Explotación de la Información

Práctica 2: Indexador

Rafael Bustos Moreno

ÍNDICE

1. Introducción

En esta práctica se nos presenta el problema de realizar un indexador de documentos completo. Explicando un poco lo anterior, debemos crear una serie de índices que almacene cierta información sobre el contenido de este, para más adelante implementar un navegador.

Lo que se nos propone es una estructura de clases que procederé a explicar:

- **Clase InfTermDoc:**

En esta clase se guardará la información relativa a cada término indexado que aparecen en los documentos. Se guardará la frecuencia de cada término en el documento (ft) y la posición de dicho término dentro del documento (cuando este activa la opcion almacenarPosTerm del indexador).

- **Clase InformacionTermino:**

Guardaremos la informacíon de cada término indexado. En concreto guardaremos la frecuencia total del término en la colección de documentos (ftc) y guardaremos los documentos en los que se encuentre dicho término.

- **Clase InfDoc:**

En esta, lo que guardaremos será la información de cada documento indexado. Siendo más exactos, guardaremos de cada documento un id asociado a él (idDoc), el número total de palabras del documento (numPal), el número total de palabras sin parada, es decir, número total de palabras del documento que no son palabras de parada (numPalSinParada), el número total de palabras diferentes (numPalDiferentes), el tamaño en bytes del documento (tamBytes) y la fecha de modificación del documento (fechaModificacion).

- **Clase InfColeccionDocs:**

Aquí guardaremos la información de la colección de documentos indexados. Lo que guardaremos será: el número total de documentos indexados (numDocs), el número total de palabras de la colección (numTotalPal), el número total de palabras sin parada, es decir, número total de palabras de la colección que no son palabras de parada (numTotalPalSinParada), el número total de palabras diferentes en la colección (numTotalPalDiferentes) y el tamaño total en bytes de los documentos (tamBytes).

- **Clase InformacionTerminoPregunta:**

En esta otra clase guardaremos la información de cada término de la pregunta indexada. De esta clase guardaremos, la frecuencia total del término en la pregunta (ft) y las posiciones del término en la pregunta (posTerm).

- **Clase InformacionPregunta:**

Por último, esta clase almacenará toda la información de la pregunta indexada. En concreto se almacenará el número total de palabras de la pregunta (numTotalPal), el número total de palabras que no son de parada (numTotalPalSinParada) y el número total de palabras diferentes en la pregunta (numTotalPalDiferentes).

- **Clase IndexadorHash:**

Esta clase es la clase principal de nuestro programa, es el que lleva a cabo la indexación y es la clase que tiene toda la información acerca de los índices. En ella guardaremos en los siguientes atributos:

- **indice:** Tabla hash con la información de todos los términos indexados. Es accesible por el nombre del término.

- **indiceDocs:** Tabla hash con la información de todos los documentos indexados. Es accesible por el nombre del documento.
- **informacionColeccionDocs:** Atributo de clase InfColeccionDocs que almacena la información de la colección de documentos indexados.
- **pregunta:** String que guarda la pregunta indexada actual.
- **indicePregunta:** Tabla hash que guarda la información de cada término de la pregunta. Es accesible por el término de la pregunta.
- **infPregunta:** Atributo de clase InformacionPregunta que guarda la información de la pregunta indexada.
- **stopWords:** Lista de string que guarda las palabras de parada extraídas del documento ficheroStopWords.
- **ficheroStopWords:** String que guarda el documento de las palabras de parada.
- **tok:** Tokenizador a utilizar a la hora de indexar.
- **directorioIndice:** String donde se almacenará el índice
- **tipoStemmer:** Int que indica que tipo de stemmer se va a utilizar en el indexador.
- **almacenarEnDisco:** Booleano que indica si la indexación se guarda en el disco o si por el contrario en memoria principal.
- **almacenarPosTerm:** Booleano que indica si almacenamos la posición de los términos en el documento.

Como hipótesis de partida comencé implementando un programa más ineficiente a la hora de leer los ficheros, guardar los ficheros o recorrer las tablas hash, cosa que posteriormente mejoré.

2. Análisis de la solución implementada de la indexación en Memoria

A continuación, voy a analizar la solución que he implementado para el método Indexar de la clase Indexador. Lo primero que hago es leer entero el documento que se pasa por parámetro a la función. Una vez abierto y leído, voy línea por línea cogiendo el nombre del documento.

Cuando obtengo el nombre del documento busco en indiceDocs si ese documento ya ha sido indexado anteriormente. En caso de que ya esté indexado, miro si la fecha del documento es más nueva que el documento ya indexado, y si es más nueva llamo al método BorraDoc para que borre la indexación del documento antigua y después llamo al métido IndexarDocumento con el idDocumento de la indexación antigua. En caso de no encontrar el documento en una indexación pasada, llamo al método IndexarDocumento con idDoc = 0;

Dentro de la función indexarDocumento, lo primero que hago es abrir el documento y verificar que el documento se ha abierto correctamente. Una vez abierto leo el documento de forma mapeada con mmap, obteniendo todo el contenido del documento. Después verifico si el idDoc pasado por parámetro es igual a 0, si este es igual a cero entonces se sefiere a que el documento no estaba indexado, por lo que se le asigna un nuevo idDoc mediante el número de documentos de la colección. En caso de que no sea igual a cero se le asigna al documento el mismo idDoc.

Después de esto, tokenizo todo el fichero sacando los tokens de todo el fichero, y uno a uno verifico si se encuentra dentro de las stopwords, y en caso de que se encuentre, se descarta. Luego, verifico si se encuentra o no dicho término indexado, en caso de que ya estuviese indexado, miro que si es del documento actual aumento su frecuencia en dicho documento y hago un push_back de la misma, en caso de que sea de otro documento, lo creo en l_docs de

dicho documento. Por otra parte, si dicho término no estaba en el índice, inserto dicho término en el índice.

Una vez recorridos todos los términos del documento actualizo la información correspondiente de la colección de documentos.

3. Justificación de la solución implementada de la indexación en Memoria

Las estructuras utilizadas en esta solución son las que se nos proporciona en la asignatura. He probado a cambiar las estructuras unordered_map por estructuras tr1::unordered_map, que mejoraban la optimización de las mismas, pero me daban muchos problemas a la hora de recorrerlas por lo que decidí no usarlas.

Por otra parte, la mejora de leer los documentos de manera mapeada la implemente para mejorar la eficiencia del programa, antes de esto utilice la estructura ifstream y leía línea por línea el documento provocando esto ineficiencia.

4. Análisis de la solución implementada de la indexación en Disco

Para la solución implementada de la indexación en disco he creado 4 nuevos atributos en la clase IndexadorHash. Los atributos implementados han sido indicesEnDisco, lista con los punteros a los ficheros de los indices guardados en disco, indicesDisco entero para llevar la cuenta de indices guardados en disco, indicesDocsEnDisco lista con los punteros a los ficheros de los indicesDocs guardados en disco, indicesDocsDisco entero para llevar la cuenta de indiceDocs guardados en disco.

La indexación en disco la realizo al acabar de leer cada documento. Lo que hago es comprobar que almacenarEnDisco esté a true, si se encuentra a true lo que hago es mirar que cantidad de memoria total tengo y que cantidad de memoria se está utilizando. Si la cantidad de memoria que se está utilizando es mayor a la memoria del sistema, guardo los indices en un fichero en el directorioIndice. Al acabar de guardarse actualizo la lista con los punteros a los ficheros en disco con cada indexación y aumento el número de ficheros en disco.

5. Justificación de la solución implementada de la indexación en Disco

He probado muchas maneras de obtener la memoria consumida, y la única que me sirvió fue esta:

```
int parseLine(char* line){
    int i = strlen(line);
    while (*line < '0' || *line > '9')  line++;
    line[i-3] = '\0';
    i = atoi(line);
    return i;
}

// Función para sacar el tamaño de memoria utilizado por
int getValue(){ //Note: this value is in KB!
```

```
    FILE* file = fopen("/proc/self/status", "r");
    int result = -1; char line[128];
    while (fgets(line, 128, file) != NULL){
        if (strncmp(line, "VmRSS:", 6) == 0){
            result = parseLine(line); break;
        }
    }
    fclose(file);
    return result;
}
```

Por otra parte como única mejora es que a la hora de guardar el documento en disco solo escribo una vez, ya que me guardo los valores de los índices en un string y escribo en el fichero una sola vez al final de recorrerlos.

6. Análisis de las mejoras implementadas

Una de las mejoras implementadas ha sido a la hora de leer los ficheros. Al principio leía los ficheros de manera que iba línea a línea leyendo la información del documento, pero con la mejora implementada lo que hago es mapear el fichero y leerlo entero de una sola vez, almacenando todo el texto del documento en un char*. Esta mejora la he implementador en los métodos Indexar para leer el documento con la lista de documentos y en el método IndexarDocumento para leer todo el contenido del documento. Este es el algoritmo:

```
int fd = open(documento.c_str(), O_RDONLY);
// Si el documento se ha abierto
if(fd != -1)
{
    // Abrimos el archivo con mmap
    int len = lseek(fd, 0, SEEK_END);   // len también es el tamaño del documento
    if(len < 0) // Si la longitud es menor que cero significa que es un documento vacio
        return true;
    // Guardamos el texto del documento en *mbuf
    char *mbuf = (char*) mmap(NULL, len, PROT_READ, MAP_PRIVATE, fd, 0);
```

Gracias a esta mejora he podido llamar al método Tokenizar del tokenizador con todo el texto entero del documento, ya que, como cuando mapeo el documento se me almacenan también los saltos de línea, esto hace que al llamar al tokenizar se me almacenen todos los tokens del texto con una sola llamada al método Tokenizar en la función IndexarDocumento.

```
// Tokenizamos el fichero entero
tok.Tokenizar(mbuf, tokens);
// Recorremos los tokens o terminos del documento
for(it = tokens.begin(); it != tokens.end(); it++)
{
```

Otra de las mejoras implementadas es en el método GuardarIndexacion(). La mejora que implemente al principio es que iba escribiendo en el fichero cada vez que realizaba una línea, sin embargo, leí por internet que lo más rápido es escribir una sola vez mucha información. Por ello, lo que hice fue inicializar un string vacío e ir concatenando todas las líneas en esa variable. Al final del todo escribo una sola vez el string haciendo que no tenga que escribir más de una de vez en el fichero.

```
if(fich.is_open())
{
    string guardar = "";
    // Empezamos a guardar términos
    guardar += ficheroStopWords + "\n"; // Fichero de stopwords
    for(unordered_set<string>::const_iterator it = stopWords.begin(); it != stopWords.end(); it++)  // Palabras de parada
    {
        guardar += *it + ",";
    }
    guardar += "\n";
    guardar += to_string(tipoStemmer) + "\n";  // Guardamos tipoStemmer
    guardar += to_string(almacenarEnDisco) + "\n"; // Guardamos almacenarEnDisco
    guardar += to_string(almacenarPosTerm) + "\n"; // Guardamos almacenarPosTermino
    guardar += directorioIndice + "\n"; // Guardamos directorioIndice
```

Como se observa debajo, solo escribo una única vez en el fichero:

```
    // Escribimos en el fichero y lo guardamos
    fich << guardar;
    fich.close();
```

7. Análisis de eficiencia computacional

Eficiencia con guardado en disco:

```
Ha tardado 3.4983 segundos

Memoria total usada: 1972 Kbytes
  "    de datos: 1840 Kbytes
  "     de pila: 132 Kbytes
```

Eficiencia con guardado en memoria:

```
Ha tardado 2.99655 segundos

Memoria total usada: 122616 Kbytes
  "    de datos: 122484 Kbytes
  "     de pila: 132 Kbytes
```

Como se observa, si guardamos en disco con las directrices que he marcado yo, la memoria desciente bastante, pero el tiempo aumenta un poco.