Database 开发文档

学号：5140379023
姓名：柯学翰

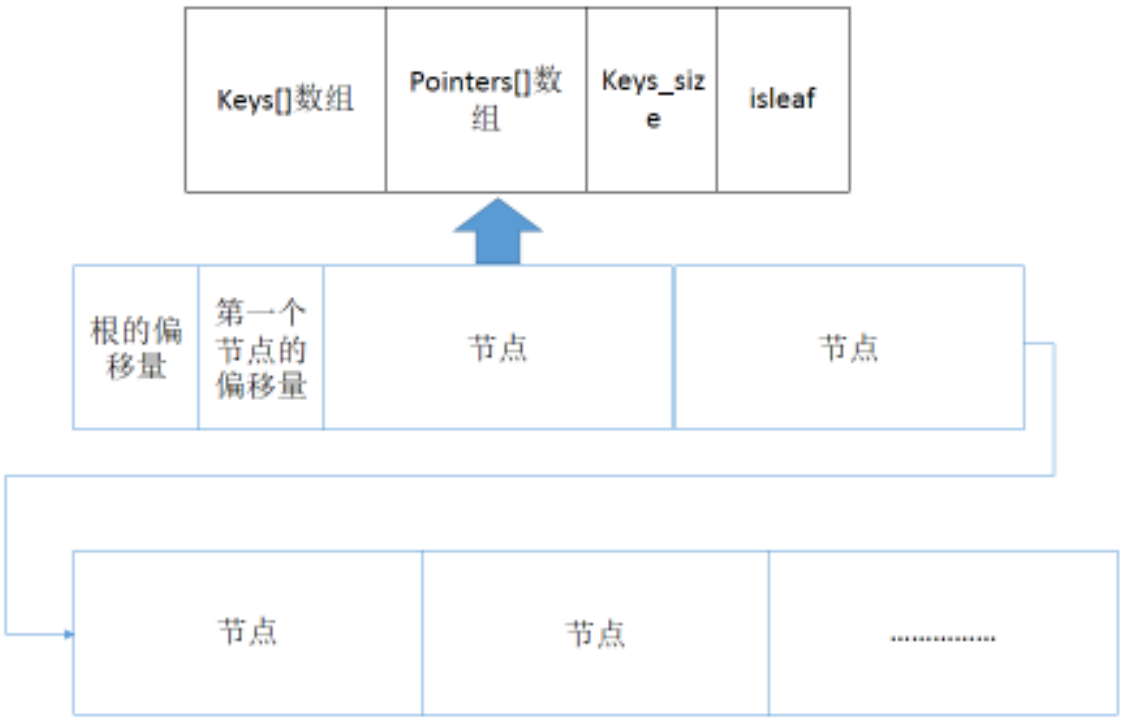
目录：

I. 概述

该数据库是用来储存学生的成绩的，关键字为 int 类型的学号，可以查到学生的名字和分数。数据库使用了 b+树作为基本的数据结构，并且用 cfile 库的 fread 和 fwrite 函数实现块状的读写。设有索引和数据文件，文件为二进制的文件，索引文件为存储在文件中的 b+树，根节点储存在内存中，以减少读写的次数。数据库实现了基本的增删查改的功能，还有附加了遍历的功能。数据的插入和删除算法为一次从上往下遍历树，不用回溯，以提高效率。然后对各种数据量下的性能进行测试，进行分析思考。

II. 基本数据结构

1 索引文件的基本结构



2 代码中定义

A. b+树节点

```
struct Node
{
    int keys[node_number + 1];
    int pointers[node_number + 2];
    int keys_size;
    bool isleaf;

};
```

B. 数据结构

```
struct Data
{
    string name;
    int score;
};
```

C. DB 类的定义

```
class DB{
public:

    DB(string filename, int flag);
```

```
private:

    string idx_name;                    //打开数据库的名字
    string dat_name;
    FILE* fst_idx;
    FILE* fst_dat;
    File_pointer idxroot_ptr;               //索引B+树的根的偏移量
    File_pointer idx_ptr;                     //索引的偏移量
    File_pointer dat_ptr;                     //文件的偏移量
    Node curr_node;
    Node root_node;                                 //根节点，根节点不用读入，保存在内存中
    bool is_replace;                  //用于是否替换
```

D. 一些说明

基本结构中，key 是 int 类型的学号，而 pointers[i]对应的是 keys[i]和 keys[i+1]之间的孩子。保存的也是孩子在索引文件中的偏移量。可以从根开始，一直寻找适当的路径，向下寻找。B+树所有的数据都储存在最底层，最底层的数据的 pointers[i]对应的是 keys[i]在数据文件中的偏移量。

III. 实现的方法

1. 基本操作

查找

```
bool DB::DB_fetch( int key)
```

插入

```
bool DB::DB_store(int key, string name, int score, int type)
{
    if (type == insert)
```

替代

```
 if (type == replace)
```

删除

```
void    DB::DB_delete(int key)
```

2. 扩展的操作

由 b+树的特点，还实现了 b+树的遍历，按顺序遍历所有的关键字。DB 中的 first_dat 记录了遍历的起点，每个叶子节点的最后一个 pointers[]记录的是下一个叶子节点的偏移量，因此可以一直遍历下去。

```
void DB::Visit()
{
    int adress = first_dat;
    while (true)
    {
        Node node;

        ReadNode(adress, node);
        for (int i = 0; i < node.keys_size; ++i)
        {
        }
        if (node.pointers[node_number + 1] <= 0)
            break;
        adress = node.pointers[node_number + 1];
    }
}
```

3. 亮点

A. 使用了 fwrite 和 fread 函数，实心了磁盘上的块状读写，大大简化代码的复杂程度。

```
void  DB::WriteNode(File_pointer address, Node &r)
{
    if (address == idxroot_ptr)
    {
        root_node = r;
        return;
    }
    fst_idx.seekp(address);
    fst_idx.write(( char*)(&r), sizeof(Node));


}
```

B. 插入和删除算法只要一次的自上往下的遍历。传统的插入算法是先找到叶子节点，插入后，若节点满了的话，再分裂，向上递归操作。本数据库的插入和删除算法是一次遍历，从上往下时，发现儿子节点要满时，则分裂。所以在叶子节点插入，一定不会是满的。删除算法同理。

C. 将根节点储存在内存中，根节点只有在创建 DB 类时进行一次读取，之后不用进行读取，减少了磁盘读取的次数，提高了效率。这是典

型的用空间换时间的，在空间和时间之间进行取舍。因为一个根节点不是很大，而之后的插入删除都能减少一次磁盘的读取（一般也就 2 到 4 次），这会大大的提高效率。若是将 b+树的前两层都读入内存中保存，内存的空间占用太大，所以最终决定只读入一层。

```
Node root_node;                    //根节点，根节点不用读入，保存在内存中
```

IV. 测试设计

正确性设计：

1) 向数据库写 n 条记录。

(2) 通过键值读回 n 条记录。

(3) 执行下面的循环 n×5 次：

(a) 随机读一条记录。

(b) 每循环 37 次，随机删除一条记录。

(c) 每循环 11 次，添加一条新记录并读回这条记录。

(d) 每循环 17 次，随机替换一条记录为新记录。在连续两次替换中，一次用同样大小的

记录替换，一次用比以前更长的记录替换。

(4) 将此子进程写的所有记录删除。每删除一条记录，随机地寻找 10 条记录。K

V. 性能测试

B+树的节点的元素的个数的不同会对 b+树的效率产生影响，下面以节点元素个数不同的情况下，插入相同个数的元素的时间相比较

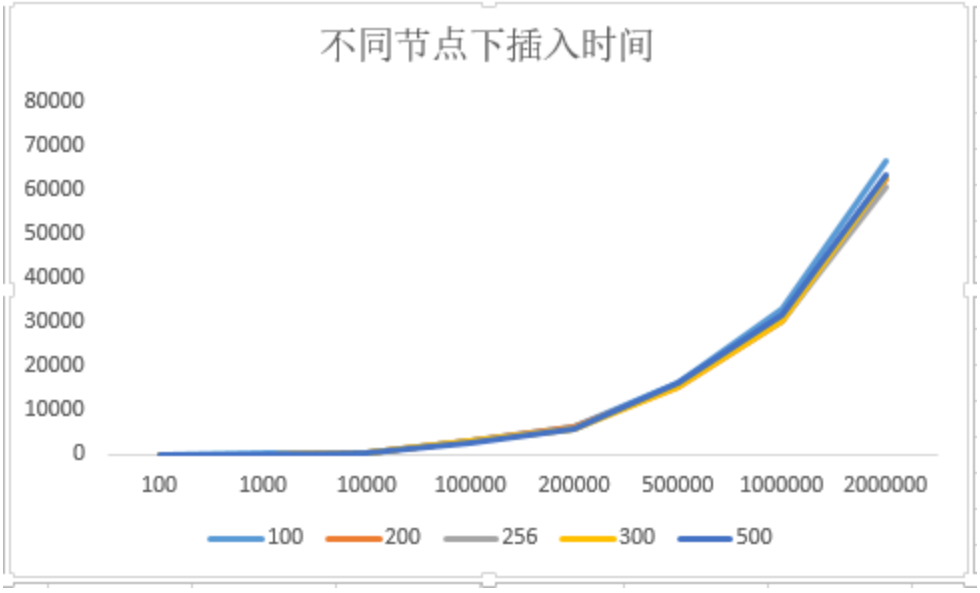
节点大小	100	200	256	300	500
数据量	插入时间/ms	插入时间/ms	插入时间/ms	插入时间/ms	插入时间/ms
100	1	0	1	0	1
1000	34	31	30	31	29
10000	312	281	304	312	307
100000	3062	2859	2829	2920	2718
200000	6062	6072	5887	5839	5886
500000	16032	15087	15616	15222	16321
1000000	32900	30316	30378	30414	31577
2000000	66800	62632	60884	62916	63641

在节点大小一定的情况下，不同数据量大小下，插入，修改，查找，删除 1000 次所需要的时间

节点大小=256				
数据量	插入10000次/ms	替换10000次/ms	查找10000次/ms	删除10000次/ms
100	243	219	78	94
1000	297	229	78	94
10000	281	235	88	109
100000	297	312	110	109
1000000	311	312	109	110
2000000	297	297	101	107
5000000	297	297	109	110

VI. 测试分析

节点大小不同情况下的效率比较图



可知节点大小为 200 到 300 之间时数据库的效率最高。当数据较小时，差别

很小，因为时间为都很快。但数据量比较大时，节点的大小过于小时，树的层数增加，操作的时间也会增加，而节点过于大时，读取节点的速度和在内存中操作的时间也增加，操作的效率也不高。所以经测试，节点大小在 200 到 300 之间，效率比较高，大于 300 后效率变化不大。于是取一个较好的数字 256 进行基本操作的性能测试。

当节点大小为 256 时，在不同数据量大小的进行 1000 次的操作，观察时间的不同。发现查找所需要的时间是最少的，插入和替换基本相同，因为这两个在代码中的执行步骤基本上也是相同。删除的时间和查找的时间几乎一样，因为删除也是一次的向下遍历的，所以用的时间和查找相近。而插入和替换虽然也是一次向下的遍历，但时间几乎也是查找的两到三倍，可能是因为算法在实现的过程中有有多余的读取，写的不够完善。

VII．思考

为什么插入等其它操作的时间远多于查找呢，达到了两倍到三倍左右？
在插入的过程中有部分情况是会读取一个节点两次的，特别是在顺序插入的过程中，还有代码有些地方可能比较费时间。不过本来插入，其他的操作也是很多的，时间本来就会比较长才对。也和同学的比较过，有的同学在没有用一次自上向下的读取以及没有将根节点储存在内存的情况下，插入的时间可能会达到查找 4 到 5 倍。这个插入的算法已经是一次读取的插入了，感觉可以改进的地方是在实现过程中，使计算机能更快的跑你的代码。