浙江大学

数据库系统实验报告

作业名称：Minisql-Buffer&Record Manager

姓　　名：

学　　号：

电子邮箱：

联系电话：

指导老师：

2020 年 6 月 18 日

Buffer&Record Manager 设计报告

一、 实验目的

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。

通过对 MiniSQL 的设计与实现，提高学生的系统编程能力，加深对数据库系统原理的理解.

二、 实验需求

1. 数据类型

只要求支持三种基本数据类型：int，char(n)，float，其中 char(n)满足 1 <= n <= 255 。

2. 表定义

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

3. 索引的建立和删除

对于表的主键自动建立 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)。

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。

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。(where 条件是范围时删除多条)

三、 实验环境

1. 操作系统
   i. 名称：Microsoft Windows 10 Pro
2. 机器信息
   i. 系统类型：x64-based PC
   ii. 物理内存：16,343 MB

四、 模块设计

1. Record Manager
   i. 功能描述
      1. 创建/删除表或索引文件
      2. 插入记录
      3. 查找并显示记录

ii. 主要数据结构

1. 表的属性类

该结构中包括属性名、类型、是否唯一以及对应的 index 名。

该结构可以用来表示表中各个属性的具体信息。

```
class Attribute
{
public:
    string name;
    int type;                   //the type of the attribut
e,-
1 represents float, 0 represents int, other positive inte
ger represents char and the value is the number of char)
    bool ifunique;
    string index;          // default value is "", represe
nting no index
    Attribute(string n, int t, bool i);

public:
    int static const TYPE_FLOAT = -1;
    int static const TYPE_INT = 0;
    string indexNameGet(){return index;}

    void print();
};
```

2. 查找时的操作类

用于记录进行查找语句时的操作，共有 5 种不同的比较操作。

对于一组 condition，可以在一个 select 语句中加入多个条件，实现多个条件查找。

```
class Condition
{

public:
    const static int OPERATOR_EQUAL = 0; // "="
    const static int OPERATOR_NOT_EQUAL = 1; // "<>"
    const static int OPERATOR_LESS = 2; // "<"
    const static int OPERATOR_MORE = 3; // ">"
    const static int OPERATOR_LESS_EQUAL = 4; // "<="
    const static int OPERATOR_MORE_EQUAL = 5; // ">="

    Condition(string a,string v,int o);

    string attributename;
    string value;           // the value to be compared
```

```
    int operate;            // the type to be compared

    bool judge(int content);
    bool judge(float content);
    bool judge(string content);
};
```

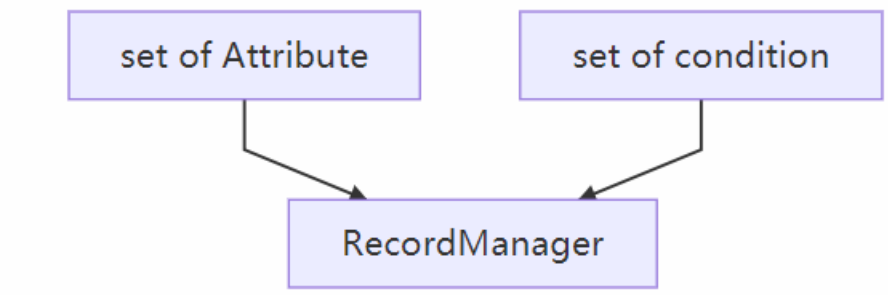
3. 管理所有 record 操作的类

Record Manager 负责管理记录表中数据的数据文件。主要功能为实现数据文件的创建与删除（由表的定义与删除引起）、记 录的插入、删除与查找操作，并对外提供相应的接口。其中记录的查找操作要求能够支持不带条件的查找和带一个条件的查找（包括等值查找、不等值查找和区间查找）。 数据文件由一个或多个数据块组成，块大小应与缓冲区块大小相同。一个块中包含一条至多条记录，为简单起见，只要求支 持定长记录的存储，且不要求支持记录的跨块存储。

```
class RecordManager{
public:
    RecordManager(){}
    BufferManager bm;
    API *api;

    int tableCreate(string tableName);
    int tableDrop(string tableName);

    int indexDrop(string indexName);
    int indexCreate(string indexName);

    int recordInsert(string tableName, char* record, int re
cordSize);

    int showRecordNum(string tableName, vector<string>* att
ributeNameVector, vector<Condition>* conditionVector);
    int recordBlockShow(string tableName, vector<string>* a
ttributeNameVector, vector<Condition>* conditionVector, int
 blockOffset);

    int recordAllFind(string tableName, vector<Condition>*
conditionVector);

    int recordAllDelete(string tableName, vector<Condition>
* conditionVector);
    int recordBlockDelete(string tableName,  vector<Conditi
on>* conditionVector, int blockOffset);
```

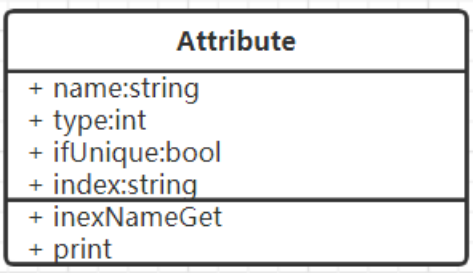
```
    int indexRecordAllAlreadyInsert(string tableName,string
 indexName);

    string tableFileNameGet(string tableName);
    string indexFileNameGet(string indexName);
private:
    int recordBlockShow(string tableName, vector<string>* a
ttributeNameVector, vector<Condition>* conditionVector, Blo
ckNode* block);
    int recordBlockFind(string tableName, vector<Condition>
* conditionVector, BlockNode* block);
    int recordBlockDelete(string tableName,  vector<Conditi
on>* conditionVector, BlockNode* block);
    int indexRecordBlockAlreadyInsert(string tableName,stri
ng indexName, BlockNode* block);

    bool recordConditionFit(char* recordBegin,int recordSiz
e, vector<Attribute>* attributeVector,vector<Condition>* co
nditionVector);
    void recordPrint(char* recordBegin, int recordSize, vec
tor<Attribute>* attributeVector, vector<string> *attributeN
ameVector);
    bool contentConditionFit(char* content, int type, Condi
tion* condition);
    void contentPrint(char * content, int type);
    int getTypeSize(int type);
    int getRecordSize(string tableName);
    int tableExist(string tableName);
    int getAttribute(string tableName, vector<Attribute>* a
ttributeVector);
    void recordIndexDelete(char* recordBegin,int recordSize
, vector<Attribute>* attributeVector, int blockOffset);
};
```

iii. 给出类图(如果有)，以及类间关系

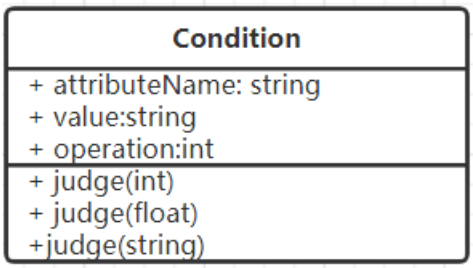


类图：

Attribute 类：



Condition 类：

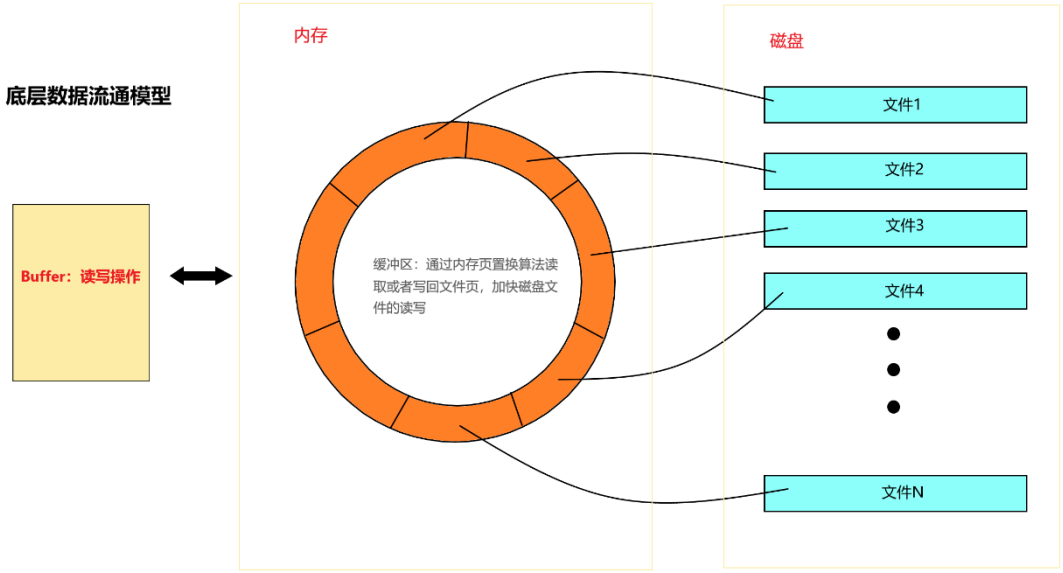


这两个类在 RecordManager 类中的函数里被使用。

2. Buffer Manager

i. 功能描述

1. 实现物理文件的读写
2. 使其他模块与物理磁盘完全隔开，使之完全转换为内存操作
3. 提供系统缓冲， 模拟操作系统内存管理功能，加快数据文件的读取和写入



ii. 主要数据结构

1. BlockNode

代表了一个内存块结构，包含块的各个属性

```
struct BlockNode
```

```
{
    int offset; // the offset number in the block list
    bool pin;  // the flag that this block is locked
    bool ifbottom; // flag that this is the end of the fi
le node
    string filename; // the file which the block node bel
ongs to
    friend class BufferManager;

private:
    char *address; // the content address
    bool reference; // the LRU replacement flag
    bool dirty; // the flag that this block is dirty, whi
ch needs to written back to the disk later
    size_t using_size; // the byte size that the block ha
ve used.
};
```

2. FileNode

   代表了一个内存中的文件及它的属性

```
struct FileNode
{
    string filename;
    bool pin; // the flag that this file is locked
    list<BlockNode> block_list; //the
block list for replacement
};
```

3. BufferManager

   缓冲管理器，提供相应的函数功能

   1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文件
   2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换
   3. 记录缓冲区中各页的状态，如是否被修改过等
   4. 提供缓冲区页的 pin 功能，及锁定缓冲区的页，不允许替换出去

```
class BufferManager
{
    private:
        FileNode *fileHead;
        FileNode file_pool[MAX_FILE_NUM];
        BlockNode block_pool[MAX_BLOCK_NUM];
        list<FileNode> file_list;
        int total_block; // the number of block that have
 been used, which means the block is in the list.
```

```
        int total_file; // the number of file that have b
een used, which means the file is in the list.
        void initBlock(BlockNode & block);
        void initFile(FileNode & file);
        BlockNode* getBlock(FileNode * file,BlockNode* po
sition,bool if_pin = false);
        void flushAll();
        void flush(string fileName,BlockNode* block);
        void cleanDirty(BlockNode &block);
        size_t getUsingSize(BlockNode* block);
        static const int BLOCK_SIZE = 4096;

    public:
        BufferManager();
        ~BufferManager();
        void deleteFileNode(string fileName);
        FileNode* getFile(string fileName,bool if_pin = f
alse);
        void setDirty(BlockNode & block);
        void setPin(BlockNode & block,bool pin);
        void setPin(FileNode & file,bool pin);
        void setUsingSize(BlockNode & block,size_t usage)
;
        size_t getUsingSize(BlockNode & block);
        char* getContent(BlockNode& block);
        static int getBlockSize() //Get the size of the b
lock that others can use.Others cannot use the block head
        {
            return BLOCK_SIZE - sizeof(size_t);
        }
        BlockNode* getNextBlock(FileNode * file,BlockNode
* block);
        BlockNode* getBlockHead(FileNode* file);
        BlockNode* getBlockByOffset(FileNode* file, int o
ffestNumber);

};
```

五、 模块实现

1. RecordManager 模块功能描述
   i. 表/索引文件的创建/删除

创建表/索引时新建文件，并检查是否顺利打开。删除表时，先删除内存中的文件结构，随后删除该文件。

表的创建和删除：

```
tableCreate( tablename )
{
    Create or open the file
    if (can't open the file successfully)
        return 0;
    else
    return 1;
}
tableDrop(tableName)
{
    Delete the file node in buffer manager
    if (can't remove the file)
        return 0;
    return 1;
}
```

ii. 插入记录

先在内存中获得对应的文件结构，随后遍历其中的块，直到找到一个块中剩余空间足够放下当前记录为止，将记录放入当前块中。

```
int RecordManager::recordInsert(string tablename,char* record
, int recorsize)
{
    Get the block head of the file
    while (true)
        if (block head is NULL)
            return -1;
        if (the space is enough)
            Put record in this block;
            Reset using size;
            Set block dirty;
            return offset;
        else
            Get next block node;
    return -1;
}
```

iii. 删除记录

对于一个内存块，得到它的内容，判断记录是否符合条件。如果记录符合条件，删除 index 中关于记录的内容，然后在 buffer 中移动内存内容，覆盖被删除内容所在内存，将后面的内存内容清空。最后设置脏块，标志块的内容被修改过。

```
int RecordManager::recordBlockDelete(string tableName,  vecto
r<Condition>* conditionVector, BlockNode* block)
{
    if (block head is NULL)
        return -1;
    Get attribute list in this table
    while (Not meet the tail)
        if(fit the condition)
            delete this record in index manager
            Refresh the buffer content;
            Set using size and dirty;

        else
            move to next record;
}
```

2. BufferManager 模块功能描述

i. 在内存中打开文件

在内存中找到一个文件结构存放当前文件的信息。如果已经存在该文件直接返回，否则，如果有空的文件结构则直接放入，如果没有，找到最长时间未使用的一个文件结构将内容写回，然后进行替换操作。

```
FileNode* BufferManager::getFile(string fileName, bool if_pin
)
{
    if(file list is not empty)
       if(file node exist)
           return node;
    if(have empty node)
           Add file in empty node;
    else
           Find the node didn’t be used with longest time and
not be locked;
        if(exist)
           Write back and replace;
        else
           throw error;
    return node;
}
```

ii. 获得可写入的内存块

寻找内存中是否还有未使用的块，如果有，直接返回该块，如果没有，在所有块中寻找未被锁定且可以进行替换的块。找到块后将内容写回原文件，将这个块加入新的文件列表中，对该块的内容进行初始化，并将新的文件中的内

容写入块中。

```
BlockNode* BufferManager::getBlock(FileNode * file,BlockNode
*position, bool if_pin)
{
    if(exist empty block)
        Find empty block;
        return block;

    else
         Find a block which can be replacement;
         Add the block into the block list;
        Set pin;
        Read the file content to the block;

    return btmp;
}
```

iii. 刷新内存

将被刷新部分的内容全部写回文件中。

```
void BufferManager::flush(string fileName,BlockNode* block)
{
    if(block is not dirty)
        return;
    else
         Open the file;
         Seek position for writing;
         Write back the content;

         return;
}
```

3. 测试方法

测试中利用已经初始化好的 attribute 及 condition 等结构，调用函数完成如下操作：

i. 新建表/索引文件
ii. 在表中放入数据，并读出观察是否成功放入文件
iii. 根据条件筛选记录，观察是否正确
iv. 删除部分记录，观察删除后文件中结果
v. 输入大量数据，观察数据是否成功写回
vi. 删除文件

六、 遇到的问题及解决方法

1. 写进文件的内容在读出时出错

解决方法：一开始没有使用二进制写模式打开文件，后更改模式。

2. 在删除记录后文件内容出错

解决方法：最开始只是将需删除的记录变为空，没有用后面的数据覆盖，导致读到空的条目以及最后一条记录没有被读取。改为所有数据向前覆盖再置最后一段内容为空

3. 操作 index 时和预期不符合
   没有统一好不同模块间的接口，检查函数接口和含义。

七、 总结

通过完成 buffer Manager 以及 Record Manager 模块，我接触到了数据库系统的底层交互方式，用文件模拟物理磁盘的读写工作，完成了一个简易的数据库底层搭建。其实在完成这部分内容时，对于 buffer 的替换策略选择实际上有很多，但是在这里只完成了一种很简单的顺序替换，没有考虑总的使用次数等因素。

希望后续有时间的话我能够完成更完善的模块或尝试完成整个工程。