75.42 Taller de Programación I

TP N°5: Archivos Ubicuos

Grupo 04

"Documentación de diseño"

Índice

1. Introducción

El siguiente documento tiene como objetivo especificar y precisar los detalles técnicos de la realización de la primera parte del trabajo práctico. Se explican los problemas surgidos a la hora del diseño y las soluciones encaradas para solventar los mismos.

Todos los archivos y códigos fuente aquí mencionados y relacionados al proyecto, así como también el presente informe, pueden ser encontrados y descargados del repositorio del grupo (`https://github.com/federicomrossi/7506-tp-grupo07`).

2. Requerimientos planteados

El trabajo solicitado se basa en la creación de un sistema para la generación de un catalogo a partir de archivos de letras de canciones de diversos autores. Dicho sistema debe ser capaz de realizar búsquedas por autor, titulo y frases, entregando como resultado los temas que coincidan con los parámetros ingresados.

A su vez se exigieron ciertas condiciones para la realización del trabajo a la hora del manejo de las estructuras. El índice para la búsqueda por titulo debía estructurarse como un hash, por autor como un árbol B+ y los archivos de texto como bloques de registros variables.

El producto final es una aplicación de consola que permite realizar las operaciones pedidas.

3. División del trabajo

El trabajo esta divido en tres capas funcionales: la física, la lógica y el front-end.

A su vez, la capa lógica esta formada por tres módulos principales, a saber, el árbol B+, el hash extensible y las estructuras de recuperación de textos. La distribución mencionada se puede observar en la *Figura 1.*

Esta división esta fundamentada en tener el menor acoplamiento posible entre las clases y sobretodo entre los módulos lógicos. A su vez, permite una fácil división de las tareas entre los integrantes del grupo de trabajo.

4. Detalles técnicos

A continuación se describirá con un mayor nivel de detalle cada uno de los módulos que componen el sistema.

4.1. Capa Física

La capa física de este trabajo se centra alrededor de dos clases: *ArchivoBloques* y *SerialBuffer*.

La primera es la encargada de la interacción directa con el disco. La misma define interfaces para poder leer y escribir un archivo en función de bloques de un tamaño fijo y parametrizable.

El problema que surgió aquí es que independientemente de utilizar el modo normal de apertura para escribir de C++ (ios::out), el archivo era truncado y se perdía su contenido, quedando solamente el último bloque escrito. La solución para esto fue que si se quería modificar un bloque ya existente, se utilizara un archivo de trabajo temporal para pasar el contenido original, intercalar el bloque a modificar y luego copiar el resto del archivo. Posteriormente se elimina el archivo original y se renombra el de trabajo. Debido a que esto es muy costoso y generalmente muchas de las operaciones de escritura consisten en agregar un nuevo bloque al final, existe un método que abre el archivo en modo append (ios::app) y agrega el nuevo bloque. La lectura en cambio no presento problema alguno.

SerialBuffer es la clase encargada de brindar los medios para poder persistir de manera ordenada los registros de cada estructura. La misma presenta dos métodos principales, pack y unpack.

El primero se encarga de agregar registros a un buffer de caracteres (cabe aclarar que se escogió por archivos de tipo binario para la persistencia de datos). Para poder recuperar la información a posteriori, antes de empaquetar cada registro carga en el buffer un prefijo de longitud para poder saber cuanto va a tener que recuperar del buffer a un objeto.

El segundo, hace el trabajo inverso y restaura la información de un buffer, que previamente se leyó desde el disco, a un objeto. Utiliza el prefijo de longitu para saber la cantidad de caracteres a pasar desde el buffer al objeto de destino.

Como puede notarse, el buffer es la estructura fundamental y puede verse al mismo como una sucesión de registros como la siguiente:

```
[prefijoLongitud(unsigned short int), reg(registro genérico de longitud variable)]
```

Donde el tamaño total queda comprendido dentro del tamaño del buffer.

Una carácteristica de esta implementación es la relación uno a uno que debe haber entre los tamaños de bloque y de los buffer, ya que si los mismos no coincidieran generarían errores de segmentación a la hora de tratar datos en memoria.

4.2. Árbol B^+

Una de las estructuras mas importantes de la aplicación es el Árbol B^+, siendo esta la base sobre la cual se sostiene el sistema de indexación. Su implementación consta de una clase central denominada *ArbolBmas* y de tres estructuras secundarias denominadas *Nodo*, *NodoInterno* y *NodoHoja*. Como puede sospecharse, las dos últimas estructuras heredan de *Nodo*, siendo esta la estructura padre. El sentido de este diseño es establecer métodos comunes de manera tal de poder aplicar polimorfismo sobre ellas desde la clase *ArbolBmas*. Más detalles sobre los nodos serán expuestos en los siguientes apartados de la sección.

El árbol se almacenará en un archivo de bloques fijos, en el cual se aparta el primer bloque (bloque cero) para poder almacenar la metadata del mismo. Esta consta del *nivel* máximo que posee (si existen los niveles 0, 1 y 2, se almacenará el valor 2), un *contador de bloques* existentes el cual toma registro del último bloque asignado de manera tal de poder proveer un nuevo bloque en una inserción posterior, y el *número de bloque de la raíz*. Este último sirve para poder cargar la raiz al abrir el árbol, es decir, la raíz se encontrará siempre en memoria. Como puede notarse, a consecuencia de esto último, el bloque raíz no es siempre el mismo, sino que variará a medida que el árbol vaya creciendo y sea necesario realizar particiones de nodos. Se decidió hacerlo de esta manera a fin de poder reducir las lecturas y evitar las transformaciones entre tipos de nodos (ya que en un principio, cuando el árbol se encuentre vació, la raíz será un nodo hoja, pero si es partida, se convertirá en un nodo interno).

4.2.1. Nodos internos y nodos hoja

Cada nodo esta representado por un bloque en el archivo, en el cual se guardarán claves en el caso de los nodos internos, y registros en nodos hoja. La determinación de cuantas claves o registros caben en los nodos se realiza tomando en cuenta el tamaño del bloque especificado por el usuario en el archivo de configuración.

Una de las problemáticas a resolver fue cómo determinar el overflow, para lo cual se decidió separar una clave o registro en cada bloque que cumpla la función de posición de overflow. Es decir, al ingresar elementos en un nodo, una vez que se llega a ocupar la posición de overflow, el nodo lo advierte y da aviso de tal evento a su nodo padre. Este comportamiento se logró devolviendo un valor Ílógico en el método *insertar()* de los nodos, siendo que, la devolución de *false* significa que el nodo no está en overflow y un retorno del valor lógico *true* implica que el nodo se encuentra sobrecargado.

Cada nodo, además, provee a su padre un método que permite realizar la división con un nodo hermano. Este método se denomina *dividir()*. Entonces, en caso de que un nodo entre en overflow, su padre deberá crear otro nodo que sea del mismo tipo que el primero, y pasárselo como parámetro al método *dividir* de aquel que se encuentra sobrecargado. De esta manera se realizará la partición que permitirá volver a un estado estable al árbol.

Otro detalle a tener en cuenta es que para poder persistir o leer los nodos, existen los métodos *guardar()* y *cargar()* respectivamente. Estos guardarán o cargarán desde el número de bloque que se encuentra asignado al objeto de tipo *Nodo*.

4.2.2. Registros genéricos

Una restricción importante que posee el árbol es que los registros a insertarse deben ser, por contrato, objetos de tipos que hereden de la clase *RegistroGenerico*. Por contrato nos referimos a que el árbol puede llegar a aceptar otros objetos de diferentes tipos, pero el comportamiento ante estos puede ser inestable debido a la necesidad de contar con ciertos métodos primordiales. Esta decisión de diseño se debe a la forma en que internamente se encuentran organizados los registros al momento de ser cargados en memoria.

Además, debe tenerse en cuenta que dichos tipos descendientes de *RegistroGenerico* deben sobrecargar los métodos de serialización para, de esta manera, poder lograr un correcto funcionamiento del árbol.

De todas maneras, la gran ventaja es que el uso de registros genéricos le da la posibilidad al usuario de poder almacenar todo aquello que desea en el árbol, pudiéndose hacer uso de este último en distintos tipos de implementaciones.

4.2.3. Depuración

Al iniciar el diseño, se tuvo que pensar en una forma de realizar las pruebas, de manera tal de poder constatar el correcto funcionamiento del árbol. Es por esto que a los nodos se les ha agregado el método *imprimir()*, el cual debe imprimir por salida estándar una representación de su contenido manteniendo un formato predeterminado. Para el caso de nodos hoja el formato es el siguiente:

```
[nivel], [numero_bloque]: ([clave1])..([claveN])[nodo_hermano],
```

mientras que para los nodos internos el formato es:

```
[nivel], [numero_bloque]: [hijo1]([clave1])[hijo2]([clave2])..[hijoN]([claveN])[hijoN+1].
```

La clase *ArbolBmas* también posee un método *imprimir()*, el cual invoca al método *imprimir()* del nodo ubicado en la raíz.

Los nodos internos, aparte de imprimirse a sí mismos, también invocan a los *imprimir()* de sus hijos, generándose así, en la salida estándar, una representación completa del árbol. Esto permite hacer sencillo seguimiento de las inserciones, particiones, etc.

4.3. Hash Extensible

Para la implementación del *Hash Extensible* se buscó poder modularizar lo más posible las estructuras que hacen a su funcionamiento.Estas son el *Block*, perteneciente a la capa física, y *BlockTable* y *HashExtensible*, pertenecientes a la capa lógica.

HashExtensible cumple el trabajo de función de dispersión, es decir, dado un ID y aplicando una función de dispersión, genera otro valor, utilizado para ver donde ubicar dicho ID.

BlockTable y *Block* son las estructuras pilares en el funcionamiento de esta estructura.

Empezando por la estructura más baja se encuentra *Block*, también llamada *Buckets* (Cubetas). Su misión es la de contener registros, pero al ser de un tamaño fijo, se pueden llenar y provocar problemas los cuales son analizados más adelante. Los bloques cuentan además con un número de bloque por el cual son referenciados y un llamado tamaño de dispersión (TD). Este número representa la cantidad de veces que un bloque es referenciado por la tabla y es calculado como

$$TD = \frac{TT}{N_i}$$

donde TT es el tamaño de la tabla y N_i la cantidad de veces que es referenciado el bloque i por la tabla.

Las cubetas están organizadas por una clase lógica, *BlockTable*, que contiene un arreglo con un número (siempre potencia de 2) de estas. Esta clase es la encargada de manejar el comportamiento del hash, haciendo que este sea extensible.

Cuando un bloque se desborda, la tabla duplica su tamaño, copiándose a sí misma en la segunda mitad, generando así varias referencias a los mismos bloques (estos siguen conservando su tamaño de dispersión).

Tanto *Block* como *BlockTable* son persistidas en un archivo de bloques y en un archivo secuencial respectivamente. Se eligió usar un archivo secuencial ya que teóricamente el arreglo de bloques no suele tomar números incontrolables en memoria, sino unos pocos bytes. El flujo para agregar un registro es el siguiente:

1. Se aplica una función de dispersión, obteniendo así la posición en la tabla de bloques (BlockTable) donde se ubicaría el registro.
2. En dicha posición se encuentra un número de bloque dado.
3. Se busca el bloque en el archivo de bloques y se lo trae a memoria.
4. Una vez bufferizado, se verifica si hay tamaño para agregar un nuevo registro.
5. Si hay tamaño:

   a) se agrega el bloque al buffer.

   Sino, se presentan dos claras situaciones:

a) Si TD = TT (es el caso en que el bloque está siendo referenciado una sola vez):
    1) Se duplica la tabla.
    2) Se crea un nuevo bloque.
    3) Se lo agrega en la posición donde estaba el bloque anterior, mientras que el viejo queda en la segunda parte de la tabla.
    4) Se redispersan los registros (aplica la función hash) entre el bloque viejo y el nuevo bloque.
    5) Se agrega el registro en el bloque correspondiente.

b) Si TD != TT (el bloque está referenciado más de una sola vez):
    1) Se crea un bloque
    2) Se lo inserta en la posición donde estaba el bloque viejo y en las siguientes TD posiciones a su derecha y TD posiciones a su izquierda.
    3) Se redispersan los registros.
    4) Se agrega el registro en el bloque correspondiente.

6. Se persiste el buffer en memoria.
7. Una vez finalizado el proceso de inserción de registros, se persiste la tabla en el archivo secuencial.

Para la búsqueda el proceso es mucho más sencillo:

1. Se aplica una función de dispersión, obteniendo así la posición en la tabla de bloques (BlockTable) donde se ubicaría el registro.
2. En dicha posición se encuentra un número de bloque dado.
3. Se busca el bloque en el archivo de bloques y se lo trae a memoria.
4. Una vez bufferizado, se serializan los registros y se busca secuencialmente por el deseado.

4.4. Indexación

La indexación de un tema fue dividida en tres partes. Un índice de títulos, implementado en la clase "*IndiceTitulo*", un índice de autores, implementado en la clase "*IndiceAutor*" y un un índice de Recuperacion de texto, implementado en la clase "*RTTgenerator*".

4.4.1. Índice de Autor

El mismo esta basado en un Árbol B^+. En este último se almacenan registros con el formato

```
(idAutor, clave1, clave2, clave3, clave4, clave5, refLista)
```

Las claves representan referencias a posiciones de canciones en el archivo maestro. La "*refLista*" representa una referencia a una posicion al archivo de listas de autores. Se eligio esta solucion ya que es factible que existan muchas canciones del mismo autor.

4.4.2. Índice de Título

El mismo esta basado en un Hash Extensible. En el mismo se almacenan registros con el formato

```
(idTitulo, clave1, clave2, clave3, refLista)
```

Las claves representan referencias a posiciones de canciones en el archivo maestro. La "*refLista*" representa una referencia a una posición al archivo de listas de títulos. Se eligió esta solución ya que, si bien es poco probable, podría suceder que existan muchas canciones con el mismo título.

4.4.3. Índice de RTT

El mismo esta principalmente basado en un Árbol B^+. Además se tienen archivos auxiliares para guardar las listas invertidas de documentos y posiciones.

En el árbol se guardan registros con el formato

```
(idPalabra, refListaDocs)
```

En el archivo de listas de documentos se guardan registros con el formato

```
(idDoc, refListaPos)
```

En el archivo de listas de posiciones se guardan las posiciones donde aparece dicho término.

Cuando se quiere resolver una consulta primero se busca en el árbol todas las listas de documentos. Se intersectan las mismas para obtener aquellos documentos donde aparezcan todos los términos. Por último, utilizando el archivo de posiciones se verifica que las posiciones relativas de la consulta sean las mismas que las posiciones relativas en el archivo original.

4.5. Front-end (Interfaz)

La interfaz esta basada en modo texto a través de la salida estandar, en nuestro caso, la consola del sistema operativo. La misma se implementa en la clase "Menu". Se imprimen 5 opciones posibles:

- *Opción 1*: Indexar canciones from scratch;
- *Opción 2*: Indexar canciones en modo append;
- *Opción 3*: Buscar canciones por Autor;
- *Opción 4*: Buscar canciones por titulo;
- *Opción 5*: Buscar canciones por frase;
- *Opción 6*: Salir.

Las resoluciones de las consultas se imprimen por pantalla, mientras que las letras recuperadas se imprimen en el archivo "*salida.out*" el cual se almacena en la carpeta destino especificada en el archivo de configuración "*config.h*" (este se encuentra en el directorio *codigo*).