

MiniSQL 设计报告

小组成员：任浩然（组长）、刘振东、秦兆祥

课程：数据库系统原理

专业：计算机科学与技术

报告作者：任浩然

笔者主要工作量：B+Tree & Index Manager

2020-06-25

目录

一、 MiniSQL 系统概述

1. 背景
2. 功能描述
3. 运行环境与设计语言

二、 MiniSQL 系统结构设计

1. 总体设计
2. database 模块
3. API 与 interpreter 实现
4. RecordManager 实现
5. CatalogManager 实现
6. IndexManager 实现
7. BufferManager 实现

三、 测试样例与结果

四、 分组与设计分工

五、 实验心得

一、 MiniSQL 系统概述

1. 背景

1.1 编写目的

通过对 **MiniSQL** 的设计与实现，提高学生的系统学习能力，加深对数据库运行原理的理解。

1.2 项目背景

设计并实现一个精简型单用户 **SQL** 引擎 (**DBMS**) **MiniSQL**，允许用户通过字符界面输入 **SQL** 语句实现表的建立/删除；索引的建立/删除以及表记录的插入/删除/查找。

2. 功能描述

2.1 数据类型

支持三种基本数据类型：**int**，**float**，**char (n)**， $1 \leq n \leq 255$ 。

2.2 表定义

一个表最多可以定义 **32** 个属性，各属性可以指定是否为 **unique**；支持单属性的主键定义。

2.3 索引的建立和删除

对于表的主属性自动建立 **B+**树索引，对于声明为 **unique** 的属性可以通过 **SQL** 语句由用户指定建立/删除 **B+**树索引（因此，所有的 **B+**树索引都是单属性单值的）。

2.4 查找记录

可以通过指定用 **and** 连接的多个条件进行查询，支持等值查询和区间查询。

2.5 插入和删除记录

支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。

2.6 错误检查

- 支持表冲突检测，检索表冲突检测，表不存在检测，索引不存在检测
- 支持对主键与 **unique** 属性的查重
- 检测输入 **SQL** 指令合法性
- 支持对重复删除某条 **Record** 的检测

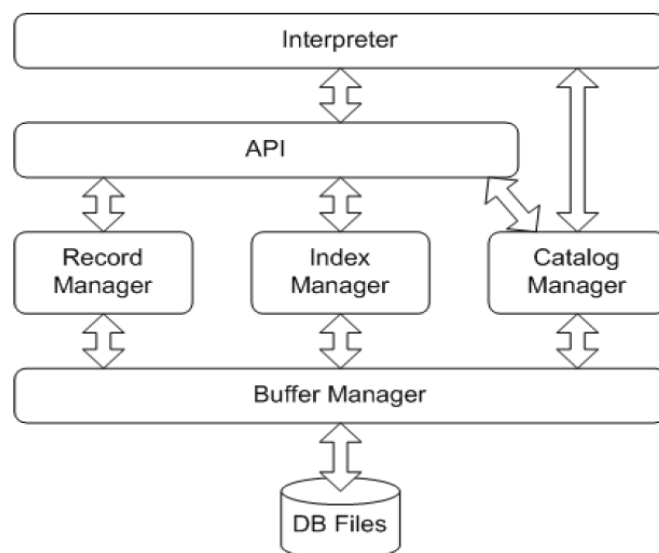
3. 运行环境与设计语言

本项目在 **Visuql Studio 2017** 完成编写与运行，全部采用 **C++** 语言进行设计

二、 MiniSQL 系统结构设计

1. 总体设计

本 **MiniSQL** 采用分层分模块设计方式，**Interpreter** 与 **UI** 完成与客户之间的指令信息传达；**API** 可根据 **Interpreter** 的结果实现对不同模块的调用；**Record Manager**, **Index Manager**, **Catalog Manager** 是对程序不同功能的具体实现；**Buffer Manager** 是整个程序的文件输入输出接口，控制着整个程序的写入写出；**DB files** 是程序的底层数据文件实现。



(图 2.1: minisql 设计框架)

2. Database 实现

2.1. 基础数据结构

在 DBMS 中,最重要的数据即为表的表示。表的构成非常复杂,就定义而言,其包括表的属性名称,表属性的数据类型,表的名称,表的元组数量。除此之外,表的数据项也是构成表的重要元素。

为了能设计出一个表类,我们首先定义了如下基础数据结构:

1)属性结构

```
1. struct Attribute{
2.     int AttrNum;                //the number of attributes
3.     short Flag[MAXATTRIBUTE];    //the flag of every attribute
4.     string AttributeName[MAXATTRIBUTE]; //the name of every attribute
5.     bool IsUnique[MAXATTRIBUTE]; //whether every attribute is unique
6. };
```

其中,Flag 表示每个属性的数据类型(-2 为 int, -1 为 float, 0—255 为 char(n)),AttributeName 数组表示属性名,IsUnique 数组表示属性是否 unique,AttrNum 表示总属性数。

2)索引结构

```
1. struct Index{
2.     int IndexNum;
3.     short Position[MAXINDEX]; //the location of the index in the attribute
4.     string IndexName[MAXINDEX];
5. };
```

IndexNum 表示索引数,Position 表示索引在属性列中的位置,IndexName 表示索引的名字。

我们分别使用上述一个结构来表示表中的对应信息。除此之外,我们还需考虑一个单独数据项的存储方式。注意到我们事先是不知道数据类型的,因此需要特殊的结构来存储数据。我们设计了如下的多态父类指针:

```
1. class MyData{
2. public:
3.     int Flag;
4.     //We use -2 to represent int type,
5.     -1 to represent float type,
6.     0 ~ 255 to represent char type
```

```

7.     virtual ~MyData() {}
8. };
9.
10. class MyInt: public MyData{
11. public:
12.     int value;
13.     MyInt(int x):value(x){
14.         Flag = -2;
15.     };
16.     virtual ~MyInt() {}
17. };
18.
19. class MyFloat: public MyData{
20. public:
21.     float value;
22.     MyFloat(float x):value(x){
23.         Flag = -1;
24.     };
25.     virtual ~MyFloat() {}
26. };
27.
28. class MyChar: public MyData{
29. public:
30.     string value;
31.     MyChar(string x):value(x){
32.         Flag = x.size();
33. //We use different Flag values to represent different length
34.     };
35.     virtual ~MyChar() {}
36. };

```

我们首先定义父类数据项 `MyData`，其中唯一变量 `Flag` 用于标识数据类型。然后分别设计三个子类，存储不同类型的数据。在数据生成时，我们只需要把返回的子类指针作为父类指针保存；而在访问时，可先通过访问父类的 `Flag` 来判断数据类型，再通过指针的强制类型转换访问对应的子类里的数据项。

3)where 结构

在 `select` 和 `delete` 操作中，需要分析 `where` 后面的语句，将其具体转化为结构，方便不同模块之间的参数传递。`Where` 结构定义如下：

```

1. typedef enum {
2.     eq, //equal

```

```

3.     leq, //less than or equal
4.     l,   //less than
5.     geq, //greater than or equal
6.     g,   //greater than
7.     neq  //not equal
8. }WHERE;
9.
10. struct where {
11.     MyData* data;
12.     WHERE flag;
13. };

```

其中 data 为数据项，flag 为枚举类型，表示比较连词，eq 表示等于，leq 表示小于等于，l 表示小于，geq 表示大于等于，g 表示大于，neg 表示不等于。

4)InsertPosition 结构

在执行选择、插入、删除等操作时，我们需要准确的定位每一个元组所在的位置，尤其是要直到该元组在硬盘中物理文件的位置以及在内存中的虚拟位置，为此，我们定义了 InsertPosition 结构来描述元组所在的位置。

```

1. struct InsertPosition{
2.     int BufferNum;        //The block number in memory
3.     int Position;        //The position in the block
4. };

```

该结构主要着重描述了元组在内存中的虚拟地址是多少。第一个整数表示了该元组在缓存区中区块的编号，第二个整数表示了该元组数据的起始地址在数据块中的偏移量（用字节数来表示）。虽然这个是数据在内存中的虚拟地址，但是可以很容易的通过 BufferManager 中的地址表来转换成相应的物理地址，两者的互相转换便于文件的读写操作的实习。

除此之外，在将元组插入索引时，还要将其转换成“长地址”，长地址(addr)的计算方法是：

$$addr = BlockOffset \times Position$$

在上述公式中，BlockOffset 指的是元组在物理文件中的块偏移量，这个值可以通过 BufferManager 来得到。每次向索引中插入 Key 或者获取 Key 的地址时都采用的是“长地址”，这样便于索引的存储，也便于数据的流通。

2.2 MyTuper 类与 MyTable 类

为了设计 `MyTable` 类，我们首先设计了 `MyTuper` 类，用于表示表中的一个元组。`MyTuper` 类的定义如下：

```
1. class MyTuper{
2. public:
3.     vector<MyData*> Data;
4.
5.     MyTuper(){}
6.
7.     MyTuper(const MyTuper& MT);
8.
9.     ~MyTuper();
10.
11.    int TuperLength() const
12.    {
13.        return (int)Data.size();
14.    } //Represent the length of the data
15.
16.    void AddData(MyData* NewData)
17.    {
18.        Data.push_back(NewData);
19.    } //Add a new data to the tuper.
20.
21.    MyData* operator[](unsigned short i);
22.    //Return the pointer to the specified data item.
23.    void DisplayTuper();
24.    //Display the data in the tuper
25. };
```

其数据项由 `MyData` 的指针型向量构成，为了便于后续操作，我们还为 `MyTuper` 类设计了几个主要的成员函数：`TuperLength()`用于返回元组长度，`AddData()`用于添加新数据项，`DisplayTuper()`用于显示元组数据，`[]`用于寻址。

在前面一系列基础结构下，我们设计了如下的 `MyTable` 类：

```
1. class MyTable{
2.
3. public:
4.     int BlockNum; //The number of blocks the table occupies
5.     string TableName; //The name of the table
6.     Attribute attributes; //Attributes in table
7.     vector<MyTuper*> tupers; //Tupers in table
8.     Index indices; //Indices in table
```



```

9.     int PrimaryLocation;
10.    //The location of primary key,
11.    -1 represent that there is no promary key

```

(此处省略了成员函数介绍)其中, BlockNum 用于记录该表所占据的数据块数, tupers 作为 MyTuper* 的向量, 用于记录表的成员数据, PrimaryLocation 记录主键的位置, indices 为索引信息, attributes 为属性信息, TableName 为表名。MyTable 类的构造函数如下:

```

1. MyTable(string s, Attribute attribute, int blocknum):
2.     TableName(s),attributes(attribute),BlockNum(blocknum)
3.     {
4.         PrimaryLocation = -1;
5.         for(int i = attributes.AttrNum;i < MAXATTRIBUTE;i++)
6.         {
7.             attribute.IsUnique[i] = false;
8.         }
9.         indices.IndexNum = 0;
10.    }

```

即只通过表名, 属性, 块数目初始化。在实际使用时, 我们会先构造空表, 然后调用其它成员函数进行主键和索引的设置。除一些基本的构造析构函数和信息访问函数外, 我们还设计了添加数据函数和显示函数以及用于计算数据大小的函数:

```

1. void DisplayTable();
2. void AddTuper(MyTuper* MT);
3. int GetAttributeSize() const
4. {
5.     return attributes.AttrNum;
6. }
7.
8. int GetTuperSize() const
9. {
10.     return (int)tupers.size();
11. }

```

2.3 异常类

由于程序的层次关系较为明显, 对于程序运行过程中可能遇到的问题, 我们不直接输出, 而采取抛出异常的方式交由上层处理。不同类中会定义不同类型的

异常：我们定义了表异常类如下：

```
1. class TableException: public exception{
2. public:
3.     TableException(string s):Message(s){}
4.
5.     string DisplayMessage()
6.     {
7.         return Message;
8.     }
9.
10. private:
11.     string Message;
12. };
```

3. API 与 interpreter 实现

API 完成了对用户输入指令的接受，并通过 interpreter 理解之后，调用到相对应的模块中，实现具体的函数。

```
class API {
private:
    CatalogManager API_Catalog;
    RecordManager API_Record;
    IndexManager API_Index;
    vector<MyTable*> MyTableDef;
public:
    API() {};
    ~API() {};

    void set_buffer(BufferManager& temp_buffer) {
        API_Catalog.set_Buffer(temp_buffer);
        API_Record.set_Buffer(temp_buffer);
        API_Index.set_Buffer(temp_buffer);
    }

    void api_insert_table(string tablename, MyTuper&tempmytupler);
    int Istable(string tablename);
    //check it exists the vector ,if not check buffer from Catalog
    void api_create_table(string tablename,Attribute &temp_attribue,string Pri_key);
    void api_create_index(string indexname, string tablename, string attrname);
    void drop_table(string tablename);
    void drop_table_fromDef(string TableName);
    void drop_index(string indexname,string tablename);
    void select_tuple_sql(string tablename);
    void select_tuple_sql(string tablename, WHERE tempwhere, string value, string AttriName);
    void delete_tuple_sql(string tablename);
    void delete_tuple_sql(string tablename, WHERE tempwhere, string value, string AttriName);
    string get_tablename_fromindex(string idnexname);
};
```

（图 3.1 API 类结构）

Interpreter 是对用户输入指令的理解与分析，包含对拼写检查，输入命令行检查等，并通过将结果传回 API 来实现对指令的正确实现。

```

class Interpreter {
private:
    API my_api;
public:
    Interpreter() {
    }
    ~Interpreter() {}
    void set_buffer(BufferManager& test_buffer) {
        my_api.set_buffer(test_buffer);
    }
    void insert_sql(string tablename, MyTuper& temptuple); //insert
    void Run();
    WHERE get_where(string test_str);
    void process_where(string test_str, int t, int type);
    //0 select
    //1 delete
    void create_table_sql(string tablename, Attribute& temp_Attribute, string Pri_key); //create table
    void create_index_sql(string indexname, string tablename, string attrname); //create index
    void test_str_process(string test_str, Attribute& temp_Attribute);
    void drop_table_sql(string tablename); //drop table
    void drop_index_sql(string indexname); //drop index
    void delete_tuple_sql(string tablename);
    void delete_tuple_sql(string tablename, WHERE tempwhere, string value, string AttriName);
    void select_tuple_sql(string tablename);
    void select_tuple_sql(string tablename, WHERE tempwhere, string value, string AttriName);

    //delete from
    //select*;
}

```

(图 3.2 Interpreter 类结构)

4. RecordManager 实现

Record Manager 是关于数据的查询、删除、添加重要的模块，其主要功能包括对数据的插入，对数据的查找与删除，值得注意的一件事是，**Record Manager** 会分析查找条件所依赖的属性是否有 **index** 索引，并以此来决定是否通过调用 **Index Manager** 来进行数据查询。

```

class RecordManager {
private:
    BufferManager* m_Buffer_manager;
    My_place the_place;
public:
    My_place get_insert_place() {
        return this->the_place;
    }
    RecordManager() {}
    ~RecordManager() {}
    void set_Buffer(BufferManager& temp_buffer) {
        this->m_Buffer_manager = &temp_buffer;
    }
    //My_Attri get_Attriindex(string tableName, string Attriname);

    int compare(WHERE tempwhere, char A[], char B[], int size);
    int get_blocknum(string tablename);
    int get_Attrinum(string tablename);
    void insert_tuple_table(MyTable& temp_table, MyTuper& temp_tuper);
    void create_table(MyTable& temp_table);
    void delete_from_table(string tablename, int blocknum);
    //void print(char temp[], int size);
    void select_or_delete_from_table(bool select, MyTable& test_table, WHERE tempwhere, string value, string Attriname);
    void serach_in_table(string tablename, MyTable& temp_table);
    void print(char test_char[], Attribute& test_Attri);
    int cmp_of_mine(char A[], char B[], int size);
    void check(MyTable& temp_table, MyTuper& temp_tuper);
    string get_table_define(MyTable& temp_table);
};

template<class T>
int compare_int_float(WHERE tempwhere, T A, T B);

```

(图 3.3 Record Manager 类结构)

5. CatalogManager 实现

Catalog Manager 模块是完成对表头信息的操控，其中表头信息主要包含该表的表名称、属性名称、属性个数与种类、索引的数目以及相对应的属性名、索引名等，是对一个表的完整信息存储模块，写出在 **head** 文件中。

```
class CatalogManager {
private:
    BufferManager* m_Buffer_manager;
public:
    CatalogManager() {};
    ~CatalogManager() {};
    MyTable* Get_My_Table(string tablename);
    void set_Buffer(BufferManager& temp_buffer);
    void create_table(MyTable& temp_table);
    string get_table_define(MyTable& temp_table);
    //string get_table_define(int tablename);
    void drop_table(string tablename);
    void create_index(MyTable& temp_table, string indexname, int indexAttri);
    void drop_index(MyTable& temp_table);
    //string find_table_name(string indexname);
};
```

(图 3.4 Catalog Manager 类实现)

6. IndexManager 实现

6.1 BTreeNode

BTreeNode 是 B+树的节点设计，主要包含三个部分：数据、指针、数据个数。另外，为了方便对整个 B+树的写出处理，**BTreeNode** 自带一个 **OutputBTreeNode** 函数，来实现对单个结点的写出操作。

BTreeNode 保存的元素是一个 **TupleData** 类，每一个 **TupleData** 包含三个基本项：数据、数据存放的块、数据的偏移量。数据本身是 **char** 类型保存的，为了贴合 **buffer manager** 的设计，使用时也需要进行必要的类型转换；**blocknum** 代表了数据存放的块，因为一个文件内容较大时，可能需要多个块来存储信息，此时我们需要标记数据存放在具体块的位置；**tuplenum** 代表了块内偏移量，当我们找到数据存放的块时，我们还需要在这个块内定位该数据，此时就需要 **tuplenum** 的帮助，完成对数据位置的查找。

在数据类型 **TupleData** 类中，有三个重载函数，分别用于判断两个数据的大小关系，值得注意的是，由于数据存放类型的不同，我们需要不同的类型转换函数来得到真实数据的大小，然后才可以进行大小比较。本次设计中采用了 **C-11** 标准中的 **reinterpret_cast** 智能指针，实现了对不同类型数据指针类型的转换，方便快捷的完成了运算符的重载操作。

```

struct TupleData {
    char tuple_data[20];           // store data
    int blocknum;                  // the blockoffset of this tuple
    int tuplenum;                  // the tuplenum in this block
    int flag;
    bool isdelete = false;
    bool operator < (const TupleData& cmp) { ... }
    bool operator == (const TupleData& cmp) { ... }
    bool operator > (const TupleData& cmp) { ... }
};

typedef struct BTreeNode* PtrtoNode;
struct BTreeNode {
    list<TupleData> data;           // store data
    list<PtrtoNode> ptr;           // store children ptr
    int ptrsum = 0, datasum = 0;    // value sum in each BTreeNode
    PtrtoNode pre, next, parent;    // ptrs
    void OutputBTreeNode(ofstream& outf);
};

```

(图 3.5 BTreeNode 结构)

6.2 B+ Tree

B+树是整个 **Index Manager** 最重要的部分，也是最基础的结构，在本次设计中，我们可以通过修改 **k** 值来实现对 **B+**树阶数的修改，以此达到查询效率的最大化。**B+**树类存储了该树所基于的表名称 (**table**)、属性名称 (**attributr**) 以及数据类型 (**flag**) 和阶数 (**k**)，每一个 **B+**树类型中只保存一个空的根节点 **Head**，并通过该根节点实现对整个 **B+**树的查询、遍历过程。

B+树重点的操作包括创建空树、插入、分裂、删除、查询和写出操作。其中一个辅助函数 **ParentEmpty** 函数是对插入操作的辅助，用于检验其是否存在一个未满的父节点，并决定插入的具体操作。

6.2.1 B+Tree Insert 实现细节

在 **B+**树的插入中,我们首先需要用 **Find** 函数找到对应数据插入的位置,然后进行判断:

- (1) 如果该节点未满,则直接将数据插入到该节点中;
- (2) 若该节点已满,但是存在一个父亲节点未满,则需要将这个节点至父亲节点中的每一个节点进行分裂 (**split**),然后将之前提到的未满足的父亲节点数据进行更新,不需要增加 **B+Tree** 的层数;
- (3) 若该节点已满且所有的父亲节点都已满,此时我们要对 **B+Tree** 的根节点进行调整,也就是 **B+Tree** 增加一层。实现细节为:从当前节点至根节点的所有结点都进行分裂,然后创建一个新的根节点,并将原来根节点分裂出来的两个结点作为新根节点的孩子。值得注意的一点在于,我的 **B+Tree Head** 是一个空结点,但是其结构与其他节点相同,这有助于我们 **split** 操作,也就是当我们对原先根节点进行 **split** 时,原 **Head** 结点中的数据数量+1,指针数量+1,我们此时可以将该 **Head** 理解为一个内部节点,所以只需新增一个空的 **Head** 结点即可,操作过程非常简单。

6.2.2 B+Tree Delete 实现细节

B+树的删除采用一个 **bool** 位 (**isdelete**) 来判断该节点是否被删除,所以并不是非常的高效,但是实现相对简便。

每一次删除仍是首先找到对应数据的存储位置,然后判断该节点存储的数据是否已经被删除。倘若已经被删除,则抛出异常;否则,将 **isdelete** 位置位,表及该数据。

6.2.3 B+Tree Find 实现细节

Find 函数是对 **B+Tree** 进行搜索的重要函数,该函数返回一个结点指针,请注意: **Find** 函数指对非叶子节点进行遍历,也就是返回的指针必定是一个叶子节点,检验是否存在数据需要对该叶子节点中的数据进行遍历,之后得到相应的结果。该实现方式有助于整体 **B+Tree** 的结构,在多个函数中使用,

容错率高、鲁棒性高。

```
class BTree {
private:
    int k; // define max values sum in each node
    int flag; // tuple data type
    string table; // on which table
    string attribute; // on which attribute
    string indexname;
    PtrtoNode Head; // head node is just a ptr
public:
    int isempty = 1;
    BTree() {} // default ctor
    BTree(string table, string attribute, string indexname, int flag, int k = 20) :table(table), attribute(attribute)
    , indexname(indexname), k(k), flag(flag) {
        Head = nullptr;
    }
    BTree() {
        this->OutputtoFile(this->indexname);
    }
    void Insert(TupleData& data); // B+Tree insert
    void InsertintoTNode(TupleData& data, PtrtoNode tmp);
    // inset into correct location in PtrtoNode tmp
    TupleData Delete(TupleData& data); // B+Tree delete
    PtrtoNode Find(TupleData& data); // B+Tree find certain data
    PtrtoNode ParentEmpty(PtrtoNode tmp); // check whether there is a parent node is not full and return PtrtoNode
    void Split(PtrtoNode tmp); // split TNode to two TNode;
    void OutputtoFile(string indexname);
};
```

(图 3.6 B+ Tree 类结构)

6.3 Index Manager

Index Manager 是对 B+树索引与其他模块之间搭建的桥梁，每一个 **IndexManager** 类中都包含一个内嵌的 **B+Tree**，以此来实现查询效率的最大化。

Index Manager 主要的功能是接受来自 **Catalog Manager** 与 **Record Manager** 的指令，完成对索引的创建、对现有索引的插入、删除、查找、写出工作，是整个 **minisql** 检索数据的核心。

笔者认为，**Index Manager** 最大的实现技术难点在于与其他模块的对接，也是整个小组作业完成中工作量最大、最耗时也最难确定的部分。举个例子，我们可能通常认为 **Index Manager** 在查询和删除时知道的是具体数据，但是可能因为其他模块接口的需要，你得到的参数并不一定令人满意，所以需要与组员之间更多的交流沟通，完成接口的规定与接口传参的处理。总而言之，最终版的 **Index Manager** 与笔者最初认为的有较大的不同，多了很多新功能，传参变化也较大，但是依旧是整个实验设计的核心所在。

```

class IndexManager {
private:
    BTree mybtree;           // IndexManager contains a btree
    BufferManager* buffer;
public:
    IndexManager() {}
    ~IndexManager() {}
    void CreateIndex(MyTable& table, string attr, string indexname); // indexname not have ".txt", just name
    void Search(char* value, MyTable& table, string& attr, string indexname, WHERE con, bool set);
    void BuildBTreeFromFile(string indexname); // indexname not have ".txt", just name
    PtrtoNode GetOneTNode(ifstream& inf, int flag);
    void DeleteAll(string indexname); // delete all data in this index
    void DropIndex(string indexname); // delete this index
    void set_Buffer(BufferManager& abuffer) { this->buffer = &abuffer; }
    void InsertBtree(MyTable& table, My_place& data, string indexname);
    void print(char* test_char, Attribute& test_Attri);
    void set0(TupleData& data, MyTable& table);
    string gettablename(string indexname) {...}
    bool IsRepeat(MyTable& table, string attr, char* value, string indexname);
};

```

(图 3.7 Index Manager 类结构)

7. BufferManager 实现

7.1 模块概述

BufferManager 是一个用于管理缓冲区的模块，被我们定义为一个硬件模块，也就是说我们要用软件来模拟这个模块实现这个模块的功能。他的主要功能就是负责内存与硬盘上文件和数据的交互。程序的所有的数据读取与写入都是直接在内存中的缓冲区进行的，操作完成之后由 BufferManager 来检查是否需要把相应的缓冲区块写回到文件中。

在每一个 BufferManager 中存放着多个 buffer，每个 buffer 对应着一块缓存区，为了保证与磁盘的交互效率达到最大化，我们将每个缓存区的大小设置为 4K，这样每次读取文件或者向缓存区中写数据时，操作的都是 4K 大小的文件块。每次要读取文件或者数据时，都要向 BufferManager 提出申请，如果 BufferManager 发现被请求的文件不在内存中，则将文件加载到内存中，然后在把内存块的编号返回给请求者。如果文件本身就在内存块中，则直接把内存块标号返回。

其他的模块获取到 BufferManager 返回的编号值以后，可以直接凭借这个编号访问对应的缓存取。类似于物理寻址的方式，BufferManager 会开辟一块专门的缓存区位置，然后所有的函数要访问数据时都需要凭借之前得到的内存块标号来在这个区域中存取数据。实现相应的功能。

7.2 主要功能

- 1) 当做整个程序与硬盘上数据的接口，每次需要读取硬盘上的文件或者是向硬盘上写文件时都需要与 `BufferManager` 进行交互。
- 2) 负责缓存区的管理，比如当某个缓存块要被替换时，要负责判断该块的内容是否需要被写回文件，并负责将脏数据写回文件，加载新的模块。在程序结束时也要把所有的脏数据写回文件中。
- 3) 模拟 LRU 算法，每次替换 block 时选择替换掉最近访问时间最早的块。

7.3 模块设计

```
1. class BufferManager{
2. public:
3.
4.     MyBuffer buffer[BLOCKNUM];
5.
6.     BufferManager();
7.
8.     ~BufferManager();
9.
10.    void WriteBlock_to_File(int BufferNum);
11.    //Write the content of block back to file,and initialize the block
12.
13.    int GetBufferNum(string filename, int BlockOffset);
14.    //Get the index of the designated block in memory
15.
16.    void ReadBlock(string filename, int BlockOffset, int BufferNum);
17.    //Read the file into block
18.
19.    void WriteBlock(int BufferNum);
20.    //We use it when the block is rewritten, and we symbolize it
21.
22.    int GetEmptyBuffer();
23.    //Get the empty buffer block in memory
24.
25.    int GetEmptyBufferExcept(string filename);
26.    //Get the empty buffer block in memory without substitute the file
27.
28.    InsertPosition GetInsertPosition(MyTable& table);
29.    //Return the available position of inserting data
30.
31.    int AddNewBlock(MyTable& table);
32.    //After inserting a new table file,
```

```

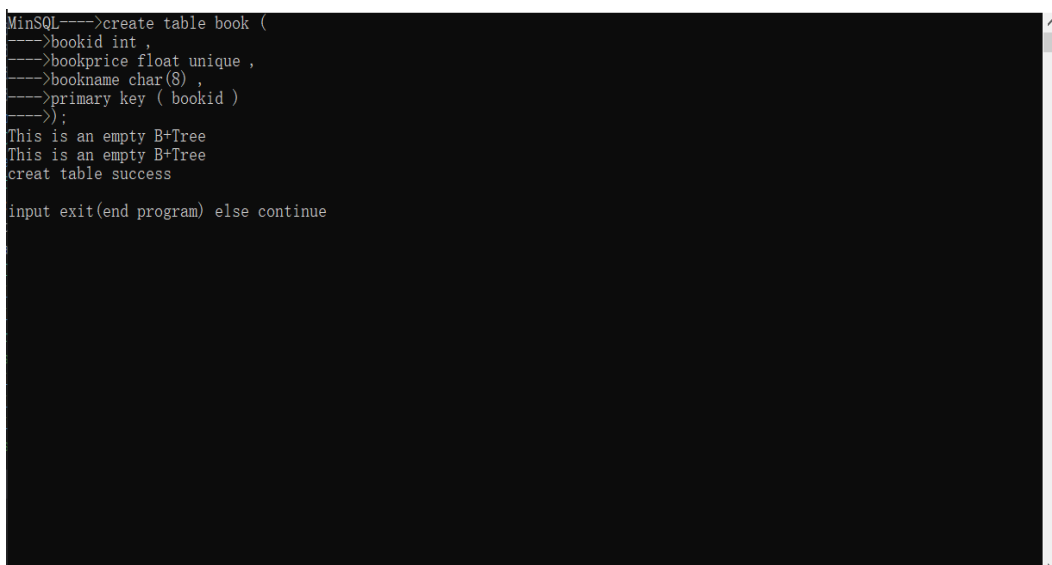
33.     return the updated index of block in memory
34.
35.     int Index_in_Buffer(string filename, int BlockOffset);
36.     //Get the index of one block in memory, return -1 if not find the block
37.
38.     void ScanWholeFile(MyTable& table);
39.     //Read the whole table file into memory
40.     (if the file is to large, it may cause collapse)
41.
42.     void SetInvalid(string filename);
43.     //Set the block of the file to invalid,
44.     used for deleting table and index
45.
46.     int AllocateBlock(string filename, int BlockOffset);
47. };

```

三、 测试样例与结果

创建表：

创建一个 **book** 表，该表同时具有 **int**、**float**、**char** 三种数据类型，体现数据类型定义的丰富性，此外，还对各属性是否 **unique** 进行了不同的设置，方便对表性能进行检验。



```

MySQL---->create table book (
---->bookid int ,
---->bookprice float unique ,
---->bookname char(8) ,
---->primary key ( bookid )
---->);
This is an empty B+Tree
This is an empty B+Tree
creat table success
input exit(end program) else continue

```

插入数据：

```
E:\My Information\study\数据库系统DB\实验lab\minisql\DBM\Debug\DBM.exe
MinSQL---->create table book (
---->bookid int ,
---->bookprice float unique ,
---->bookname char(8) ,
---->primary key ( bookid )
---->);
This is an empty B+Tree
This is an empty B+Tree
creat table success

input exit(end program) else continue
continue
MinSQL---->insert into book values (0,0.0,'abcdef') ;
insert success

input exit(end program) else continue
continue
```

删除数据:

```
E:\My Information\study\数据库系统DB\实验lab\minisql\DBM\Debug\DBM.exe
MinSQL---->create table book (
---->bookid int ,
---->bookprice float unique ,
---->bookname char(8) ,
---->primary key ( bookid )
---->);
This is an empty B+Tree
This is an empty B+Tree
creat table success

input exit(end program) else continue
continue
MinSQL---->insert into book values (0,0.0,'abcdef') ;
insert success

input exit(end program) else continue
continue
MinSQL---->delete from book where bookid = 0 ;
Find and delete!
delete from bookwhere ... success
delete success

input exit(end program) else continue
continue
```

查询:

```
E:\My Information\study\数据库系统DB\实验lab\minisql\DBM\Debug\DBM.exe
MinSQL---->insert into book values (1,1.1,'abcdef') ;
insert success

input exit(end program) else continue
continue
MinSQL---->insert into book values (2,2.2,'abcdef') ;
insert success

input exit(end program) else continue
continue
MinSQL---->insert into book values (3,3.3,'abcdef') ;
insert success

input exit(end program) else continue
continue
MinSQL---->insert into book values (4,4.4,'abcdef') ;
insert success

input exit(end program) else continue
continue
MinSQL---->select * from book ;
bookid  bookprice bookname
0        0         'abcdef'
1        1.1       'abcdef'
2        2.2       'abcdef'
3        3.3       'abcdef'
4        4.4       'abcdef'

input exit(end program) else continue
continue
```

删除表:

```
E:\My Information\study\数据库系统DB\实验lab\minisql\DBM\Debug\DBM.exe
input exit(end program) else continue
MinSQL---->insert into book values (3,3.3,'abcdef') ;
insert success
input exit(end program) else continue
MinSQL---->insert into book values (4,4.4,'abcdef') ;
insert success
input exit(end program) else continue
continue
MinSQL---->select * from book ;
bookid    bookprice bookname
0         0         'abcdef'
1         1.1       'abcdef'
2         2.2       'abcdef'
3         3.3       'abcdef'
4         4.4       'abcdef'
input exit(end program) else continue
continue
MinSQL---->drop table book ;
delete the table head file named book
delete the table file named book
delete the index named book_bookid_index
drop table success
input exit(end program) else continue
```

删除索引：

```
E:\My Information\study\数据库系统DB\实验lab\minisql\DBM\Debug\DBM.exe
input exit(end program) else continue
MinSQL---->insert into book values (2,2.2,'abcdef') ;
insert success
input exit(end program) else continue
MinSQL---->insert into book values (3,3.3,'abcdef') ;
insert success
input exit(end program) else continue
MinSQL---->insert into book values (4,4.4,'abcdef') ;
insert success
input exit(end program) else continue
continue
MinSQL---->select * from book ;
bookid    bookprice bookname
0         0         'abcdef'
1         1.1       'abcdef'
2         2.2       'abcdef'
3         3.3       'abcdef'
4         4.4       'abcdef'
input exit(end program) else continue
1
MinSQL---->drop index book_bookid_index
```

四、 分组与设计分工

任浩然：负责整体小组的分工、设计整个实验的模块细节、指定函数接口与底层接口、独立完成 B+Tree 与 Index Manager 模块。

刘振东：完成 Catalog Manager, Record Manager, api, Interpreter

秦兆祥：完成 Database 与 Buffer Manager

五、 实验心得

通过本次 minisql 实验，我挑战自我，独立完成了 B+Tree 的设

计，并担任小组长角色，统筹安排小组的分工与工作进程，是对自己的一次极大的挑战，同时我也收获了非常多，对 minisql 的理解深入了很多，收获颇丰。