浙江大学

数据库系统实验报告

作业名称：	MiniSQL IndexManager
姓　　名：	王晨宇
学　　号：	3180104919
电子邮箱：	646645903@qq.com
联系电话：	13506707122
指导老师：	孙建伶

2020 年　6 月　20 日

MiniSQL IndexManager

一、 实验目的

设计并实现一个精简型单用户 SQL 引擎(DBMS) MiniSQL，允许用户通过字界面/图形界面输入 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. **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）。

2. **索引的插入和删除对应的节点**

   对于建立了索引的记录，在插入或删除一条记录时，也要相应地在 B+中删除相应的节点。

3. **索引的查询**

   在查询语句涉及到建立了索引的键值时，通过在 B+树上查询，提高查询速度

三、 实验环境

操作系统：Windows 10
编程语言：C++
开发环境：Visual Studio 2019

四、 Index Manager 模块设计

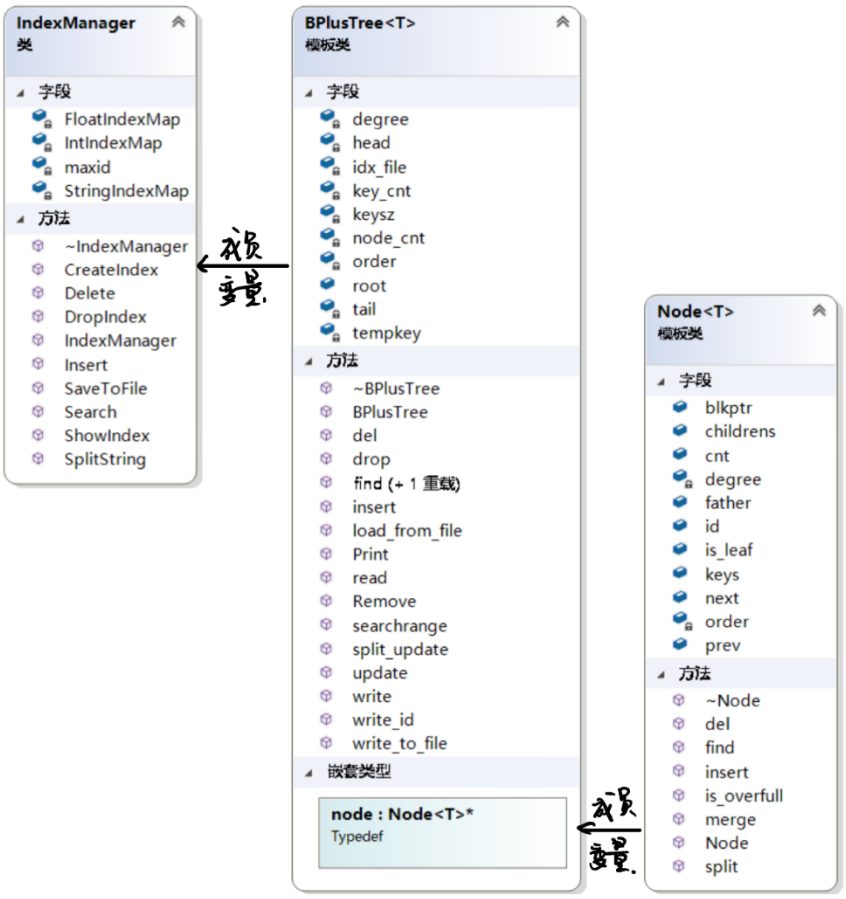
1. 功能描述

```
//根据API提供的文件名创建索引
void CreateIndex(string filename);
//根据API提供的文件名删除索引
void DropIndex(string filename);
//根据API提供的文件名、键值和关系运算符在相应的B+树中查找
// 返回查到的一系列记录在文件中的偏移地址
vector<int> Search(string filename, index_node val, OP op);
//根据API提供的文件名在相应的B+树中插入键值
void Insert(string filename, index_node val, int blockoffset);
//根据API提供的文件名在相应的B+树中删除键值
void Delete(string filename, index_node & val);
```

2. 主要数据结构
   1. Index Manager : STL map<T, int> 实现索引文件和 B+树的查找
   2. BPlusTree: BPlusTreeNode *，实现一棵 B+树
   3. BPlusTreeNode: STL vector<T> key; STL vector <BPlusTreeNode*> father,

children;　在一个树节点中保存键值和儿子等信息

3. 类图与类间关系



五、 模块实现

1. 阐述该模块所使用的数据结构，说明实现各个功能的核心代码(给出简化后的代码截图)，包括重要的函数、提供给 API 模块的接口(如果有)等。
   1. 数据结构
      1. Index Manager:

```
class IndexManager {
   private:
    //每种类型的索引需要一个不同的map 映射文件名到b+树
    map<string, BPlusTree<int>*> IntIndexMap;
    map<string, BPlusTree<float>*> FloatIndexMap;
    map<string, BPlusTree<string>*> StringIndexMap;
    map<string, int> maxid;
   public:
    IndexManager();
~IndexManager();
//创建索引
void CreateIndex(string filename);
//删除索引
```

```
void DropIndex(string filename);
//打印索引（调试用）
    void ShowIndex(string filename);
    //保存索引到本地
    void SaveToFile();
    //查询键值
vector<int> Search(string filename, index_node val, OP op);
//插入键值
    void Insert(string filename, index_node val, int blockoffset);
    //删除键值
    void Delete(string filename, index_node & val);
    //判断键值的数据类型
    IFStype SplitString(string str);
};
```

因为用 template 实现 B+树对各种类型键值的适配，所以需要不同类型的 map 来迅速找到是否存在一个需要的索引。

2. BPlusTreeNode:

```
template <typename T>
class Node {
   private:
    int order, degree; //树的容量和度数
   public:
    int cnt; //键值总数
    int id;  //标号
Node* father; // 父亲节点
vector<T> keys; // 节点内所有键值
vector<int> blkptr; // 叶子节点中键值对应的record 偏移地址
vector<Node*> childrens; // 儿子节点
Node* next; // 后继节点（兄弟）
Node* prev; // 前驱节点（兄弟）
    bool is_leaf; //是否为叶子

    Node(int id = 0, int ord = 3, int deg = 3, int _cnt = 0, Node* _fa = NULL, Node
* _nxt = NULL, Node* _pre = NULL, bool is_new_leaf = false);
    ~Node();
    bool is_overfull(); //是否超过容量
    bool find(T key, int& pos); //查找一个键值
    Node* split(T& val); //分裂
    void insert(T key, int blockoffset = 0, Node* ch = NULL); //插入键值
    void del(int pos); //删除某一位置上的键值
    void merge(); //合并相邻的兄弟
```

```
};
```

3. BplusTree：

```
template <typename T>
class BPlusTree {
    typedef Node<T>* node;

   private:
    //底层的链表头与尾
node head, tail;
//度数、容量节点大小等信息
    int keysz, degree, order;
int key_cnt, node_cnt;
//存储文件名
    string idx_file;
    T tempkey;

   public:
    //根节点
    node root;
    BPlusTree(string _idx_file_ = "", int _keysz = 8, int _degree = 4,
              int _order = 4);
~BPlusTree();
//插入键值和对应的偏移地址
void insert(T val, int blockoffset);
//树内查找
pni find(T val, node now);
//重载find，为与indexmanager 的接口
int find(T val);
//删除一个键值
int del(node now, node brother, T Parentkey);
//与indexmanager 的接口
    int Remove(T val);
    //插入后分裂并维护
    void split_update(node now);
    //打印自己（调试用）
void Print();
    //删除后维护
    void update(node now, T oldval, T newval);
    void write_to_file();
    void write_id(ofstream &fp, node now);
    void write(ofstream &fp, node now);
    bool read(ofstream &fp, node &now);
```

```
    bool load_from_file();
    //删除自己，释放空间
    void drop(node now);
    //范围查询，为与 indexmanager 的接口
    vector<int> searchrange(T val, OP op);
}
```

2. 接口部分核心代码
    1. 与 API 的接口：
       由于所有函数均涉及到 int、float、string 三种类型的索引，篇幅限制我只列出 int 类型索引的功能部分：

```
//根据文件名创建索引
void CreateIndex(string filename);
//根据文件名删除索引
void DropIndex(string filename);
//根据文件名和关系运算符在索引中查找
vector<int> Search(string filename, index_node val, OP op);
//根据文件名在索引中插入键值
void Insert(string filename, index_node val, int blockoffset);
//根据文件名在索引中删除键值
void Delete(string filename, index_node & val);
```

    2. 与 Buffer 的接口

```
//从 buffer 中获得树中的一个节点
void  build_node();
```

3. B+树中的核心代码
    1. 查找：API 调用 Index Manager，在 Search 中调用如下的查找函数进行查询。

```
Function find(int id)
        Apply Binary search on records.
        If record with the search key is found
            return required record
        Else if current node is leaf node and key not found
            print Element not Found
```

    2. 插入：API 调用 Index Manager，在 Insert 中调用如下插入函数插入一个键值到索引。

```
Function INSERT(int id)
    If the bucket is not full (at most b 1 entries after the insertion), add the re
cord.
    Otherwise, split the bucket.
        Allocate new leaf and move half the buckets elements to the new bucket.
```

```
        Insert the new leafs smallest key and address into the parent.
        If the parent is full, split it too.
                Add the middle key to the parent node.
        Repeat until a parent is found that need not split.
    If the root splits, create a new root which has one key and two pointers.
```

3. 删除：

   API 调用 Index Manager，在 Insert 中调用如下插入函数删除索引的一个键值。在遇到的问题和解决方法中有详细的图示。

```
Function remove(int K)
    Start at the root and go up to leaf node containing the key K
    Find the node n on the path from the root to the leaf node containing K
        If n is root, remove K
            if root has mode than one keys, done
            if root has only K
                if any of its child node can lend a node
                   Borrow key from the child and adjust child links
                Otherwise merge the children nodes it will be new root
            If n is a internal node, remove K
                If n has at lease ceil(m/2) keys, done!
                If n has less than ceil(m/2) keys,
                    If a sibling can lend a key,
                        Borrow key from the sibling and adjust keys in n and the pa
rent node
                        Adjust child links
                    Else
                        Merge n with its sibling
                        Adjust child links
            If n is a leaf node, remove K
                If n has at least ceil(M/2) elements, done!
                    In case the smallest key is deleted, push up the next key
                If n has less than ceil(m/2) elements
                If the sibling can lend a key
                    Borrow key from a sibling and adjust keys in n and its parent n
ode
                Else
                    Merge n and its sibling
                    Adjust keys in the parent node
```

2. 模块的测试代码

   模块的测试主要是 B+树对各种操作的正确性。

   1. 首先实现了一个检查 B+树经过各种操作之后是否仍然满足 B+树的各种性质

的函数：bool SelfCheck：

```
template <typename T>
bool BPlusTree<T>::SelfCheck(node now, T l, T r){
    if (!now) return true;
    检查自己；
    检查所有孩子；
    return flag;
}
```

它可以检查每个节点内是否满足 b+树的要求，同时检查相对的大小关系、叶子节点的顺序等。简言之，检查经过操作后这颗 B+树是否仍是 B+树。

2. 为了直观地看出 b+树结构上的问题，实现了一个自我打印的函数:

```
template <typename T>
void BPlusTree<T>::Print() {
    if (!root) {puts("NULL!"); return;}
    以 BFS 的方式遍历树中所有的点并输出；
}
```

配合 Node 类中重载的输出函数，效果如下（按层打印，共三层）：

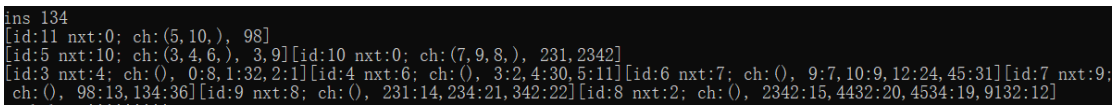


Figure 1 输出 B+树效果展示

3. 为了检验经过删除、插入后查询结果的正确性，写了一个由 set 实现的简易 b+树配合一个 Shell 脚本进行大数据量的插入、删除和查询操作，并比对结果：

```
set<int> s;
int main() {
    freopen("1.in", "r", stdin);
    int n, order; cin >> n >> order;

    for (int j = 1; j <= n; j++) {
        bool flag = true;
        int x, y, op; cin >> op;
        switch (op)
        {
        case 0: cin >> x; s.erase(x); break;
        case 1: cin >> x; s.insert(x); break;
        case 2: cin >> x >> y;
            for (auto i : s) if (i >= x && i <= y) cout << i << " ";
            puts("");
        }
    }
}
```

检查的内容包括各种度数（order）、操作的数量（n）、数据的类型（int、string、float）

和重复键值、删除不存在的键值等特殊情况等。经过长时间的验证，保证了实现的正确性

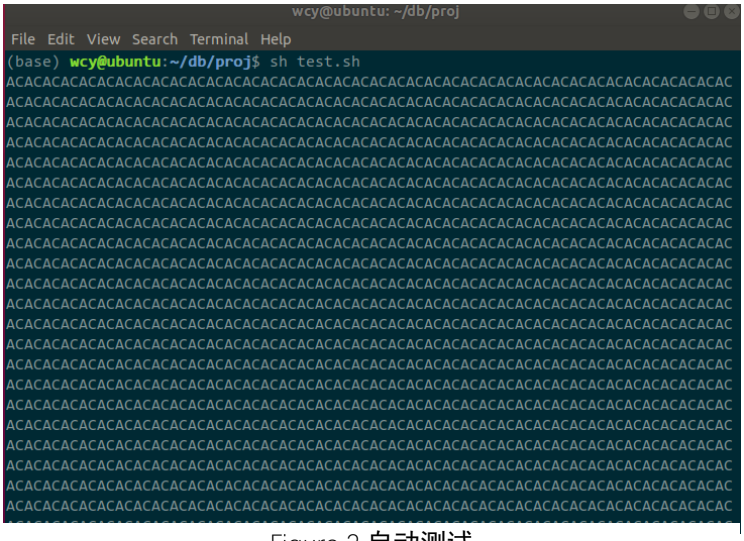


Figure 2 **自动测试**

六、 遇到的问题及解决方法

1. 面向对象编程过程中的一些问题和解决方法

由于采用 C++实现，在面向对象编程的过程中也遇见了很多问题。这部分更像是 oop 的知识学习和经验总结。

1. 使用 template 定义的类的所有成员变量成员函数必须写在同一个文件中，不能像通常一样定义在头文件中而把成员函数写到另一个文件里。
2. 使用模板的参数在作为函数参数在传参之前必须先显式调用构造函数，再传参：

```
T x = T();//T is a template
return del(root, NULL, x);
```

3. 对于模板类的判断，没有找到非常好的办法。在一个函数中也不能通过分类讨论定义若干个类型不同的独立的变量进行操作。所以最后还是采用了通过传递一个包含了所有类型的结构体和一个类型信息，分类讨论进行读取。

2. B+树实现时的一些问题和解决方法

对于 B+树的实现，书上的伪代码比较抽象，偏向描述性，在实现过程中遇见了比较多的困难。想在互联网中找到一些相关的资料，但是经过仔细分析后发现大多数的现成模板都存在着一些问题，没能完整地实现 B+树的功能。最后还是自己独立地实现了所有功能。

1. 在基础的插入查询操作的实现过程中，因为诸如 lower_bound 返回的具体大小和位置关系、分裂时指针的指向、根的调整等遇到了一些问题，插入和查询不能正常进行。通过上一章节提到的测试代码，实现了比较复杂、全面的测试方式，一方面可以输出整棵树的结构，可视化地调试；另一方面也有大数据量的复杂操作组合测试。最后保证了基础操作的正确性。

2. 删除操作的正确性。删除操作是 B+树实现过程中最复杂的部分。一方面是节点的递归合并操作，这部分的正确性同样可以通过测试代码得到检验，同时我也自己做了图来详细给出每一种情况的删除过程前后的节点、元素位置、键值大小关系，更好地弄清每一步的具体操作；但是还有一种情况比较难以

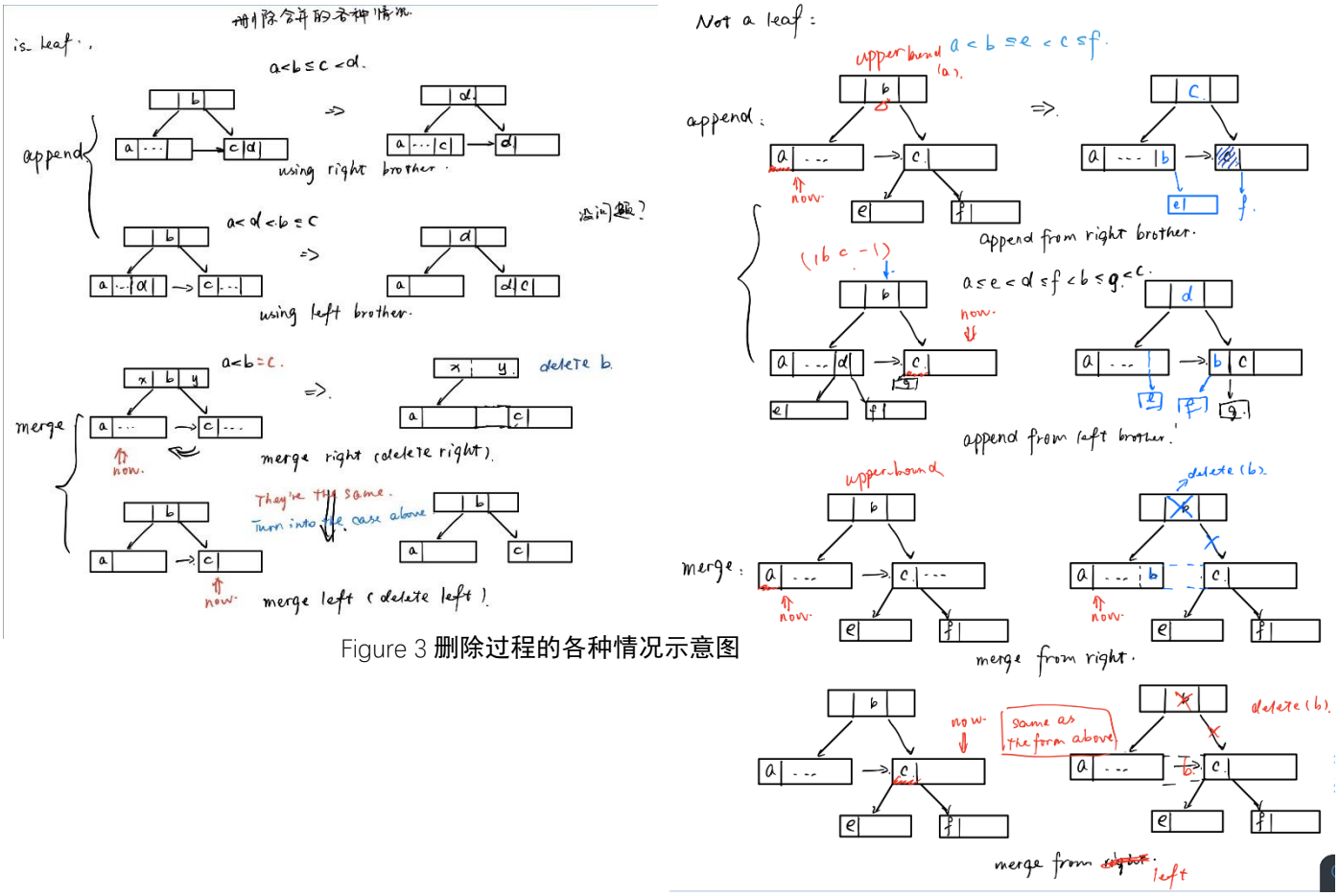


Figure 3 删除过程的各种情况示意图

处理，这也是我在参考许多代码发现很多人都没有很好解决或者根本就没有解决的地方。若要删除的元素在非叶子节点中当作一个比较的中间键值，那么我们删除的时候也应该把它出现在非叶节点中的位置删去。但是这就涉及到新的中间键值的调整。在处理中我把删除及合并之后的叶子节点第一个键值作为可能需要替换的新中间键值，向上递归，在祖先节点的 entries 中查找，若找到了，用新的中间键值替换。这一部分的代码如下图所示：

```
template <typename T>
void BPlusTree<T>::update(node now, T oldval, T newval){
    if (!now) return;
    int pos = 0;
    //在当前点中找到，则修改
    if (now->find(oldval, pos)) {now->keys[pos] = newval; return;}
    //否则继续递归
    update(now->father, oldval, newval);
}
```

完成上述步骤后再从最初的叶子节点开始向上调整（合并）。

3. 和其它模块对接的过程中遇到的一些问题。因为这一部分主要是和 API 及 Buffer 模块对接，和 buffer 的对接没有太大的问题。和 API 对接的模块大多是打包了的一个语句信息，和测试时的查询没有太大的区别。但是在验收的时候也遇见了删除后查询出一个无效信息的情况。也曾被助教认为是 B+树的

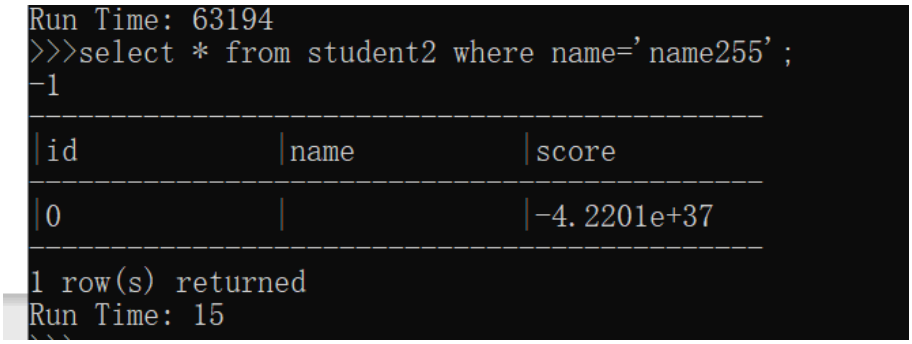


Figure 4 **验收时遇见的问题**

删除操作写的有问题。但是经过调试后发现是传递的查询键值没有对上。这部分的解决办法基本上都是输出中间变量或是单步调试。因为大家各自都不熟悉对方模板内的成员函数，所以虽然这样效率不高但是一方面帮助大家了解了各自模块的具体内容，一方面便于找到其它的 bug。

七、 总结

这次的 miniSQL 实现大概是我第一次参与这样分工明确的较大的项目开发。很高兴能和两个优秀的队友完整地实现了要求的内容以及部分 bonus point。在这个过程中我收获很多，同时也有很多经验教训。在此总结如下：

1. 算法的实现过程

因为在《高级数据结构与算法分析》课程作业中，我们实现了一个只有插入和查询的度数确定的 B+树，在有些简略地实现了它之后我以为完整的 B+树的实现应该不是非常困难，所以有些轻视它。在原来的基础上拓展的过程也耗费了比较长的时间。

后来在写到删除部分时遇到了比较大的困难，调试了好几个晚上发现之前实现的是一个假算法。有点打击信心，也曾经想过找个看起来能用的现成代码改一改。但是在理清了删除的逻辑、画出图来直观地看整个过程之后我还是很快地重写出了正确的算法。并通过大量的测试保证了整个过程的正确性。让之后的整合、调试工作减少了工作量。

一开始小看了它的实现难度，本来还打算帮队友干点别的，感觉有点点愧疚。在实现之前就应该先画出核心部分的流程图，理清过程的逻辑，会让之后的实现变得简单很多；复杂的测试是必要的。没有人比我更懂 B+树，这大概是测试完没有问题之后我最想说的话。

2. 和队友的整合、调试过程

一开始我们还是用 github 作为版本控制和文件共享的工具，但是因为我对 github 不是非常熟练，使得它不仅没有提高效率，反而使过程变得繁琐麻烦。在后来的调试之中因为每次修改的内容很少，干脆就在 qq 群或是电话中通知大家同步修改。但是这样还是会出现一些版本不统一的情况。还是应该熟练掌握 git 的各种功能，这样在以后的小组合作中可以更加优雅、高效地合作。

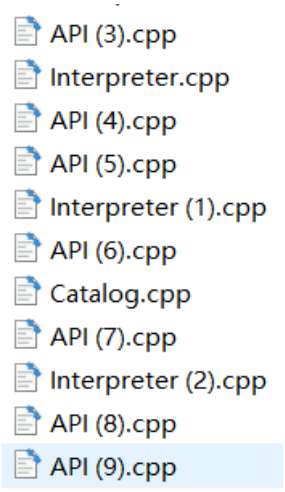


Figure 6 用 qq 共享的若干版本

Figure 5 逐渐被放弃的 repository

在开发工具上，我第一次尝试使用了 Visual Studio。以前都是写一些文件较少的小项目，并且都是自己写的，调试起来比较知道问题大概会出现在哪里，可以比较快地通过输出中间变量或是针对性的数据找到问题所在。但是这次的合作项目规模较大，并且对于其它模块也不太熟悉，所以就比较依赖于 vs 提供的单步调试。我第一次知道了这个 IDE 有这么强大的功能。但是它生成的 DEBUG 文件过于占用硬盘资源，让本来就不富裕的硬盘空间雪上加霜。

3. 功能实现的总结和拓展的思考

我们完整地实现了指导书中所要求的功能，在完全由自己造轮子的情况下我觉得非常不容易。尤其是卢佳盈同学在实现 interpreter 模块时完整地实现了一套异常处理，并且在完成之后动手做了一个图形界面。非常感谢她。当然因为时间比较短，许多基础功能的实现方法可能都比较简单直接，没有更进一步地考虑效率。但是利用索引的等值查询还是做到了毫秒级别，我觉得还比较优秀。

在功能的拓展上，对中文的支持得益于我们实现时使用了比较通用、鲁棒性更高的结构，算是意料之外，情理之中。本来有一个对于联合索引的一个初步的设想：索引的每个键值都是一个 vector，保存了一串结构体（含三种类型），每个结构体对应了原关系中的一个键。同时一个类型 vector 保存了对应的每个键具体是什么数据类型。这样就可以把原来的单键值索引扩展到联合索引；同时可以自动生成一个额外的隐式键，这样就可以支持在数据不确定为 unique 的键上创建索引。但是这样对于 interpreter、API、和 Index Manager 来说都是一个比较大的工作量，而且它的效率也不一定会更高。最后只好放弃实践的尝试。

总而言之，这次的小组合作我收获非常多，对 DBMS 的认识也更加深刻。数据库不是简简单单的读取文件和记录比对——那是我之前脑子里的刻板印象——它的背后还有很多高深的原理，值得我在之后的学习、开发之中去体会、探索、发现。我们实现的 miniSQL 相比较商业数据库来时还是过于 mini 了，在之后的学习中我也会继续努力。