Explotación de la Información
Indexador

Jorge Donis del Álamo

19 de abril de 2020

Índice

1. Talla del problema

Consideraremos las siguientes variables de entrada como definitorias de la talla del problema:

$$\text{Número de tokens por documento } n$$
$$\text{Número de documentos en el corpus } m$$

Además, los cálculos se desprenden del análisis de `bool IndexadorHash::Indexar(const string ficheroDocumentos)`. Finalmente, se considera el coste de inserción, búsqueda y borrado de la tabla hash como constante.

2. Almacenamiento en memoria principal

En este caso, no se realizan accesos a disco, salvo para leer los tokens de los documentos.

2.1. Complejidad temporal

El mejor caso vendría dado por una colección de m documentos en la que ningún documento estuviera previamente indexado y todos los tokens con el correspondiente tratamiento de stemming fueran *palabras de parada*. El coste vendría dado por crear m entradas en la tabla hash `indiceDocs` (con coste constante) y tokenizar los m documentos con coste n.

$$\begin{aligned} C &= m(\text{insertar en indiceDocs} + \text{tokenizar}) \\ &= m(1+n) \in \Omega(mn) \end{aligned}$$

El peor caso sería un corpus en el que todos los documento estuvieran previamente indexados. Además, estos documentos contendrían todos n tokens (después del stemming) únicos. Entonces habría que crear m n entradas en `indice` y además todos los tokens tendría una entrada en lista (que tendría que ser insertada) `l_docs` (que relaciona los tokens con los documentos en los que aparecen). Evidentemente, para que se de la reindexación, los documentos han de ser más recientes que los previamente indexados. El coste de la reindexación radica en el borrado de un documento. Para borrar un documento hay que borrar de todos los tokens sus ocurrencias en dicho documento. Si el token sólo apareciera en dicho documento, además habría que borrar el token del índice. Esto siempre va a suceder, ya que los tokens son únicos. El coste total sería:

$$C = m(\underbrace{mn}_{\text{BORRADO DOCUMENTO}} + \underbrace{n}_{\text{BORRADO TOKENS}} + \underbrace{n}_{\text{TOKENIZAR}} + n(\underbrace{1}_{\text{ACTUALIZAR INDICE E INDICEDOCS}}))$$

$$= m^2n + 3mn$$
$$\in O(m^2n)$$

2.2. Complejidad espacial

En mi implementación, las únicas variables que dependen de la talla del problema son las tablas hash `indice` e `indiceDocs` y además guardo en memoria todos los tokens de un documento en concreto como `char*`. Esta memoria se reserva y libera convenientemente.

El mejor caso sería que todos los tokens fueran *palabras de parada*. Así no habría ninguna entrada en `indice`, aunque seguiría habiendo m entradas en `indiceDocs`. El coste sería $m + n \in \Omega(m + n)$

El peor caso sería aquel en que todos los tokens son únicos (y no son palabras de parada). El coste sería $mn + m + n \in O(mn)$

3. Almacenamiento en memoria secundaria

Esta versión supone que la colección de documentos no cabe en memoria principal. En teoría es posible almacenar todas las palabras del español (en torno a 80 mil palabras, suponiendo 10 caracteres por palabra, nos daría 781 MB), sin embargo, el número de documentos es potencialmente infinito.

En mi implementación, se guarda un fichero por token y un fichero por documento, ambos almacenarán, respectivamente, la información de término y de documento. Los nombres de los ficheros se calculan a partir del nombre original y una función hash. Con lo que la complejidad temporal teórica viene a ser la misma que usando una tabla hash, ya que la búsqueda, inserción y borrado son constantes. La realidad es que los tiempos de ejecución son astronómicos. Esto se debe a las múltiples escrituras en disco, todas ellas de pocos bytes.

4. Posibles mejores para el almacenamiento en memoria secundaria

El principal objetivo es **reducir el número de accesos a disco**. Tampoco podríamos dejar todos los descriptores de archivo abiertos, ya que el sistema operativo tiene un límite para esto. La clave radicaría en usar estrategias similares a las empleadas en Sistemas de Gestión de Bases de Datos, tanto relacionales como no relacionales. Al fin y al cabo, el índice no deja de ser una colección de datos de la que queremos escribir y leer con velocidad y conservando la integridad de los datos.

Si seguimos el ejemplo de las bases de datos relacionales (SQL), éstas hacen uso de estructuras de datos como árboles-B para realizar sus índices (que mejoran la velocidad de las queries en una determinada tabla). El problema surge de la naturaleza dinámica de nuestro sistema de información. Cada token puede tener una longitud potencialmente infinita, lo mismo para el número de documentos en el que aparece. La solución más sencilla sería reducir todas las clases a una sola tabla `InformacionTermino`, la cual tendría un tamaño constante. Entonces sí que se podrían realizar inserciones "en grupo" de la información de varios tokens a la vez. Es decir, se guardaría en memoria principal una porción (lo más grande posible) del índice, hasta que no cupiera, en cuyo momento se escribiría en disco. Los árboles-b son ordenados, esto sería posible gracias a una ordenación por un identificador autoincrementado (ID) o simplemente una función hash para cada token (sería muy raro que se diera una colisión). El árbol-b puede ser de distintos órdenes, según cuántos hijos pueda tener cada nodo. La clave sería escoger un orden lo suficientemente grande como para reducir el número de escrituras en disco, por ejemplo 500.

En el caso de las bases de datos no relacionales, como por ejemplo MongoDB, también se hace uso de caché para mejorar la eficiencia. Podríamos conservar la naturaleza dinámica de nuestro sistema de información si empleamos un búfer temporal. El tamaño de este búfer temporal no se podría saber a priori (no se podría reservar en el heap), pero podríamos usarlo a modo de `vector<char*>`, haciéndolo crecer cuando fuera necesario. Todas las escrituras y lecturas se realizarían sobre memoria principal, hasta que llegara el momento de escribir en disco, cuando se vaciaría el búfer temporal.