MiniSQL 设计报告

——2017-2018 春夏数据库系统期末设计

3150104951 黄柯熹 3150105197 谭琦烨

1. MiniSQL 总体概述

根据本学期所学，与同组组员合作设计一个 minisql，支持 SQL 语法，支持表的建立、删除，索引的建立、删除还有表的增删改查等常规操作。

a) MiniSQL 实现功能分析

i. 功能概述：使用命令行与用户进行交互，读取用户输入的 SQL 语句，由内部的处理器转化为内部语言，经过分析可以实现表的建立，数据的插入，索引建立等等操作。

ii. 支持数据：该 MiniSQL 支持三种数据：INT、CHAR(N)、FLOAT，其中 1<=N<=255。

iii. 定义表：表的数量上限为 40 张，每一张不超过 15 个属性，每个属性可以选择是否 UNIQUE，支持单属性的主键定义。

iv. 索引(index)：UNIQUE 和 PRIMARY KEY 的属性，可以在其上建立 B+TREE 索引。

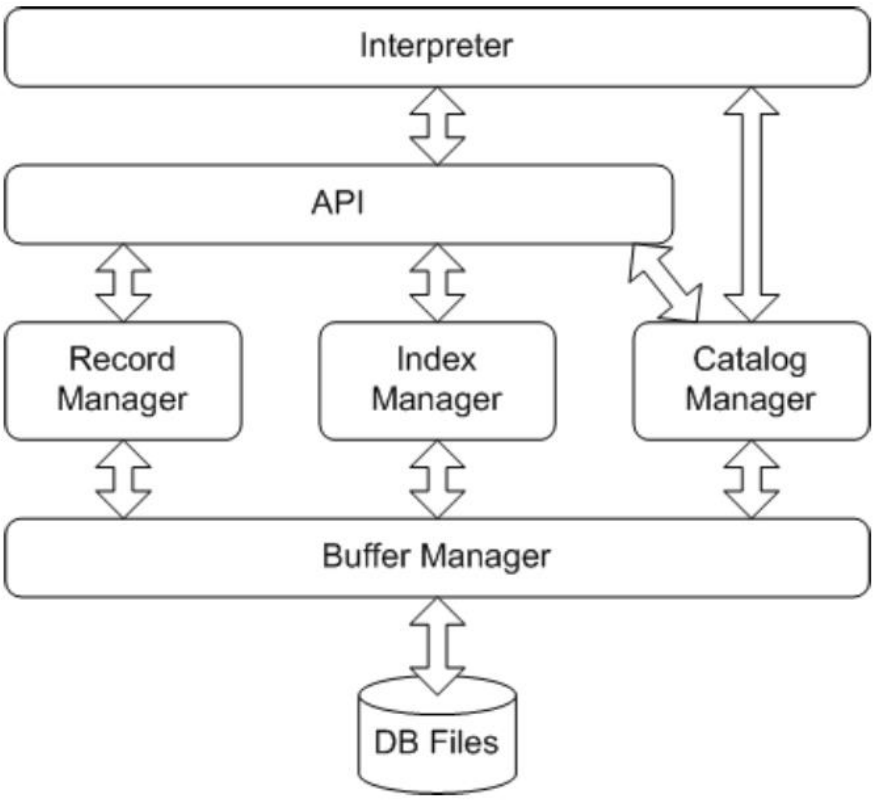
v. 记录查找：可以用查找特定条件的记录，条件形式支持数据的判断，操作符支持：=, <=, >=, <, >, <>六种。

vi. 记录插入和删除：支持一条数据的插入，支持一条或多条记录的删除。

b) MiniSQL 系统体系结构

i. 该项目由六个模块组成：Interpreter, API, Rocord Manager, Index Manager, Catalog Manager, Buffer Manager

这六个模块的关系如下：



c) 设计语言及实现平台

i. 开发语言：C++

ii. 运行平台：Windows/OS X

iii. 编译器：VC++/gcc

d) 程序界面介绍

本项目使用清新易懂的字符界面与用户进行交互。

语句用分号结束。不管语句是否合法，MiniSQL 会对其进行分析，若合法，则输出执行结果，反之，则输出错误信息。

2. MiniSQL 模块叙述

a) BUFFER MANAGER

BUFFER MANAGER 是缓冲区的管理模块，他的作用在于：

i. 根据需求，读取数据；

ii. 使用缓冲区的替换算法，将数据输出到文件；

iii. 定义并且记录缓冲区中页的状态；

iv. 提供缓冲区页的 pin 功能。

在本项目中，BUFFER MANAGER 最多容纳 128 个 block，每个 block 大小为 4KB。

b) CATALOG MANAGER

CATALOG MANAGER 负责管理表的模式信息。如一张表的名字、字段数、主键、索引等。对于表中的属性，需要记录类型、是否 UNIQUE 等。

c) RECORD MANAGER

RECORD MANAGER 负责管理记录表中数据的数据文件。

主要功能：

i. 数据文件的创建和删除；

ii. 记录的插入、删除、查找，对外提供相应的接口；

记录的查找最多允许带一个条件的查找。

数据文件由 Block 组成，大小也为 4KB。

d) INDEX MANAGER

INDEX MANAGER 负责 B+TREE 索引的实现，可以实现 B+TREE 的创建、删除、查找、插入键值、删除键值等操作，对外提供相应的接口。通过 INDEX MANAGER，用户可以快速定位符合条件的条目在数据库中的位置，大幅增加查找速度。

e) INTERPRETER

INTERPRETER 负责读取用户的输入，并生成内部的数据结构表示。若语句不合法，则给出相应的错误信息。

f) API

API 是整个 MiniSQL 的核心。API 的功能为：

i. 提供执行 SQL 语句的接口，供 INTERPRETER 调用；

ii. 根据 INTERPRETER 的结果，根据 CATALOG MANAGER 提供的信息确定执行规则；

iii. 调用 RECORD MANAGER、INDEX MANAGER、CATALOG MANAGER 提供的接口执行；

iv. 将结果返回给 INTERPRETER，将结果输出给用户。

3. MiniSQL 内部数据形式

a) INTERPRETER 解析树

```
struct ParseTree {
int opcode;
string table_name;
string index_name;
string attr_name;
string primarykey;
string filename;
vector<Attribute> attributes;
vector<Conditionp> conditions;
RowData rowdata;
Table tableinfo;
Index indexinfo;
};
```

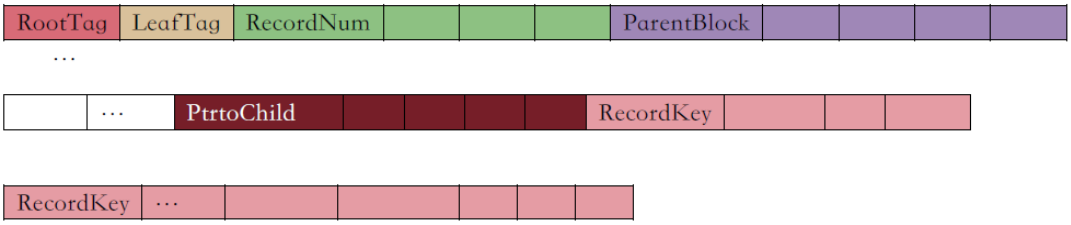
其中，OPCODE 代表着 SQL 语句的操作方式。具体映射如下：

```
#define STARTOPCODE 10
```

```
#define INSERT (STARTOPCODE + 1)
#define DELETE (STARTOPCODE + 2)
#define DROPINDEX (STARTOPCODE + 3)
#define DROPTABLE (STARTOPCODE + 4)
#define CREINDEX (STARTOPCODE + 5)
#define CRETABLE (STARTOPCODE + 6)
#define SELECT (STARTOPCODE + 7)
#define QUIT (STARTOPCODE + 8)
#define EXECFILE (STARTOPCODE + 9)
#define SHOWTABLE (STARTOPCODE + 10)
#define CLEARBUFFER (STARTOPCODE + 11)
```

b) INDEX MANAGER 中的 B+ Tree 结构

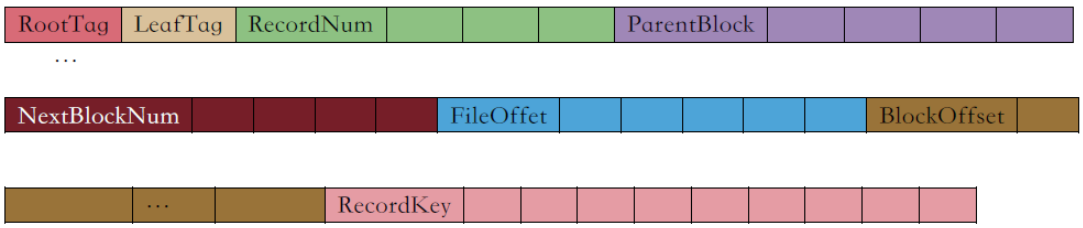
i. 分支节点结构



其中一个格子代表数组中的一个元素，每个一字节，我们用一字节来判断根，一字节来判断叶，四字节表示该节点中有多少记录，五字节表示父亲的节点号。

我们在整个 block 里面的记录量为$\frac{4096-11}{(2*5+RecordLength)}$，其中，每个记录都有两个指针，nextblock 的编号，children 的编号，record 的键值。

ii. 叶子节点结构



可以看到，叶子节点中，不仅要记录这个 record 的 key，还要记录 FileOffset 与 BlockOffset，即在 DBFile 中的第几个 Block 和在此 Block 中的第几条记录，这样一来，我们便不用对该表进行线性搜索，减少了昂贵的 I/O 次数。

c) BUFFER MANAGER BLOCK 结构

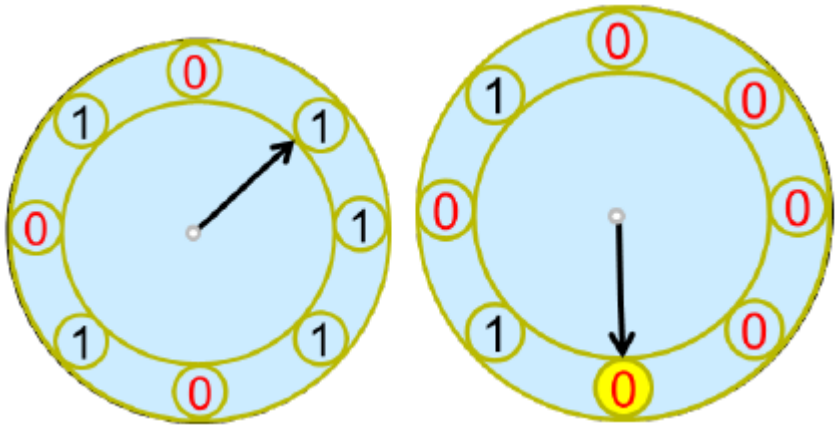
```
class BlockNode
{
public:
BlockNode();
BlockNode(string fileName,int offset,bool pin);
string fileName;//in which file
int offset;//offset from the head of the file
BYTE BlockData[BLOCK_SIZE];
```

```
bool pin;
bool LRUtag;//true-- reference=1, false -- reference=0
bool dirty;
BlockNode *preBlock;
BlockNode *nextBlock;
char *address;//the content address
size_t usingSize; // the byte size that the block have used.
The total size of the block is BLOCK_SIZE .
};
```

其中 BlockData 是要写入磁盘的 block。LRUtag 是 LRU 算法的标记，dirty 是脏标记。

d) BUFFER MANAGER LRU 时钟算法

当 buffer 已满而又需要读入新的 BlockNode 时，需要从缓冲区内找到某一块进行替换。这里用的是 clock 的算法，一种 LRU 算法的近似实现。



如图，缓冲区被组织成类似时钟的环状。其中的 1 和 0 是是否被访问的标记。如果当前 block 的访问标记为 false，然后该 block 被访问，则设为 1。

Buffermanager 还有一个 private 的变量 referenceloc，即当前时钟的“指针”指向哪里。在寻找替换出去的 block 时，从当前指针位置开始，如果指向的 block 被锁定则跳过；如果指向的 block 访问位（在这里定义为布尔变量 LRUtag）为 true，则设为 false，并移向下一位置，直到遇到某个位置的 LRUtag 为 false。则该 block 被替换出去。

如果寻找了一圈回到初始位置而没有找到 LRUtag 为 false 的 block，则将当前 block 替换出去。

e) RECORD MANAGER

i. 块定位

在文件内容管理过程中，定位文件中的块是必有且关键的一步。在本系统中是运用 FileNode 类记录文件名和所用的数据块，BlockNode 类记录块归属于哪个文件和已占用字节。两者都以链表结果存储。由此即可通过表名，依次定位到 FileNode 和 BlockNode。

ii. 插入函数

对于插入函数及传入的数据，在找到空位之前，我们得对插入的数据进行判断，是否在主键和唯一的属性上是不是有重复的值，如果有重复的值便输出错误，插入失败。如果没有重复的值，我们便找到空位插入，然后更新索引。

iii. 情况选择

根据传入的条件进行选择，首先遍历查询的条件中是否有建有索引，如果有索引就先使用索引条件返回一些数据的块号和偏移量，然后再根据后续的条件再对用索引的数据进行匹配，这样会减小有索引的条件的搜索时间。对于无索引的数据，用一条一条遍历的方法对每一条数据进行匹配，选择符合条件的输出。

iv. 删除函数

对于删除操作，我们需要删除符合条件的值，其实是执行一个 select 操作，然后再进行删除。对于删除每条记录的操作之后留下的空隙，我采取的并不是将这个空间赋值为空，而是将这个块中后面的记录依次前移，同时更新索引。当块内全部数据被删除后，删除此块。这样使得块的利用率升高，且较为规整，节约空间且便于空间整合。

f) 程序主循环

```
// MiniSql.cpp : Defines the entry point for the console
application.
//

#include "stdafx.h"
#include "API.h"

int main()
{
    API api;
    while (1)
    {
        api.getInstruction();
        clock_t start = clock();
        api.InstructionExcute();
        clock_t end = clock();
        printf("time elapsed:   %lfs    \n", ((double)(end - start)
/ CLOCKS_PER_SEC));
        api.ShowResult();
    }
    return 0;
}
```

使用一个 while(1)，重复读入用户的输入，除非用户输入“quit”，该程序不会停止运行。获取指令，若指令合法，则进行指令的执行，统计执行时间。若成功则显示结果，失败则显示错误信息。

4. MiniSQL 内部接口形式

a) BUFFER MANAGER

i. `BlockNode* Fch_Block(string fileName, int offset);`

当用户需要获取某个 block 内的信息时，给出文件名和对应的块偏移量。若已在 buffer 内，直接返回指针；否则读取文件内的数据，加载入缓冲区，返回指针。

ii. `int create_Block(string fileName);`

在文件末创建一个 block 的空间，并返回对应偏移量。

iii. `bool create_File(string fileName);`
创建文件。

iv. `bool delete_File(string fileName);`
删除文件。

v. `int buf_loc();`
当缓冲区满时，找到应替换出去的 block 并写回，同时返回该已经被空出来的位置的下标（存在数组中）。

vi. `bool pin(BlockNode& block);`
锁定 block。

vii. `bool del_pin(BlockNode& block);`
删除 block 的锁定。

viii. `int file_Size(string fileName);`
获取文件大小。

ix. `bool buf_ret();`
将 buffer 的 block 写回文件。

x. `bool dir_ret(BlockNode block);`
将修改过的 block 写回文件。

xi. `void buf_clr();`
清空 buffer，写回脏块，初始化 buffer 数组里的每个位置。

b) CATALOG MANAGER

i. `int create_Table(string tableName, vector<Attribute> attributes,string primaryKey, int attributeNum);`
创建表，同时创建表的详细信息文件，维护表单列表，并修改缓冲区信息。

ii. `int create_Index(string indexName, string tableName, string attributeName);`
创建索引，同时创建索引的详细信息文件，维护索引列表，并修改缓冲区信息。

iii. `int Table_ext(string tableName);`
判断表是否存在。

iv. `int Index_ext(string indexName);`
判断索引是否存在。

v. `int drop_Table(string tableName);`
删除表。删除该表对应的详细信息文件、删除建立在该表上的索引的信息、删除表单列表中该表的表明，同时修改缓冲区内信息。

vi. `int drop_Index(string indexName);`
删除索引。删除该索引在索引列表中的信息，维护该索引所建立的表的详细信息，同时修改缓冲区内信息。

vii. `Table get_Table(const string tableName);`
搜索表，返回 Table 结构。

viii. `Index getIndex(const string indexName);`
搜索索引，返回 Index 结构。

ix. `Table Table4Log(const string tableName);`
从表信息文件读取表，返回 Table 结构。

x. `Index Index4Log(const string indexName);`

从索引信息文件读取索引，返回 Index 结构。

xi. `int Attribute_in_Table(string tableName, string attributeName);`

判断一个属性是都在一个表中。

xii. `bool Index_in_Table(string tableName, string indexName);`

判断一个索引是否在一个表中。

xiii. `void print_Tables();`

输出所有表的详细信息。

xiv. `void print_Indexes();`

输出所有索引的详细信息。

xv. `int print_Index_in_Table(string tableName);`

输出一个表中所有索引的信息。

xvi. `int print_Attributes_in_Table(string tableName);`

输出一个表中所有属性的信息。

xvii. `void create_Table_Log(Table table, ofstream& out);`

根据表，创建表的详细信息文件。

c) RECORD MANAGER

i. `int table_create(string tableName);`

创建表。

ii. `int table_drop(string tableName);`

删除表。

iii. `int index_create(string indexName);`

创建索引。

iv. `int index_drop(string indexName);`

删除索引。

v. `int insert_record(string tableName, BYTE* record, int recordSize, int& offset);`

插入数据。

vi. `int show_all(string tableName, Attribute attributes[MAX_ATTRIBUTE_NUM], vector<Condition>* conditionVector);`

根据输入，显示符合条件的数据。

vii. `int show_block(string tableName, Attribute attributes[MAX_ATTRIBUTE_NUM], vector<Condition>* conditionVector, int blockOffset);`

根据输入，显示 block 内符合条件的数据。

viii. `int find_all(string tableName, Attribute attributes[MAX_ATTRIBUTE_NUM], vector<Condition>* conditionVector);`

根据输入，选择表内符合条件的数据。

ix. `int del_all(string tableName, Attribute attributes[MAX_ATTRIBUTE_NUM], vector<Condition>* conditionVector);`

根据输入，删除表内符合条件的数据。

x. `int print_record(string tableName, Attribute attributes[MAX_ATTRIBUTE_NUM],Attribute attributesPrint[MAX_ATTRIBUTE_NUM]);`

根据输入，显示表内部分特征的数据。

xi. `int print_block(string tableName, Attribute attributes[MAX_ATTRIBUTE_NUM], Attribute attributesPrint[MAX_ATTRIBUTE_NUM], int blockOffset);`

根据输入，显示 block 内部分特征的数据。

d) INDEX MANAGER

i. `void createIndex(const Table tableInfo, Index indexInfo);`

创建索引。

ii. `void dropIndex(Index indexInfo);`

删除索引。

iii. `IndexBranchNode insertKeyB(Index indexInfo, IndexLeafNode node, int blockOffset = 0);`

通过建立 B+TREE 进行键值插入。

iv. `outputdata searchB(const Table& tableInfo, const Index& indexInfo, string key, int blockOffset = 0);`

通过建立 B+TREE 进行等值查找。

e) INTERPRETER

i. `string get_inst();`

获取用户输入指令。

ii. `RC check_inst(string instruction);`

检测用户指令是否合法。

iii. `ParseTree get_ParseTree();`

获取解析树。

iv. `void cln_inst();`

清除指令并获取下一条指令。

5. MiniSQL 测试报告

a) 测试 float、int、char 的等值查询

```
Instruction executed.
MiniSQL>> select * from student2 where score=98.5;
1080100003 name3 98.50
1080100046 name46 98.50
1080100085 name85 98.50
1080100137 name137 98.50
1080100201 name201 98.50
1080100242 name242 98.50
1080100284 name284 98.50
1080100430 name430 98.50
1080100456 name456 98.50
1080100488 name488 98.50
1080100652 name652 98.50
1080100683 name683 98.50
1080100686 name686 98.50
1080100795 name795 98.50
1080100799 name799 98.50
1080100868 name868 98.50
1080100876 name876 98.50
1080100882 name882 98.50
1080100945 name945 98.50
1080100953 name953 98.50
1080100972 name972 98.50
1080100989 name989 98.50
1080101022 name1022 98.50
1080101045 name1045 98.50
1080101230 name1230 98.50
1080101363 name1363 98.50
1080101375 name1375 98.50
1080101463 name1463 98.50
```

b) 测试 index 创建和性能

```
MiniSQL>> select * from student2 where name='name245';
1080100245 name245 62.50
time elapsed:  0.003000s
Instruction executed.
```

没有创建索引前，查询某条记录的时间为 0.003s。

```
MiniSQL>> select * from student2 where name='name245'
1
1080100245 name245 62.50
time elapsed:  0.001000s
Instruction executed.
```

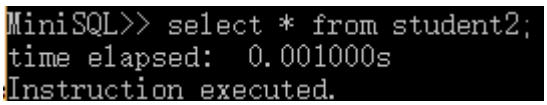
创建了索引之后，查询某条记录的时间只需要 0.001s，为创建前的 1/3.

c) 测试 unique 键约束

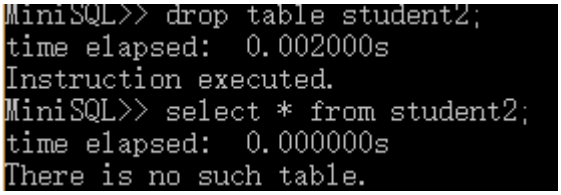
```
MiniSQL>> insert into student2 values(1080100245,'name245',100);
1
time elapsed:  0.001000s
No duplicated unique key allowed.
```

可见，unique 键约束阻止了重复数据的插入。

d) 测试记录的删除与表的删除



删除表中的记录后，再进行查找，则没有记录。



删除表后，再进行查找，则返回“There is no such table.”

6. MiniSQL 分工

谭琦烨：RECORD MANAGER、CATALOG MANAGER、API、BUFFER MANAGER

黄柯熹：INTERPRETER、INDEX MANAGER、测试及报告