Administración de Bases de Datos

PRÁCTICA 1 – ORGANIZACIÓN DE FICHEROS

11 de abril de 2025

Prof. David Criado Ramón

Autor:

Ana Aragón Jerónimo

UNIVERSIDAD DE GRANADA

E.T.S. de Ingenierías Informática y de Telecomunicación

Índice

Introducción

Archivos Secuenciales Físicos

Un Archivo Secuencial Físico (ASF) es un archivo almacenado en disco que está compuesto por una secuencia de bloques, donde los registros se organizan según el orden físico en el que se insertaron.

Este tipo de organización permite almacenar registros de longitud fija en bloques homogéneos, facilitando una lectura secuencial eficiente. Se utiliza cuando el acceso a los datos se realiza en orden y no requiere una inserción o eliminación frecuente en posiciones intermedias del fichero.

Estructuras de datos necesarias

El TDA fileASF

VARIABLES Y CONSTRUCTORES

Clase FileHeader

Para la estructura de datos llamada fileHeader se ha utilizado una clase llamada FileHeader:

```
class FileHeader{
    private:
        int tam_file_header;    //Tamaño de la cabecera del fichero
        int n_block;            //número de bloques del fichero
        int n_reg;              //número de registros del fichero
        int tam_block;          //tamaño del bloque
        int tam_block_header;   //tamaño de la cabecera del bloque
        int tam_reg_header;     //tamaño de la cabecera del registro
        int n_atrib_reg;        //número de atributos del registro
        list<int> field_type;   //tipo de cada campo del registro
        list<int> tam_field;    //longitud (tamaño) de cada campo

    public:
        FileHeader();           //Constructor sin parámetros
        FileHeader(int tam_file_header, int n_block, int n_reg, int tam_block, int tam_block_header,
            int tam_reg_header, int n_atrib_reg, list<int> field_type, list<int> tam_field);
        //Constructor con parámetros
};
```

Los atributos son los siguientes:

-**int tam_file_header** → tamaño de la cabecera del fichero.

-**int n_block** → número de bloques del fichero.

-**int n_reg** → número de registros del fichero.

-**int tam_block** → número que contiene el tamaño del bloque (único para todos los bloques del fichero, al ser bloques homogéneos).

-**int tam_block_header** → número que contiene el tamaño de la cabecera del bloque (único para todos los bloques del fichero).

-**int tam_reg_header** → número que contiene el tamaño de la cabecera del registro (único para todos los registros del fichero).

-**int n_atrib_reg** → número que contiene el número de atributos del registro (único para todos los registros del fichero).

-**list<int> field_type** → lista que contiene el tipo de cada campo del registro (igual para todos los registros del fichero). Por ejemplo, el 0 se puede corresponder con lógico, 1 con carácter, 2 con entero, 3 con real y 4 con cadena de caracteres.

-**list<int> tam_field** → lista que contiene la longitud de cada campo del registro (igual para todos los registros del fichero).

Además, cuenta con 2 constructores, con y sin parámetros. (Los getters/setters se presuponen que vienen implementados).

Clase MemoryBlock

Para el bloque en memoria se han usado las siguientes clases:

```
class MemoryBlock{
    private:
        int id_block;           //cabecera del bloque --> identificador
        list<MemoryReg> regs;   //secuencia de registros

    public:
        MemoryBlock();          //Constructor sin parámetros
        MemoryBlock(int id, list<MemoryReg> regs);
        //Constructor con parámetros
};
```

La clase MemoryBlock, que cuenta con:

-**int id_block** → identificador del bloque.

-**list<MemoryReg> regs** → lista de registros, de la clase MemoryReg.

Además, cuenta con 2 constructores, con y sin parámetros. (Los getters/setters se presuponen que vienen implementados).

Clase MemoryReg

```
class MemoryReg{
    private:
        HeaderReg header;       //cabecera del registro
        list<int> fields;       //representa los valores de los campos

    public:
        MemoryReg();            //Constructor sin parámetros
        MemoryReg(int id, int status, list<int> fields);
        //Constructor con parámetros
};
```

La clase MemoryReg, que cuenta con:

-**HeaderReg header** → cabecera del registro, formada por 2 números enteros, almacenados en la clase HeaderReg.

-**list<int> fields** → lista con los valores de cada uno de los campos del registro.

También cuenta con 2 constructores, con y sin parámetros. (Los getters/setters se presuponen que vienen implementados).

Clase HeaderReg

```
class HeaderReg{
    private:
        int id_reg;             //identificador del registro
        int status;             //estado del registro

    public:
        HeaderReg();            //Constructor sin parámetros
        HeaderReg(int id, int status);
        //Constructor con parámetros
};
```

Y otra clase llamada HeaderReg, en la que se encuentran:

-**int id_reg** → identificador del registro dentro del fichero.

-**int status** → estado del registro (borrado-0, activo-1, y si hay más estados se les asignan los valores en orden creciente de la misma manera).

Al igual que las otras clases, cuenta con 2 constructores, con y sin parámetros. (Los getters/setters se presuponen que vienen implementados).

Clase FileASF

```
class FileASF{
    private:
        string diskFilename;    //cadena de caracteres diskFilename
        FILE* diskFile;         //estructura de tipo file (fichero)
        FileHeader fileHeader;      //estructura de datos fileHeader
        MemoryBlock currentBlock;//bloque de memoria
        bool headerSentinel;    //Centinela para el contenido de la cabecera
        bool blockSentinel;     //Centinela para el contenido del bloque memoria

        void readHeader();      //Lee la cabecera del fichero del disco
        void writeHeader();     //Escribe la cabecera del fichero en el disco
        void readBlock(int n);      //Lee un bloque del fichero en el disco
        void writeBlock(int n);     //Escribe un bloque del fichero al disco
        bool isValidBlock(int id);  //Comprueba si un bloque es válido
        void openFileASF();     //Abre el fichero
        void closeFileASF();    //Cierra el fichero

    public:
        FileASF();              //Constructor sin parámetros
        FileASF(string filename, FileHeader header);
        //Constructor con parámetros

        MemoryReg recoverReg(int id_reg);   //recupera el registro dado el identificador
        MemoryReg nextReg(int id_reg);      //da el siguiente registro dado el identificador
        void insertReg(MemoryReg reg);      //inserta un nuevo registro
        void updateReg(int id_reg, list<int> field_type, list<int> tam_field,
            int status);                    //actualiza un registro
        void readFileASF();                 //lee el fichero ASF (secuencialmente)
        void reorganizeFileASF();           //reorganiza el fichero ASF
};
```

La última clase es FileASF, siendo esta la clase principal. Los atributos serán los siguientes:

-**string diskFilename** → cadena de caracteres que almacena el nombre y ruta del fichero almacenado en disco que gestionará la estructura de datos.

-**FILE* diskFile** → puntero (tipo fichero) para gestionar el fichero almacenado en disco.

-**FileHeader fileHeader** → estructura de tipo FileHeader que almacena todo lo mencionado anteriormente en dicha clase.

-**MemoryBlock currentBlock** → bloque en memoria que almacena secuencialmente todo lo mencionado en dicha clase.

-**bool headerSentinel** → centinela que indica si el contenido de la cabecera del fichero ha cambiado desde la última escritura de bloque al fichero.

-**bool blockSentinel** → centinela que indica si el contenido del bloque en memoria ha cambiado desde la última escritura de bloque al fichero.

LISTADO DE OPERACIONES PÚBLICAS

El listado de operaciones públicas de esta clase son las que vienen a continuación:

```
public:
    FileASF();              //Constructor sin parámetros
    FileASF(string filename, FileHeader header);
    //Constructor con parámetros

    MemoryReg recoverReg(int id_reg);   //recupera el registro dado el identificador
    MemoryReg nextReg(int id_reg);      //da el siguiente registro dado el identificador
    void insertReg(MemoryReg reg);      //inserta un nuevo registro
    void updateReg(int id_reg, list<int> field_type, list<int> tam_field,
        int status);                    //actualiza un registro
    void readFileASF();                 //lee el fichero ASF (secuencialmente)
    void reorganizeFileASF();           //reorganiza el fichero ASF
```

-**MemoryReg recoverReg(int id_reg)** → este método se encarga de recuperar el registro dado el identificador de dicho registro.

-**MemoryReg nextReg(int id_reg)** → pasando como parámetro el identificador del registro, te da el siguiente registro (dado el orden secuencial del fichero).

-**void insertReg(MemoryReg reg)** → este método inserta el registro pasado como parámetro (colocándolo al final).

-**void updateReg(int id_reg, list<int> field_type, list<int> tam_field, int status)** dados los parámetros de identificador de registro, las listas de tipos de capo y de tamaño de campo, junto con el estado, se actualiza el registro identificado (se sobreescribe).

-**void readFileASF()** → este método permite leer el fichero (secuencialmente).

-**void reorganizeFileASF()** → reorganiza el fichero.

Al igual que todas las demás clases, también cuenta con 2 constructores, uno con parámetros y otro sin ellos. (Los getters/setters se presuponen que vienen implementados).

LISTADO DE OPERACIONES PRIVADAS

Por otro lado, también cuenta con un listado de operaciones privadas:

```
class FileASF{
    private:
        string diskFilename;    //cadena de caracteres diskFilename
        FILE* diskFile;         //estructura de tipo file (fichero)
        FileHeader fileHeader;      //estructura de datos fileHeader
        MemoryBlock currentBlock;//bloque de memoria
        bool headerSentinel;    //Centinela para el contenido de la cabecera
        bool blockSentinel;     //Centinela para el contenido del bloque memoria

        void readHeader();      //Lee la cabecera del fichero del disco
        void writeHeader();     //Escribe la cabecera del fichero en el disco
        void readBlock(int n);      //Lee un bloque del fichero en el disco
        void writeBlock(int n);     //Escribe un bloque del fichero al disco
        bool isValidBlock(int id);  //Comprueba si un bloque es válido
        void openFileASF();     //Abre el fichero
        void closeFileASF();    //Cierra el fichero
```

-**void readHeader()** → este método permite leer la cabecera del fichero del disco.

-**void writeHeader()** → escribe la cabecera del fichero en el disco

-**void readBlock(int n)** → lee un bloque del fichero en el disco.

-**void writeBlock(int n)** → este método permite escribir un bloque del fichero al disco.

-**bool isValidBlock(int id)** → con este método se comprueba si un bloque es válido o no.

-**void openFileASF()** → abre el fichero.

-**void closeFileASF()** → cierra el fichero.

El TDA fileRecord

Este TDA cuenta con una única clase, la ***clase FileRecord***.

```
class FileRecord{
    private:
        int id_reg;         //identificador del registro
        int status;         //estado del registro
        list<int> fields;   //representa los valores de los campos

    public:
        FileRecord();   //Constructor sin parámetros
        FileRecord(int id, int status, list<int> fields)
        //Constructor con parámetros
        void display() const;   //muestra el contenido del registro
        void updateFields(const list<int>& fields); //actualiza los campos (sobreescribiendo)
        void modifyField(int pos, int value);       //modifica un campo dada una posición
};
```

VARIABLES

Cuenta con los siguientes atributos:

-**int id_reg** → identificador del registro dentro del fichero.

-**int status** → estado del registro (borrado-0, activo-1, y si hay más estados se les asignan los valores en orden creciente de la misma manera).

-**list<int> fields** → lista con los valores de cada uno de los campos del registro.

LISTADO DE OPERACIONES PÚBLICAS

```
public:
    FileRecord();   //Constructor sin parámetros
    FileRecord(int id, int status, list<int> fields)
    //Constructor con parámetros
    void display() const;   //muestra el contenido del registro
    void updateFields(const list<int>& fields); //actualiza los campos (sobreescribiendo)
    void modifyField(int pos, int value);       //modifica un campo dada una posición
```

Por otro lado, además tiene un listado de operaciones públicas:

-**void display() const** → muestra el contenido del registro.

-**void updateFields(const list<int>& fields)** → actualiza los campos (sobreescribiéndolos).

-**void modifyField(int pos, int value)** → modifica un campo con el valor dado y en la posición indicada en los parámetros.

Esta clase también cuenta con 2 constructores, con y sin parámetros. (Los getters/setters se presuponen que vienen implementados).

Operaciones

Métodos públicos

¿CUÁLES REQUIEREN REALIZAR LECTURA DESDE EL DISCO O ESCRITURA AL DISCO?

La operación recoverReg requiere de lectura desde el disco.
Dado que cualquier registro puede encontrarse en cualquier bloque, será necesario acceder al bloque correspondiente para recuperar el registro. Como mínimo se leerá un bloque completo, que contiene múltiples registros. Los bytes leídos contienen los datos de todos los registros almacenados en dicho bloque.
La cantidad de bytes leídos será una cantidad comprendida entre:

- La sumatoria de los tamaños de los campos del registro.
- El producto de la sumatoria de los tamaños de los campos del registro, el número de registros y el número de bloques.

La operación nextReg requiere de lectura desde el disco.
Esta operación puede necesitar cargar en memoria el bloque siguiente si el registro buscado no se encuentra en el bloque actual. Como mínimo se leerá un bloque, que contiene los registros secuenciales.
La cantidad de bytes leídos será igual que la del método anterior.

La operación insertReg requiere de lectura y escritura en disco.
En primer lugar, puede leerse el último bloque para comprobar si hay espacio disponible. Después, se escribirá el bloque actualizado con el nuevo registro. Además, se actualizará la cabecera fileHeader incrementando el contador n_reg. Por tanto, se escribe una cantidad de bytes igual a la suma de los tamaños de los campos del registro.

La operación updateReg requiere de lectura y escritura en disco.
Al igual que en recoverReg, se debe localizar el bloque que contiene el registro a modificar, por lo que se leerá como mínimo un bloque. Luego, se sobrescribe el bloque actualizado con los cambios aplicados al registro.
La cantidad de bytes leídos será una cantidad comprendida entre:

- La sumatoria de los tamaños de los campos del registro.
- El producto de la sumatoria de los tamaños de los campos del registro, el número de registros y el número de bloques.

Y se escribirá una cantidad de bytes igual a la suma de los tamaños de los campos del registro.

La operación readFileASF requiere de lectura desde el disco.
Esta operación recorre todo el archivo secuencialmente, cargando cada bloque en memoria y mostrando sus registros. Se leerán todos los bloques existentes en el fichero, por lo que el número de bytes leídos será una cantidad comprendida entre:

- La sumatoria de los tamaños de los campos del registro.
- El producto de la sumatoria de los tamaños de los campos del registro, el número de registros y el número de bloques.

La operación reorganizeFileASF requiere de lectura y escritura en disco.
Esta operación tiene como objetivo eliminar registros obsoletos. Se leen todos los bloques y se reescriben solamente aquellos que contienen registros válidos. También se actualiza la cabecera fileHeader con los nuevos valores de n_block y n_reg. Por tanto, se leerán tam_block * n_block bytes, y se escribirán tantos bloques como se mantengan (menos o igual a n_block), además de tam_file_header.

¿CUÁLES REQUIEREN ESCRIBIR LA CABECERA FILEHEADER AL DISCO?

La operación insertReg actualiza fileHeader para incrementar el número de registros (n_reg). Se escribe una cantidad de bytes igual a tam_file_header, normalmente al inicio del fichero.

La operación reorganizeFileASF también actualiza fileHeader al modificar la cantidad de bloques y registros válidos tras la reorganización. También se escriben tam_file_header bytes.

¿CUÁLES REQUIEREN APERTURA/CIERRE DEL FICHERO DISKFILE?

Las operaciones recoverReg, nextReg, insertReg y updateReg requieren apertura del fichero en caso de no estar abierto previamente. Esto se debe a que necesitan cargar un bloque desde disco o escribir en él, y para ello se requiere que el fichero esté disponible.

Las operaciones readFileASF y reorganizeFileASF también requieren que el fichero esté abierto, ya que se accede a todos sus bloques. Si el fichero no está abierto previamente, estas operaciones lo abrirán. También requieren del cierre del fichero.

¿CUÁLES REQUIEREN DESPLAZAR LA POSICIÓN ACTUAL DEL FICHERO?

La operación recoverReg desplaza la posición al inicio del registro buscado. El desplazamiento es el producto de la suma de los tamaños de los campos del registro y el número de registros consultados hasta localizar el registro deseado. Este cálculo es tamRegistro * numRegistros, donde tamRegistro es la sumatoria de los tamaños de los campos del registro y numRegistros es la cantidad de registros consultados hasta alcanzar el registro que se quiere recuperar.

La operación nextReg puede provocar un desplazamiento similar si se necesita acceder a un nuevo bloque. Se calcula de la misma forma que el método anterior.

La operación insertReg no requiere desplazamiento.

La operación updateReg desplaza el puntero al bloque correspondiente al registro actualizado, usando el mismo cálculo que en recoverReg.

La operación readFileASF desplaza el puntero secuencialmente por todo el fichero, empezando en tam_file_header y avanzando de tam_block en tam_block.

Esta operación se utiliza para leer un archivo secuencialmente. El proceso comienza desplazando la posición actual del archivo al inicio del mismo, y luego va avanzando a medida que se leen los registros, uno por uno. Cada vez que se lee un registro, la posición se mueve de acuerdo con el tamRegistro (la sumatoria de los tamaños de los campos del registro) y el número de registros que se han leído hasta ese momento. Es decir, el desplazamiento se calcula como tamRegistro * numRegistros, donde tamRegistro es la suma de los tamaños de los campos del registro, y numRegistros es la cantidad total de registros leídos. Este desplazamiento asegura que la posición en el archivo se actualice correctamente para leer los registros secuencialmente hasta el final del archivo.

La operación reorganizeFileASF realiza desplazamientos múltiples para leer y escribir bloques válidos, recorriendo el archivo secuencialmente desde tam_file_header.

Cada vez que se lee un registro, la posición se mueve de acuerdo con el tamRegistro (la sumatoria de los tamaños de los campos del registro) y el número de registros que se han leído hasta ese momento. Es decir, el desplazamiento se calcula como tamRegistro * numRegistros, donde tamRegistro es la suma de los tamaños de los campos del registro, y numRegistros es la cantidad total de registros leídos. Este desplazamiento asegura que la posición en el archivo se actualice correctamente para leer los registros secuencialmente hasta el final del archivo.

Métodos privados

¿CUÁLES REQUIEREN REALIZAR LECTURA DESDE EL DISCO O ESCRITURA AL DISCO?

La operación readHeader realiza lectura desde el disco.
Este método se encarga de leer la cabecera del fichero, que tiene un tamaño de tam_file_header.

- Se leen exactamente tam_file_header bytes desde el inicio del fichero.
- Los datos leídos corresponden a la información de la cabecera, que incluye el número de bloques y número de registros.
- No se modifica el contenido del fichero, solo se cargan los datos en memoria.

La operación writeHeader realiza escritura en disco.
Este método actualiza la cabecera del fichero con nuevos valores (por ejemplo, al insertar o reorganizar).

- Se escriben tam_file_header bytes desde el inicio del fichero.
- El contenido escrito es la nueva versión de la estructura fileHeader.

La operación readBlock realiza lectura desde el disco.
Se encarga de cargar un bloque completo desde disco hacia la estructura block.

- Se leen tam_block bytes desde una posición determinada del fichero.
- La posición depende del número de bloque (i) y se calcula como tam_file_header + i * tam_block.
- Los datos leídos corresponden a los registros dentro de ese bloque.

La operación writeBlock realiza escritura en disco.
Escribe un bloque completo (almacenado en memoria en la variable block) en la posición correspondiente del archivo.

- Se escriben tam_block bytes desde una posición calculada igual que en readBlock.
- Los datos escritos contienen los registros actualizados de ese bloque.

¿CUÁLES REQUIEREN ESCRIBIR LA CABECERA FILEHEADER AL DISCO?

El único que lo hace es precisamente el método que realiza esta tarea, el método writeHeader(), pero ningún método privado realiza la escritura de la cabecera.

¿CUÁLES REQUIEREN APERTURA/CIERRE DEL FICHERO DISKFILE?

Ninguno de los métodos privados por sí mismo abre o cierra el fichero diskFile. Se asume que el fichero ya está abierto por el método público que los llama. Sin embargo, dependen de que el fichero esté abierto correctamente.

Por ejemplo:

- readHeader y writeHeader solo funcionarán si el fichero ya ha sido abierto en modo lectura o escritura.
- En caso de error en la apertura, estos métodos fallarían o retornarían algún indicador de fallo.

¿CUÁLES REQUIEREN DESPLAZAR LA POSICIÓN ACTUAL DEL FICHERO?

Las operaciones readHeader y writeHeader. Estos métodos desplazan el puntero al inicio del fichero con un offset de 0 bytes.

Las operaciones readBlock y writeBlock. Estos métodos desplazan el puntero a la posición del bloque deseado, calculando el desplazamiento como tam_file_header + i * tam_block (donde i es el número del bloque). Este desplazamiento se hace desde el principio del fichero, y los datos se leen o escriben en el bloque correspondiente.

> Como el fichero es secuencial físico, estos desplazamientos se realizan de forma progresiva en lectura o escritura, sin retroceso.