Buffer 模块详细设计报告

一．设计概述

1. 目标：实现物理文件的读写，包括两类接口：
   一类实现对任意文件任意位置读写任意长度的数据
   另一类实现对固定长度数据的读写操作，并自动管理文件空间
2. 程序文件：Buffer.h Buffer.cpp
3. 功能：
   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
   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
   记录缓冲区中各页的状态，如是否被修改过等
   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
4. 接口实现：
   实现文件的创建和删除
   实现数据（记录）在文件中的读取和写入
   实现在文件中添加和删除数据时的空间自动管理提供对上层模块的统一接口
5. 工作原理：
   所有对于文件的读写都通过 Buffer 类提供的接口实现
   Buffer 类将磁盘文件以页的形式换入内存缓冲
   当待读写的文件数据所在的文件页之前已经被读入内存，则直接对该缓冲数据进行操作
   当待读写的文件数据所在的文件页不存在内存中：
   1. 如果内存缓冲页未满，则将需要读写的文件页读入内存缓冲页，执行读写操作
   2. 如果内存缓冲页已满，则通过内存页置换算法，将部分之前被使用的内存页写回到原文件，将待读写的文件页换入内存页
6. 文件格式：
   索引文件和数据文件有统一的文件格式
   每个文件都以若干固定大小的文件页的形式存在
   除了零号页面，所有文件页都由页头和数据区两部分组成
   页头由页号和常驻内存标志组成。
   页号表示该页在文件中的位置，常驻内存标志用于内存页置换算法
   零号文件页除了页号和常驻内存标志外，还拥有文件头信息数据区，文件头信息记录了该文件可以写入新数据的位置信息，以及被删除的数据的位置信息。此外，还有一块预留区域保留以后使用

二．实现详解

Buffer 模块主要由以下几个类实现：

1. PAGEHEAD ，作为页头信息，用以标识文件页。
2. FileAddr，作为文件地址，定位在文件中的位置。存储页的编号和页内偏移量。
3. FILECOND，作为文件头，用以存储文件中各页存储状况。
4. MemPage，作为内存页，即一个 MemPage 为一个固定内存页。存储文件号和文件页号，即指向的文件和在文件中为第几页，并且储存是否是最近一次访问内存和是

否为脏页信息，以完成页替换功能，同时该类存储实际物理文件地址。以及是否钉住的信息，实现钉住功能。

5. PageManager，作为内存页管理类，即该类存储所有页，当文件申请新页时，即从该类中提取页分配给文件。在该类中实现页置换算法。
6. MemFile，作为文件类，即将已经打开的物理文件提取到内存中，放入缓冲区。在该类中实现读写删除更新数据等功能。并且与 PageManger 一起实现申请页并将数据写入该页。
7. BUFFER，作为文件类的汇总，负责打开文件并存储已经打开文件，即所有其他模块均从该类进入，获得想要操作的文件。实现缓冲区的功能。

下面对几个主要模块的代码实现进行解释:

1. MemPage 类

```
class MemPage
{
    friend class MemFile;
    friend class PageManager;
    friend class BUFFER;
public:
    MemPage();
    ~MemPage();
private:
    void BacktoFile() const;            // 把内存中的页写回到文件中
    bool SetModified();                 // 设置为脏页
    bool SetPinned();
public:
    unsigned long fileId;               // 文件指针，while fileId==0 时为被抛弃的页
    unsigned long filePageID;           // 文件页号

    mutable bool bIsLastUsed;           // 最近一次访问内存是否被使用，用于PageManager算
法
    mutable bool isModified;            // 是否脏页
    mutable bool isPinned;             // 是否钉住
    void *PtrtoPageBegin;               // 实际保存物理文件数据的地址
    PAGEHEAD *pageHead;                 // 页头指针
    FILECOND* GetFileCond()const;       // 文件头指针（while filePageID == 0）
        };
```

以上是该类的定义。其中解释一下 mutable 去定义两个变量的原因。是因为在 BacktoFile 时加了 const 关键字，但是需要改变上述两个变量，所以需要有可变定义。

具体函数感觉只需要解释 BacktoFile 的实现即可。以下是具体实现:

```
void MemPage::BacktoFile() const
{
    // 脏页需要写回
    if (this->isModified && this->fileId>0)
    {
        int temp = 0;
```

```
            temp = lseek(this->fileId, this->filePageID*FILE_PAGESIZE, SEEK_SET);
            if (temp == -1)throw SQLError::LSEEK_ERROR();
            temp = write(this->fileId, this->PtrtoPageBegin, FILE_PAGESIZE); // 写回文件
            if (temp != FILE_PAGESIZE) throw SQLError::WRITE_ERROR();  // 写失败
            isModified = false;
            bIsLastUsed = true;
        }
}
```

在这里使用了两个库函数，lseek 和 write，lseek 主要是用于设置一个文件的偏移量，因为我们是每次从内存写回本地文件时，一次写一整页，而一个文件可能由很多页组成，所以在写之前需要找到该页的偏移量，当然也很好找，因为每次都写一页，所以偏移量为一页的 size 乘该页的 ID，write 即从该页的开始写起，写一页的 size 即可。同时需要将该页设为非脏页，因为已经写出，且设置为最近使用过的页。

2. MemFile 类

```
class MemFile
{
    friend class BUFFER;
    friend class BTree;
    friend bool DropTable(std::string table_name, std::string path);
public:
    const void* ReadRecord(FileAddr *address_delete)const;          // 读取某条记录,返回记录指针(包括记录地址数据)
    void* ReadWriteRecord(FileAddr *address_delete);          // 读取某条记录,返回记录指针(包括记录地址数据)
    FileAddr AddRecord(const void* const source_record, size_t sz_record);
// 返回记录所添加的位置
    FileAddr DeleteRecord(FileAddr *address_delete, size_t);                // 返回删除的位置
    bool UpdateRecord(FileAddr *address_delete, void *record_data, size_t record_sz);
    // 构造
    MemFile(const char *file_name, unsigned long file_id);
    // 写入数据
    void* MemRead(FileAddr *mem_to_read);                               // 读取内存文件,返回读取位置指针
    FileAddr MemWrite(const void* source, size_t length);              // 在可写入地址写入数据
    FileAddr MemWrite(const void* source, size_t length, FileAddr* dest);
    void MemWipe(void*source, size_t sz_wipe, FileAddr *fd_to_wipe);
    MemPage * AddExtraPage();                                           // 当前文件添加一页空间
    MemPage* GetFileFirstPage();                                        // 得到文件首页
    char fileName[MAX_FILENAME_LEN];
    unsigned long fileId;                                              // 文件指针
    unsigned long total_page;                                          // 目前文件中共有页
```

数

```
    };
```

该类实现较为复杂，首先从其存储数据看起。其存储 filename，fileid，和文件一共有多少页等信息。这些应该都很好理解。

函数 ReadRecord 即获得一个文件内容的地址，这里不是指单独的文件，而是文件中的一些内容的地址，代码也非常简单。

ReadWriteRecord 这个函数不同在于它读取的是将要用于改变的数据的地址，所以会多一步，即将该页设置为脏页，其余完全与上一函数相同。

MemRead 也与上述函数相同，其主要作用是在插入删除时读取作用，故需要将该页设置为 lastused。

插入与删除相似，在此主要拿一个函数详细说明：

```
FileAddr MemFile::AddRecord(const void* const source, size_t sz_record)
{
    auto pMemPage = GetGlobalPageManager()->GetMemAddr(this->fileId, 0);
    auto pFileCond = pMemPage->GetFileCond();
    FileAddr fd; // 写入的位置
    void *tmp_source;
    if (pFileCond->DelFirst.offSet == 0 && pFileCond->DelLast.offSet == 0)
    {
        // 没有被删除过的空余空间，直接在文件尾插入数据
        // 将添加的新地址作为记录数据的一部分写入
        tmp_source = malloc(sz_record + sizeof(FileAddr));
        memcpy(tmp_source, &pFileCond->NewInsert, sizeof(FileAddr));
        memcpy((char*)tmp_source + sizeof(FileAddr), source, sz_record);
        auto real_pos = MemWrite(tmp_source, sz_record+ sizeof(FileAddr));
        MemWrite(&real_pos,sizeof(FileAddr), &real_pos);
        fd = real_pos;
    }
    else if(pFileCond->DelFirst == pFileCond->DelLast)
    {
        // 在第一个被删除的数据处，填加新数据
        tmp_source = malloc(sz_record + sizeof(FileAddr));
        memcpy(tmp_source, &pFileCond->DelFirst, sizeof(FileAddr));
        memcpy((char*)tmp_source + sizeof(FileAddr), source, sz_record);
        MemWrite(tmp_source, sz_record+sizeof(FileAddr), &pFileCond->DelFirst);
        fd = pFileCond->DelFirst;
        pFileCond->DelFirst.offSet = 0;
        pFileCond->DelLast.offSet = 0;
    }
    else
    {
        auto first_del_pos = pFileCond->DelFirst;
        fd = pFileCond->DelFirst;
        pFileCond->DelFirst = *(FileAddr*)MemRead(&pFileCond->DelFirst);
```

```
            tmp_source = malloc(sz_record + sizeof(FileAddr));
            memcpy(tmp_source, &first_del_pos, sizeof(FileAddr));
            memcpy((char*)tmp_source + sizeof(FileAddr), source, sz_record);
            MemWrite(tmp_source, sz_record+sizeof(FileAddr), &first_del_pos);
        }
        delete tmp_source;
        pMemPage->SetModified();
        return fd;
}
```

以上是插入函数的详细代码。一开始所作是从全局的内存页中获得该文件的内存页的地址，而后获取该文件页文件头的信息，而后根据是否存在删除过的记录进行插入。这里使用 memcpy 函数进行对申请的内存赋值，使用两次的主要原因是第一次对文件头等部分进行赋值，而后赋值内容。

下面同时介绍 MemWrite 函数的作用。MemWrite 函数有两种，一种是在 FileAddr 所确定的地方写，另一种是在可以继续写入的地方写。可以看到，后面有 FileAddr 指针的 MemWrite 即为在 FileAddr 确定的地方写，这种情况是记录有删除的情况使用。而无记录删除即在可以写入地方写入的情况，只需调用不含 FileAddr 指针的函数即可。而在第一个部分中我们再次调用 MemWrite 其实并不是想要重写，而是将其设置为脏页和最近使用，且返回值为可写入的下一个位置。

删除函数相对较为容易理解,在此只说明一下 DelFirst 与 DelLast 的关系。首先 DelFirst 与 DelLast 都是 FileAddr 类型，即储存的是 fileid 与 offset，当未删除过时由于初始化两者均为 0，只需判断两者是否均为 0 即可判断一个文件是否存在删除过的记录，而后一旦有删除过的记录，第一次两者均相等，而从第二次开始，两者开始不相等，这里我们通过储存删除的记录的 FileAddr 来实现从 DelFirst 一路到 DelLast，即我们在每次储存 DelLast 应当指向的记录地址时候，将申请的 DelLast 设置为 offset 与 fileid 均为 0，但是本身 DelLast 还是指向删除记录的，这样前一个 Del 储存的内容是指向这个 DelLast 的，也就是说，形成一个类似链表的结构。这样既可以从 DelFirst 一路走到 DelLast，同时判断 DelFirst 与 DelLast 相等时条件不变，更加易于代码书写。

这里设置为 offset 为 0 的另一个重要原因是可以通过此来判断记录是否已经删除，因为我们可以通过 fileid 与 offset 知道一个 fileAddr 的物理地址，而这里我们存储的是实际内容，通过这样就会导致存储的 fileAddr 指向的物理地址的内容与存储的实际内容不等，也就可以判断这个记录是否已经删除。排除重复删除情况，因为这里理应使用惰性删除，后面还会用到这部分内存，不应当删除直接释放掉，我想了这么一招来判断是否记录已经删除。

3. PageManager 类

```
class PageManager
{
    friend class MemFile;
    friend class BUFFER;
    friend class BTree;
    friend bool InsertRecord(TB_Insert_Info tb_insert_info, std::string path);
    friend bool DropTable(std::string table_name, std::string path);
public:
    PageManager();
    ~PageManager();
    // 返回磁盘文件内存地址
    MemPage* GetMemAddr(unsigned long fileId, unsigned long filePageID);
```

```
    // 创建新页，适用于创建新文件或者添加新页的情况下
    MemPage* CreatNewPage(unsigned long fileId, unsigned long filePageID);
private:
    // 返回一个可替换的内存页索引
    // 原页面内容该写回先写回
    unsigned int GetReplaceablePage();
    // 如果目标文件页存在内存缓存则返回其地址，否则返回 nullptr
    MemPage* GetExistedPage(unsigned long fileId, unsigned long filePageID);
    MemPage* LoadFromFile(unsigned long fileId, unsigned long filePageID);
    // PageManager置换算法
    unsigned long PageManagerSwap();
    MemPage* MemPages[MEM_PAGEAMOUNT+1];  // 内存页对象数组
        };
```

之前提到该类是储存并管理整个内存页的类，其关键部分在于获取一个页与页替换的算法。这里就详细介绍一下。

首先是获取一个页:

```
MemPage* PageManager::GetExistedPage(unsigned long fileId, unsigned long filePageID)
{
    // look up for the page in memPage list
    for (int i = 1; i <= MEM_PAGEAMOUNT; i++)
    {
        if (MemPages[i] && MemPages[i]->fileId == fileId && MemPages[i]->filePageID ==
filePageID)
            return MemPages[i];
    }
    return nullptr;
}
```

以上是 GetExistedPage 函数的代码，很简单直观，就是遍历内存页检查是否存在。

```
MemPage* PageManager::LoadFromFile(unsigned long fileId, unsigned long filePageID)
{
    unsigned int freePage = GetReplaceablePage();
    MemPages[freePage]->fileId = fileId;
    MemPages[freePage]->filePageID = filePageID;
    MemPages[freePage]->isModified = false;
    MemPages[freePage]->bIsLastUsed = true;

    try {
        assert(fileId > 0);
        assert(filePageID>=0);
        long offset_t = lseek(fileId, filePageID*FILE_PAGESIZE, SEEK_SET);       // 定
位到将要取出的文件页的首地址
        if (offset_t == -1)throw SQLError::LSEEK_ERROR();
        long byte_count = read(fileId, MemPages[freePage]->PtrtoPageBegin,
FILE_PAGESIZE);          // 读到内存中
```

```
        if (byte_count == 0)throw SQLError::READ_ERROR();
    }
    catch (const SQLError::BaseError &e)
    {
        DispatchError(e);
    }
    return MemPages[freePage];
}
```

LoadFromFile 函数相对麻烦，因为首先他需要获取可以插入的新的页的 index，这里需要另一个函数 GetReplaceablePage()；我们下面再说。这里获取到可以写入的页之后，通过 lseek 与 read 函数读入内存，这之前有讲 write 与 lseek，这里与那里基本一致。

这样有这两个函数就可以获取文件的页了，生成新页的函数较为简单，不再详细叙述。

下面讲述 GetReplaceablePage()：

```
unsigned int PageManager::GetReplaceablePage()
{
    // 查找没有分配的内存页
    for (int i = 1; i <= MEM_PAGEAMOUNT; i++)
    {
        if (MemPages[i] == nullptr)
        {
            MemPages[i] = new MemPage();
            return i;
        }
    }
    // 查找被抛弃的页
    for (int i = 1; i <= MEM_PAGEAMOUNT; i++)
    {
        if (MemPages[i]->fileId == 0)
            return i;
    }
    // LRU算法
    unsigned int i = PageManagerSwap();
    if (i == 0)i++;
    MemPages[i]->BacktoFile();
    return i;
}
```

即首先判断是否有空页，而后判断是否有被抛弃的页，都没有即缓冲区满之后，开始用 LRU 算法替换页。

PageManagerSwap()：

```
unsigned long PageManager::PageManagerSwap()
{
    static unsigned long index = 1;
    assert(MemPages[index] != nullptr);
    while (true)     // 最近被使用过
```

```
    {
        if (!MemPages[index]->bIsLastUsed&&!MemPages[index]->isPinned)break;
        MemPages[index]->bIsLastUsed = 0;
        index = (index + 1) % MEM_PAGEAMOUNT;
        if (index == 0)index++;
    }
    auto res = index;
    MemPages[index]->bIsLastUsed = 1;
    index = (index + 1) % MEM_PAGEAMOUNT;
    if (index == 0)index++;
    return res;
}
```

只有不是最近使用过的块且不是 pinned 的页才可以拿出，这样就找到了应该被替换页的 index，用上述的函数即可实现页替换算法。并且对于脏页还要写回。

4. BUFFER 类

```
class BUFFER
{
    friend bool DropTable(std::string table_name, std::string path);
public:
    BUFFER() = default;
    ~BUFFER();
    MemFile* operator[](const char *fileName);      // 打开文件，打开失败返回 nullptr

    void CreateFile(const char *fileName);          // 创建文件，并格式化
    void CloseFile(const char *FileName);
    void CloseAllFile();
private:
    // 返回文件所映射的内存文件
    MemFile* GetMemFile(const char *fileName);
    std::vector<MemFile*> memFiles;  // 保存已经打开的文件列表
};
```

BUFFER 类相对较为简单，其本质就是一个整合，将所有打开的 memfiles 整合到一个 BUFFER 类中，各个函数非常简单明了，没有太多需要说明的地方，直接看代码即可：

```
MemFile* BUFFER::GetMemFile(const char *fileName)
{
    // 如果文件已经打开
    for (size_t i = 0; i < memFiles.size(); i++)
    {
        if ((strcmp(memFiles[i]->fileName, fileName) == 0))
            return memFiles[i];
    }
    // 文件存在但是没打开
    int PtrtoFile = open(fileName, _O_BINARY | O_RDWR, 0664);
    if (PtrtoFile != -1)
```

```
    {

        MemFile* newFile = new MemFile(fileName,PtrtoFile);
        memFiles.push_back(newFile);
        return newFile;
    }
    // 文件不存在
    return nullptr;

    }
```

该函数即获取文件，若没有即打开并加入缓冲区。其余几个函数与其基本一致，不再赘述，该类的主要用途就是存储文件，真正的困难的部分已经介绍过了。

稍作总结，以上就是 Buffer 模块的基本实现思路和主要代码解释，这一部分花费较多时间，并且与数据库书本知识也有不小联系，最困难的部分我已经进行了较为详细的说明，其余部分可进入工程查看，也有较为详细的注释。

record 模块详细设计报告

一．设计概述

1.目标：负责记录数据的读写

2.程序文件：Record.h Record.cpp

3.调用其他模块：Buffer模块

4.功能：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、更新与查找操作。

二．实现详解

该模块主要由以下几类实现：

1. KeyAttr 该类用来储存不同类型的变量，同时可以进行各个变量之间的比较操作。
2. Column_Cell 定义每个字段的单元数据字段单元记录了该字段的数据类型、数据的值、以及该字段的下一个字段指针，如果保存的字段是字符串类型，则字符串的前三个字符表示表创建时定义好字符串长度。
3. RecordHead 定义一条记录的头结构，唯一标志一条记录，记录的头记录了该记录的第一个字段的地址，记录各个字段以链表的形式形成整条记录，若是字符串类型字段，字符串前三个字符是表示该字段字符串的长度。
4. Record 定义所有的记录数据操作。记录读写包括索引部分(.idx)和数据部分(.dbf)。索引部分的修改由 B+树对应的模块负责，本模块的所有操作只修改记录的实际数据部分。该类所有的记录均以文件地址唯一标志。删除和查找记录的地址由索引树提供
5. CompareCell 储存比较的字段名称，比较的字段类型，比较关系，比较的值。

除开类外，为实现查找操作定义了三个函数，分别为：

```
std::vector<std::pair<KeyAttr, FileAddr>> Search(CompareCell compare_cell, std::string table_name, std::string path
= std::string("./"));
```

std::vector<std::pair<KeyAttr, FileAddr>> KeySearch(CompareCell compare_cell, std::string table_name, std::string path = std::string("./")); //索引查找

std::vector<std::pair<KeyAttr, FileAddr>> RangeSearch(CompareCell compare_cell, std::string table_name, std::string path = std::string("./")); // 遍历查找

这里我们直接实现区间查找与等值查找，无需重载函数，原因在于CompareCell这个类。首先讲述一下这个类的实现。

```
enum Operator_Type { B, BE, L, LE, E, NE };
Operator_Type GetOperatorType(std::string s);
// 比较的字段名称，比较的字段类型，比较关系，比较的值
class CompareCell                       //一个字段比较单元
{
public:
    CompareCell(Operator_Type t, Column_Cell cc) :OperType(t), cmp_value(cc) {}
    bool operator()(const Column_Cell &cc);
    Operator_Type     OperType;           //比较关系关系运算符
    Column_Cell       cmp_value;
};
```

可以看到，其储存的内容为Operator_Type与Column_Cell，Operator_Type是之前定义的枚举类，分别代表bigger，bigger and equal,less,less and equal,equal,not equal七种所有比较关系，Column_Cell是之后要介绍的类，其本质是记录字段的数据。这样，我们有了字段和比较关系，就可以在一个函数中完成不同的比较关系的处理。Search函数其实就是如果有索引则进行索引搜索，调用KeySearch函数，否则调用RangeSearch进行遍历查找。这里暂且不谈详细代码，先介绍其他类的实现。

```
class KeyAttr
{
public:
    using Key_Value = union {
        char                StrValue[ColumnNameLength];       //字符串指针
        int                  IntValue;                        //整形值
        double               DoubleValue;                      //浮点型值
    };
    Column_Type type;
    Key_Value value;

    bool operator<(const KeyAttr &rhs)const;
    bool operator>(const KeyAttr &rhs)const;
    bool operator==(const KeyAttr &rhs)const;
    bool operator<=(const KeyAttr &rhs)const;
    bool operator>=(const KeyAttr &rhs)const;
    bool operator!=(const KeyAttr &rhs)const;

};
```

以上是KeyAttr类的声明代码，我们使用联合体来降低占用内存，提高代码复用。除此之外进行了运算符重载，使得该类可以进行比较，这里的具体代码很容易理解，不放上来了就。

```
class Column_Cell
{
public:
    Column_Cell();
    Column_Cell(KeyAttr key);
    Column_Cell(const Column_Cell& rhs); // 拷贝构造
    Column_Cell& operator=(const Column_Cell&rhs); // 拷贝赋值
    Column_Cell(Column_Cell&& rhs) = delete; // 移动构造
    Column_Cell& operator=(Column_Cell&&rhs) = delete; // 移动赋值
    size_t size()const;
    void* data()const;
    ~Column_Cell();
    Column_Type column_type;
    std::string columu_name;
    Column_Value column_value;
    Column_Cell *next;
    size_t sz = 0;   // 保存字符串字段的长度
    // 类型转换
    operator KeyAttr()const;
};
```

以上是 Column_Cell 类的声明，正如之前所说，字段单元记录了该字段的数据类型、数据的值、以及该字段的下一个字段指针，构造函数等不必多说，size()即返回相应变量的长度。Data 即返回相应指针。

```
class RecordHead
{
public:
    RecordHead();
    RecordHead(const RecordHead &rhs); // 拷贝构造
    RecordHead& operator=(const RecordHead&rhs);  // 拷贝赋值
    RecordHead(RecordHead &&rhs); // 移动构造
    RecordHead& operator=(RecordHead&&rhs);  // 移动赋值
    ~RecordHead();

    void AddColumnCell(const Column_Cell &cc);
    size_t size()const;                          // 返回整条记录的大小
   Column_Cell* GetFirstColumn()const;

private:
    Column_Cell *phead;  // 指向记录的第一个字段
    Column_Cell *pLast;

};
```

以上是RecordHead类，定义一条记录的头结构，唯一标志一条记录记录的头记录了该记录的第一个字段的地址，记录各个字段以链表的形式形成整条记录。构造函数等无需多说，

在此说明一下AddColumnCell函数。

```
void RecordHead::AddColumnCell(const Column_Cell &cc)
{
    if (!phead)
    {
        phead = new Column_Cell;
        *phead = cc;
        phead->next = nullptr;
        pLast = phead;
    }
    else
    {
        pLast->next = new Column_Cell;
        *(pLast->next) = cc;
        pLast = pLast->next;
        pLast->next = nullptr;
    }
}
```

大致就是使用链表在尾部插入一个新Column_Cell。即生成链表操作。其余部分也非常简单，size()即返回整个record的大小，GetFirstColumn即返回头指针，指向第一个字段的指针。

```
class Record
{
public:
    // 插入新的记录，返回新插入记录的所在数据文件的地址
    FileAddr InsertRecord(const std::string dbf_name, const RecordHead &rd);

    // 删除记录，返回删除的记录所在数据文件的地址
    FileAddr DeleteRecord(const std::string dbf_name, FileAddr fd, size_t);

    // 更新整条记录,若是字符串类型字段，字符串前三个字符是表示该字段字符串的长度
    // 实际写入时并不将字符串的长度写进文件
    bool UpdateRecord(const std::string dbf_name, const RecordHead &rd, FileAddr fd);

private:
    // 将以链表的形式传入的记录数据读取并保存为一个整数据块以便于写入数据文件
    // tuple的第一个元素为记录数据的大小，第二个元素为数据的指针
    std::tuple<unsigned long, char*> GetRecordData(const RecordHead &rd);
};
```

以上是record类的声明，这里没有将查找记录放入record类中是因为查找与插入删除更新的操作有较大不同，查找相对困难许多并且需要用到其他模块的类，所以我们小组想直接放在API中。这里使用元组得到传出的data的值。

```
std::tuple<unsigned long, char*> Record::GetRecordData(const RecordHead &rd)
{
```

```
    // 记录数据的副本
    unsigned long data_size = rd.size();
    char *rd_data = (char*)malloc(data_size);
    memset(rd_data, 0, data_size);
    auto pcolumn = rd.GetFirstColumn();

    unsigned long offset = 0;
    while (pcolumn)
    {
        memcpy(rd_data + offset, pcolumn->data(), pcolumn->size());
        offset += pcolumn->size();
        pcolumn = pcolumn->next;
    }
    assert(data_size == offset);

    auto tp = std::make_tuple(data_size, rd_data);
    return tp;
}
```

如此，得到size和data。插入、删除与更新等基本相似，在此只详细介绍插入。

```
FileAddr Record::InsertRecord(const std::string dbf_name, const RecordHead &rd)
{
    // 记录数据的副本
    auto tp = GetRecordData(rd);

    // 插入记录
    auto fd =  GetGlobalFileBuffer()[dbf_name.c_str()]->AddRecord(std::get<1>(tp),
std::get<0>(tp));
    delete std::get<1>(tp);  // 释放副本内存
    return fd;
}
```

首先获得记录数据，存储在元组中，然后调用buffer模块，插入数据，即可完成一次插入数据。删除等也与此类似。查找函数由负责API同学实现，在此略过。

总结：

由于设计问题，导致我在写buffer时把许多record应该做的事也做了，导致record较为简单，但是本身这两者的功能合并起来是与要求无二的，且都是一个人，也就是我负责的模块。至于查找功能，在和小组成员商讨后决定放在API中，因为需要用到B+树，catalog，record等模块，相对较为复杂，为了实现更好的查找优化，决定放在API中，由小组另一个负责API的成员完善。

catalog 模块详细设计报告

三．设计概述

1.目标：负责管理数据库所有的模式信息

2.程序文件：catalog.h catalog.cpp

3.调用其他模块：Record模块 B+tree 模块

4.功能：

数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。

表中每个字段的定义信息，包括字段类型、是否唯一等。

数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

四．具体实现

首先是管理数据库信息的类:

```
// 目录定位和切换 用于数据库的使用
class CatalogPosition
{
    friend bool UseDatabase(std::string db_name, CatalogPosition &cp);
public:
    CatalogPosition();
    bool ResetRootCatalog(std::string root_new);  // 重置根目录
    void SwitchToDatabase();// 转到数据库列表目录下
    bool SwitchToDatabase(std::string db_name);// 转到具体的数据库下
    std::string GetCurrentPath()const;
    std::string GetRootPath()const;
    std::string SetCurrentPath(std::string cur);
    bool GetIsInSpeDb() { return isInSpeDb; }
    bool SetInInSpeDb(bool new_b) { isInSpeDb = new_b; return new_b; }
private:
    static bool isInSpeDb;          //是否在某个具体的数据库目录下
    std::string root; // 根目录，数据库文件的保存位置
    std::string current_catalog;
    };
```

在此，我们设计了可以创建多个数据库的功能，在这里实现时就需要对不同的数据库进行操作，这个类主要管理各个数据库的路径，同时直接用_access 函数判断是否存在相应数据库，然后返回值。

而关于表信息的管理我们分为了几类:

创建表信息，插入表信息，删除表信息，更新表信息，查找表信息。

每个基本理念都相似，在此只详细说明创建表信息。

```
    struct TB_Create_Info
{
    using ColumnInfo = struct ColumnInfo                // 新建表的字段信息
    {
```

```
        std::string name;
        Column_Type type;
        bool isPrimary;                                 // 是否主键
        bool isUnique;
        bool haveIndex;
        int length;                                     // 字段数据长度
    };
    std::string table_name;                             // 新建的表名
    std::vector<ColumnInfo> columns_info;               // 表的各个字段
};
```

在这里，我们使用这一的结果定义一个表的信息，可以看到，整个表的所有信息均可以获取到。然后定义一个检查函数，检查生成的结构是否符合。

```
void Check_TB_Create_Info(const TB_Create_Info &tb_create_info)
{
    // 表名
    std::string table_name = tb_create_info.table_name;
    std::string idx_file = GetCp().GetCurrentPath() + table_name + ".idx";
    // 判断表是否已经存在
    if (_access(idx_file.c_str(), 0) != -1) {  //表存在
        throw SQLError::TABLE_ERROR("The table already exists!");
    }

    // 检查每个字段的信息
    for (int i = 0; i < tb_create_info.columns_info.size(); i++)
    {
        auto &column = tb_create_info.columns_info[i];

        // 检查字段名称长度
        if (column.name.size() >= ColumnNameLength)
            throw SQLError::TABLE_ERROR("Error!Column name length overflow");

        // 检查字段类型
        if (column.type != Column_Type::C && column.type != Column_Type::D&&
column.type != Column_Type::I)
            throw SQLError::TABLE_ERROR("Column data type error!");
    }

    // 检查是否多个关键字
    int primary_count = 0;
    for (auto &e : tb_create_info.columns_info)
        if (e.isPrimary) primary_count++;

    if (primary_count > 1)
        throw SQLError::TABLE_ERROR("Error!More than one primary key!");
```

```
    // 检查字段个数
    if (tb_create_info.columns_info.size() > RecordColumnCount)
        throw SQLError::TABLE_ERROR("Error!Column count is overflow!");
}
```

首先要通过物理地址获取到文件是否存在，若不存在该表，则开始创建，顺便 check 一下各个字段类型信息等是否符合条件。

最后是索引的管理。

```
// 索引文件头信息管理类
class TableIndexHeadInfo
{
public:
    TableIndexHeadInfo(BPlusTree &_tree) :tree(_tree) {}
    // 表的字段个数
    size_t GetColumnCount()const;
    // 各个字段名字
    std::vector<std::string> GetColumnNames()const;
    // 各个字段类型
    std::vector<Column_Type> GetColumnType()const;
    Column_Type GetColumnType(std::string column_name)const;
    // 各个字段的大小
    std::vector<int> GetColumnSize()const;
    // 第i个字段的数据大小
    int GetColumnSizeByIndex(int i)const;
    // 主键字段的索引
    int GetPrimaryIndex()const;
    // 判断该字段名是不是表的字段
    bool IsColumnName(std::string column_name)const;
    // 返回该字段在所有字段中的索引位置
    int GetIndex(std::string column_name)const;
    // 判断字段名是否为主键字段
    bool IsPrimary(std::string column_name)const;
    // 返回给定字段名字段距离数据头地址的偏移
    int GetColumnOffset(std::string column_name);
private:
    BPlusTree &tree;
};
```

该索引文件信息包含了所有需要包含的内容，其实不止索引，也包含了其他内容。如字段名称等，也可以判断一个字段是否为主键。通过 record，获得各个字段的信息。实现相对较为简单，大多的获取信息和判断是否存在，没有特别的算法需要说明。

总结:

Catalog 模块我感觉是一个比较简单的信息储存模块，同时带有判断的功能，没有很特别的

算法，大部分代码没有需要详细说明的地方，可以较为轻松理解阅读。需要说明的我觉得是关于那个表信息的管理我们分为了几类，这里我的想法是便于 API 调用，因为本身我们这个模块就存储了各个表的信息，同时在这里进行一下表创建插入删除更新查找的检查的一部分是比较方便的，比如表是否已经存在于这个磁盘上了，表的内容是否不合适，表的字段等是否违反了规定等等，建立结构体是为了传出信息方便，因为一个表的插入需要的信息是不少的，而且创建插入删除更新查找这几个操作的每一个操作需要的内容都不尽相同，想要一致传入很困难，所以我就定义了结构体方便操作，这样也方便管理。虽然不是内容要求的，但是我觉得这样也是一种合理的优化。