

MiniSQL 总实验报告

蒋仕彪 3170102587 求是科学班（计算机）1701

1. 实验说明

- 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。

- 实验分工

- 所有内容均由我（蒋仕彪）独立完成。

2. 实验平台

- 编程语言：`C++`

- 有良好的工程性，支持复杂模块的编写。
- 一些特性可以简化编程，如 `指针`、`STL` 等。

- 开发工具：`VS2017`。

- 适合工程化文件的编写和管理。
- 提供自动补全等操作，并实时检测并提醒语法错误。

- 管理工具：`GitHub Desktop`

- 支持多版本的保存和回退，防止意外发生。
- 支持快速查看新修改的代码。

3. 总体框架：

3.1 支持的功能

- 创建一个表

- 格式：

```
create table 表名 (
    列名 类型 ,
```

列名 类型 ,

列名 类型 ,

primary key (列名)

);

- 若执行成功则创建表。

- **primary key** 必须存在且放在最后；里面只能放一个列的名字。

- 某些类型后面可以紧跟着 **unique**，表示该键值不能重复。

- **删除一个表**

- 格式：

```
drop table 表名 ;
```

- 若执行成功则删除表。

- 删除表时，会自动遍历表中所有索引，一并将其删除。

- **创建索引**

- 格式：

```
create index 索引名 on 表名 ( 列名 );
```

- 若执行成功则创建索引。

- 当一个表创建的时候，其 **primary key** 会自动创建一个索引（不能被删）。

- 创建索引的列名必须是 **unique** 的。

- 创建索引时，会遍历表中所有数据，并给出的列名为关键字建立一棵 **B+树** 在对应文件夹里。之后插入或者删除时，会自动对这棵 **B+树** 进行相应的操作。

- **删除索引**

- 格式：

```
drop index 索引名;
```

- 若执行成功则删除对应索引。

- 回收这个索引对应的 **B+** 树，同时移除对应文件。

- **选择语句**

- 格式：

```
select * from 表名;
```

或

```
select * from 表名 where 条件1 and 条件2 and ... and 条件n;
```

- 若执行成功将所有符合要求的记录打印在屏幕上。返回结果的第一行为属性名，其余每一行表示一条记录。
- 每一则条件由 `列名 运算符 常量` 构成，常量的类型要和列名的定义一致。
- 运算符只能是 `< , > , <= , >= , = , !=` 中的一种。
- 只支持 `select *` 模式，不支持选择其中部分的列；只支持直接询问或按 `and` 连接的条件询问。**
- 内部实现细节：
 - 普通的 `select` 会遍历表里的每一个 `record` 并检查是否满足条件（如果满足就打印出来），效率是线性的。
 - 如果查询的格式满足，只有一个 `=` 条件且列名已经建立索引，那么程序会在该索引的 `B+树` 内部查询是否有这个叶子，效率是 `log` 级别的。

- 插入一条记录

- 格式：

```
insert into 表名 values ( 值1 , 值2 , ... , 值n );
```

- 若执行成功则插入这条记录，否则返回详细的错误类型。
- 会对所有 `unique` 的属性进行合法性判断。
- 会扫描表中所有已创建索引的列，将这条记录加到对应的 `B+树` 里。

- 删除记录

- 格式：

```
delete from 表名;
```

或

```
delete from 表名 where 条件1 and 条件2 and ... and 条件n;
```

- 删除符合要求的记录，并返回一共删除了几条记录。
- 会扫描表中所有已创建索引的列，将这条记录在相应的 `B+树` 里删掉。
- 内部实现细节：
 - 普通的 `delete` 会遍历表里的每一个 `record` 并检查是否满足条件（如果满足就删除），效率是线性的。
 - 如果删除的格式满足，只有一个 `=` 条件且列名已经建立索引，那么程序会在该索引的 `B+树` 内部查询是否有这个叶子，并把对应的 `record` 删掉，效率是 `log` 级别的。

- 执行脚本文件

- 格式：

```
execfile 文件名;
```

- 执行对应文件里所有的SQL语句，默认以文本文件的形式打开。
- 文件里的 执行脚本文件语句 不会执行（为了防止循环嵌套等问题）。
- 文件里每一条语句执行完，都会在屏幕里输出结果或者报错信息。但是防止终端输出过多的冗余信息，当指令数超过 20 条的时候，只保留前 20 条的结果（之后的会被隐藏）。

- 保存结果

- 格式：

```
save;
```

- 保存之前对数据库的所有操作。
 - 将 Cache 里的所有信息写回硬盘，包括插入的记录、删除的记录、创建的表和索引，以及所有的 B+ 树。

- 退出

- 格式：

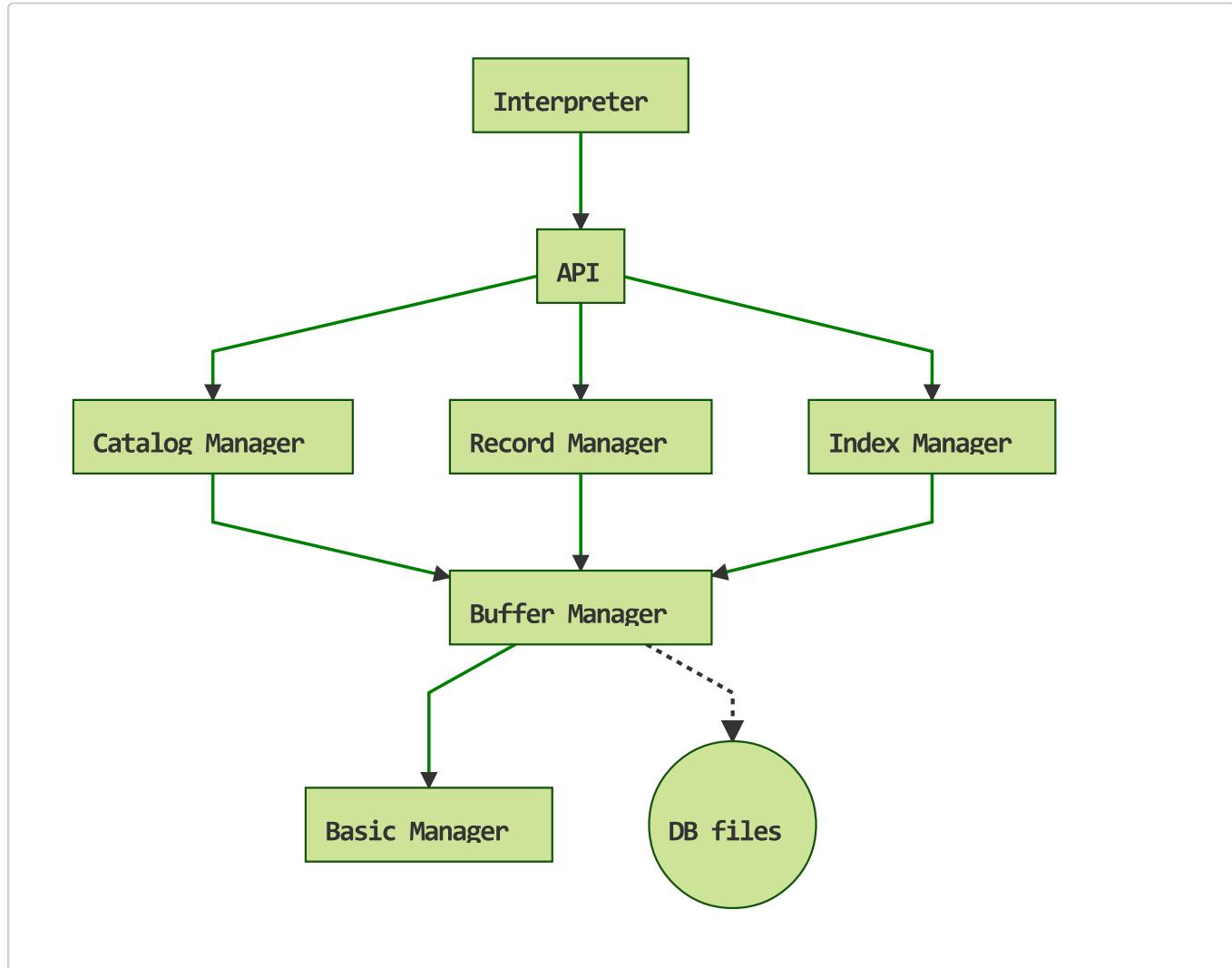
```
quit;
```

- 保存之前对数据库的所有操作并退出。
 - 相当于做了一遍 save 操作后安全退出。
 - 如果强退的话，之前的操作可能会丢失（还未从Cache写回硬盘）。

- 功能细节：

- 每个操作若成功执行，会输出执行时间。
 - 所有语句如果执行失败，都会返回详细的错误信息。
 - 一条语句可以写在一行，也可以写在多行，结尾用 ; 标记。
 - 关键字必须完全匹配（包括大小写）。
 - 字符串可以用 ' 或 " 包起来表示，但左右必须匹配。
 - 如果语句执行失败，会返回错误信息。为避免混淆，错误信息中的引用一律用 [] （而不是引号）包起来。

3.2 系统体系结构:



4. 各部分实现细节 (自底向顶介绍)

4.0 预备技能：处理文件

- 文件操作

- 我们需要对一个文件执行读、写（末尾或中间摸个位置）操作，所以最适合的 `fopen` 模式是 `r+` 或 `rb+`。我们还可以用一些函数来辅助输出。
 - `fseek(file, offset, pos)` 将文件指针定位到指定位置的后 `offset` 位。
`SEEK_SET` , `SEEK_CUR` , `SEEK_END` 分别表示开头, 当前位置和结尾。
 - `ftell(file)` 计算出当前文件指针距离文件开头的字节数。

- 一个文件操作优化

- 该程序会涉及到大量的文件读写操作。
- C++的 `fopen` 和 `fclose` 特别慢，短时间内频繁开关甚至会无效。
- 我在全局维护一个名叫 `File` 的 `map` 来管理文件指针。它的第一维是文件名称，第二维是目前该文件的文件指针。每当打开或关闭 `minisQL` 时，统一打开或关闭

所有文件。 `save` 的时候会重新 `fclose` 和 `fopen` 每个文件，确保别的程序能读取新文件。

- 在上述需求下的文件开关代码

```
map<string, FILE*>::iterator CreateFile(string name, int binary) {
    auto it = File.find(name);
    if (it != File.end()) {
        fclose(it->second);
        File.erase(it);
    }
    FILE *tmp = fopen((name + ".txt").c_str(), binary ? "wb" : "w");
    fclose(tmp);
    string t = binary ? "rb+" : "r+";
    File[name] = fopen((name + ".txt").c_str(), t.c_str());
    return File.find(name);
}

void DeleteFile(string name) {
    auto it = File.find(name);
    if (it != File.end()) {
        fclose(it->second);
        File.erase(it);
        remove((name + ".txt").c_str());
    }
}
```

4.1 Basic Manager 模块（基本结构与约定）

- 如何支持类型不同的数据

- 在本实验中，我们一共要支持 `int`, `float`, `char()` 三种类型，那么一组数据该如何来定义呢？如果我们直接采用 C++ 里对应的结构，免不了会有很多 `if` (特别是在 `select` 和 `delete` 时的条件获得)。
- 不同类型的长度控制也很麻烦。`char()` 甚至可以开到 255 位。
- 对此，我想到了一个很巧妙的设计。对于任意一个单位数据，我用固定的 256 字节去刻画。其中，第一位 `TYPE` 表示这是什么类型，后 255 位用来储存数据。对于 `int` 和 `float` 类型，我在程序里将其视作 `char()` (这样读入输出和判断都十分方便)。这个数字还正好是 2 的幂次，可以减少一些数据交互时的空间浪费。那要怎么判断两个数据的大小关系呢？我只要重载这个 `struct` 的运算符，根据不同的 `TYPE`，用 C++ 自带的 `atoi` 和 `atof` 来比较。

```
#define NONE 0
#define LESS 1
#define MORE 2
#define LESSQ 3
#define MOREQ 4
```

```
#define EQUAL 5
#define NOTEQUAL 6
#define INT 1
#define FLOAT 2
#define CHAR 3
```

- 以上是我对 TYPE 以及运算符 OP 的约定。OP 会在重载中用到，例如：

```
int operator < (const Element &b) const {
    return tp == CHAR ? strcmp(d, b.d) < 0 : (tp == INT ? atoi(d) < atoi(b.d) : atof(d) < atof(b.d));
}
```

• 定义一张表时要储存的信息

- 考虑一个属性应该如何定义。它会有一个属性名 name，会有一个类型 tp。我们还要用一些 bool 值记录它是否是 key，是否 uni（是否有 index 我将额外存储，在这里不记录）。此外，如果类型是 Char，我们还要用 len 记录名字长度的限制。

```
struct InputData {
    char tp, uni, key; int len; char name[248];
    InputData() {
        tp = uni = key = 0; len = 255;
        memset(name, 0, sizeof(name));
    }
    InputData(string _name, int _tp, int _key, int _uni, int _len){
        strcpy(name, _name.c_str());
        tp = _tp, uni = _uni | _key;
        key = _key, len = _len;
    }
};
```

- 注意三种类型我是从 1 开始标的。那时因为，如果遇到了删除表，我会简单地将其类型标为 NONE(0)，意为“已删除”。

• 其它约定

```
#define DATASIZE 256
#define MAXELEMENT 16
#define BLOCKSIZE 4096
#define BUFFERSIZE 1024
```

DATASIZE 表示每个单位数据的大小； MAXELEMENT 表示每个表的属性上限；
BLOCKSIZE 是与硬盘交互时每块 block 的大小上限 ($4096 = 256 \times 16$ ，因为要保证
每个 block 里至少放得下一组记录)。
BUFFERSIZE 是 cache 里能共存的 block 上限。

4.2 Buffer Manager 模块 (缓冲区管理)

• 简单的buffer实现

- 实际的 `buffer` 处理会很复杂，我对此做了一定的简化。
- 实验要求中，不要求单组数据跨 `block` 储存。因此，对于一张表的一组 `record`，我单独开一个 `block` 来保存它。
- 如果固定了 `block` 上限（比如 `4096`），对于那些属性很少的表就会造成大量的空间浪费。所以我设的 `block` 大小是可变的，为 `DATASIZE * DATANUMBER`。其中 `DATASIZE = 256`，`DATANUMBER` 是当前表的属性个数。

• block 的表示

- 对于一个 `block`，首要任务是知道它在那里。我对于每一个表都会新开一个数据表为维护，所以就可以用 `<filename, offset>` 来描述一个 `block` 了。因为是 可变 `block`，我额外用一个 `num` 记录 `block` 的实际长度，以及对应的数据信息。注意这里的信息是用 `char` 数组编码的。

```
struct Block {  
public:  
    Block(string _fileName, int _offset, int _num) {  
        num = _num;  
        memset(data, 0, BLOCKSIZE);  
        offset = _offset;  
        fileName = _fileName;  
    }  
private:  
    char data[BLOCKSIZE];  
    string fileName;  
    int offset, num;  
};
```

• LRU 管理

- 内存里能存放的 `block` 个数是有限的，当存放的 `block` 数量超过一定大小的时候，我们不得不替换数据。
- 这里我采用了计算机中常用的 `LRU` 的管理方法：删除“激活最早”的那个 `block`。我用 C++ 的 `list`（链表）和 `map` 配合，来高效地维护这个操作。在全局维护：
 - 一个 `list<block>`。
 - 一个 `map<pair<filename, offset>, list_iterator`。
- `map` 的第一维就是 `block` 的 `pair` 标识符，每当我需要一个 `buffer` 的时候，我先在 `map` 里看看它是否在 `cache` 里。如果不在，我就从硬盘里读入。操作完毕后，我在链表里将当前激活的 `block` 移动到结尾。`map` 的访问是 $O(\log)$ 的，其余的单步时间都是 $O(1)$ 的。

```

bufferIter BFM::BufferManagerRead(const string &fileName, int offset,
int num)
{
    auto it = table.find(make_pair(fileName, offset));
    if (it != table.end()) {
        Block tmp = *(it->second);
        buffer.erase(it->second);
        buffer.push_back(tmp);

        bufferIter B = --buffer.end();
        tag T = make_pair(tmp.fileName, tmp.offset);
        return table.find(T)->second = B;
    }
    else {
        auto in = File.find(fileName);
        try {
            fseek(in->second, offset, SEEK_SET);
            Block tmp = Block(fileName, offset, num);
            fread(&tmp.data, num * DATASIZE, 1, in->second);
            return InsertBlock(tmp);
        }
        catch (...) {
            printf("The impossible thing happens in buffer read.\n");
        }
    }
}

```

- buffer 的编码和解码

- 因为采用特殊的数据存储，编码和解码会变得异常简单：

```

Element BFM::LoadData(bufferIter it, int pos) {
    Element ret;
    ret.tp = (*it).data[pos];
    for (int i = 0; i < 255; i++)
        ret.d[i] = (*it).data[pos + i + 1];
    return ret;
}
void BFM::SetData(bufferIter it, int pos, Element val) {
    it->data[pos] = val.tp;
    for (int i = 0; i < 255; i++)
        it->data[pos + i + 1] = val.d[i];
}

```

- 注意到，LRU管理的 map 里存放的直接是 list 的迭代器。我们在修改一个 block 的时候，是直接对迭代器的地址修改的（Cache里时刻保存着最新的数据）

- buffer 的输出

- buffer的输出比较直观：

```
void BFM::BufferManagerWrite(const Block &b)
{
    try {
        auto in = File.find(b.fileName);
        assert(in->second != nullptr);
        fseek(in->second, b.offset, SEEK_SET);
        fwrite(&b.data, b.num * DATASIZE, 1, in->second);
    }
    catch (...) {
        printf("The impossible thing happens in buffer write.\n");
    }
}
```

- 注意硬盘里的数据会滞后，“断电”后会丢失更新信息。因此，每当用户执行 `save` 或者 `quit` 的时候，我们要把内存里的信息写会硬盘：

```
void BFM::BufferManagerFlush()
{
    for (bufferIter T = buffer.begin(); T != buffer.end();)
        BufferManagerWrite(*T++);
    buffer.clear();table.clear();
}
```

4.3 Catalog Manager 模块（表目录管理模块）

- 建立和删除表

- 注意到，尽管数据库规模会很大，不同表的个数必然不大。我们不必用 `buffer` 去管理表头信息，可以直接进行 `C++` 的文件读写操作。
- 建立表时，直接向 `__table.txt` 的结尾输出该表的各项信息。

```
void Catalog::CreateTable(string TableName, vector<InputData>Table) {
    auto out = File.find("__table__");
    if (out == File.end()) out = CreateFile("__table__", 0);
    fseek(out->second, 0, SEEK_END);
    fprintf(out->second, "%s 1 %d", TableName.c_str(), Table.size());
    for (auto t : Table) fprintf(out->second, " %s %d %d %d", t.name,
        (int)t.tp, t.key, t.uni, t.len);
    fprintf(out->second, " ");
}
```

- 上述代码会输出一个 `1` 的标识符，表示该表是被激活的状态。为什么这么做呢？

这样删除表的时候，只需找到对应的位置，将 `1` 标识符复写成 `0`，意为这张表已经被废弃。这种做法使得删除变得更加高效。每当数据库空闲的时候，可以遍历 `__table__.txt`，将所有 `0` 标记删除，别的数据移位上来。

- 所以删除表就很简单。配合 `fseek` 和 `ftell` 来复写位置。

```
int Catalog::DropTable(string TableName) {
    auto out = File.find("__table__");
    fseek(out->second, 0, SEEK_SET);
    char curTableName[252];
    while (fscanf(out->second, "%s", curTableName) != EOF) {
        int number, valid, loc = ftell(out->second) + 1;
        fscanf(out->second, "%d %d", &valid, &number);
        int same = (string)curTableName == TableName && valid;
        while (number--) {
            char name[252]; int tp, uni, key, ind, len;
            fscanf(out->second, "%s %d %d %d %d", name, &tp, &uni, &key, &len);
        }
        if (same) {
            fseek(out->second, loc, SEEK_SET);
            fprintf(out->second, "0");
            return 1;
        }
    }
    return 0;
}
```

• 获得一张表的定义信息

- 在 `insert`，`select` 和 `delete` 时，我们都需要首先获得一张表的定义信息，才能详细判断用户行为是否有错误。
- 这就需要在 `Catalog Manager` 里写一份“获取表头定义信息”的代码：

```
vector<InputData> Catalog::GetTable(string TableName) {
    auto in = File.find("__table__");
    vector<InputData>ret;
    char curTableName[252];
    fseek(in->second, 0, SEEK_SET);
    while (fscanf(in->second, "%s", curTableName) != EOF) {
        int number, valid;
        fscanf(in->second, "%d %d", &valid, &number);
        int same = (string)curTableName == TableName && valid;
        for (int id = 0; id < number; id++) {
            char name[252]; int tp, uni, key, len;
            fscanf(in->second, "%s %d %d %d %d", name, &tp, &key, &uni, &len);
        }
    }
}
```

```

        InputData tmp = InputData(name, tp, key, uni, len);
        if (same) ret.push_back(tmp);
    }
    if (same) return ret;
}
return ret;
}

```

4.4 Index Manager 模块（索引和 B+ 树模块）

- B+ 树的实现

- B+ 树 是这个实验比较核心的部分。
- B+ 树 实时满足的性质
 - 对于 M 阶 B+ 树，每个节点拥有最多 M 个孩子， $M - 1$ 个键值。
 - 对于叶节点，储存 $\lceil \frac{M-1}{2} \rceil \sim M - 1$ 条记录。
 - 任意一个非根非叶节点，有 $\lceil \frac{M}{2} \rceil \sim M$ 个孩子。
 - 如果根不是叶节点，至少要有两个孩子。
- B+ 树 的节点定义

```

struct BPlusTree {
    Element key[M];
    BPlusTree *son[M + 1], *fa, *Left, *Right;
    int pos[M];
    int num, isleaf, isroot;
    int id; //记录这个点的编号
    static int tot;
    BPlusTree() {
        num = 0; id = ++tot;
        for (int i = 0; i <= M; i++) son[i] = nullptr;
        fa = Left = Right = nullptr;
        isleaf = isroot = 0;
    }
};

```

- key 表示每一层的键值，son 表示指向孩子的指针。
- Left 和 Right 表示左兄弟和右兄弟的指针，fa 表示父亲。
- pos 表示对应数据在文件里的 offset。

- B+ 树 的插入操作

1. 如果插入后键值个数不超过 $M - 1$ ，结束插入操作。
2. 如果当前是叶节点，将数据记录其分裂成等长的两块。在父亲处新建一个索引，其键值为右儿子第一条数据的值。

3. 如果当前是非叶节点，将索引分裂成等长的两块。将分裂中间的那个键值向父
亲传递。

- 的分裂过程的细节如下：

```
void Split(BPlusTree *cur) {
    BPlusTree *add = new BPlusTree();
    add->isleaf = cur->isleaf;
    add->fa = cur->fa;
    Element mid;
    if (cur->Right != nullptr)
        cur->Right->Left = add, add->Right = cur->Right;
    cur->Right = add; add->Left = cur;

    if (cur->isleaf) {
        mid = cur->key[(M + 1) / 2];
        add->num = M / 2;
        cur->num = M - add->num;
        for (int i = 0; i < add->num; i++)
            add->key[i] = cur->key[i + cur->num], add->pos[i] = c
ur->pos[i + cur->num];
    }
    else {
        mid = cur->key[M / 2];
        add->num = (M - 1) / 2;
        cur->num = M - 1 - add->num;
        for (int i = 0; i < add->num; i++) {
            add->key[i] = cur->key[i + cur->num + 1];
            add->son[i] = cur->son[i + cur->num + 1];
            add->son[i]->fa = add;
        }
        add->son[add->num] = cur->son[M];
        add->son[add->num]->fa = add;
    }

    if (cur->isroot) {
        cur->isroot = add->isroot = 0;
        BPlusTree *root = new BPlusTree();
        root->isroot = 1;
        root->isleaf = 0;
        root->num = 1;
        root->key[0] = mid;
        root->son[0] = cur;
        root->son[1] = add;
        cur->fa = add->fa = root;
        rt[rtIndexName] = root;
    }
}
```

```

    }
    else {
        InsertNode(cur->fa, mid, add);
    }
}

```

- B+ 树 的删除操作

1. 如果删除后，键值个数至少有 $\lfloor \frac{M-1}{2} \rfloor$ 个，结束删除操作。
 2. 找到任意一个同父亲的兄弟节点。
 3. 如果兄弟节点键值有盈余，直接分一个键值（和一个索引）给他，并结束删除操作。
 4. 否则他们合并后一定不会超过限制。直接将它们合并，并删除父亲对应的一条键值信息，递归地往上删除。
- 这是叶节点的部分删除代码：

```

if (cur->fa->son[0] == cur) {
    int id = GetID(cur->fa, cur);
    if (cur->Right->num > least) {
        SimpleInsert(cur, cur->num, cur->Right->key[0], nullptr,
cur->Right->pos[0]); //表示往 cur 的第 cur->num 位插入
        SimpleDelete(cur->Right, 0); //删除cur->Right开头
        cur->fa->key[id] = cur->Right->key[0];
    }
    else {
        for (int i = 0; i < cur->Right->num; i++)
            SimpleInsert(cur, cur->num, cur->Right->key[i], nullptr,
cur->Right->pos[i]);
        DeleteNode(cur->fa, id); //递归向上
        DelLink(cur->Right);
    }
}

```

- 这是非叶节点的部分删除代码：

```

if (cur->fa->son[0] == cur) {
    int id = GetID(cur->fa, cur);
    if (cur->Right->num > least) {
        SimpleInsert(cur, cur->num, cur->fa->key[id], cur->Right-
>son[0]);
        cur->Right->son[0]->fa = cur;
        cur->fa->key[id] = cur->Right->key[0];
        SimpleDelete(cur->Right, -1);
    }
    else {
        SimpleInsert(cur, cur->num, cur->fa->key[id], cur->Right-
>son[0]);
    }
}

```

```

        cur->Right->son[0]->fa = cur;
        for (int i = 0; i < cur->Right->num; i++) {
            SimpleInsert(cur, cur->num, cur->Right->key[i], cur->
Right->son[i + 1]);
            cur->Right->son[i + 1]->fa = cur;
        }
        DeleteNode(cur->fa, id);
        DelLink(cur->Right);
    }
}

```

- **B+ 树接口定义**

- **B+ 树** 已经实现完毕，并给 **Index Manager** 留下下列接口：

```

int CalcSize(BPlusTree *cur);
//计算以 cur 为根的树的大小
int InsertLeaf(BPlusTree *cur, Element key, int offset);
//往 B+ 树插入节点，键值是 key，值是 offset（对应的 block 位置）
void DeleteLeaf(BPlusTree *cur, Element key);
//往 B+ 树里删除一个键值是 key 的元素。
void LoadTree(string name);
//从 index 叫做 name 的这个文件里读入一棵 B+ 树。
void SaveTree(string name);
//将 index 叫做 name 的这棵 B+ 树写入对应文件里。
void Free(BPlusTree *cur);
//释放以 cur 为根的整棵 B+ 树的内存。

```

- **B+ 树的根节点维护**

- 显然，内存里同一时间可能存在多棵 **B+ 树**，而每次与 **B+ 树** 交互的时候，都需要知道当前树的根节点，所以有必要且只要维护所有树的根节点。
- 我们不妨在全局开一个 **map**，

```
map<string, BPlusTree*>rt;
```

- 即对于每一个 **indexName**，都保存其对应的根节点。

- **index 文件管理**

- 就像管理表头一样，**index** 文件管理由主文件 **_index_.txt** 和每个 **index** 构成的独立文件构成。
- 函数 **ReadIndex()** 在每次打开数据库 **exe** 时候执行（只做一次）。它会去访问 **_index_.txt** 文件的每一行，读取所有 **index** 的定义：每一行由 **fileName, attributeName, indexName** 组成。
- 全局有两个用来管理的 **map**：

```
map<pair<string, string>, string>IndexID1;
map<string, pair<string, string>>IndexID2;
```

- 它们分别是 `pair<fileName, attributeName> -> indexName` 和 `indexName -> pair<fileName, attributeName>`。显然它们是互逆的，修改也是同时的。之所以写两个，因为不同时候这两个都有需求。在 `ReadIndex()` 时要顺便维护好这两个 `map`。

```
void ReadIndex() {
    FILE *tmp = fopen("__index__.txt", "r+");
    if (tmp == NULL) return;
    File["__index__"] = tmp;
    auto in = File.find("__index__");
    fseek(in->second, 0, SEEK_SET);
    int number;
    if (fscanf(in->second, "%d", &number) == EOF) return;
    IndexID1.clear(); IndexID2.clear();
    while (number--) {
        char a[256], b[256], c[256];
        fscanf(in->second, "%s %s %s", &a, &b, &c);
        IndexID1[make_pair(a, b)] = c;
        IndexID2[c] = make_pair(a, b);
        LoadTree(c);
    }
}
```

4.5 Record Manager 模块 (数据记录和处理模块)

- 访问数据
 - 通过分析发现，有三类过程需要访问数据：
 - 在 `select` 操作时，找出符合要求的数据并打印。
 - 在 `create index` 操作时，建立 B+ 树要遍历所有的数据。
 - 在 `insert` 时，如果属性没有索引，要扫描一遍数据看是否有重复。
 - 以上三个过程可以写在一个函数里。
 - 正常扫描数据并判断条件是否成立。
 - 可以看做是条件为空的查询。
 - 可以看做是条件只有一个 `=` 的查询。
 - 同时增加两个参数 `visual` 和 `ins`，表示是否打印结果、是否建立 B+ 树

```
int Record::FindRecord(string TableName, vector<InputData>Table, vector<Condition>Cond, int visual = 0, int ins = 1) {
    //找到至少一个返回1, 没找到返回0, 类型出错返回-1
    vector<int>Index;
    Index.resize(Cond.size());
```

```

    for (int i = 0; i < Cond.size(); i++) {
        Index[i] = -1;
        for (int k = 0; k < Table.size(); k++)
            if (Table[k].name == Cond[i].name)
                Index[i] = k;
        if (Index[i] == -1) return -1;
    }
    vector<pair<string, int>> edit;
    for (int i = 0; i < Table.size(); i++) {
        auto it = IndexID1.find(make_pair(TableName, (string)Table[i].
name));
        if (it != IndexID1.end())
            edit.push_back(make_pair(it->second, i));
    }
    int appear = 0;

    auto in = File.find(TableName);
    if (in == File.end()) return 0;
    fseek(in->second, 0, SEEK_END);
    int where = ftell(in->second);
    int blocksize = DATASIZE * Table.size();
    int offset = where / blocksize;

    for (int i = 0; i < offset; i++) {
        bufferIter it = BFM::BufferManagerRead(TableName, i * blocksize,
Table.size());
        vector<Element> rec;
        for (int j = 0; j < Table.size(); j++) {
            Element cur = BFM::LoadData(it, j * DATASIZE);
            if (cur.tp != NONE) rec.push_back(cur);
        }
        if (rec.size() > 0) {
            //printf("%d %d\n", rec.size(), Table.size());
            assert(rec.size() == Table.size());
            int ok = 1;
            for (int t = 0; t < Cond.size(); t++){
                Element ret = rec[Index[t]], k = Cond[t].p;
                switch (Cond[t].op) {
                    case LESS: ok &= ret < k; break;
                    case MORE: ok &= ret > k; break;
                    case LESSQ: ok &= ret <= k; break;
                    case MOREQ: ok &= ret >= k; break;
                    case EQUAL: ok &= ret == k; break;
                    case NOTEQUAL: ok &= ret != k; break;
                    default: assert(0);
                }
            }
        }
    }
}

```

```

    }
    if (ok) {
        if (visual) {
            if (!appear) {
                printf("The result is:\n");
                for (auto k : Table)
                    printf("%12s", k.name);
                printf("\n");
            }
            for (auto k : rec)
                printf("%12s", k.d);
            printf("\n");
        }
        if (ins)
            for (auto k : edit)
                InsertLeaf(rt[rtIndexName = k.first], rec[k.se
cond], i * blocksize);
            appear = 1;
        }
        rec.clear();
    }
}
if (visual) printf("\n");
return appear;
}

```

- 删除数据

- 原理和访问数据差不多，只是在这个过程是为 `delete` 服务的。我们每遇到一个符合要求的记录，就要将它的 `type` 标记为 `0`，意为它被删除了。

- 插入数据

- 每次插入数据的时候，就要在对应文件的末尾新开出一段空间并输出。同时在 `Cache` 里增加这个 `block`，为以后的调用加速。

```

int Record::Insert(string TableName, vector<Element>Data) {
    try {
        auto out = File.find(TableName);
        fseek(out->second, 0, SEEK_END);
        int offset = ftell(out->second);

        Block *tmp = new Block(TableName, offset, Data.size());
        for (int i = 0; i < Data.size(); i++)
            BFM::SetData(tmp, i * DATASIZE, Data[i]);
        bufferIter it = BFM::InsertBlock(*tmp);
    }
}

```

```

        BFM::BufferManagerWrite(*tmp);
        return offset;
    }
    catch (...) {
        printf("The impossible thing happens in buffer insert.\n");
    }
}

```

4.6 API 模块 (核心程序控制)

- API模块是数据操作的主要逻辑代码

- 通过调用 `Index` , `Catalog` 和 `Record` 模块, 整合处理各种数据库操作。
- API的函数接口其实就是数据库的各项操作:

```

string DropTable (string TableName);
string CreateTable(string TableName, vector<InputData> Table);
string CreateIndex(string TableName, string AttributeName, string IndexName);
string DropIndex(string IndexName, int compulsory);
string Insert(string TableName, vector<Element>vec);
string Delete(string TableName, vector<Condition>Cond);
string QuickSelect(string TableName, vector<InputData>Table, string name, Element key);
string Select(string TableName, vector<Condition>Cond);

```

- `QuickSelect` 表示借助 `B+树` 来快速做 `select` 操作。
- 这里的数据格式已经被 `Interpreter`模块 翻译成最方便的格式。
 - 一组 `table` 的定义用 `vector<InputData>` 表示。
 - 一组条件用 `vector<Condition>` 表示。
 - 一组记录用 `vector<Element>` 表示。

- 举例: `DropTable()` 函数

- 分析一下删表时需要做的事:
 1. 首先判断这个 `table` 是否存在。
 2. 在 `bufser`模块里, 把当前 `cache` 里和这个表有关的 `block` 删除。
 3. 关闭指向当前表数据的文件指针, 并删除该文件。
 4. 在表头定义文件里删除该 `table`。
 5. 找到与这个表相关的所有 `index`, 将这些 `index` 都删除。

```

string API::DropTable(string TableName) {
    if (!CM->ExistTable(TableName))
        return "Table [" + TableName + "] doesn't exist.";
    BFM::deleteFile(TableName);
    DeleteFile(TableName);
}

```

```

CM->DropTable(TableName);
while (true) {
    auto it = IndexID1.lower_bound(make_pair(TableName, ""));
    if (it == IndexID1.end() || (it->first).first != TableName) break;
    DropIndex(it->second, 1);
}
return "Drop Table [" + TableName + "] successfully.";
}

```

- 举例：CreateIndex() 函数

1. 首先判断这个 `table` 是否存在。
2. 获得 `table` 的定义信息，找到要创建 `index` 的属性。
3. 判断该属性是否是 `unique`，是否已经建立过 `index`。
4. 判断该 `index` 的名字是否已经被使用过。
5. 建立索引文件，并在两个 `map` 里插入新的 `index` 信息。
6. 初始化 `B+` 树的根节点。
7. 调用 `RM->FindRecord` 扫描所有记录，建立 `B+` 树。

```

string API::CreateIndex(string TableName, string AttributeName, string name) {
    if (!CM->ExistTable(TableName)) return "Fail! Table [" + TableName + "] hasn't been created yet.";

    vector<InputData>Table = CM->GetTable(TableName);
    int k = -1;
    for (int i = 0; i < Table.size(); i++)
        if (Table[i].name == AttributeName) k = i;
    if (k == -1)
        return "Fail! Can't find [" + AttributeName + "] in the table.";

    if (Table[k].uni == 0)
        return "Fail! Attribute [" + AttributeName + "] isn't unique.";
    if (IndexID1.find(make_pair(TableName, AttributeName)) != IndexID1.end())
        return "Fail! The index of primary key has been created by default.";
        else return "Fail! The attribute [" + AttributeName + "] has created the index yet.";
    }

    if (IndexID2.find(name) != IndexID2.end())
        return "Fail! Index name [" + name + "] is used.";
    IndexID1[make_pair(TableName, AttributeName)] = name;
    IndexID2[name] = make_pair(TableName, AttributeName);
}

```

```

CreateFile(name, 0);

BPlusTree::tot = 0;
BPlusTree *cur = new BPlusTree();
cur->isroot = cur->isleaf = 1; cur->num = 0;
rt[rtIndexName = name] = cur;

vector<Condition>Null;
RM->FindRecord(TableName, Table, Null, 0, 1);

return "Index [" + name + "] is created successfully.";
}

```

4.7 Interpreter 模块（语法翻译器）

- 和用户交互

- Interpreter 是和用户交互的模块，会涉及到很多词法分析。
- 因为没学过编译原理，我采用“严格根据语法规则一步一步分析”的策略，把用户的需要解析成我的 minisQL 想要的格式。
- 因为是分类讨论一点点逼近每一条指令，我会大量用到 substr 这个函数。此外，为了处理多余空格，我手动实现了 python 里的 strip 函数。
- 以下是中枢模块：

```

string Interpreter::check(string cur, int canFile = 1) {
    cur = strip(cur);
    if (cur.substr(0, 6) == "create") {
        cur = strip(cur.substr(6));
        if (cur.substr(0, 5) == "table")
            return TryCreateTable(strip(cur.substr(5)));
        if (cur.substr(0, 5) == "index")
            return TryCreateIndex(strip(cur.substr(5)));
    }
    else if (cur.substr(0, 4) == "drop") {
        cur = strip(cur.substr(4));
        if (cur.substr(0, 5) == "table")
            return TryDropTable(strip(cur.substr(5)));
        if (cur.substr(0, 5) == "index")
            return TryDropIndex(strip(cur.substr(5)));
    }
    else if (cur.substr(0, 6) == "insert")
        return TryInsert(strip(cur.substr(6)));
    else if (cur.substr(0, 6) == "delete")
        return TryDelete(strip(cur.substr(6)));
    else if (cur.substr(0, 6) == "select")

```

```

        return TrySelect(strip(cur.substr(6)));
    else if (cur.substr(0, 8) == "execfile") {
        if (canFile) return SolveFile(strip(cur.substr(8)));
        return "You can't execute file in the file! This instruction will be ignored.";
    }
    return "Can't recognize your instruction!";
}

```

- 对于每一条指令分别写一个函数。我采用的这个做法不是很优美，代码会比较复杂，但是能给出详细的报错信息。拿 `create table` 函数举例：

```

string Interpreter::TryCreateTable(string cur) {
    if (cur.back() != ')') return "Fail! Find no right bracket.";
    cur.erase(--cur.end());
    int LeftBracket = cur.find('(');
    if (LeftBracket == -1) return "Fail! Find no left bracket.";
    if (LeftBracket == 0) return "Fail! Miss the table name.";
    string TableName = strip(cur.substr(0, LeftBracket));
    cur = cur.substr(LeftBracket + 1);
    if (cur == "") return "Fail! Table can't be empty.";
    vector<InputData>Table;
    while (true) {
        int comma = cur.find(',');
        if (comma == -1) break;
        string block = strip(cur.substr(0, comma));
        cur = strip(cur.substr(comma + 1));
        InputData now;
        if (block.size() > 6 && block.substr(block.size() - 6) == "unique")
            now.uni = 1, block = strip(block.substr(0, block.size() - 6));
        int space = block.find(' ');
        if (space == -1) return "Fail! Find no space to split type and name.";
        strcpy(now.name, strip(block.substr(0, space)).c_str());
        string type = reduce(block.substr(space + 1));
        if (type == "int") now.tp = INT;
        else if (type == "float") now.tp = FLOAT;
        else if (type.substr(0, 4) == "char" && type.substr(4, 1) == "(" & type.back() == ')') {
            try {
                int number = stoi(type.substr(5, type.length() - 6));
                if (number >= 1 && number <= 255)
                    now.tp = CHAR, now.len = number;
                else return "Fail! The number in char() is out of range.";
            }
            catch (...) {

```

```

        return "Fail! The number in char() can't be recognized.";
    }
}
else return "Fail! No such type like [" + type + "].";
for (auto t : Table)
    if (strcmp(t.name, now.name) == 0)
        return "Fail! The names of Attributes are duplicated.";
Table.push_back(now);
}
if (Table.size() > MAXELEMENT)
    return "Fail! The number of Attributes exceeds.";
if (cur.substr(0, 7) != "primary")
    return "Fail! The format of [primary key] is wrong.";
cur = strip(cur.substr(7));
if (cur.substr(0, 3) != "key")
    return "Fail! The format of [primary key] is wrong.";
cur = strip(cur.substr(3));
if (!cur.size() || cur[0] != '(' || cur.back() != ')')
    return "Fail! The bracket of primary key is not correct.";
cur = strip(cur.substr(1, cur.length() - 2));
int Findkey = 0;
for (auto &t : Table)
    if ((string)t.name == cur) {
        t.uni = t.key = 1;
        Findkey = 1;
    }
if (!Findkey)
    return "Fail! primary key [" + cur + "] is not in the table.";
return api->CreateTable(TableName, Table);
}

```

- 整个程序构架

- 加载和保存数据

- 在 `exe` 刚开始启动的时候，我们需要管理文件指针，并从一些数据文件中读取数据（`index` 信息，`table` 信息）。
 - 由以下 `Load` 函数实现。

```

void Interpreter::Load() {
    ReadIndex();
    api->CM->ReadTable();
}

```

- 在保存或者关闭 `exe` 时，我们需要保存数据（`B+` 树，`cache` 里的 `block`）和关闭文件指针。由以下 `save()` 函数实现。

```

void Interpreter::Save() {
    BFM::BufferManagerFlush();
    SaveIndex();
    SaveAllTree();
    for (auto t : File)
        fclose(t.second);
    File.clear();
    Load();
}

```

- 注意如果是 `save` 的话，还需要重新 `load` 一遍。

- 主程序实现

- 注意每次是默认读到分号的，所以没有见到分号就要一直读入（包括回车）。
- 但是每遇到回车后，就要像传统的命令行 `sql` 一样也换个行。

```

int main() {
    Interpreter *cur = new Interpreter();
    cur->Load();
    while (true) {
        string Command = "";
        printf("minysql-->");
        while (true) {
            char ch = getchar();
            if (ch == '\n') break;
            Command += ch;
        }
        int ed = Command.find(';');
        if (ed != -1) {
            Command = Command.substr(0, ed);
            break;
        }
        printf("      -->");
    }
    if (Command.substr(0, 4) == "quit") break;
    int Time1 = clock();
    if (Command.substr(0, 4) == "save") {
        cur->Save();
        printf("Save successfully.\n");
        int Time2 = clock();
        printf("Time cost : %.4fs\n", (Time2 - Time1) * 1.0 / CLOCKS_PER_SEC);
        continue;
    }
}

```

```
    string Message = cur->check(Command);
    if (Message != "") {
        printf("%s\n", Message.c_str());
        int Time2 = clock();
        printf("Time cost : %.4fs\n", (Time2 - Time1) * 1.0 / CLOCKS_
PER_SEC);
    }
    cur->Save();
    return 0;
}
```

5. 实验结果与测试

- 创建一个表

```
test1.sql test2.sql test3.sql test4.sql
1 create table t1
2 (
3     id int ,
4     name char(20) unique,
5     age int ,
6     salary float,
7     primary key(id)
8 );
```



```
E:\ZJU大学生活\课程学习\数据库\miniSQL\Try\Debug\Try.exe
minisql-->execfile test1. sql;
Create Table [t1] successfully.
Execute file [test1. sql] successfully!
Time cost : 0.0050s
minisql-->
```

- 插入数据

```
test1.sql test2.sql test3.sql test4.sql test5.sql
1 insert into t1 values(1,'Jim',20,2000.00);
2 insert into t1 values(2,'Kate',24,1800.00);
3 insert into t1 values(3,'John',34,4000.00);
4 insert into t1 values(4,'Marry',20,3000.00);
5 insert into t1 values(5,'Tom',24,1850.00);
6 insert into t1 values(6,'Queen',28,24000.00);
7 insert into t1 values(7,'Porry',17,1000.00);
8 insert into t1 values(8,'Green',24,8000.00);
9 insert into t1 values(8,'Jim',20,4000.00);
10 insert into t1 values(9,'Green',22,4000.00);
```



```
minisql-->execfile test2. sql;
Insert Values into [t1] successfully.
Fail! Unique value [id] has existed yet.
Fail! Unique value [name] has existed yet.
Execute file [test2. sql] successfully!
Time cost : 0.0150s
minisql-->
```

- 注意到，最后两条记录分别违反 `primary key` 和 `unique` 属性的唯一性。

- 基本的运算符检验

```
test3.sql test4.sql test5.sql test6.sql test7.sql test8.sql
1 select * from t1;
2
3 select * from t1 where id > 6;
4 select * from t1 where id < 3;
5 select * from t1 where id >= 6;
6 select * from t1 where id <= 3;
7 select * from t1 where id = 6;
8 select * from t1 where id != 6;
9
10 select * from t1 where salary < 1850.00;
11 select * from t1 where salary >= 4000.00;
12 select * from t1 where salary = 3000.00;
13 select * from t1 where salary != 3000.00;
```

E:\ZJU大学生活\课程学习\数据库\miniSQL\Try\Debug\Try.exe

```
minisql-->execfile test3.sql;
The result is:
    id      name      age      salary
    1       Jim       20      2000.00
    2       Kate      24      1800.00
    3       John      34      4000.00
    4       Marry     20      3000.00
    5       Tom       24      1850.00
    6       Queen     28      24000.00
    7       Porry     17      1000.00
    8       Green     24      8000.00

The result is:
    id      name      age      salary
    7       Porry     17      1000.00
    8       Green     24      8000.00

The result is:
    id      name      age      salary
    1       Jim       20      2000.00
    2       Kate      24      1800.00

The result is:
    id      name      age      salary
    6       Queen     28      24000.00
    7       Porry     17      1000.00
    8       Green     24      8000.00

The result is:
    id      name      age      salary
    1       Jim       20      2000.00
    2       Kate      24      1800.00
    3       John      34      4000.00

The result is:
    id      name      age      salary
    6       Queen     28      24000.00
```

```

The result is:
      id      name      age    salary
      6       Queen     28   24000.00
The result is:
      id      name      age    salary
      1       Jim       20   2000.00
      2       Kate      24   1800.00
      3       John      34   4000.00
      4       Marry     20   3000.00
      5       Tom       24   1850.00
      7       Porry     17   1000.00
      8       Green     24   8000.00

The result is:
      id      name      age    salary
      2       Kate      24   1800.00
      7       Porry     17   1000.00

The result is:
      id      name      age    salary
      3       John      34   4000.00
      6       Queen     28   24000.00
      8       Green     24   8000.00

The result is:
      id      name      age    salary
      4       Marry     20   3000.00

The result is:
      id      name      age    salary
      1       Jim       20   2000.00
      2       Kate      24   1800.00
      3       John      34   4000.00
      5       Tom       24   1850.00
      6       Queen     28   24000.00
      7       Porry     17   1000.00
      8       Green     24   8000.00

```

Execute file [test3.sql] successfully!
Time cost : 0.1050s

- 条件连接的检验

```

test4.sql test5.sql test6.sql test7.sql test8.sql
1  select * from t1 where id > 4 and salary >= 2000.00;
2  select * from t1 where age > 25 and age < 30;

```

minisql-->execfile test4.sql;

The result is:

id	name	age	salary
6	Queen	28	24000.00
8	Green	24	8000.00

The result is:

id	name	age	salary
6	Queen	28	24000.00

Execute file [test4.sql] successfully!
Time cost : 0.0300s

- 字符串比较和 index 创建

```
test5.sql x test6.sql x test7.sql x test8.sql x
1 select * from t1 where name = 'Jim';
2 select * from t1 where name != 'Jim';
3 select * from t1 where name >= 'Queen';
4 select * from t1 where name <= 'Jim';
5
6 create index iname on t1(name);
7
8 select * from t1 where name = 'Jim';
```

minisql-->execfile test5.sql;

The result is:

id	name	age	salary
1	Jim	20	2000.00

The result is:

id	name	age	salary
2	Kate	24	1800.00
3	John	34	4000.00
4	Marry	20	3000.00
5	Tom	24	1850.00
6	Queen	28	24000.00
7	Porry	17	1000.00
8	Green	24	8000.00

The result is:

id	name	age	salary
5	Tom	24	1850.00
6	Queen	28	24000.00

The result is:

id	name	age	salary
1	Jim	20	2000.00
8	Green	24	8000.00

Index [iname] is created successfully.

The result is:

id	name	age	salary
1	Jim	20	2000.00

Execute file [test5.sql] successfully!

Time cost : 0.0290s

- **删除记录**

```
test5.sql x test6.sql x test7.sql x test8.sql x
1 delete from t1 where id > 7;
2 delete from t1 where id <= 2;
3 delete from t1 where id < 4 and salary > 3000.00;
4 select * from t1;
```

```

minisql-->execfile test6.sql;
Delete 1 records successfully.
Delete 2 records successfully.
Delete 1 records successfully.
The result is:
    id      name      age      salary
    4      Marry      20      3000.00
    5      Tom        24      1850.00
    6      Queen      28      24000.00
    7      Porry      17      1000.00

Execute file [test6.sql] successfully!
Time cost : 0.0140s

```

- 删除索引和表

```

1 drop index iname;
2
3 drop table t1;
4 select * from t1;

```

```

minisql-->execfile test7.sql;
Index [iname] is dropped successfully.
Drop Table [t1] successfully.
Fail! Table [t1] hasn't existed yet!
Execute file [test7.sql] successfully!
Time cost : 0.0370s

```

- B+ 树检验：插入数据

- 在插入数据的时候，如果一个属性是 `unique` 的，每次插入都要扫描一遍表里的所有元素，复杂度是平方级别的。但是如果预先建立了 `index`，插入时只要在 `B+` 里检验是否重复即可，耗时可以忽略不计。

```

freopen("test8.sql", "w", stdout);
printf("create table t2(id int, id2 int unique, primary key(id));\n");
//printf("create index work2 on t2(id2)\n");
for (int i = 1; i <= 400; i++)
    printf("insert into t2 values(%d, %d);\n", i, i);

```

- 以上代码输出到 `test8.sql` 中。先不建立 `index` 插入数据，然后删除数据库信息，创建 `index`（去掉注释）并重新插入数据：

```
E:\ZJU大学生活\课程学习\数据库\miniSQL\Try\Debug\Try.exe
minysql-->execfile test8.sql;
Create Table [t2] successfully.
Insert Values into [t2] successfully.
The number of instructions is too large, the rest results will be hidden.
Execute file [test8.sql] successfully!
Time cost : 10.6620s
```

```
E:\ZJU大学生活\课程学习\数据库\miniSQL\Try\Debug\Try.exe
minysql-->execfile test8.sql;
Create Table [t2] successfully.
Index [work2] is created successfully.
Insert Values into [t2] successfully.
The number of instructions is too large, the rest results will be hidden.
Execute file [test8.sql] successfully!
Time cost : 0.2910s
```

- 容易发现，两者运行时间有着显著差别。

- **B+ 树检验：查询数据**

- 先往 B+ 数里插入 20000 条数据。

```
freopen("test9.sql", "w", stdout);
printf("create table t3(id int, id2 int unique, primary key(id));\n");
printf("create index work3 on t3(id2)\n");
for (int i = 1; i <= 20000; i++)
    printf("insert into t3 values(%d, %d);\n", i, i);
```

- 删除索引|查询和建立索引|查询：

```
minysql-->drop index work3;
Index [work3] is dropped successfully.
Time cost : 0.0090s
minysql-->select * from t3 where id2 = 15000;
The result is:
      id      id2
     15000     15000

Time cost : 7.3160s
minysql-->create index work3 on t3(id2);
Index [work3] is created successfully.
Time cost : 7.9140s
minysql-->select * from t3 where id2 = 15000;
The result is:
      id      id2
     15000     15000
Time cost : 0.0090s
```

- 没有索引的时候需要 7.3s 才能查询到数据。而建立索引后，单组查询只要 0.01s。