Grado en Ingeniería Informática - Universidad de Alicante

Práctica 2: Indexador

Explotación de la Información. 2022-2023

Andrei Eduard Duta
28-4-2023

CONTENIDO

1 INTRODUCCIÓN

En esta memoria se presenta la solución implementada para el problema de la indexación de una colección de documentos. Los indexadores son herramientas que se utilizan para organizar grandes cantidades de información en un sistema de recuperación de información. Su objetivo es crear una estructura de datos que permita la búsqueda eficiente de información en una colección de documentos. Para ello se ha desarrollado una clase en C++, llamada *IndexadorHash*, que utiliza estructuras de datos basadas en tablas hash para generar índices tanto en memoria como en disco.

La hipótesis de partida es que la implementación de una solución eficiente de indexación permitirá acelerar el proceso de búsqueda en grandes colecciones de documentos.

2 ANÁLISIS DE LA SOLUCIÓN IMPLEMENTADA - MEMORIA

El algoritmo implementado en la indexación en memoria se basa en la utilización de la clase *Tokenizador* previamente desarrollada, que divide los documentos en tokens. A continuación, se utiliza una tabla hash (*unordered_map*) para almacenar la información de los tokens y las posiciones en las que aparecen en cada documento. Adicionalmente, se utiliza un conjunto hash (*unordered_set*) para almacenar la información de los documentos y sus títulos.

El **algoritmo** seguido para indexar una colección de documentos en memoria es el siguiente:

1. Se recorren todos los documentos a indexar de la colección
2. Se tokeniza el documento a indexar, obteniendo la lista de tokens.
3. Se recorre la lista de tokens del documento, se aplica el stemmer, se eliminan los que sean stop-words y se almacenan todos los términos diferentes que aparecen en un índice. Para cada término se almacena su frecuencia de aparición en la colección de documentos (***ftc***) y cada documento en el que aparece (***fd***) con la frecuencia de aparición en el documento (***ft***) y las posiciones donde aparece dicho termino en el documento.
4. Los términos repetidos actualizan sus respectivas entradas del índice
5. Cuando se termina de indexar un documento se guardan sus datos recogidos en el índice de documentos: nombre, número de palabras que contiene, número de palabras eliminando las stop-words, número de palabras diferentes, tamaño en bytes y fecha de modificación.
6. Por último, después de indexar un documento, también se guarda la información de la colección: número de documentos indexados, número total de palabras indexadas, número total de palabras indexadas sin contar las stop-words, número total de palabras diferentes y tamaño total en bytes.

Como la eficiencia de este algoritmo depende en gran parte de las estructuras de datos usadas para los índices, se usan **tablas hash**.

3 JUSTIFICACIÓN DE LA SOLUCIÓN ELEGIDA - MEMORIA

Como alternativas al algoritmo implementado se podría la paralelización del algoritmo secuencial. Con esto se podría incrementar mucho la eficiencia de la indexación, sin embargo, no se ha implementado de esta forma ya que no es el objetivo de la asignatura.

En cuanto a las **estructuras de datos** utilizadas, las **tablas hash** son las más eficientes en cuanto a tiempo, ya que ofrecen una complejidad de inserción, borrado y búsqueda constante (salvo en caso de redimensionar la tabla – O(n)). Además, *unordered_map* implementa dispersión abierta que en caso de colisión utiliza una lista enlazada. La tabla se redimensiona cuando se supera cierto umbral. Cabe destacar que si tomamos en cuenta la complejidad del cálculo del hash, como este debe tomar en cuenta todos los caracteres de una palabra, la complejidad temporal total de búsqueda, inserción y borrado sería O(n) siendo n el número de caracteres de la palabra.

Se han considerado como alternativa las estructuras **trie**. Estos son árboles de búsqueda en los que cada nodo es una letra. La inserción y búsqueda tienen complejidad O(n). Sin embargo, los tries añaden una complejidad adicional al tener que acceder a memoria por cada carácter, mientras que en una tabla hash solamente se accede a la memoria una vez. Por otro lado, cuando una palabra no está indexada en un trie, la búsqueda acaba antes de recorrer todos los caracteres, por lo que en este caso el trie sería más rápido.

La complejidad espacial depende de la pre-alocación de memoria que se hace en las tablas hash para evitar colisiones y de la cantidad de prefijos comunes en las palabras de la colección para los tries.

Para una aplicación que haga uso de búsquedas parciales o de búsquedas que devuelvan todas las palabras que comparten un prefijo, los tries serían una mejor solución.

Por último, existen los tries **PATRICIA**, que, aunque no mejoran la velocidad de búsqueda de las tablas hash, consumen menos memoria.

4 ANÁLISIS DE LA SOLUCIÓN IMPLEMENTADA – DISCO

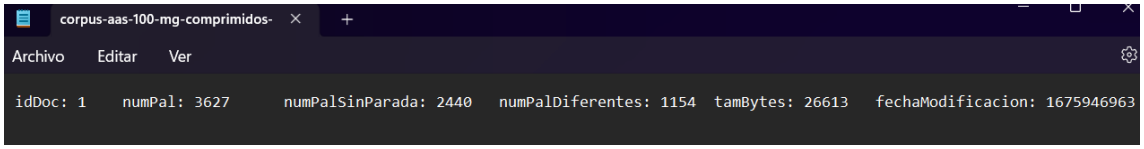
Para solucionar el problema de indexar grandes colecciones que no caben en memoria principal se ha implementado una versión para almacenar la indexación en disco.

El algoritmo es el mismo que para la solución en memoria, pero se añaden tres tablas hash adicionales. Cada vez que se supera cierto umbral predeterminado de palabras indexadas, se almacena todo lo indexado en

disco. Para ello se llama a una función que recorre el índice de palabras y crea un archivo para cada una donde se escribe la información relacionada con la palabra.

```
Frecuencia total: 24970 fd: 688
      Id.Doc: 709 ft: 26      60
      415   446   457   491   520
      528   712   1078  1093  1122
      1350  1402  1717  1831  1841
      1853  1885  1894  1928  1946
      2149  2213  2234  2259  2264
      Id.Doc: 708 ft: 31      173
      389   851   873   896   919
      1295  1317  1323  1360  1373
      1531  1544  1550  1836  2120
      2129  2137  2163  2212  2220
      2230  2284  2293  2374  2802
      2817  2865  2912  2952  3040
      Id.Doc: 707 ft: 31      173
      389   851   873   896   919
      1295  1317  1323  1360  1373
      1531  1544  1550  1836  2120
      2129  2137  2163  2212  2220
      2230  2284  2293  2374  2802
      2817  2865  2912  2952  3040
      Id.Doc: 706 ft: 31      173
      389   851   873   896   919
```

También se almacena cada documento indexado con su información.



Para tener una referencia de las palabras y los documentos guardados en disco, se utilizan tres tablas hash adicionales:

- *indice_guardados*: esta tabla almacena cada palabra junto con un identificador único que es el nombre del archivo.
- *indiceDocs_guardados*: esta tabla almacena cada documento guardado en disco. En este caso el nombre del archivo será el mismo que el del documento.
- *indice_actualizar*: esta tabla se utiliza como almacenamiento temporal para aquellas palabras que al encontrarse en un documento que se está indexando, se comprueba que ya han sido indexadas, pero se encuentran almacenadas en disco. Cuando se llama a la función de almacenar en disco, las palabras de este índice se recuperan de sus respectivos archivos, se actualizan y se guardan de nuevo.

5 JUSTIFICACIÓN DE LA SOLUCIÓN ELEGIDA – DISCO

Se han considerado varias alternativas para el almacenamiento en disco. Como primera solución se consideró almacenar todas las palabras en **un mismo archivo**. Esto reduciría tiempos ya que no se necesitan tantos accesos a memoria secundaria. Sin embargo, esta solución implica cargar un gran número de datos cada vez que se quiera modificar algo de una palabra de las almacenadas, además de la complejidad de deserializar todo el archivo para buscar la palabra que se quiere modificar y volver a serializar todo. Por otro lado, sería impracticable ya que implica cargar en memoria todos los datos, justo lo que queremos evitar con esta solución.

Una posible mejora sería guardar palabras por **lotes**, pero esto implicaría tener un índice de palabras para cada lote y recorrer cada uno de estos índices cuando se quiere buscar una palabra.

La solución elegida fue guardar **cada palabra en un archivo**. Inicialmente esto se implementó usando un *unordered_set* para guardar las palabras almacenadas y nombrar a los archivos con el nombre de las palabras. Esto supuso un problema ya que algunas palabras pueden contener caracteres como "/" que se debe escapar para usarlo en el nombre de un archivo. Esto implicaría recorrer cada palabra carácter a carácter lo que añadiría demasiada complejidad temporal. Se decidió cambiar el *unordered_set* por *unordered_map* y así guardar un identificador único para cada palabra que sería el nombre del archivo.

Para optimizar la velocidad de indexación, se debería ajustar el momento de almacenar en el disco en función de la memoria principal disponible de la máquina. Para ello se puede comprobar la cantidad de memoria usada o disponible a través de alguna librería o comandos y almacenar en disco cuando estemos llenando la memoria disponible. Como no disponemos de un corpus tan grande, pero se quiere comprobar el funcionamiento de esta solución, en esta práctica se ha implementado de tal manera que después de indexar cierto número de documentos (100) se almacene la indexación.

5.1 ACTUALIZACIÓN DE LAS PALABRAS

Uno de los problemas encontrados ha sido que después de indexar los 100 primeros documentos y almacenar en disco, cada vez que había una palabra ya indexada se cargaba el fichero para actualizarla y se volvía a escribir. Esto aumentaba los tiempos de una forma prohibitiva (2-3 segundos por documento).

Para evitar acceder al fichero cada vez que se quiere actualizar una palabra se ha implementado un nuevo *unordered_map (**indice_actualizar**)* con la misma estructura y funcionamiento que el índice principal, pero con la diferencia que solamente almacenará palabras que se tienen que actualizar, pero han sido almacenadas en disco.

De esta manera, si en los próximos 100 documentos, aparece una misma palabra varias veces, en lugar de acceder cada vez a disco, esta se indexará desde 0 en memoria principal y en la próxima llamada a *AlmacenarEnDisco()* se actualizarán los ficheros correspondientes sumando los nuevos datos.

6 ANÁLISIS DE LAS MEJORAS REALIZADAS EN LA PRÁCTICA

Para optimizar el código se han realizado varias mejoras como usar punteros en lugar de copiar los objetos a la hora de actualizar un término, usar paso por referencias y variables const, usar iteradores para los bucles. También se ha revisado el código eliminando asignaciones innecesarias e incluso búsquedas repetidas.

Todas estas mejoras han supuesto una aceleración de 1,42 sobre el código original pasando a indexar el corpus de 4s a 2,8s.

También se ha probado a reservar memoria para los índices, pero no se ha observado ninguna mejoría. Esto probablemente se deba a que no disponemos de una colección suficientemente grande para provocar el redimensionamiento de las tablas hash.

Algunas de las mejoras mencionadas también han afectado positivamente a la memoria usada como el paso por referencia y el uso de punteros.

7 ANÁLISIS DE EFICIENCIA COMPUTACIONAL

Las pruebas para realizar el análisis se han realizado sobre el mismo PC, en el mismo estado y compilando sin las opciones de optimización o debug. Se han ejecutado 5 veces cada prueba realizando una media. Los tiempos se han medido en segundos y la memoria en Kbytes.

	Sin almacenamiento	**Con almacenamiento**
Temporal	2,79 s	12.49 s
Espacial	122092 Kb	24112 Kb

Podemos observar que la versión con almacenamiento supone un incremento de casi 5 veces (4,47) en el tiempo a cambio de un consumo 5 veces menor de memoria.

Finalmente, concluimos que los análisis arrojan un resultado satisfactorio puesto que hemos conseguido una relación lineal de intercambio entre el tiempo que tarda y la cantidad de memoria que consume. De esta forma, y a falta de realizar pruebas sobre un corpus más grande, hemos conseguido solucionar la problemática de indexar una colección de documentos que no cabe en memoria principal sin perder eficiencia temporal.