简易数据存储系统说明文档

515030910038　　张子扬

2016.7.25

一、　概述

本简易数据存储系统由散列表实现，在本文档中，分别介绍了散列函数的选择与算法原理，散列冲突的解决方法，索引文件与数据文件的组织方法，索引结构体的实现与数据库类各功能的实现以及测试方法与效率、正确性分析。

二、　散列表的实现

在散列表与 B+树中，选择散列表的原因是散列表比较容易实现，逻辑结构更清楚。散列表的大小与数据库规模有关，要求测试 100 万条数据，我的默认大小设为 2000 万，。这只是为了减少测试时间，散列表大小是用宏写的，方便修改。散列表并未以实在的数据结构出现（如数组），而是借助其概念，在组织索引文件时应用。因此，散列表越大，索引文件也越大。针对散列表大小（空间）与性能（时间）的矛盾会在第九部分分析。每个散列表的元素都是一个索引结构体，即我实现的索引结构体，散列表的每个索引结构体包含的内容如图所示：

```
struct Idx {
    char key[KEYSIZE_MAX];      //索引字符串
    unsigned int value_off;     //数据在数据文件中偏移量
    unsigned int off;           //索引在索引文件中偏移量
    unsigned int off_next;      //索引链表中当前索引的下一跳索引的偏移量
    int len_key;                //索引字符串的长度
    int len_value;              //数据字符串的长度
    bool isDelete;
};
```

其中重要的内容是索引偏移量`unsigned int off;　　//索引在索引文件中偏移量`，数据偏移量`unsigned int value_off;　　//数据在数据文件中偏移量`，它们都是表示对应的内容在文件中的位置，在读文件定位文件指针用到。前者实际上就是通过计算键值的散列值处理后的结果，在解决散列冲突时有重要应用；后者是由一个数据库的私有变量不断赋值得到，在第六部分会分析。

三、　散列函数分析

散列函数的选择是重要的主题，针对不同的键的数据类型，有不同适合的散列函数，本数据库采用 char*字符串类型的键值，使用了下列算法:

```
unsigned int DB::hash(const char* key) {
    unsigned int seed = 1313131;
    unsigned int h = 0;
    while (*key) {
        h = h*seed + (*key++);
    }
    return ((h & 0x7FFFFFFF) % HASH_SIZE);
}
```

本算法参考了著名的 BKDR 算法，由一个字符串得到其散列值，为了减少冲突，应该使该字符串中每个字符都参与散列值计算，使其符合雪崩效应。直接求和容易键值冲突，解决方法是对字符间的差距进行放大，取定一个系数，用系数的 n 次方

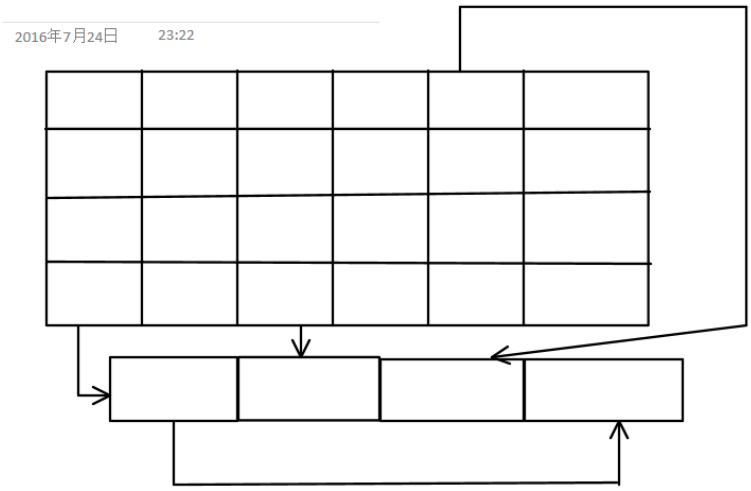
作为每个字符的系数。我大大降低了碰撞的发生，假设有个字符数组 p，有 n 个元素，那么：

SUM(p$\binom{n}{0}$) = 系数^(n-1) * p(n-1) +系数^(n-2) * p(n-2) + … + 系数^1 * p(1) + 系数^0 * p(0)

系数的选择是奇数，选择原因不再赘述，选择奇数会充分利用每一位，减少键值冲突，这里选择的是 1313131。将得到的 h 与 0x7fffffff（即最大整形）位与运算，再与散列表大小取余数，得到一个无符号整形。

四、 散列冲突解决方案

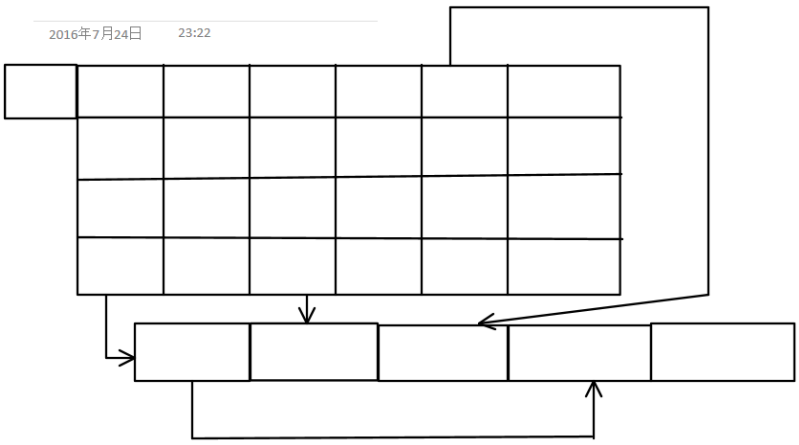
使用链表法解决键值冲突问题，示意图如图所示：



首先填充常规位置，即散列表的预留空间，如图上方表格的一栏。发生键值冲突时，将新的索引结构体添加到文件末尾，如图下方的新添加的栏。上一个冲突的键值的结构体的 next_key_off 赋值为新的结构体偏移量，形成关联，相当于形成逻辑上的链表。如再发生键值冲突，如前反复即可。如果插入时发现链表中某个结构体索引已被删除了，会自动填充进去。

五、 索引文件组织形式

索引文件用二进制读写方法，采用了 C 语言的 fopen_s、fwrite、fread、fflush、fseek、fclose 函数完成文件读写，索引文件组织形式如图所示：



索引文件第一个位置是一个 unsigned int 类型变量的值，记录有多少个索引结构体写在文件里，一般情况下这个值与写入文件的索引结构体个数相等，当删除索引时，这个值不会发生变化，这是为了插入新的键值到文件尾部，删除的空间会在合适的时机填充，这部分详见第七部分。每次打开书库会载入索引文件中的这个值，赋值给数据库对象的私有成员 last_idx_off 中，在打开数据库进行操作时，私有成员的值会不断变化。关闭数据库时，将更新好的值重新写入索引文件中。

索引文件后面均为索引结构体，每个索引结构体都是长度恒定的，每次都将一个完整的索引结构体用 fwrite 函数写入索引文件，读取时也按结构体读取。长度恒定的结构体方便直接用散列函数计算出偏移量。在初始化时索引字符串全部填为'\0'，其他属性为 0，然后写入文件。在散列表常规位置插入值时，需要更新这些变量。修改索引字符串的相关位置的值。初始化的作用是为了加快第一次的插入速度。初始化 2000 万索引结构体的时间相比大量插入的速度忽略不计。

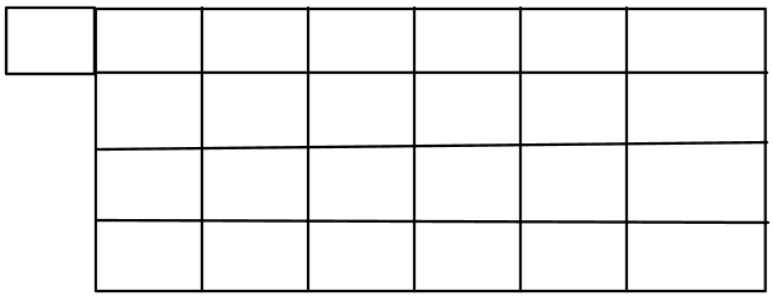
初始化的情况如图所示：

```
last_dat_off = 0;
last_idx_off = HASH_SIZE;
//
fwrite(&last_dat_off, sizeof(int), 1, fp2);
fwrite(&last_idx_off, sizeof(int), 1, fp1);
Idx index;
for (int i = 0; i < KEYSIZE_MAX - 1; i++)
    index.key[i] = '\0';
index.key[KEYSIZE_MAX - 1] = '\0';
index.value_off = 0;
index.off = 0;
index.off_next = 0;
index.len_key = 0;
index.len_value = 0;
index.isDelete = true;
for (int i = 0; i < HASH_SIZE; i++)
    fwrite(&index, sizeof(index), 1, fp1);
```

还有比较重要的一个变量，就是 isDelete，初始化为 true，表示该索引已删除。插入新值后修改为 false，表示索引存在。该变量在数据库的 insert 和 find 函数中有重要应用，用于在删除的位置填充新的索引结构体（可以理解为覆盖，因为索引结构体定长）。

六、 数据文件组织形式

数据文件结构相对简单，组织形式如图：



第一个位置是一个 unsigned int 类型变量的值，记录有多少个索引结构体写在文件里，一般情况下这个值与写入文件的索引结构体个数相等，当删除索引时，这个值不会发生变化，这是为了插入新的键值到文件尾部，每次打开书库会载入索引文件中的这个值，赋值给数据库对象的私有成员 last_dat_off 中，在打开数据库进行操作时，私有成员的值会不断变化。关闭数据库时，将更新好的值重新写入索引文件中。

之后所有位置都是数据字符串，与索引字符串不同的是，它不限定长度，这是用动态数组实现的，字符串尾部均有'\0'，因此 fwrite 写入时长度比字符串长度多一。

数据文件是没有删除操作的，也就是说数据文件会不断变大，不论进行 delete 操作还是 replace 操作，都只是在数据文件末尾写入数据字符串，利用 last_dat_off 可以为索引结构体的数据偏移量 value_off 赋值。不考虑数据文件的删除是因为删除的成本较高，虽然可以模仿索引文件的删除方法，但数据文件的磁盘大小相比索引文件小太多，因此不考虑数据文件的删除操作。

七、 一致刷新方法

索引文件和数据文件在每次增删查改操作后会执行 fflush 刷新文件内容，打开数据库的过程中，只有两个文件的头部 unsigned int 会保存在数据库的私有变量中，所以数据库对内存几乎没有需求，主要内存占用是性能测试时生成随机字符串的 map，与数据库无关。在早期测试中，曾尝试导入内存再批量插入操作，但利用内存与纯磁盘读写速度差异不大，主要性能问题在后面会介绍。

八、 数据库类

数据库类实现的操作如图：

```
class DB {
public:
    DB(string fileName);                    //创建两个文件且初始化数据库，若文件已存在则什么都不做
    unsigned int hash(const char* key);     //散列函数
    int open();                             //打开数据库的两个文件，并载入last_idx_off和last_value_off两个值
    int close();                            //保存关闭数据库的两个文件，并存入last_idx_off和last_value_off的值到文件
    char* find(const char* key);            //寻找key对应的value，返回value字符串指针,否则返回NULL
    bool del(char* key);                    //删除key，返回true，否则返回false
    int insert(char* key, char* value);     //插入key及value，返回1，若已存在该key，返回0
    bool replace(char* key, char* value);   //替换key对应的value，找不到key返回false，替换成功返回true
    void traversal();                       //打印数据库所有key及对应value
private:
    Idx* find_key(const char*key);          //寻找key对应索引结构体,返回结构体的指针，否则返回NULL
    unsigned int last_idx_off;              //索引文件总偏移量
    unsigned int last_dat_off;              //数据文件总偏移量
    string fileName;                        //数据库名
    string idxName, datName;                //两个文件名
    FILE* fp1;                              //索引文件流
    FILE* fp2;                              //数据文件流
    errno_t err1, err2;                     //fopen_s 的返回值，用于判断两文件是否存在
};
```

find_key 函数是四大函数的基本函数，它的作用是找到 key 对应的索引结构体并返回它的指针。利用散列值与偏移量的对应关系，很方便地找到散列表常规位置的索引结构体，接下来只需要遍历冲突链表即可（这里实际上是用 off_next 和 off 两个偏移量赋值实现），比较索引长度和索引的每一位，相等则找到，返回读出来的索引结构体的指针，找不到（判断条件是读到文件结尾，这是因为每次冲突时，新写入的索引结构体的 off_next 总设为 last_idx_off，因此读到文件尾就算读完这条链了）则返回 NULL。

Insert 函数是四大函数最复杂的，因为要判断多个条件，首先是调用 find_key 函数

查找该键值是否已存在，存在就直接返回 0，否则判断散列表常规位置是否初始化空闲，是否是删除空闲，是则在该处插入，否则遍历该冲突链表，找到删除空闲，或到达文件尾部，插入即可。

find 函数依托于 find_key 函数，find_key 函数返回对应结构体指针，只需用其 value_off 即可在数据文件中定位对应数据字符串，并利用索引结构体的 value_length 和动态数组返回数据字符串。

replace 函数同样依托于 find_key 函数，若找不到（返回 NULL）就返回 false，否则利用 find_key 返回的索引结构体指针，更新索引结构体的 value_off,value_length，将数据直接写在数据文件末尾，不覆盖原数据。

Traversal 函数将数据库中所有键值对打印出来，这只需不停读取索引结构体并利用 value_off 读取对应的数据字符串即可，非常简单。

类的构造函数用于文件不存在时初始化数据库，否则什么都不做。这个函数相对独立，会自己打开关闭文件。

Open 和 close 函数是载入与保存两个文件的头部 unsigned int ，并执行打开文件，关闭文件操作。其它函数，除构造函数均不会打开关闭文件，只会在每次调用函数后用 fflush 函数刷新修改的文件，这样是为了提升效率。

Clear 函数删除两个文件。

九、 测试与分析

测试分为性能测试与正确性测试，性能测试中已经涵盖了部分正确性测试的要求。首先是实现了随机字符串生成函数，可以生成制定最大长度的 key 和 value，将它们存入 STL 的 map 中用来测试，一共生成 100 万个键值对符合测试要求。遍历 map，调用 insert 函数即可插入数据库；再次遍历 map，调用 find 函数即可查找；遍历 map，调用随机字符串生成函数进行 replace 操作；遍历 map，调用 del 函数删除；最后调用 traversal 函数，遍历数据库发现已经全部删除。每种测试都执行 TEST_NUM 次，这个宏的值设为 100 万，可以修改。

正确性测试在我写的测试函数中与性能测试是一样的。

我保留了一个小测试，是最初写好用来测试数据库的，可以视为一个全自动的交互测试。

另外，控制台界面可以用户交互输入要求的命令并执行，可以视为正确性测试。

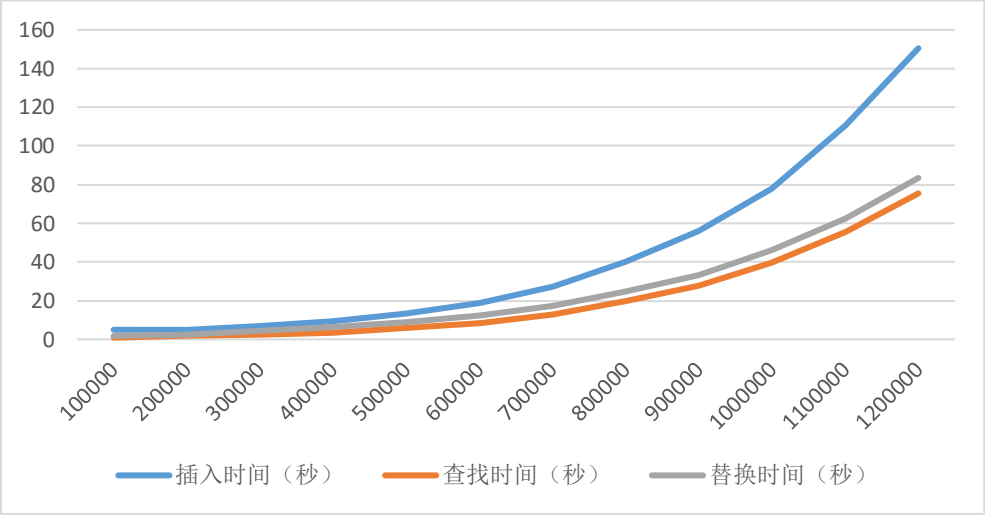
测试数据：

插入，查询，查找的情况如图所示（仅列出部分有代表性的数据，全部数据在文本文档中）

数据规模（条）	散列表大小	索引文件大小	插入时间（秒）	查询时间（秒）	替换时间（秒）
300000	20000003	20000763	7.026	2.373	4.262
200000	20000003	20000381	5.062	1.75	2.652
1000000	20000003	20007565	79.426	39.643	46.261
700000	20000003	20003754	27.402	12.897	17.304
100000	20000003	20000173	4.982	0.891	1.803
500000	20000003	20002156	13.501	5.909	8.982

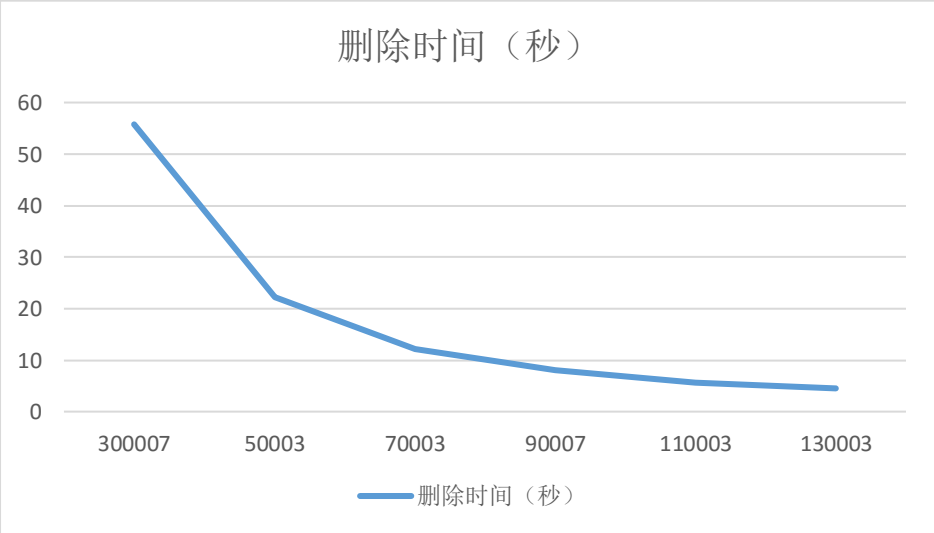
测试时，key 和 value 最大长度均设为 8，这是为了加大效率。

绘制图形后：



发现图线基本满足线性递增，是因为数据量较大时发生键值冲突，需要遍历冲突链表耗时增加。在 10 万到 50 万基本无键值冲突，图线是平的，因此可以认为，不发生键值冲突时基本满足插入查找替换时间复杂度 O（1）的要求。当然，影响最大的是键值冲突问题，在表格中，索引文件大小减去散列表大小就是发生键值冲突的索引结构体个数，100 万数据量时大小为 7000 多，耗费时间也增大很多。

解决键值冲突的方法是增大散列表，如插入 10 万条数据时，散列表大小与插入时间的关系图形为：

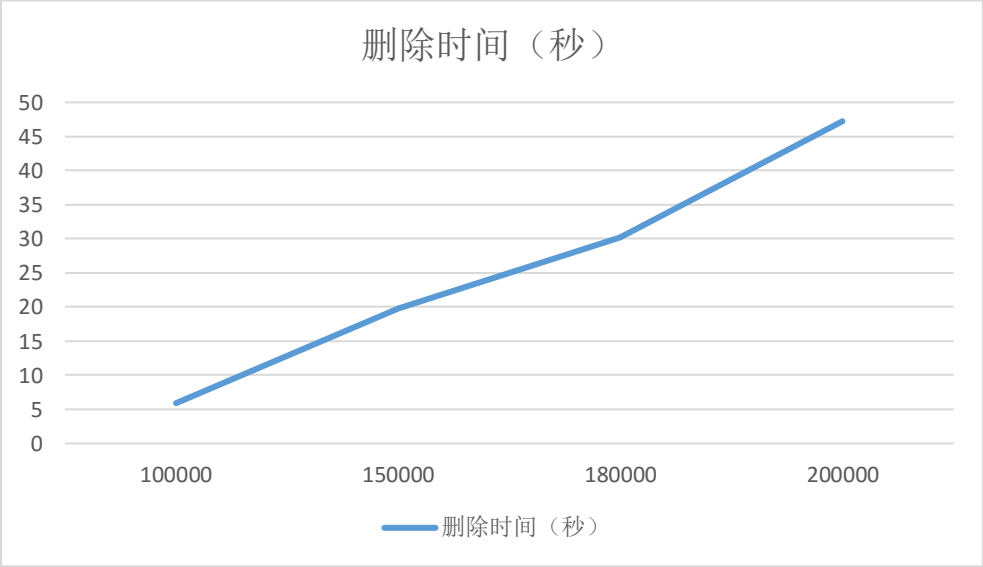


发现散列表较小时耗时很大，经过分析，这是因为键值冲突遍历的问题，散列表很大时，耗时几乎不会改变，这是因为对键值冲突的解决已到达极限。100 万数据时，选择的散列表大小为 20000003，这是多次尝试的结果。当然可以更大，例如还尝试过 50000017，这样索引文件会高达 2GB 多，而且比较浪费。

删除操作的情况为：

数据规模（条）	散列表大小	索引文件大小	删除时间（秒）
100000	20000003	20000096	5.903
150000	20000003	20000208	19.808
180000	20000003	20000271	30.209
200000	20000003	20000381	47.226

删除操作的图形为：



删除操作在进行性能测试时有很大的问题，耗时与其他函数相比明显不正常，且数据规模越大，耗时会剧烈递增。删除操作与替换操作并无二异，目前仍没有找到问题的原因。但是删除操作是正确的，可以在交互测试时证明正确性。

十、 总结评价

总体来说，用散列表实现的数据库比较容易构思，却不太好实现，主要是文件偏移量的操作，比较难把握。另外值得注意的是性能与散列表大小的问题，这实际上是时间与空间的矛盾，散列表大小最终确定为 20000003，这方便我对性能进行测试。当然可以更大，例如还尝试过 50000017，这样索引文件会高达 2GB 多，而且比较浪费。

仍然存在的几个问题是：

（1）没有根据装填因子动态改变散列表大小，如 100 万测试数据量发生键值冲突的在 8000 条左右，这部分是很影响性能的，因为需要遍历链表。若可以根据装填因子动态改变散列表大小，减少键值冲突，或者采用其他策略，如修正散列函数，也许可以解决，但碍于能力与时间没有实现。

（2）删除操作的问题，删除函数是正确的，没有写错，而且删除函数与替换函数没有太大差别，但两者耗费时间差异巨大，超过 20 万的数据量删除函数耗时很长，原因未知。

（3）磁盘读写速度问题，在多个电脑上测试，发现均有速度下降问题，一开始测试时读写很快，但会逐渐下降，几分钟后会达到一个稳定的最低点，大约为开始时速度的百分之一，这个问题暂时只能用增大散列表大小使开始读写速度提高，减慢下降速度。推测可能与读写冲突链表有关。

总体来说，这次实现的数据库算法上没有太大难度，操作细节却很多，比较难下手，是一次很好的锻炼。