实验名称: Project_MiniSQL

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 系统需求

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

一个表最多可以定义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（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

VScode

四 模块设计

本次小组项目中主要负责RecordManager、BufferManager、DB Files等模块设计

4.1 Record Manager

1) 功能描述

Record Manager 负责管理记录表中数据的数据文件，主要功能有：

- 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。

2) 主要数据结构

主要数据结构为vector，没有使用到自己定义的数据结构。

3) 设计的类和类间关系

这一模块设计的类是RecordManager，所调用的类有CatalogManager、HeapFile

4.2 Buffer Manager

1) 功能描述

Buffer Manager 负责缓冲区的管理，主要功能有：

- 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，不允许替换出去为提高磁盘I/O 操作的效率，缓冲区与文件系统交互的单位是块，块的大小应为文件系统与磁盘交互单位的整数倍，一般可定为4KB 或8KB。

2) 主要数据结构

struct BlockNode 定义链表节点存储Data Block

3) 设计的类和类间关系

这一模块设计的类是BufferManager

4.3 DB Files

1) 功能描述

DB Files 指构成数据库的所有数据文件，主要由记录数据文件、索引数据文件和Catalog 数据文件组成。

2) 主要数据结构

主要数据结构为vector，没有使用到自己定义的数据结构。

3) 设计的类和类间关系

这一模块设计的类是HeapFile，所调用的类有BufferManager

五 模块实现

5.1 Record Manager

RecordManager 负责管理记录表中数据的数据文件，实现最终对文件内记录的增查删改操作

5.1.1 主要函数实现

```
// 查找
int RecordManager::select(const char* tableName,
    const vector<string>* colName,
    const vector<int>* cond,
    const vector<string>* operand,
    vector<char*>* record,
    vector<int>* ids)
{
    CatalogManager* manager = MiniSQL::getCatalogManager();
    Table* table = manager->getTable(tableName);
    if (table == NULL)
        return -1;

    HeapFile* file = new HeapFile(("record/" + string(tableName)).c_str());
    int recordLength = table->getRecordLength();

    int id, hitCount = 0;
    char* dataIn = new char[recordLength];

    while ((id = file->getNextRecord(dataIn)) >= 0)
        if (checkRecord(dataIn, tableName, colName, cond, operand))
        {
            char* hit = new char[recordLength];
            memcpy(hit, dataIn, recordLength);
            record->push_back(hit);
            ids->push_back(id);
            hitCount++;
        }

    delete[] dataIn;
    delete file;
    return hitCount;
}
// 插入
int RecordManager::insert(const char* tableName, const char* data)
{
    CatalogManager* manager = MiniSQL::getCatalogManager();
    Table* table = manager->getTable(tableName);
    if (table == NULL)
        return -1;
    HeapFile* file = new HeapFile(("record/" + string(tableName)).c_str());

    char* exist = new char[table->getRecordLength()];
    while (file->getNextRecord(exist) >= 0)
    {
        int colId;
        if ((colId = table->checkConsistency(data, exist)) >= 0)
        {
            cerr << "ERROR: [RecordManager::insert] Duplicate values in unique column `" << tabl
            delete[] exist;
            delete file;
```

```
            return -1;
        }
    }
    int ret = file->addRecord(data);

    delete[] exist;
    delete file;
    return ret;
}

// 删除
bool RecordManager::remove(const char* tableName, const vector<int>* ids)
{
    HeapFile* file = new HeapFile(("record/" + string(tableName)).c_str());
    for (int i = 0; i < (int)ids->size(); i++)
        file->deleteRecord(ids->at(i));
    delete file;
    return true;
}

// 建表
bool RecordManager::createTable(const char* tableName)
{
    CatalogManager* manager = MiniSQL::getCatalogManager();
    Table* table = manager->getTable(tableName);
    if (table == NULL)
        return false;
    HeapFile::createFile(("record/" + string(tableName)).c_str(), table->getRecordLength());
    return true;
}

// 删表
bool RecordManager::dropTable(const char* tableName)
{
    Utils::deleteFile(("record/" + string(tableName)).c_str());
    return true;
}

// Check
bool RecordManager::checkRecord(const char* record,
    const char* tableName,
    const vector<string>* colName,
    const vector<int>* cond,
    const vector<string>* operand)
{
    CatalogManager* manager = MiniSQL::getCatalogManager();
    Table* table = manager->getTable(tableName);
    if (table == NULL)
        return false;

    int condCount = colName->size();
```

```
    for (int i = 0; i < condCount; i++)
    {
        char dataOut[MAX_VALUE_LENGTH];
        short type = table->getValue(colName->at(i).c_str(), record, dataOut);

        if (type <= TYPE_CHAR)
        {
            if (!charCmp(dataOut, operand->at(i).c_str(), cond->at(i)))
                return false;
        }
        else if (type == TYPE_INT)
        {
            if (!intCmp(dataOut, operand->at(i).c_str(), cond->at(i)))
                return false;
        }
        else if (type == TYPE_FLOAT)
        {
            if (!floatCmp(dataOut, operand->at(i).c_str(), cond->at(i)))
                return false;
        }
    }
    return true;
}
```

5.1.3 接口

```
public:
    // 执行Select命令
    int select(
        const char* tableName, const vector<string>* colName,
        const vector<int>* cond, const vector<string>* operand,
        vector<char*>* record, vector<int>* ids
    );
    // 执行Insert命令
    int insert(const char* tableName, const char* data);
    // 执行Remove命令
    bool remove(const char* tableName, const vector<int>* ids);
    // 执行Create Table命令
    bool createTable(const char* tableName);
    // 执行Drop Table命令
    bool dropTable(const char* tableName);
    // 执行Check命令
    bool checkRecord(
        const char* record, const char* tableName,
        const vector<string>* colName, const vector<int>* cond,
        const vector<string>* operand
    );
private:
    // 比较string型
    bool charCmp(const char* a, const char* b, int op);
    // 比较int型
    bool intCmp(const char* a, const char* b, int op);
    // 比较float型
    bool floatCmp(const char* a, const char* b, int op);
```

5.2 Buffer Manager

BufferManager缓冲区主要由BlockNode结构以及一系列的成员方法进行管理。

```
struct BlockNode
{
    Block* block;
    BlockNode* pre;
    BlockNode* nxt;

    BlockNode(Block* _block): block(_block) {}
    ~BlockNode() { remove();}

    // 插入节点
    void add(BlockNode* node)
    {
        pre = node; nxt = node->nxt;
        node->nxt->pre = this; node->nxt = this;
    }
    // 删除节点
    void remove()
    {
        pre->nxt = nxt;
        nxt->pre = pre;
    }
};
```

5.2.1 主要函数实现

```
// 删除文件中所有Block
void BufferManager::removeBlockByFilename(const char* filename)
{
    BlockNode* nxtNode;
    for (BlockNode* node = lruHead->nxt; node != lruTail; node = nxtNode)
    {
        nxtNode = node->nxt;
        if (node->block->filename == filename)
            deleteNodeBlock(node, false);
    }
}

// 删除某一节点Block
void BufferManager::deleteNodeBlock(BlockNode* node, bool write)
{
    Block* block = node->block;
    writeBlock(block->filename.c_str(), block->id);
    nodeMap.erase(block->filename + "`" + to_string(block->id));
    delete node;
    delete block;
    blockCnt--;
}

// 加载Block
Block* BufferManager::loadBlock(const char* filename, int id)
{
    Block* block = new Block(filename, id);
    FILE* file = fopen(("data/" + string(filename) + ".mdb").c_str(), "rb");
    fseek(file, id*BLOCK_SIZE, SEEK_SET);
    fread(block->content, BLOCK_SIZE, 1, file);
    fclose(file);

    BlockNode* node = new BlockNode(block);
    node->add(lruHead);
    nodeMap[string(filename) + "`" + to_string(id)] = node;
    blockCnt++;

    return block;
}

// 写入Block
void BufferManager::writeBlock(const char* filename, int id)
{
    Block* block = nodeMap[string(filename) + "`" + to_string(id)]->block;
    if (block->dirty == false)
        return;

    FILE* file = fopen(("data/" + string(filename) + ".mdb").c_str(), "rb+");
    fseek(file, id*BLOCK_SIZE, SEEK_SET);
    fwrite(block->content, BLOCK_SIZE, 1, file);
```

```
    fclose(file);
}
```

5.2.2 接口

```
public:
    // 根据id获取Block
    Block* getBlock(const char* filename, int id);
    // 根据Filename删除Block
    void removeBlockByFilename(const char* filename);

private:
    // 删除
    void deleteNodeBlock(BlockNode* node, bool write = true);
    // 从File中加载Block
    Block* loadBlock(const char* filename, int id);
    // 从File中写入Block
    void writeBlock(const char* filename, int id);
```

5.2.3 测试代码

```
// Print block filename and id
void BufferManager::debugPrint() const
{
    BlockNode* node = lruHead->nxt;
    cerr << "DEBUG: [BufferManager::debugPrint]" << endl;
    for (; node != lruTail; node = node->nxt)
        cerr << "Block filename = " << node->block->filename << ", id = " << node->block->id <<
    cerr << "----------------------------------------" << endl;
}
```

5.3 DB Files

5.3.1 主要函数实现

```
// 新建文件
void HeapFile::createFile(const char* _filename, int _recordLength)
{
    _recordLength++;

    FILE* file = fopen(("data/" + string(_filename) + ".mdb").c_str(), "wb");
    char data[BLOCK_SIZE] = {0};
    memcpy(data, &_recordLength, 4);
    // Set record count to 0
    memset(data + 4, 0, 4);
    // Set first empty record to -1
    memset(data + 8, 0xFF, 4);
    fwrite(data, BLOCK_SIZE, 1, file);
    fclose(file);
}

// 访问记录
const char* HeapFile::getRecordById(int id)
{
    if (id >= recordCount)
        return NULL;

    loadRecord(id);
    bool invalid = *(reinterpret_cast<char*>(block->content + bias + recordLength - 1));
    if (invalid)
        return NULL;

    return block->content + bias;
}

// 新增记录
int HeapFile::addRecord(const char* data)
{
    loadRecord(firstEmpty >= 0 ? firstEmpty : recordCount);

    if (firstEmpty >= 0)
        firstEmpty = *(reinterpret_cast<int*>(block->content + bias));
    else
        recordCount++;

    memcpy(block->content + bias, data, recordLength-1);
    memset(block->content + bias + recordLength - 1, 0, 1);
    block->dirty = true;

    updateHeader();
    return ptr;
}

//删除记录
bool HeapFile::deleteRecord(int id)
{
```

```
    if (id >= recordCount)
    {
        cerr << "ERROR: [HeapFile::deleteRecord] Index out of range!" << endl;
        return false;
    }

    loadRecord(id);
    bool invalid = *(reinterpret_cast<char*>(block->content + bias + recordLength - 1));
    if (invalid)
    {
        cerr << "ERROR: [HeapFile::deleteRecord] Record already deleted!" << endl;
        return false;
    }

    memcpy(block->content + bias, &firstEmpty, 4);
    memset(block->content + bias + recordLength - 1, 1, 1);
    block->dirty = true;

    firstEmpty = ptr;
    updateHeader();
    return true;
}
```

5.3.2 接口

```
public:
    // 新建文件
    static void createFile(const char* _filename, int _recordLength);
    // 访问记录
    const char* getRecordById(int id);
    // 新增记录
    int addRecord(const char* data);
    // 删除记录
    bool deleteRecord(int id);

private:
    // 当前block
    Block* block;
    void updateHeader();
    void loadRecord(int id);
```

六 遇到的问题及解决方法

对c++语言的运用掌握不太熟练，查询了很多库的调用，极大简化了程序，C++也较好解决了多人协作程序的整合问题。
Record模块实现较为顺利，Buffer模块前期因为部分概念不甚清晰导致在调试的过程出现较多bug，最后程序的实现也帮助我更理解缓冲区的概念。

七 总结

总的来说本次实验看似复杂、工程量大，协作完成之后发觉整体难度并不大，重点在于对程序各个模块的功能实现。在和队友充分讨论后确立了各模块的框架、给定了各功能函数的接口，随后由模块进行分工，各自实现程序功能，最后整合，进而成功完成了本次大作业。