del documento asociado y su puntuación parcial, esto es, relativa sólo a la información que contiene el *posting*. Una vez hecho esto se introducen estos datos en un *heap* del tamaño de la consulta, ordenado de manera que la raíz del mismo corresponde al documento con menor identificador.

Cuando ya se tiene el *heap* construido, se van recorriendo las listas de *postings* cosecuencialmente, extrayendo el identificador del documento asociado y su puntuación parcial. Una vez extraídos estos datos de un *posting* se introducen en el *heap* tras extraer su raíz. Si el identificador de documento de la raíz cambia, se introduce en un *min-heap* de tamaño el número de documentos a devolver. En caso contrario, se actualiza el valor de la puntuación sumando las puntuaciones parciales.

Para recorrer los *postings* de forma cosecuencial se utiliza un array del tamaño de la consulta, que mantiene los índices de acceso a los elementos de las listas de *postings* de todos los términos de la consulta.

Una vez ya se han procesado todos los *postings*, quedan en el *min-heap* los documentos con mayor puntuación.

Para calcular las puntuaciones parciales a partir de un *posting* se utiliza:

$$score = \frac{1}{|d|} \frac{log_2|D|}{log_2|D_t|} (1 + log_2(frec(t,d)))$$

donde:

- $frec(t,d)$ es el tamaño de la lista de posiciones del *posting*, es decir, la frecuencia del término en el documento.
- $|D|$ es el número de documentos en la colección.
- $|D_t|$ es el número de documentos que contienen al término.

- $|d|$ es el módulo del documento.

Todos estos datos se obtienen utilizando los métodos del índice.

2.1.2. Buscador literal

Para implementar el buscador literal se han procesado los términos de la consulta secuencialmente del siguiente modo. En primer lugar se obtiene la lista de *postings* del primer término. Tras ello, se obtiene la lista de *postings* del segundo y se crea una lista nueva a partir de ambas.

Esta nueva lista estará formada por *postings* que se construyen de la siguiente manera. En primer lugar, se buscan *postings* en las dos listas con el mismo identificador de documento. Para encontrar coincidencias de documentos se recorre una de las listas y se utiliza búsqueda binaria en la otra, puesto que los *postings* en una lista están en el índice ordenados por identificador de documento. Una vez se tienen dos *postings* con el mismo documento asociado, se recorren las listas de posiciones obteniendo aquellas que sean consecutivas.

Este proceso se itera hasta haber procesado todas las listas de *postings* de los términos, obteniendo finalmente una lista de *postings* en la que los documentos asociados contendrán el literal completo.

Una vez obtenida esta lista de *postings*, se calculan sus puntuaciones de la misma forma que en el buscador anterior, obteniendo una lista de todos los documentos, que se ordenará en forma decreciente de puntuaciones.

2.2. Estructuras de datos

Se ha utilizado una cola de prioridad de Java (que está basada en un heap) como implementación del heap que se usa en el buscador *TF-IDF*. Esta estructura se inicializa con un tamaño igual que el tamaño de la consulta, y siempre se realiza una extracción de la raíz antes de la inserción de un elemento, por lo que nunca se excede el tamaño dado en la inicialización.

Para el *min-heap* se ha realizado una implementación propia utilizando también la cola de prioridad de Java. En esta implementación, el heap se inicializa con un tamaño máximo. Una vez alcanzado, no se permite la inserción de elementos que sean menores que el situado en la raíz, que es a su vez el menor del heap. En caso de que el elemento a insertar sea mayor que la raíz, esta última se extrae antes de la inserción.

De esta manera, en dicho heap siempre estarán los documentos que mayor valor tengan, obteniendo así un gran ahorro de memoria, pues el tamaño del *min-heap* se limitará al número de resultados a devolver. Además, también se obtiene un gran ahorro de tiempo de ordenación por dicha limitación.

2.3. Ejemplos de búsqueda

Ambos buscadores devuelven una lista que contiene objetos representando los documentos con mayor puntuación ordenados de forma decreciente. En el

método principal de cada clase, tras introducir una consulta y obtener los documentos, se muestra para documento su identificador, su ruta, su puntuación, y parte de su contenido, cargando parte del documento en busca de algún término de la consulta y mostrándolo parcialmente.

A continuación se muestran algunos ejemplos de búsqueda de ambos buscadores con la colección de 1K documentos. En primer lugar se va a comparar los resultados de los distintos índices para el buscador *TF-IDF*.

Para la consulta "paste" el primer resultado obtenido es el siguiente:

```
ID: 490 Name: clueweb09-en0000-61-13089.html  Score: 0.094
Content: paste is vacuum dried and then additives such as
perfume and ant
```

Al utilizar el índice stem, obtememos como primer resultado:

```
ID: 819 Name: clueweb09-en0009-93-3724.html  Score: 0.0867
Content: past experiences
```

Se puede observar que al aplicar el índice *stem* se obtenido un documento donde el término buscado se ha reducido a su raíz: "past".

En cuanto al índice *stopword*, si buscamos con el buscador básico "this is for obama" obtenemos como primer resultado el siguiente:

```
ID: 786 Name: clueweb09-en0001-02-21241.html  Score: 0.1780
Content: obama family tree and genealogy of senator obama
```

En esta búsqueda se han filtrado los términos "this is for", de manera que se le da importancia sólo al término "obama". Para el índice básico, esta misma consulta devolverá como documento con mayor puntuación:

```
ID: 319 Name: clueweb09-en0010-79-2218.html  Score: 0.196
Content: for the management of that trivia section topi
```

Para el buscador literal con el índice básico, si ejecutamos, por ejemplo, la consulta "getting organized" obtenemos como primer resultado el siguiente, donde observamos que el documento obtenido contiene el literal:

```
ID: 298 Name: clueweb09-en0002-44-10182.html  Score: 0.284
Content: getting organized clothing closets dymo
tips advice
```