PRÁCTICA 3 DE BASES DE DATOS

16/12/2022

PABLO ANTÓN ALAMILLO

EDUARDO JUNOY ORTEGA

LABORATORIO 08b

2.OBJETIVO DE ESTA PRÁCTICA.

El objetivo de esta práctica es comprender e implementar el uso de una base de datos mediante un árbol y sus índices.
Se debe lograr que los datos y los índices se almacenen en cada uno de sus ficheros correspondientes de manera continua para buscar e insertar información. De esta forma, el fichero se escribe tras una modificación en este y, tras una modificación o inserción en este fichero, se lee en el momento en el que sea necesario. Por lo tanto, el uso de los datos de este fichero puede ser compartido simultáneamente por distintos programas.

3.EXPLICACIÓN DE CADA ALGORITMO.

Funciones createTable y changeFileExtension:

```
bool createTable(const char * tableName){
    FILE *f = NULL;
    int deleted = NO_DELETED_REGISTERS;
    char * indexName = NULL;
    bool out;

    if(!tableName) return NULL;

    f = fopen(tableName, "rb");
    if(!f){
        f = fopen(tableName, "wb+");
        fwrite(&deleted, sizeof(int), 1, f);
    }
    if (!f)
    {
        return false;
    }
    fclose(f);

    indexName = changeFileExtension(tableName);

    out = createIndex(indexName);
    free(indexName);
    return out;
}

char *changeFileExtension(const char* fileName){
    int len;
    char *outName = NULL;
    outName = strdup(fileName);
    if(!outName) {
        return false;
```

```
    }
    len = strlen(fileName);
    outName[len-3] = '\0';
    strcat(outName, "idx");
    return outName;
}
```

Esta función crea una tabla. Para ello, abre el archivo dado y hace los respectivos controles de errores. A indexName se le asigna el resultado de la función auxiliar changeFileExtension, que modifica la extensión del archivo duplicando el nombre del fichero con strdup() y quitando los tres últimos caracteres para añadir el .idx con la función strcat(). Por último, libera memoria y devuelve el booleano que devuelve la función createIndex().

Función createIndex:

```
bool createIndex(const char * indexName){
    FILE *f = NULL;
    int deleted = -1;
    if(!indexName) return false;

    f = fopen(indexName, "rb");
    if(!f){
        f = fopen(indexName, "wb+");
        fwrite(&deleted, sizeof(int), 1, f);
        fwrite(&deleted, sizeof(int), 1, f);
    }
    if (!f)
    {
        return false;
    }
    fclose(f);
    return true;
}
```

Para crear un índice nuevo creamos el fichero que funcionará como índice, lo abrimos, hacemos los controles de errores correspondientes y lo cerramos. Si todo se ha realizado correctamente devuelve true.

Función findKey:

```
bool findKey ( const char * book_id , const char * indexName, int *
nodeIDOrDataOffset ){
    Node n;
    FILE *index = fopen(indexName, "rb");
    int cmp;
```

```
    int root;

    if(!book_id || !indexName || !nodeIDOrDataOffset) return false;

    fseek(index, 0, SEEK_SET);
    fread(&root, sizeof(int), 1, index);
    if(root == -1) return false;

    n = read_node(index, root);
    while(true){
        cmp = strncmp(book_id, n.book_id, PK_SIZE);
        if(cmp > 0){
            if(n.right == -1){
                (*nodeIDOrDataOffset) = (n.offset -
INDEX_HEADER_SIZE)/sizeof(Node);
                fclose(index);
                return false;
            }
            n = read_node(index, n.right);
        }
        else if(cmp < 0){
            if (n.left == -1)
            {
                (*nodeIDOrDataOffset) = (n.offset -
INDEX_HEADER_SIZE)/sizeof(Node);
                fclose(index);
                return false;
            }
            n = read_node(index, n. left);
        }
        else{
            (*nodeIDOrDataOffset) = n.offset;
            fclose(index);
            return true;
        }
    }
}
```

En primer lugar, nos aseguramos de que los argumentos de la función existen y no son erróneos, nos situamos en el offset inicial con las funciones fseek() y fread() para encontrar el nodo raíz, si existe. Después, almacena en el parámetro nodeIDOrDataOffset el identificador del último nodo visitado si no se ha encontrado la clave y devuelve false y el offset del registro de datos si se ha encontrado la clave, devolviendo true.

Función addTableEntry:

```
bool addTableEntry(Book * book, const char * tableName, const char *
indexName){
    FILE *table = NULL;
    int nodeIdOrDataOffset = 0;
    int bookOffset = 0;
    int deleted = 0;

    if(!tableName || !book || !indexName) return false;

    if(findKey(book->book_id, indexName, &nodeIdOrDataOffset)) return
false;

    table = fopen(tableName, "ab");
    if(!table) return false;

    /*reads the pointer to the deleted list*/
    fread(&deleted, sizeof(int), 1, table);
    /*there exists no delete funcion in this module so this
    condition is unlikely to be met*/
    if(deleted != -1){

    }
    else{
        /*finds the offset of the end of the file*/
        fseek(table, 0, SEEK_END);
        bookOffset = ftell(table);
    }
    writeData(book, table, bookOffset);

    return addIndexEntry(book->book_id, bookOffset, indexName);
}
```

En esta función se inicializa las variables necesarias con sus respectivos controles de errores. Después abre el fichero que contiene la tabla, lee el puntero a la lista eliminada, aunque esta condición sea muy poco probable. Si no hay una lista eliminada busca el offset del final del archivo. A continuación, escribe los datos del libro en el offset correspondiente a la tabla con writeData(). Finalmente, devuelve el resultado booleano de la función AddIndexEntry().

Función writeData:

```
void writeData(Book * book, FILE * table, int Offset){
    fseek(table, Offset, SEEK_SET);

    /*writes the book information to the table file*/
    fwrite(book->book_id, sizeof(char), PK_SIZE, table);
    fwrite(&book->title_len, sizeof(int), 1, table);
    fwrite(book->title, sizeof(char), book->title_len, table);
    fclose(table);
}
```

Esta función escribe todos los datos del diccionario necesarios en el buffer.

```
int get_book_size(Book* b){
    return PK_SIZE + sizeof(int) + b->title_len*sizeof(char);
}
```

También hemos creado esta función auxiliar para obtener el tamaño del libro.

```
bool addIndexEntry(char * book_id, int bookOffset, const char *
indexName){
    FILE *index = NULL;
    bool aux;
    int id;
    Node n;
    int root = 0;
    int offset;
    int deleted = -2;

    if (!book_id || bookOffset < 0  || !indexName) return false;

    aux = findKey(book_id, indexName, &id);
    if(aux) return false;

    index = fopen(indexName, "ab+");
    if(!index) return false;

    /*reads deleted*/
    fread(&root, sizeof(int), 1, index);
    fread(&deleted, sizeof(int), 1, index);
    if(root == -1){
        fseek(index, 0, SEEK_SET);
        fwrite(&root, sizeof(int), 1, index);
```

```
        strncpy(n.book_id, book_id, PK_SIZE);
        n.offset = bookOffset;
        n.right = -1;
        n.left = -1;
        n.parent = id;

        fseek(index, 0, SEEK_END);
        offset = ftell(index);

        /*write the data to the file*/
        write_node(index, &n, offset);
        return true;
    }
    /*there exists no delete funcion in this module so this
    condition is unlikely to be met*/
    if(deleted >= 0){
        n = read_node(index, deleted);
        /*update header*/
        fseek(index, sizeof(int), SEEK_SET);
        fwrite(&n.left, sizeof(int), 1, index);
        /*use the deleted node as an offset*/
        offset = deleted;
    }
    else{
        /*find the end of a file*/
        fseek(index, 0, SEEK_END);
        offset = ftell(index);
    }

    /*fill the data on the node to be inserted*/
    strncpy(n.book_id, book_id, PK_SIZE);
    n.offset = bookOffset;
    n.right = -1;
    n.left = -1;
    n.parent = id;

    /*write the data to the file*/
    write_node(index, &n, offset);
    return true;
}
```

Después de inicializar las variables y hacer controles de errores. Se busca el id del libro en el archivo y después de abrirlo lee los borrados con fread(). En el caso de que no exista un

nodo raíz, se sitúa en el offset inicial, crea un nodo raíz y escribe los datos en el fichero. Si hay algún borrado actualiza el encabezado y utiliza el nodo borrado como un offset. Si no, encuentra el final del archivo. Por último, guarda los datos en el nodo a insertar y estos en el fichero.

Función printTree():

```
void printTree(size_t level, const char * indexName){
    FILE *index = NULL;
    Node n;
    int root;
    char aux[PK_SIZE + 1];

    index = fopen(indexName, "rb");
    if (!index)
    {
        return ;
    }
    fread(&root, sizeof(int), 1, index);
    n = read_node(index, root);
    strncpy(aux, n.book_id, PK_SIZE);
    aux[PK_SIZE] = '\0';
    printf("%s (%d): %d\n", n.book_id, root, n.offset);

    if(n.left != -1){
        printf("\tl");
        _printTreeRec(level, 1, n.left, index);
    }

    if(n.right != -1){
        printf("\tr");
        _printTreeRec(level, 1, n.right, index);
    }
}
```

Esta función muestra por pantalla el árbol binario de búsqueda almacenado en el fichero de índice. Posteriormente, se realiza lo necesario para que la impresión del árbol tenga el mismo formato que el que se muestra en el enunciado de la práctica.

Función printTreeRec():

```
void _printTreeRec(size_t level, size_t depth, int node_act_id, FILE*
index){
    Node n;
```

```
    int i;
    char aux[PK_SIZE + 1];
    if(node_act_id == -1 || depth >= level) return;
    n = read_node(index, node_act_id);

    strncpy(aux, n.book_id, PK_SIZE);
    aux[PK_SIZE] = '\0';
    printf("%s (%d): %d\n", aux, node_act_id, n.offset);

    if(n.left != -1){
        for(i = 0; i<depth; ++i) printf("\t");
        printf("l");
        _printTreeRec(level, depth+1, n.left, index);
    }

    if(n.right != -1){
        for(i = 0; i<depth; ++i) printf("\t");
        printf("r");
        _printTreeRec(level, depth+1, n.right, index);
    }
}
```

Esta función es alternativa a la anterior, con la única diferencia que se ejecuta de manera recursiva.

Funciones read_node y write_node:

```
Node read_node(FILE *index, int node_id){
    Node n;
    fseek(index, INDEX_HEADER_SIZE + node_id*sizeof(Node), SEEK_SET);
    fread(&n, sizeof(Node), 1, index);

    return n;
}

void write_node(FILE *index, Node* n, int Offset){
    fseek(index, Offset, SEEK_SET);
    fwrite(n, sizeof(Node), 1, index);
}
```

Estas dos funciones se utilizan como funciones auxiliares para utilizarlas en las funciones anteriores. En el caso de la primera, realiza la función de leer un nodo. Sin embargo, la segunda escribe el índice de un nodo.

FUNCIONES DEL MENÚ:

Función get_input():

```
int get_input() {
    int nSelected = 0;
    char buf[16];

    do {

        printf(
        "1. Use\n"
        "2. Insert\n"
        "3. Print\n"
        "4. Exit.\n"
        );

        printf("Enter a number that corresponds to your choice > ");
        if (!fgets(buf, 16, stdin))
            /* reading input failed, give up: */
            nSelected = 0;
        else
            /* have some input, convert it to integer: */
            nSelected = atoi(buf);
        printf("\n");


        if ((nSelected < 1) || (nSelected > 4)) {
            /*printf("%d", nSelected);*/
            printf("You have entered an invalid choice. Please try
again\n\n\n");
        }
    } while ((nSelected < 1) || (nSelected > 4));

    return nSelected;
}
```

Esta función es similar a la de la anterior práctica. Muestra por pantalla el menú y solicita al usuario que seleccione una de las cuatro opciones. Si el usuario no escribe ninguna de esas cuatro opciones, se vuelve a preguntar al usuario volviendo a imprimir el menú por pantalla.

Función commandUse():

```
char* commandUse(){
    char *tableName = NULL;
```

```
    tableName = (char*)malloc(sizeof(char) * BufferLength);
    if(!tableName) return NULL;

    printf("Introduzca la tabla que quieres utilizar
(archivo.dat):\n");
    if (fgets(tableName, BufferLength, stdin)){
        tableName[strcspn(tableName, "\n")]= '\0';
    }
    printf("Nombre del .dat: %s\n", tableName);

    if (createTable(tableName) == false)
    {
        printf("Fallo en la ejecucion de createTable.\n");
        return NULL;
    }

    return tableName;
}
```

Esta función solicita al usuario que introduzca el fichero de datos de la tabla y comprueba que se crea una tabla en el fichero correctamente.

Función commandInsert:

```
int commandInsert(char *tableName, char *indexName){
    char *key;
    char *titulo;
    Book book;
    key = (char*)malloc(sizeof(char) * BufferLength);
    titulo = (char*)malloc(sizeof(char) * BufferLength);

    printf("Introduzca la clave a almacenar:\n");
    if(fgets(key, BufferLength, stdin)){
        key[strcspn(key, "\n")] = '\0';
    }
    printf("Clave: %s\n", key);

    printf("Introduzca el titulo a almacenar\n");
    if (fgets(titulo, BufferLength, stdin)){
        titulo[strcspn(titulo, "\n")] = '\0';
    }
    printf("Titulo: %s\n", titulo);

    strncpy(book.book_id, key, PK_SIZE);
```

```
    book.title_len = strlen(titulo);
    book.title = titulo;

    if (addTableEntry(&book, tableName, indexName) == false)
    {
        printf("Fallo en la ejecucion de addTableEntry.\n");
        return -1;
    }

    free(key);
    free(titulo);

    return 0;
}
```

Solicita al usuario la clave a introducir y el título de la obra a almacenar. Estos los mete en el correspondiente libro y comprueba que funciona correctamente addTableEntry().

Función commandPrint():

```
void commandPrint(char *indexName){
    size_t level;
    char buffer[BufferLength];

    printf("Introduzca nivel del arbol a imprimir:\n");
    if(fgets(buffer, BufferLength, stdin)){
        buffer[strcspn(buffer, "\n")] = '\0';
    }
    level = (size_t) atoi(buffer);
    printf("Arbol: \n");
    printTree(level, indexName);
}
```

Solicita al usuario el nivel del árbol a imprimir, continúa con un fgets() para guardar la respuesta del usuario e imprime el árbol correspondiente.