FUNDAMENTOS DE BASES DE DATOS

MEMORIA DE LA PRÁCTICA 3

Fecha de Entrega: 13/12/2024

Grupo 1203

Jaime Sánchez Esteban

Cristobal Revuelta Mayordomo

Tommaso Genovese

Índice:

1. Introducción

2. Material Presentado

3. Fichero Makefile

4. Estructura de la Práctica

5. Implementación de la lógica

    5.1 Create Table

    5.2 Create Index

    5.3 Replace Extension By IDX

    5.4 Print Tree

    5.5 Print Node

    5.6 Find Key

    5.7 Add Table Entry

    5.8 Add Index Entry

1. Introducción

En esta práctica se nos solicita el implementar un sistema gestor de bases de datos capad de operar con ficheros binarios.

Para ello, emplearemos un sistema simplificado en el cual usaremos índices para manejar la navegación entre la información y un fichero de datos que individualmente contiene todos los datos de la base de datos.

A su vez, se solicita la elaboración de una interfaz de usuario básica que permita a este realizar una serie de funciones sobre la propia base de datos.

2. Material Presentado

En la entrega hemos proporcionado, aparte de la memoria aquí presente, una serie de ficheros de prueba, los cuales se nos habían proporcionado previa a la realización de la práctica y tres ficheros de nombres menu.c, utils.c y utils.h que contienen el material que hemos implementado.

A su vez se proporciona un fichero makefile que permite ejecutar una serie de comandos que se explicarán en el próximo apartado.

3. Fichero Makefile

Como se comentó previamente, se proporciona junto a la práctica un fichero makefile.cod, el cual, a diferencia de un fichero habitual makefile se ejecuta con el comando:

```
make -f makefile.cod <comando>
```

Donde "<comando>" corresponde con una de las siguientes instrucciones:

- menu: Compila y crea el ejecutable correspondiente con el menú.
- tester: Compila y crea el ejecutable correspondiente con el programa de pruebas.
- clean: Elimina los ficheros de los programas.
- rm_databases: Elimina los ficheros .dat y .idx.
- clean_all: Elimina tanto ficheros de programas como de bases de datos.
- run_menu: Compila y ejecuta el menú
- run_tester: Compila y ejecuta el tester

4. Estructura de la Práctica

La práctica entregada contiene dos secciones definidas, el menú, contenido en menu.c, que contiene la interfaz de usuario y la lógica de la base de datos contenida en utils.c.

En cuanto al menú, este ofrece al usuario que escoja una de cuatro opciones, las cuales se explican a continuación:

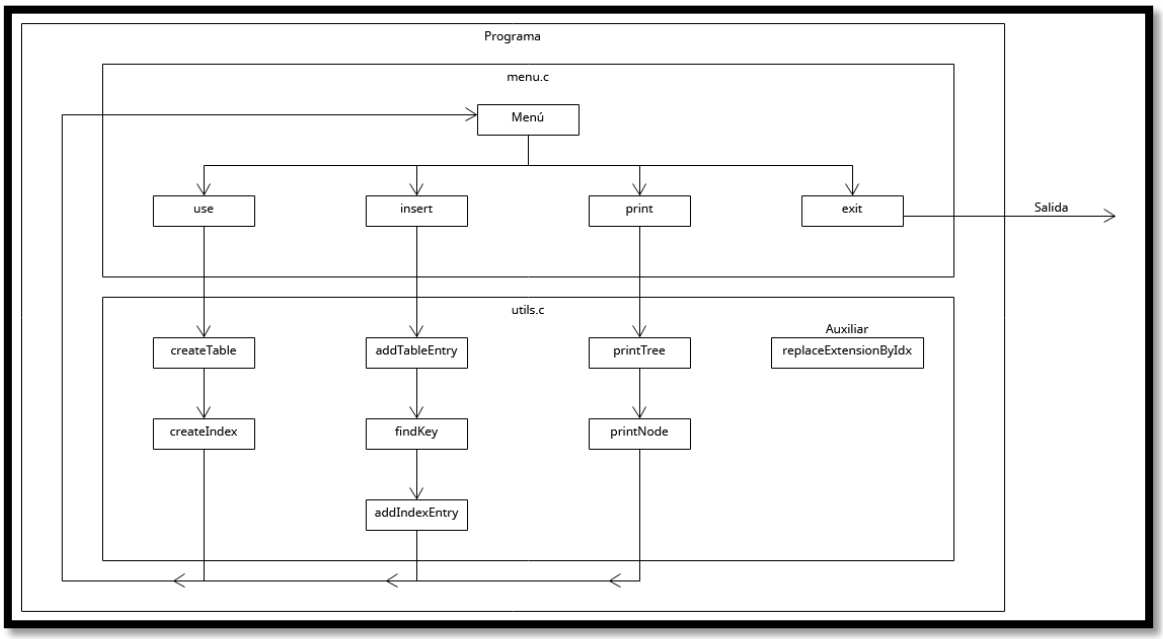
- use: Permite al usuario seleccionar una base de datos para operar sobre ella. En su forma inicial, el programa se inicia sin ninguna base de datos seleccionada, por lo que es necesario que se selecciona una empleando este comando previo a cualquiera de los otros (a excepción del de salida). Si se trata de ejecutar otro comando sin haber seleccionado base de datos se le advertirá al usuario y no se permitirá el proceder.
  En cuanto al comando en sí, se le solicitará al usuario que introduzca una cadena de caracteres correspondiente al nombre de la base de datos que quiere usar o crear, el cual debe de estar en un formato .dat.
  En caso de que el fichero seleccionado no contenga dicha terminación, se le avisará al usuario y se le dará la opción de añadirlo automáticamente y proceder o regresar al menú.
  Una vez seleccionado el nombre, el programa llamará a la función createTable y en caso de que no se produzcan errores asignará la base de datos solicitada/creada como la seleccionada y desbloqueará los otros comandos.

- insert: Permite al usuario agregar nuevos elementos a la base de datos. Para ello solicita un nombre para la clave primaria del elemento a almacenar y el título del elemento.
  Una vez obtenidos estos valores llamará a la función addTableEntry y almacenará el nuevo elemento.

- print: Permite al usuario generar una representación gráfica de los nodos de su base de datos. Previo a la generación, solicita al usuario que introduzca el nivel que se desea imprimir y procede a la generación.

- exit: Permite al usuario detener la ejecución del programa y salir de este.

A continuación, se adjunta una representación gráfica del procedimiento de ejecución de la práctica, el cual contiene las funciones empleadas y en que fichero se ubican:



5. Implementación de la lógica

En este apartado procederemos a explicar cómo hemos implementado las funciones empleadas para operar con la base de datos.

5.1 Create Table

Esta función recibe el nombre de la tabla que se desea abrir/crear y abre/crea el fichero de datos correspondiente, creando su cabecera si es necesario.

En primer lugar, se asegura de que el nombre solicitado contenga él .dat necesario para los ficheros de datos. Si el nombre es correcto procede a usar fopen para abrir el archivo si ya existe o crearlo en caso contrario.

Una vez abierto el fichero, comprueba que este no se encuentre vacio.

Si se encuentra vacío procede a añadir una cabecera con valor -1 que corresponde con el puntero a los registros borrados.

Tras esto, llama a createIndex para abrir/crear un fichero de índices equivalente.

Al retornar de esta llamada cierra el fichero y retorna.

```
bool createTable(const char *tablename)
{
    FILE *binary_file;
    int size = -1;
    char ret[BUFFLEN] = "DEFAULT";
    char dummy[BUFFLEN] = "DEFAULT";
    int len = 0;
    len = strlen(tablename);
    if (len >= BUFFLEN)
    {
        return false;
    }
    strcpy(dummy, tablename);
    strcpy(ret, tablename);
    if ((ret[len - 3] != 'd') && (ret[len - 2] != 'a') && (ret[len - 1] != 't'))
    {
        return false;
    }
    binary_file = fopen(dummy, "ab+");
    if (binary_file == NULL)
    {
        return false;
    }
    fseek(binary_file, 0, SEEK_END);
    size = ftell(binary_file);
    if (size == 0)
    {
        if (fwrite(&no_deleted_registers, HEADER_SIZE, 1, binary_file) != 1)
        {
            fclose(binary_file);
            return false;
        }
    }
    if (createIndex(dummy) == false)
    {
        fclose(binary_file);
        return false;
    }

    fclose(binary_file);
    return true;
}
```

5.2 Create Index

Esta función recibe el nombre del índice que se desea abrir/crear y abre/crea el fichero de índices correspondiente, creando sus cabeceras si es necesario.

En primer lugar comprueba que el nombre proporcionado corresponda con el tipo de archivo .idx. En caso contrario, llama a la función replaceExtensionByIdx para que lo cambie.

Una vez asegurado que el nombre es correcto, de forma similar a createTable, comprueba si el fichero se encuentra vacio y si este es el caso lo inicializa, añadiendo una cabecera compuesta por dos -1, correspondientes con el elemento raíz del índice y el puntero de registros borrados respectivamente.

Tras esto, cierra el archivo y retorna.

```
bool createIndex(const char *indexName)
{
    FILE *binary_file;
    int size = -1;
    char ret[BUFFLEN] = "DEFAULT";
    int len = 0;
    len = strlen(indexName);
    if (len >= BUFFLEN)
    {
        return false;
    }
    strcpy(ret, indexName);
    if ((ret[len - 3] != 'i') && (ret[len - 2] != 'd') && (ret[len - 1] != 'x'))
    {
        replaceExtensionByIdx(indexName, ret);
    }
    binary_file = fopen(ret, "ab+");
    if (binary_file != NULL)
    {
        fseek(binary_file, 0, SEEK_END);
        size = ftell(binary_file);
        if (size == 0)
        {
            size_t write_result = fwrite(&no_deleted_registers, sizeof(no_deleted_registers), 1,
binary_file);
            if (write_result != 1)
            {
                fclose(binary_file);
                return false;
            }
                write_result  =  fwrite(&no_deleted_registers,  sizeof(no_deleted_registers),  1,
binary_file);
            if (write_result != 1)
            {
                fclose(binary_file);
                return false;
            }
        }
    }
    else
    {
        return false;
    }
    fclose(binary_file);
    return true;
}
```

5.3 Replace Extension By IDX

Esta función recibe la cadena que se desea transformar y una segunda donde se almacenará la primera con la extensión modificada.

La función en si es muy simple y se emplea con propósito auxiliar, simplemente calcula la longitud de la primera cadena, la copia en la segunda y remplaza los tres últimos caracteres por ‘i’, ‘d’ y ‘x’ respectivamente.

```
void replaceExtensionByIdx(const char *fileName, char *indexName)
{
    int len = 0;
    len = strlen(fileName);
    strcpy(indexName, fileName);
    indexName[len - 3] = 'i';
    indexName[len - 2] = 'd';
    indexName[len - 1] = 'x';
    return;
}
```

5.4 Print Tree

Esta función recibe el nivel al que se desea buscar el árbol y el nombre del fichero de índices que lo contiene. Tras ello encuentra su raíz y envía a la función auxiliar printNode con los hijos derecho e izquierdo para que los impriman.

Para ello, nuevamente comprobamos que la extensión del nombre que se ha pasado como argumento es correcta, si ese es el caso abrimos el fichero en modo lectura y leemos el primer puntero para saber dónde se encuentra la raíz del árbol.

Una vez leída la posición, leemos el nodo raíz contenido en dicha posición y lo imprimimos en el formato solicitado. Tras esto, si el nivel solicitado es mayor que 0, buscamos en los hijos izquierdo y derecho llamando a la función printNodes, pasando como argumento a cada una la posición del nodo correspondiente.

Una vez se acaba la búsqueda de los hijos la función cierra el fichero y retorna.

```
void printTree(size_t level, const char *indexName)
{
    FILE *binary_file;
    char ret[BUFFLEN] = "DEFAULT";
    int len = 0, root = -1;
    Node node;
    strcpy(ret, indexName);
    len = strlen(indexName);
    if (len >= BUFFLEN)
    {
        return;
    }
    if ((ret[len - 3] != 'i') && (ret[len - 2] != 'd') && (ret[len - 1] != 'x'))
    {
        printf("The file extension is not the index one\n");
        return;
    }
    binary_file = fopen(ret, "rb+");
    if (binary_file == NULL)
    {
        printf("Error opening index file\n");
        return;
    }
    fseek(binary_file, 0, SEEK_SET);
    fread(&root, sizeof(int), 1, binary_file);
    if (root == -1)
    {
        printf("There is no root node, returning");
        return;
    }
    fseek(binary_file, 8, SEEK_SET);
    fseek(binary_file, root * ((4 * sizeof(int)) + 4), SEEK_CUR);
    fread(&node, sizeof(Node), 1, binary_file);
    printf("%.4s (%d): %d\n", node.book_id, root, node.offset);
    if (level > 0)
    {
        if (node.left != -1)
        {
            printnode(1, level, binary_file, node.left, 'l');
        }
        if (node.right != -1)
        {
            printnode(1, level, binary_file, node.right, 'r');
        }
    }
    fclose(binary_file);
    return;
}
```

5.5 Print Node

De forma similar a printTree, esta función imprime un nodo solicitado por su argumento node_id.

Cada vez que es llamado, comprueba en qué nivel se encuentra, y en caso de haber llegado a uno mayor del solicitado retorna sin imprimir nada.

Si por el contrario se encuentra dentro de un nivel en el cual puede imprimir, busca el nodo en cuestión y lo imprime, con el añadido respecto a printTree de que indica si se está imprimiendo un hijo derecho o izquierdo.

A su vez, en función del nivel en el que se encuentre, añadirá previo a la impresión una serie de entradas de tabulador para simular la profundidad en la impresión y mostrar más claramente las relaciones de padre-hijo.

Tras finalizar la impresión, se llama nuevamente a si misma para el hijo derecho o izquierdo, cambiando los argumentos de forma que el nuevo nivel sea reconocido como el actual más uno.

```
void printnode(size_t _level, size_t level, FILE *indexFileHandler, int node_id, char side)
{
    Node node;
    size_t i = 0;
    if (_level > level)
    {
        return;
    }
    fseek(indexFileHandler, 8, SEEK_SET);
    fseek(indexFileHandler, node_id * ((4 * sizeof(int)) + 4), SEEK_CUR);
    fread(&node, sizeof(Node), 1, indexFileHandler);

    for (; i < _level; i++)
    {
        printf("   ");
    }
    printf("%c %.4s (%d): %d\n", side, node.book_id, node_id, node.offset);
    if (node.left != -1)
    {
        printnode(_level + 1, level, indexFileHandler, node.left, 'l');
    }

    if (node.right != -1)
    {
        printnode(_level + 1, level, indexFileHandler, node.right, 'r');
    }

    return;
}
```

5.6 Find Key

Esta función busca en el fichero de índices si existe un determinado libro en función de su id, y en caso afirmativo devuelve el offset de datos de este.

Para ello, desde la raíz, realiza una búsqueda binaria de los diferentes nodos almacenados en el fichero, iterando hasta encontrar la posición donde supuestamente se debería encontrar el nodo solicitado.

Si no es posible encontrarlo, se devuelve en su lugar el node_id.

```
bool findKey(const char *book_id, const char *indexName, int *nodeIDOrDataOffset)
{
    FILE *binary_file = NULL;
    int root = -1;
    int ret;
    Node node;
    size_t offset = 0, r_size = 0;

    binary_file = fopen(indexName, "rb+");

    fseek(binary_file, 0, SEEK_SET);
    fread(&root, sizeof(int), 1, binary_file);
    if (root == -1)
    {
        (*nodeIDOrDataOffset) = root;
        return false;
    }
    r_size = sizeof(Node);
    *nodeIDOrDataOffset = root;

    while (true)
    {
        offset = (*nodeIDOrDataOffset) * r_size + INDEX_HEADER_SIZE;
        fseek(binary_file, offset, SEEK_SET);
        fread(&node, sizeof(Node), 1, binary_file);
        ret = strncmp(book_id, node.book_id, PK_SIZE);
        if (ret == 0)
        {
            (*nodeIDOrDataOffset) = node.offset;
            fclose(binary_file);
            return true;
        }
        else if (ret < 0)
        {
            if (node.left == -1)
            {
                fclose(binary_file);
                return false;
            }
            else
            {
                (*nodeIDOrDataOffset) = node.left;
            }
        }
        else
        {
            if (node.right == -1)
            {
                fclose(binary_file);
                return false;
            }
            else
            {
                (*nodeIDOrDataOffset) = node.right;
            }
        }
    }
    return true;
}
```

5.7 Add Table Entry

Esta función añade una nueva entrada en el fichero de datos.

Para ello busca en primer lugar, empleando findKey, si ya se encuentra registrado un elemento con la misma id. En caso positivo detiene la ejecución y avisa del fallo.

Si no se encuentra registrado, abre el fichero de datos y comprueba si hay registros borrados con tamaño libre suficiente. En caso afirmativo guarda en el registro encontrado la información y ajusta la cadena de borrados para que futuras inserciones no sean problemáticas.

En caso negativo simplemente lo almacena al final del fichero.

Una vez almacenado el nuevo elemento en el fichero de datos, llama a la función addIndexEntry para que cree la entrada en el fichero de índices correspondiente.

```
bool addTableEntry(Book* book, const char* dataName, const char* indexName) {
    FILE* binary_file = fopen(dataName, "rb+");
    if (!binary_file) {
        perror("Error opening file");
        return false;
    }
    int current_offset = 0, next_offset = 0, prev_offset = 0, new_deleted_offset = 0;
    size_t deleted_size = 0;
    bool found_deleted = false;
    if (findKey(book->book_id, indexName, &current_offset)) {
        handle_error("The book id is already registered", binary_file);
        return false;
    }
    fseek(binary_file, 0, SEEK_SET);
    if (fread(&current_offset, sizeof(int), 1, binary_file) != 1) {
        handle_error("Error reading the initial offset.", binary_file);
        return false;
    }

    while (current_offset != NO_DELETED_REGISTERS && !found_deleted) {
        fseek(binary_file, current_offset, SEEK_SET);
        if (fread(&next_offset, sizeof(int), 1, binary_file) != 1 || fread(&deleted_size, sizeof(int), 1, binary_file) != 1) {
            handle_error("Error reading deleted record information.", binary_file);
            return false;
        }

        if (deleted_size >= book->title_len + PK_SIZE + sizeof(int)) {
            found_deleted = true;
        } else {
            prev_offset = current_offset;
            current_offset = next_offset;
        }
    }
    if (!found_deleted) {
        fseek(binary_file, 0, SEEK_END);
    } else {
        fseek(binary_file, current_offset, SEEK_SET);
    }
    current_offset = ftell(binary_file);
    if (fwrite(book->book_id, PK_SIZE, 1, binary_file) != 1 || fwrite(&book->title_len, sizeof(int), 1, binary_file) != 1 || fwrite(book->title, book-
>title_len, 1, binary_file) != 1) {
        handle_error("Error writing the book entry to the file.", binary_file);
        return false;
    }

    if (found_deleted) {
        new_deleted_offset = ftell(binary_file);
        size_t new_deleted_size = deleted_size - book->title_len - (PK_SIZE + sizeof(int));
        if (new_deleted_size > 0) {
            if (fwrite(&next_offset, sizeof(int), 1, binary_file) != 1 || fwrite(&new_deleted_size, sizeof(int), 1, binary_file) != 1) {
                handle_error("Error updating deleted entry information.", binary_file);
                return false;
            }
            fseek(binary_file, prev_offset, SEEK_SET);
            if (fwrite(&new_deleted_offset, sizeof(int), 1, binary_file) != 1) {
                handle_error("Error writing updated deleted entry information.", binary_file);
                return false;
            }
        } else {
            fseek(binary_file, prev_offset, SEEK_SET);
            if (fwrite(&next_offset, sizeof(int), 1, binary_file) != 1) {
                handle_error("Error writing updated deleted entry information.", binary_file);
                return false;
            }
        }
    }
    if (!addIndexEntry(book->book_id, current_offset, indexName)) {
        handle_error("Error updating the index file.", binary_file);
        return false;
    }
    fclose(binary_file);
    return true;
}
```

5.8 Add Index Entry

Esta función almacena en el fichero de índices una nueva entrada.

Si el fichero de índices no cuenta con ningún otro elemento almacenado previamente, se asignará como raíz el nuevo nodo.

Tras esto, se procede a buscar una posición para los datos a almacenar, recorriendo el registro de borrados hasta encontrar uno que pueda servir. Una vez encontrado, se ajusta el puntero para que no sea problemático para futuras inserciones. Si no se encuentra se almacena al final del fichero.

Si hay un elemento raíz ya definido, realizará una búsqueda binaria partiendo de la raíz para encontrar en que posición del árbol se debería almacenar.

Una vez encontrada dicha posición, almacena en el padre el offset correspondiente al suyo para contar como uno de los hijos y guarda en su puntero parent cual es el valor correspondiente al offset del padre.

```
bool addIndexEntry(char *book_id, int bookOffset, const char *indexName) {
    int root, current, parent, cmp, ret, del, delaux, loop = 0, right = 2;
    FILE *indexFile = fopen(indexName, "rb+");
    Node newNode = {0};
    Node currentNode;
    if (!indexFile) {
        printf("Error opening index file");
        return false;
    }
    fseek(indexFile, 0, SEEK_SET);
    fread(&root, sizeof(int), 1, indexFile);
    strncpy(newNode.book_id, book_id, PK_SIZE);
    newNode.left = -1;
    newNode.right = -1;
    newNode.parent = -1;
    newNode.offset = bookOffset;
    if (root == -1) {
        root = 0;
        fseek(indexFile, 0, SEEK_SET);
        fwrite(&root, sizeof(int), 1, indexFile);
        fseek(indexFile, INDEX_HEADER_SIZE, SEEK_SET);
        fwrite(&newNode, sizeof(Node), 1, indexFile);
    } else {
        current = root;
        parent = -1;
        do {
            fseek(indexFile, current * sizeof(Node) + INDEX_HEADER_SIZE, SEEK_SET);
            fread(&currentNode, sizeof(Node), 1, indexFile);
            parent = current;
            cmp = strncmp(book_id, currentNode.book_id, PK_SIZE);
            if (cmp == 0) {
                fclose(indexFile);
                return false;
            } else if (cmp < 0) {
                if (currentNode.left == -1) {
                    ret = ftell(indexFile) / sizeof(Node);
                    currentNode.left = ret;
                    newNode.parent = parent;
                    loop = 1;
                    right = 0;
                }
                if (loop == 0) {
                    current = currentNode.left;
                }
            } else if (cmp > 0) {
                if (currentNode.right == -1) {
                    ret = ftell(indexFile) / sizeof(Node);
                    currentNode.right = ret;
                    newNode.parent = parent;
                    loop = 1;
                    right = 1;
                }
                if (loop == 0) {
                    current = currentNode.right;
                }
            }
        } while (loop == 0);
        fseek(indexFile, 4, SEEK_SET);
        fread(&del, sizeof(int), 1, indexFile);
        if (del != -1) {
            fseek(indexFile, del * sizeof(Node) + INDEX_HEADER_SIZE, SEEK_SET);
            fseek(indexFile, HEADER_SIZE, SEEK_CUR);
            fread(&delaux, sizeof(int), 1, indexFile);
            fseek(indexFile, del * sizeof(Node) + INDEX_HEADER_SIZE, SEEK_SET);
            fwrite(&newNode, sizeof(Node), 1, indexFile);
            fseek(indexFile, 4, SEEK_SET);
            fwrite(&delaux, sizeof(int), 1, indexFile);
        } else {
            fseek(indexFile, 0, SEEK_END);
            del = ftell(indexFile) / sizeof(Node);
            fwrite(&newNode, sizeof(Node), 1, indexFile);
        }
        if (right == 1) {
            currentNode.right = del;
            fseek(indexFile, (current * sizeof(Node)) + INDEX_HEADER_SIZE, SEEK_SET);
            fwrite(&currentNode, sizeof(Node), 1, indexFile);
        } else if (right == 0) {
            currentNode.left = del;
            fseek(indexFile, (current * sizeof(Node)) + INDEX_HEADER_SIZE, SEEK_SET);
            fwrite(&currentNode, sizeof(Node), 1, indexFile);
        } else {
            return false;
        }
        fseek(indexFile, del * sizeof(Node) + INDEX_HEADER_SIZE, SEEK_SET);
        fread(&newNode, sizeof(Node), 1, indexFile);
    }
    fclose(indexFile);
    return true;
}
```