**数据库系统实现**

**——btree索引**

2018051409001 黄俊勋

**目录**

[第一章 项目介绍 1](#_Toc50313062)

[1.1 项目名称 1](#_Toc50313063)

[1.2 项目详细功能描述 1](#_Toc50313064)

[1.3 项目完成情况 1](#_Toc50313065)

[第二章 项目需求分析与设计 2](#_Toc50313066)

[2.1 聚集存储与btree索引 2](#_Toc50313067)

[2.2 项目需求分析 2](#_Toc50313068)

[2.3 b+tree结构分析与设计 3](#_Toc50313069)

[2.3.1 节点 3](#_Toc50313070)

[2.3.2 树 3](#_Toc50313071)

[2.3.3 约束条件 4](#_Toc50313072)

[2.4 BPlusTree类分析与设计 4](#_Toc50313073)

[2.4.1 查找 5](#_Toc50313074)

[2.4.2 插入 5](#_Toc50313075)

[2.4.3 删除 5](#_Toc50313076)

[第三章 项目需求设计与实现 6](#_Toc50313077)

[3.1 索引条目的查找 6](#_Toc50313078)

[3.2 记录的插入和索引条目的插入 7](#_Toc50313079)

[3.2.1 记录的插入 7](#_Toc50313080)

[3.2.2 索引条目的插入 9](#_Toc50313081)

[3.3 记录的删除和索引条目的删除 15](#_Toc50313082)

[3.3.1 记录的删除 15](#_Toc50313083)

[3.3.2 索引条目的删除 20](#_Toc50313084)

[第四章 项目测试 25](#_Toc50313085)

[4.1 插入功能测试 25](#_Toc50313086)

[4.2 删除功能测试 27](#_Toc50313087)

[第五章 项目总结与展望 29](#_Toc50313088)

[5.1 项目总结 29](#_Toc50313089)

[5.2 项目展望 29](#_Toc50313090)

第一章 项目介绍

1.1 项目名称

btree索引实现

1.2 项目详细功能描述

根据btree的知识设计并实现一个磁盘存储的索引：

1. 查找
2. 插入一个条目
3. 删除一个条目

1.3 项目完成情况

完成了索引条目的查找、插入、删除，实现b+tree的磁盘存储索引。

第二章 项目需求分析与设计

2.1 聚集存储与btree索引

数据库聚集存储实验，我们完成了对记录的插入、删除、修改、枚举。回顾记录的插入、删除过程，我们需要先定位block，然后再进行记录的相关操作。但是，在定位的时候，我们是以线性方式一个一个的比较定位，时间开销比较大。

因此，在btree索引实验中，我们引入了b+tree，在实验一原有的block链表上套上一层树的结构，来加速block的定位过程，提高系统的效率。



图1·聚集存储与btree索引

2.2 项目需求分析

根据上述聚集存储与btree索引关系，我们可以知道，btree索引实验建立在聚集存储实验之上，把聚集存储中block定位部分用b+tree来实现。在聚集存储实验中，我们用Table类来实现记录的操作，并提供记录的插入、删除、修改、枚举，在btree索引实验中，我们也可以用BPlusTree类来实现索引条目的操作，并提供索引条目的查找、插入、删除。

Table类与BPlusTree类的关系如下：Table类在进行记录的操作时，先调用BPlusTree类的查找方法来定位操作的block，然后，在完成了block中记录的操作之后，Table类调用BPlusTree中的插入/删除方法实现对b+tree的更新。

2.3 b+tree结构分析与设计

2.3.1 节点

b+tree的节点我们使用IndexBlock来表示，IndexBlock和聚集存储实验的block结构基本相同。

b+tree的节点有n个键和n+1个指针，为了能够把这些信息存储在一个节点blcok中，我们采用如下策略：

1. 每个键与其右指针绑定在一起，作为一条record插入到block中。
2. 剩余的一个指针（最左边的指针），放在block的nextid字段。

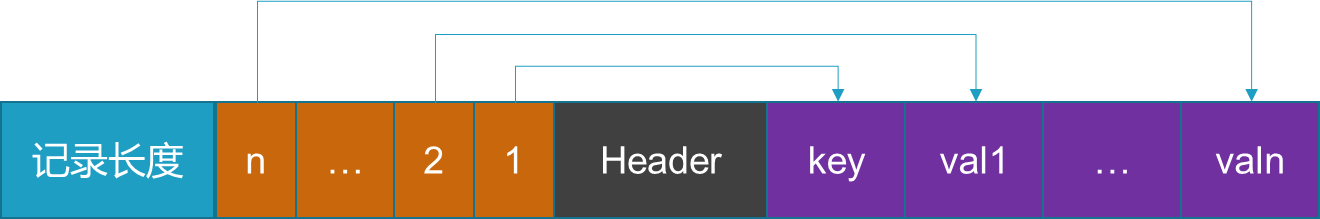


图2·record的结构



图3·键和值的分配

2.3.2 树

b+tree有内部节点、叶子节点，在聚集存储与btree索引的框架下，b+tree的内部节点是IndexBlock，存放的是索引条目，b+tree的叶子节点是DataBlock，存放的是数据记录，而且，索引和数据分两个文件存放。

因此，Table类中实现了b+tree的叶子节点，并串成了block链表，也符合b+tree的要求。BPlusTree类中，实现的是除了叶子节点之外的树结构，也就是说，BPlusTree类中的树只有两种节点：普通的内部节点和指向叶子节点的内部节点：

1. 普通的内部节点：普通的内部节点的指针的空间是针对index file中的IndexBlock的blockid。
2. 指向叶子节点的内部节点：指向叶子节点的内部节点虽然在index file中，但是他的指针空间是针对data file中的DataBlock的blockid，是连接index file和data file的中间桥梁。



图4·b+tree结构

2.3.3 约束条件

在btree索引实验中，b+tree的节点用block来表示，但是，record是变长的，所以，我们无法确定一个block能够存放多少个record，也就是无法知道一个节点有多少个值，这也导致了我们无法得到确切的n值。因而，无法用b+tree的约束条件来约束我们对b+tree的操作。

因此，我们需要对原有的b+tree的约束条件进行等价的近似修改，我们用占用空间和初始空闲空间的关系来替换n的约束条件。

1. 节点中记录的占用空间应该要超过50%。
2. 同层次节点管理的key范围不相交，并且同层次节点管理范围的“或”构成整个key空间。

2.4 BPlusTree类分析与设计

在BPlusTree类中，我们实现索引条目的查找、插入、删除功能。

2.4.1 查找

查找时，我们需要根据主键在b+tree上查找，而且用一个栈来保存查找路径：

int sraech(struct iovec &field, std::stack<int> &path);

2.4.2 插入

插入时，按照查找的路径递归地从下往上插入，插入的内容就是value值以及其绑定的右指针：

int insert(struct iovec &field, int rightid, std::stack<int> &path);

2.4.3 删除

删除时，按照查找的路径递归地从下往上删除：

int remove(struct iovec &field, std::stack<int> &path);

第三章 项目需求设计与实现

因为项目的核心部分为Table类和BPlusTree类的实现，因此，本部分将重点阐述Table类和BPlusTree类有关设计和实现。

3.1 索引条目的查找

我们在进行记录的插入、删除等操作时，需要先对操作的DataBlock进行定位。在btree的结构下，我们就是通过索引条目的查找来定位DataBlock。

再次提出【普通的内部节点】和【指向叶子节点的内部节点】的区别：

1. 普通的内部节点：普通的内部节点的指针的空间是针对index file中的IndexBlock的blockid。
2. 指向叶子节点的内部节点：指向叶子节点的内部节点虽然在index file中，但是他的指针空间是针对data file中的DataBlock的blockid，是连接index file和data file的中间桥梁。

在进行索引条目的查找时，我们在b+tree上从上往下进行查找，直到【指向叶子节点的内部节点】为止，从【指向叶子节点的内部节点】得到所定位的DataBlock的blockid。

int BPlusTree::sraech(**struct** **iovec** &field, std::stack<int> &path)

{

bool ret = initial();

**if** (ret) **return** ret;

IndexBlock index;

path.push(root\_);

index.attach(buffer\_);

int pointer = index.getNextid(); *//返回的指针，也就是定位的DataBlock的blockid*

*// 索引条目：key---right pointer*

**struct** **iovec** rec[2];

unsigned char header;

*//从上往下进行查找*

**while** (1) {

int rows = index.getSlotsNum();

*//枚举key字段,寻找第一个键值大于等于key的位置*

**for** (int i = 0; i < rows; i++) {

Record record;

unsigned short reoff = index.getSlot(i);

record.attach(buffer\_ + reoff, Block::BLOCK\_SIZE);

record.ref(rec, 2, &header);

*//找到第一个键值大于等于key的位置*

**if** (!relationInfo->fields[relationInfo->key].type->compare(

rec[0].iov\_base,

field.iov\_base,

rec[0].iov\_len,

field.iov\_len)) {

**break**;

}

*//更新pointer*

pointer = \*((int \*) rec[1].iov\_base);

}

*//如果查询进行到了指向叶子节点的内部节点，则退出*

**if** (index.getNodeType() == NODE\_TYPE\_POINT\_TO\_LEAF) **break**;

*//把blockid加入栈，保存查询路径*

path.push(pointer);

*//读下一个indexblock*

size\_t offset = (pointer - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->indexFile.read(

offset, (char \*) buffer\_, Block::BLOCK\_SIZE);

}

*//返回所得到的DataBlock的blockid*

**return** pointer;

}

3.2 记录的插入和索引条目的插入

3.2.1 记录的插入

和聚集存储实验一样，记录的插入需要先定位然后再插入，只是这里定位通过b+tree的索引条目的查找方法来快速定位。和聚集存储实验一样，如果插入失败，则分裂后再插入。

int Table::insert(**const** unsigned char \*header, **struct** **iovec** \*record, int iovcnt)

{

*//打开block*

bool ret = initial();

**if** (ret) **return** ret;

unsigned int key = relationInfo->key;

iovec &keyField = record[key];

DataBlock data;

*// TODO:检查是否重复插入*

*//路径*

std::stack<int> path;

*//定位，插入位置的blockid*

int insertid = index\_.sraech(record[key], path);

readDataBlock(insertid);

data.attach(buffer\_);

*//插入*

ret = data.allocate(header, record, iovcnt);

*//插入失败则分裂*

**if** (!ret) {

**struct** **iovec** field;

int newid;

splitDataBlock(data.blockid(), newid, &field);*//分裂*

*//判断插入的block的位置*

**if** (relationInfo->fields[key].type->compare(

keyField.iov\_base,

field.iov\_base,

keyField.iov\_len,

field.iov\_len))

insertid = insertid;

**else**

insertid = newid;

readDataBlock(insertid);

data.attach(buffer\_);

ret = data.allocate(header, record, iovcnt);

**if** (!ret) **return** S\_FALSE;

*//更新b+tree*

ret = index\_.insert(field, newid, path);

**if** (ret) **return** ret;

}

*// TODO:更新schema*

*// 排序*

std::vector<unsigned short> slotsv;

**for** (int i = 0; i < data.getSlotsNum(); i++)

slotsv.push\_back(data.getSlot(i));

**if** (relationInfo->fields[key].type == NULL)

relationInfo->fields[key].type =

findDataType(relationInfo->fields[key].fieldType.c\_str());

Compare cmp(relationInfo->fields[key], key, \*this);

std::sort(slotsv.begin(), slotsv.end(), cmp);

**for** (int i = 0; i < data.getSlotsNum(); i++)

data.setSlot(i, slotsv[i]);

*// 处理checksum*

data.setChecksum();

*//写block*

ret = writeDataBlock(insertid);

**if** (ret) **return** ret;

**return** S\_OK;

}

3.2.2 索引条目的插入

因为DataBlock和IndexBlock共同构成了一棵b+tree，DataBlock是叶子节点，IndexBlock是其他节点，所以，如果记录的插入引起了DataBlock的分裂，则必须根据之前查找的路径，从叶子节点到根节点往上更新，在b+tree中插入这个新分裂的DataBlock。

BPlusTree类中索引条目的插入方法就是来实现这样一个功能。在Table类的记录插入方法中，如果实现过程中出现了节点（DataBlock）的分裂，则调用BPlusTree类中索引条目的插入方法，把新分裂的节点（DataBlock）插入到b+tree中。

对于IndexBlock，插入的索引条目的内容就是一个包含一个键及其绑定的右指针的record：

1. 键：下一层分裂的blcok的最小主键值
2. 右指针：下一层分裂的blcok的blcokid

索引条目（record）插入的具体过程隐藏在Block类的插入记录的方法中。在BPlusTree类中，我们直接调用这个方法即可实现索引条目（record）的插入。插入后，我们对slots排序即可完成整个插入过程。

int BPlusTree::insert(**struct** **iovec** &field, int rightid, std::stack<int> &path)

{

int insertid = path.top();

path.pop();

**struct** **iovec** retField;

IndexBlock block;

readIndexBlock(insertid);

block.attach(buffer\_);

*// 插入record字段：key---right pointer*

unsigned char insertHeader = 0x00;

**struct** **iovec** insertRecord[2];

insertRecord[0].iov\_base = field.iov\_base;

insertRecord[0].iov\_len = field.iov\_len;

insertRecord[1].iov\_base = &rightid;

insertRecord[1].iov\_len = **sizeof**(int);

int ret = block.allocate(&insertHeader, insertRecord, 2);

*//插入成功*

**if** (ret) {

*// 排序*

std::vector<unsigned short> slotsv;

**for** (int i = 0; i < block.getSlotsNum(); i++)

slotsv.push\_back(block.getSlot(i));

treeCompare cmp(relationInfo->fields[relationInfo->key], 0, \*this);

std::sort(slotsv.begin(), slotsv.end(), cmp);

**for** (int i = 0; i < block.getSlotsNum(); i++)

block.setSlot(i, slotsv[i]);

*//写block*

ret = writeIndexBlock(insertid);

**if** (ret) **return** ret;

*//返回成功*

**return** S\_OK;

}

...... *// 其他代码*

}

以上的插入过程与Table类中的插入记录方法的实现过程基本相同，就是插入的记录不同而已。但是，BPlusTree索引条目的插入方法的难点在于分裂IndexBlock。如果IndexBlock插入索引条目（record）失败了，说明IndexBlock已经满了，需要分裂。假设IndexBlcok主键值最中间的两个键值为key1（其绑定的右指针为pointer1）和key2（其绑定的右指针为pointer2），插入条目的键的值为key，绑定的右指针为pointer。当IndexBlock分裂为IndexBlock和newIndexBlock时，IndexBlock劈一半，前一半留在原来的IndexBlock，后一半分配到newIndexBlock。

1. 如果key1<key<key2，则该IndexBlock分裂后，说明key作为newIndexBlock的最小键值，因为发生了分裂，所以需要递归地往上插入，插入的键就是key，该键绑定的右指针就是newIndexBlock的blockid。与此同时，key原来绑定的右指针pointer作为newIndexBlock的最左边指针（nextid字段）。
2. 如果key2<key，则该IndexBlock分裂后，说明key2作为newIndexBlock的最小键值，因为发生了分裂，所以需要递归地往上插入，往上插入的键就是key2，该键绑定的右指针就是newIndexBlock的blockid。与此同时，key2原来绑定的右指针pointer2作为newIndexBlock的最左边指针（nextid字段）。最后，分裂后，我们把要插入的key-pointer插入到newIndexBlock中。
3. 如果key<key1，则该IndexBlock分裂后，说明key1作为newIndexBlock的最小键值，因为发生了分裂，所以需要递归地往上插入，往上插入的键就是key1，该键绑定的右指针就是newIndexBlock的blockid。与此同时，key1原来绑定的右指针pointer1作为newIndexBlock的最左边指针（nextid字段）。最后，分裂后，我们把要插入的key-pointer插入到IndexBlock中。

实际的代码实现中，情况2和3可以合并成一种情况。

int BPlusTree::insert(**struct** **iovec** &field, int rightid, std::stack<int> &path)

{

...... *// 其他代码*

*// IndexBlock几个特殊record的key字段*

unsigned short slotsNum = block.getSlotsNum();

Record record;

**struct** **iovec** halfField, halfPlusField;

unsigned short recOffset = block.getSlot(slotsNum / 2 - 1);

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

record.specialRef(halfField, 0);

recOffset = block.getSlot(slotsNum / 2);

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

record.specialRef(halfPlusField, 0);

*// IndexBlock分裂成block1和block2*

IndexBlock block1, block2;

unsigned char db1[Block::BLOCK\_SIZE];

unsigned char db2[Block::BLOCK\_SIZE];

block1.attach(db1);

block1.clear(insertid);

block1.setNextid(block.getNextid()); *//设置block1最左边指针*

block1.setNodeType(block.getNodeType());

int newid = ++IndexBlockCnt;

block2.attach(db2);

block2.clear(newid);

block2.setNodeType(block.getNodeType());

*//情况1:field在中间位置*

**if** (relationInfo->fields[relationInfo->key].type->compare(

field.iov\_base,

halfPlusField.iov\_base,

field.iov\_len,

halfPlusField.iov\_len) &&

relationInfo->fields[relationInfo->key].type->compare(

halfField.iov\_base,

field.iov\_base,

halfField.iov\_len,

field.iov\_len)) {

*//分裂IndexBlock，得到block1*

**for** (unsigned short index = 0; index < slotsNum / 2; index++) {

unsigned short recOffset = block.getSlot(index);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

size\_t fields = record.fields();

**struct** **iovec** \*iov = (**struct** **iovec** \*) malloc(**sizeof**(iovec) \* fields);

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, (int) fields, &header);

*// 插入block1*

block1.allocate(&header, iov, (int) fields);

free(iov);

}

*//分裂IndexBlock，得到block2*

**for** (unsigned short index = slotsNum / 2; index < slotsNum; index++) {

unsigned short recOffset = block.getSlot(index);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

size\_t fields = record.fields();

**struct** **iovec** \*iov = (**struct** **iovec** \*) malloc(**sizeof**(iovec) \* fields);

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, (int) fields, &header);

*// 插入block2*

block2.allocate(&header, iov, (int) fields);

free(iov);

}

block2.setNextid(rightid); *//设置block2最左边指针*

*//设置返回字段*

retField.iov\_base = malloc(field.iov\_len);

::memcpy(retField.iov\_base, field.iov\_base, field.iov\_len);

retField.iov\_len = field.iov\_len;

}

*//情况2:field不在中间位置*

**else** {

int pos = 0;

**if** (relationInfo->fields[relationInfo->key].type->compare(

field.iov\_base,

halfField.iov\_base,

field.iov\_len,

halfField.iov\_len))

pos = slotsNum / 2 - 1;

**else**

pos = slotsNum / 2;

*//分裂IndexBlock，得到block1*

**for** (unsigned short index = 0; index < pos; index++) {

unsigned short recOffset = block.getSlot(index);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

size\_t fields = record.fields();

**struct** **iovec** \*iov = (**struct** **iovec** \*) malloc(**sizeof**(iovec) \* fields);

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, (int) fields, &header);

*// 插入block1*

block1.allocate(&header, iov, (int) fields);

free(iov);

}

*//分裂IndexBlock，得到block2*

**for** (unsigned short index = pos + 1; index < slotsNum; index++) {

unsigned short recOffset = block.getSlot(index);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

size\_t fields = record.fields();

**struct** **iovec** \*iov = (**struct** **iovec** \*) malloc(**sizeof**(iovec) \* fields);

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, (int) fields, &header);

*// 插入block2*

block2.allocate(&header, iov, (int) fields);

free(iov);

}

*// pos位置的record*

unsigned short recOffset = block.getSlot(pos);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

size\_t fields = record.fields();

**struct** **iovec** \*iov = (**struct** **iovec** \*) malloc(**sizeof**(iovec) \* fields);

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, (int) fields, &header);

*//设置block2最左边指针*

block2.setNextid(\*((int \*) iov[1].iov\_base));

*//设置返回字段*

retField.iov\_base = malloc(iov[0].iov\_len);

::memcpy(retField.iov\_base, iov[0].iov\_base, iov[0].iov\_len);

retField.iov\_len = iov[0].iov\_len;

*//插入record*

**if** (pos == slotsNum / 2 - 1) {

ret = block1.allocate(&insertHeader, insertRecord, 2);

**if** (!ret) **return** ret;

*// 排序*

std::vector<unsigned short> slotsv;

**for** (int i = 0; i < block1.getSlotsNum(); i++)

slotsv.push\_back(block1.getSlot(i));

treeCompare cmp(relationInfo->fields[relationInfo->key], 0, \*this);

std::sort(slotsv.begin(), slotsv.end(), cmp);

**for** (int i = 0; i < block1.getSlotsNum(); i++)

block1.setSlot(i, slotsv[i]);

} **else** {

ret = block2.allocate(&insertHeader, insertRecord, 2);

**if** (!ret) **return** ret;

*// 排序*

std::vector<unsigned short> slotsv;

**for** (int i = 0; i < block2.getSlotsNum(); i++)

slotsv.push\_back(block2.getSlot(i));

treeCompare cmp(relationInfo->fields[relationInfo->key], 0, \*this);

std::sort(slotsv.begin(), slotsv.end(), cmp);

**for** (int i = 0; i < block2.getSlotsNum(); i++)

block2.setSlot(i, slotsv[i]);

}

}

*//写block*

size\_t offset;

offset = (block1.blockid() - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->indexFile.write(

offset, (**const** char \*) db1, Block::BLOCK\_SIZE);

offset = (block2.blockid() - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->indexFile.write(

offset, (**const** char \*) db2, Block::BLOCK\_SIZE);

*// 直到根结点都满了，新生成根结点*

**if** (path.empty()) {

IndexBlock newroot;

newroot.attach(buffer\_);

newroot.clear(++IndexBlockCnt);

newroot.setNextid(insertid); *//设置newroot最左边指针*

newroot.setNodeType(NODE\_TYPE\_INTERNAL);

*//插入的记录*

insertRecord[0].iov\_base = retField.iov\_base;

insertRecord[0].iov\_len = retField.iov\_len;

insertRecord[1].iov\_base = &newid;

insertRecord[1].iov\_len = **sizeof**(int);

*//插入记录*

ret = newroot.allocate(&insertHeader, insertRecord, 2);

*//更新b+tree root*

root\_ = newroot.blockid();

*// 写newroot*

relationInfo->indexFile.write(

(root\_ - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE,

(**const** char \*) buffer\_,

Block::BLOCK\_SIZE);

*//更新文件root*

ret = writeRoot(root\_);

**if** (ret) **return** ret;

**return** S\_OK;

}

*//更新root*

ret = writeRoot(0);

**if** (ret) **return** ret;

*//递归插入*

ret = insert(retField, newid, path);

**if** (ret) **return** ret;

free(retField.iov\_base);

**return** S\_OK;

}

3.3 记录的删除和索引条目的删除

3.3.1 记录的删除

和聚集存储实验一样，记录的删除也需要先定位然后再删除，只是这里定位通过b+tree的索引条目的查找方法来快速定位。

int Table::remove(**struct** **iovec** keyField)

{

*//打开block*

bool ret = initial();

**if** (ret) **return** ret;

unsigned int key = relationInfo->key;

DataBlock data;

*//路径*

std::stack<int> path;

*//定位，目标位置的blockid*

int targetid = index\_.sraech(keyField, path);

readDataBlock(targetid);

data.attach(buffer\_);

*//删除*

int deleteIndex;

deleteIndex = data.recDelete(&keyField, relationInfo);

writeDataBlock(targetid);

*//其他代码*

}

与聚集存储实验不同的是，在这里，我们的DataBlock有约束条件，需要满足填充度大于50%，分为如下几种情况：

1. 删除后，节点填充度大于50%

判断一下删除的记录是否是原本DataBlock的第一个记录，如果是，则更新父节点（IndexBlock）中指向这个DataBlock的右指针对应的键值。

1. 删除后，节点填充度小于50%，但是兄弟节点填充度大于50%

从兄弟节点借一条记录来满足约束条件。

1. 删除后，节点填充度小于50%，而且兄弟节点填充度小于50%

合并当前节点和兄弟节点。

在实际的实现中，理论上50%在实际上存在问题，因为record是变长的，可能增加一个record就超过50%，不要这个record就不到50%。在这里，采用了一种待续优化完善的策略：一个block的填充度要大致处于1/3~2/3之间。

int Table::remove(**struct** **iovec** keyField)

{

*// 其他代码*

*// 删除后结点填充度仍>=50%*

**if** (data.getUsedspace() >= data.INITIAL\_FREE\_SPACE\_SIZE / 3) {

*//更新index*

**if** (deleteIndex ==

0) *//如果删除的记录是原本的第一条记录，那么blcok的最小键值发生了改变*

{

*//获取删除后blcok的最小键值*

**struct** **iovec** updateField;

unsigned short recOffset = data.getSlot(0);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

record.specialRef(updateField, key);

*//更新父节点（IndexBlock）中指向这个DataBlock的右指针对应的键值*

ret = index\_.updata(path.top(), keyField, updateField, targetid);

**if** (ret) **return** ret;

}

**return** S\_OK;

}

*//找到一个最近的兄弟节点*

int brotherid, isRight; *//兄弟节点的blockid，兄弟节点是否是右兄弟节点*

index\_.getBrother(path.top(), targetid, brotherid, isRight);

*//如果没有兄弟节点，则直接删除即可，无需其他合并、借操作*

**if** (brotherid == -1) {

*//更新index*

**if** (deleteIndex ==

0) *//如果删除的记录是原本的第一条记录，那么blcok的最小键值发生了改变*

{

*//获取删除后blcok的最小键值*

**struct** **iovec** updateField;

unsigned short recOffset = data.getSlot(0);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

record.specialRef(updateField, key);

*//更新父节点（IndexBlock）中指向这个DataBlock的右指针对应的键值*

ret = index\_.updata(path.top(), keyField, updateField, targetid);

**if** (ret) **return** ret;

}

**return** S\_OK;

}

*//读兄弟节点到db*

unsigned char db[Block::BLOCK\_SIZE];

size\_t offset = (brotherid - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->dataFile.read(offset, (char \*) db, Block::BLOCK\_SIZE);

DataBlock brother;

brother.attach(db);

*// 兄弟结点填充度>50%,从兄弟节点借*

**if** (brother.getUsedspace() > brother.INITIAL\_FREE\_SPACE\_SIZE \* 2 / 3) {

**if** (isRight) *//如果是右兄弟节点*

{

unsigned short recOffset = brother.getSlot(0);

Record record;

record.attach(db + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

size\_t fields = record.fields();

**struct** **iovec** \*iov = (**struct** **iovec** \*) malloc(**sizeof**(iovec) \* fields);

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, (int) fields, &header);

data.allocate(&header, iov, (int) fields);

*// 排序*

std::vector<unsigned short> slotsv;

**for** (int i = 0; i < data.getSlotsNum(); i++)

slotsv.push\_back(data.getSlot(i));

Compare cmp(relationInfo->fields[key], key, \*this);

std::sort(slotsv.begin(), slotsv.end(), cmp);

**for** (int i = 0; i < data.getSlotsNum(); i++)

data.setSlot(i, slotsv[i]);

*//删除兄弟节点所借的记录*

brother.recDelete(&iov[key], relationInfo);

*//写兄弟节点*

size\_t offset =

(brotherid - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->dataFile.write(

offset, (**const** char \*) db, Block::BLOCK\_SIZE);

*//更新兄弟的父节点*

**struct** **iovec** updateField;

recOffset = brother.getSlot(0);

record.attach(db + recOffset, Block::BLOCK\_SIZE);

record.specialRef(updateField, key);

ret = index\_.updata(path.top(), iov[key], updateField, targetid);

free(iov);

**if** (ret) **return** ret;

} **else** *//如果是左兄弟节点*

{

unsigned short recOffset =

brother.getSlot(brother.getSlotsNum() - 1);

Record record;

record.attach(db + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

size\_t fields = record.fields();

**struct** **iovec** \*iov = (**struct** **iovec** \*) malloc(**sizeof**(iovec) \* fields);

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, (int) fields, &header);

*//更新父节点*

**struct** **iovec** oldField;

Record rec;

recOffset = data.getSlot(0);

rec.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

rec.specialRef(oldField, key);

ret = index\_.updata(path.top(), oldField, iov[key], targetid);

**if** (ret) **return** ret;

*//借到的记录插入*

data.allocate(&header, iov, (int) fields);

*// 排序*

std::vector<unsigned short> slotsv;

**for** (int i = 0; i < data.getSlotsNum(); i++)

slotsv.push\_back(data.getSlot(i));

Compare cmp(relationInfo->fields[key], key, \*this);

std::sort(slotsv.begin(), slotsv.end(), cmp);

**for** (int i = 0; i < data.getSlotsNum(); i++)

data.setSlot(i, slotsv[i]);

*//删除兄弟节点所借的记录*

brother.recDelete(&iov[key], relationInfo);

*//写兄弟节点*

size\_t offset =

(brotherid - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->dataFile.write(

offset, (**const** char \*) db, Block::BLOCK\_SIZE);

free(iov);

}

*//写block*

ret = writeDataBlock(targetid);

**if** (ret) **return** ret;

**return** S\_OK;

}

*//兄弟结点填充度<=50%*

**struct** **iovec** field;

**if** (isRight) *//如果是右兄弟节点*

combineDataBlock(targetid, brotherid, &field, 1);*//兄弟节点合并到当前节点*

**else** *//如果是左兄弟节点*

combineDataBlock(brotherid, targetid, &field, 0);*//当前节点合并到兄弟节点*

*//更新b+tree*

ret = index\_.remove(field, path);

**if** (ret) **return** ret;

**return** S\_OK;

}

其中，合并节点时，需要把被合并的节点的所有记录读到内存中，然后插入到合并的节点，然后进行slots排序。此外，还需要调整blcok链表，通过调整合并的节点的nextid字段即可。合并节点方法返回的是被合并节点的第一个键值，以方便后续对b+tree的索引条目的删除。

int Table::combineDataBlock(

int blockid,

int comblockid,

**struct** **iovec** \*field,

int isRight)

{

unsigned int key = relationInfo->key;

*//原block*

DataBlock block;

readDataBlock(blockid);

block.attach(buffer\_);

*//合并的block*

DataBlock comBlock;

unsigned char db[Block::BLOCK\_SIZE];

comBlock.attach(db);

size\_t offset = (comblockid - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->dataFile.read(offset, (char \*) db, Block::BLOCK\_SIZE);

unsigned short slotsNum = comBlock.getSlotsNum();

**for** (unsigned short index = 0; index < slotsNum; index++) {

unsigned short recOffset = comBlock.getSlot(index);

Record record;

record.attach(db + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

size\_t fields = record.fields();

**struct** **iovec** \*iov = (**struct** **iovec** \*) malloc(**sizeof**(iovec) \* fields);

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, (int) fields, &header);

*//得到删除block的第一个字段*

**if** (index == 0) {

field->iov\_base = malloc(iov[key].iov\_len);

::memcpy(field->iov\_base, iov[key].iov\_base, iov[key].iov\_len);

field->iov\_len = iov[key].iov\_len;

}

int ret = block.allocate(&header, iov, (int) fields);

**if** (!ret) **return** S\_FALSE;

free(iov);

}

*// 排序*

std::vector<unsigned short> slotsv;

**for** (int i = 0; i < block.getSlotsNum(); i++)

slotsv.push\_back(block.getSlot(i));

**if** (relationInfo->fields[key].type == NULL)

relationInfo->fields[key].type =

findDataType(relationInfo->fields[key].fieldType.c\_str());

Compare cmp(relationInfo->fields[key], key, \*this);

std::sort(slotsv.begin(), slotsv.end(), cmp);

**for** (int i = 0; i < block.getSlotsNum(); i++)

block.setSlot(i, slotsv[i]);

*//调整nextid*

**if** (isRight) {

block.setNextid(comBlock.getNextid());

writeDataBlock(blockid);

} **else** {

writeDataBlock(blockid);

DataBlock prepreBlock;

**auto** bit = blockBegin();

**for** (; bit != blockEnd(); ++bit) {

prepreBlock = \*bit;

**if** (prepreBlock.getNextid() == comblockid) **break**;

}

**if** (bit == blockEnd()) **return** S\_FALSE;

prepreBlock.setNextid(blockid);

writeDataBlock(prepreBlock.blockid());

}

**return** S\_OK;

}

3.3.2 索引条目的删除

因为DataBlock和IndexBlock共同构成了一棵b+tree，DataBlock是叶子节点，IndexBlock是其他节点，所以，如果记录的删除引起了DataBlock的分并，也就是说有DataBlock从b+tree中删除，则必须根据之前查找的路径，从叶子节点到根节点往上更新，在b+tree中删除这个被合并（删除）的DataBlock。

BPlusTree类中索引条目的删除方法就是来实现这样一个功能。在Table类的记录删除方法中，如果实现过程中出现了节点（DataBlock）的合并，则调用BPlusTree类中索引条目的删除方法，把被合并的节点（DataBlock）从b+tree中删除。

对于IndexBlock，删除的索引条目的内容就是一个包含一个键及其绑定的右指针的record：

1. 键：下一层被合并（删除）的blcok的最小主键值
2. 右指针：下一层被合并（删除）的blcok的blcokid

索引条目（record）删除的具体过程隐藏在Block类的删除记录的方法中。在BPlusTree类中，我们直接调用这个方法即可实现索引条目（record）的删除。

int BPlusTree::remove(**struct** **iovec** &field, std::stack<int> &path)

{

unsigned int key = relationInfo->key;

*//删除的索引条目所在的block*

int deleteid = path.top();

path.pop();

IndexBlock block;

readIndexBlock(deleteid);

block.attach(buffer\_);

*//删除*

int deleteIndex;

deleteIndex = block.recDelete(&field, relationInfo);

**if** (deleteIndex == -1) **return** S\_FALSE;

writeIndexBlock(deleteid);

*//其他代码*

}

与前面阐述的记录的删除类似，我们的IndexBlock也有约束条件，需要满足填充度大于50%，分为如下几种情况：

1. 删除后，节点填充度大于50%

判断一下删除的记录是否是原本IndexBlock的第一个记录，如果是，则更新父节点（IndexBlock）中指向这个IndexBlock的右指针对应的键值。

1. 删除后，节点填充度小于50%，但是兄弟节点填充度大于50%

从兄弟节点借一条记录来满足约束条件。

1. 删除后，节点填充度小于50%，而且兄弟节点填充度小于50%

合并当前节点和兄弟节点。

在实际的实现中，理论上50%在实际上存在问题，因为record是变长的，可能增加一个record就超过50%，不要这个record就不到50%。在这里，采用了一种待续优化完善的策略：一个block的填充度要大致处于1/3~2/3之间。

int BPlusTree::remove(**struct** **iovec** &field, std::stack<int> &path)

{

*//其他代码*

**if** (path.empty()) *//到根节点*

{

**return** S\_OK;

}

*// 删除后结点填充度仍>=50%*

**if** (block.getUsedspace() >= block.INITIAL\_FREE\_SPACE\_SIZE / 4) {

*//更新index*

**if** (deleteIndex == 0) {

**struct** **iovec** updateField;

unsigned short recOffset = block.getSlot(0);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

record.specialRef(updateField, key);

int ret = updata(path.top(), field, updateField, deleteid);

**if** (ret) **return** ret;

}

**return** S\_OK;

}

*//找到一个最近的兄弟节点*

int brotherid, isRight;

getBrother(path.top(), deleteid, brotherid, isRight);

**if** (brotherid == deleteid) **return** S\_FALSE;

*//如果父节点只剩下一个儿子节点*

**if** (brotherid == -1) {

*//更新index*

**if** (deleteIndex == 0) {

**struct** **iovec** updateField;

unsigned short recOffset = block.getSlot(0);

Record record;

record.attach(buffer\_ + recOffset, Block::BLOCK\_SIZE);

record.specialRef(updateField, key);

int ret = updata(path.top(), field, updateField, deleteid);

**if** (ret) **return** ret;

}

**return** S\_OK;

}

*//读兄弟节点到db*

unsigned char db[Block::BLOCK\_SIZE];

size\_t offset = (brotherid - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->dataFile.read(offset, (char \*) db, Block::BLOCK\_SIZE);

IndexBlock brother;

brother.attach(buffer\_);

*// 兄弟结点填充度>50%,从兄弟节点借*

**if** (brother.getUsedspace() > brother.INITIAL\_FREE\_SPACE\_SIZE \* 2 / 3) {

*// TODO:从兄弟节点借*

**return** S\_FALSE;

}

*//兄弟结点填充度<=50%*

**struct** **iovec** delField;

**if** (isRight) {

int ret = combineIndexBlock(deleteid, brotherid, path.top(), &delField);

**if** (ret) **return** ret;

} **else** {

int ret = combineIndexBlock(brotherid, deleteid, path.top(), &delField);

**if** (ret) **return** ret;

}

*//递归删除*

int ret = remove(delField, path);

**if** (ret) **return** ret;

**return** S\_OK;

}

其中，合并节点时，与DataBlock合并节点的实现过程基本一致，但是有关细微的差别就在于Index的最左边指针。IndexBlock在合并的时候，被合并的block的最左边指针的键值必须从父亲节点得到，并且，必须把这个键-指针插入到合并的IndexBlock。

int BPlusTree::combineIndexBlock(

int blockid,

int comblockid,

int fatherid,

**struct** **iovec** \*field)

{

IndexBlock block;

readIndexBlock(blockid);

block.attach(buffer\_);

*//被合并的block*

IndexBlock comBlock;

unsigned char db[Block::BLOCK\_SIZE];

comBlock.attach(db);

size\_t offset = (comblockid - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->indexFile.read(offset, (char \*) db, Block::BLOCK\_SIZE);

*// comblock的key-pointer记录*

unsigned short slotsNum = comBlock.getSlotsNum();

**for** (unsigned short index = 0; index < slotsNum; index++) {

unsigned short recOffset = comBlock.getSlot(index);

Record record;

record.attach(db + recOffset, Block::BLOCK\_SIZE);

*// 先分配iovec*

**struct** **iovec** iov[2];

unsigned char header;

*// 从记录得到iovec*

record.ref(iov, 2, &header);

int ret = block.allocate(&header, iov, 2);

**if** (!ret) **return** S\_FALSE;

}

*// comblock的最左边指针*

int leftPointer = comBlock.getNextid();

*//要插入的key-pointer*

unsigned char insertHeader = 0x00;

**struct** **iovec** insertRecord[2];

insertRecord[0].iov\_base = NULL;

insertRecord[0].iov\_len = 0;

insertRecord[1].iov\_base = &leftPointer;

insertRecord[1].iov\_len = **sizeof**(int);

*// comblock的父亲节点*

IndexBlock faBlock;

faBlock.attach(db);

offset = (fatherid - 1) \* Block::BLOCK\_SIZE + Root::ROOT\_SIZE;

relationInfo->indexFile.read(offset, (char \*) db, Block::BLOCK\_SIZE);

*//从父节点得到comblock的最左边指针对应的键值*

slotsNum = faBlock.getSlotsNum();

**for** (unsigned short index = 0; index < slotsNum; index++) {

unsigned short recOffset = faBlock.getSlot(index);

Record record;

record.attach(db + recOffset, Block::BLOCK\_SIZE);

**struct** **iovec** iov[2];

unsigned char header;

record.ref(iov, 2, &header);

int bid = \*((int \*) iov[1].iov\_base);

**if** (bid == comblockid) {

insertRecord[0].iov\_base = iov[0].iov\_base;

insertRecord[0].iov\_len = iov[0].iov\_len;

**break**;

}

}

**if** (insertRecord[0].iov\_base == NULL) **return** S\_FALSE;

*//返回字段*

field->iov\_base = malloc(insertRecord[0].iov\_len);

::memcpy(

field->iov\_base, insertRecord[0].iov\_base, insertRecord[0].iov\_len);

field->iov\_len = insertRecord[0].iov\_len;

*//插入*

int ret = block.allocate(&insertHeader, insertRecord, 2);

**if** (!ret) **return** S\_FALSE;

*// 排序*

std::vector<unsigned short> slotsv;

**for** (int i = 0; i < block.getSlotsNum(); i++)

slotsv.push\_back(block.getSlot(i));

treeCompare cmp(relationInfo->fields[relationInfo->key], 0, \*this);

std::sort(slotsv.begin(), slotsv.end(), cmp);

**for** (int i = 0; i < block.getSlotsNum(); i++)

block.setSlot(i, slotsv[i]);

writeIndexBlock(blockid);

**return** S\_OK;

}

第四章 项目测试

4.1 插入功能测试

向关系表中逆序插入100000个record，主键是long long类型的id号，即从100000到1的顺序插入。插入后，用迭代器枚举record，判断记录是否是从1排到100000。

SECTION("insert")

{

std::ofstream outputfile;

outputfile.open("insert.txt");

Table table;

int ret = table.open("tablee");

REQUIRE(ret == S\_OK);

ret = table.initial();

REQUIRE(ret == S\_OK);

std::cout << "block:" << table.blockNum() << std::endl;

std::cout << "index:" << table.indexBlockNum() << std::endl;

std::cout << "freelength:" << table.freelength() << std::endl;

std::cout << "slotsNum:" << table.slotsNum() << std::endl;

**for** (int i = 100000; i > 0; i--) {

**struct** **iovec** iov[3];

long long id = i;

iov[0].iov\_base = &id;

iov[0].iov\_len = **sizeof**(long long);

char \*phone = "13534500702";

iov[1].iov\_base = (void \*) phone;

iov[1].iov\_len = strlen(phone) + 1;

char \*name =

"JunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixx"

"xxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJuni"

"JunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixx"

"JunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixx"

"xxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJuni"

"xxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJuni"

"xxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJunixxxxJun"

"i";

iov[2].iov\_base = (void \*) name;

iov[2].iov\_len = strlen(name) + 1;

unsigned char header = 0x84;

ret = table.insert(&header, iov, 3);

REQUIRE(ret == S\_OK);

std::cout << "insert:" << i << std::endl;

outputfile << "-----insert:" << i << "-----" << std::endl;

outputfile << "blockNum:" << table.blockNum() << std::endl;

outputfile << "indexNum:" << table.indexBlockNum() << std::endl;

outputfile << "blockid:" << table.blockid() << std::endl;

outputfile << "freelength:" << table.freelength() << std::endl;

outputfile << "slotsNum:" << table.slotsNum() << std::endl;

outputfile << "indexslotsNum:" << table.indexSlotsNum()

<< std::endl;

outputfile << std::endl;

}

long long cnt = 1;

**for** (**auto** it1 = table.blockBegin(); it1 != table.blockEnd(); ++it1) {

**for** (**auto** it2 = table.begin(it1); it2 != table.end(it1); ++it2) {

Record record = \*it2;

iovec keyField;

iovec Field;

record.specialRef(keyField, 0);

long long \*keyFieldPointer = (long long \*) keyField.iov\_base;

REQUIRE(\*keyFieldPointer == cnt++);

record.specialRef(Field, 1);

char \*FieldPointer = (char \*) Field.iov\_base;

char \*phone = "13534500702";

REQUIRE(

strncmp(FieldPointer, phone, strlen(FieldPointer)) == 0);

*// std::cout << "ACK:" << cnt << std::endl;*

}

std::cout << "blockACK:" << (\*it1).blockid() << std::endl;

}

table.close("tablee");

outputfile.close();

}

测试结果表明，100000个record插入了5555个DataBlock，13个IndexBlock。

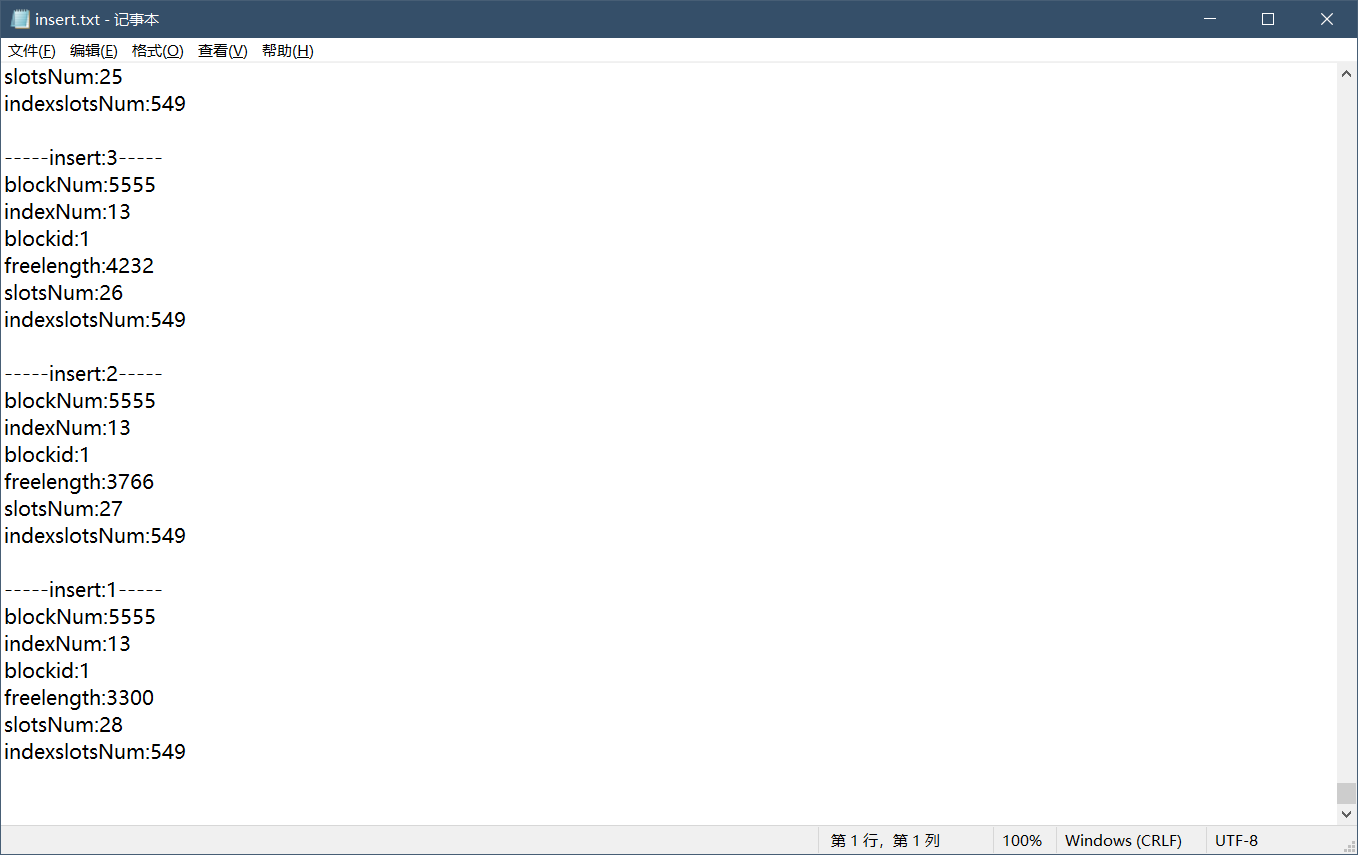


图7·输出的insert.txt文件

4.2 删除功能测试

在插入100000条record后，删除79999条record，删除的主键从1到7999。对于这种顺序删除，每删除一条record后，假定对应的主键为i，通过迭代器判断目前表中的第一个record是否是i+1。

SECTION("remove")

{

std::ofstream outputfile;

outputfile.open("remove.txt");

Table table;

int ret = table.open("tablee");

REQUIRE(ret == S\_OK);

ret = table.initial();

REQUIRE(ret == S\_OK);

**for** (long long i = 1; i < 80000; i++) {

iovec field;

long long id = i;

field.iov\_base = &id;

field.iov\_len = **sizeof**(long long);

ret = table.remove(field);

REQUIRE(ret == S\_OK);

**auto** bit = table.blockBegin();

DataBlock block = \*bit;

**while** (block.getSlotsNum() == 0) {

++bit;

block = \*bit;

}

**auto** it = table.begin(bit);

Record record = \*it;

iovec Field;

record.specialRef(Field, 0);

long long \*keyFieldPointer = (long long \*) Field.iov\_base;

REQUIRE(\*keyFieldPointer == i + 1);

record.specialRef(Field, 1);

char \*FieldPointer = (char \*) Field.iov\_base;

char \*phone = "13534500702";

REQUIRE(strncmp(FieldPointer, phone, strlen(FieldPointer)) == 0);

std::cout << "remove:" << i << std::endl;

outputfile << "-----remove:" << i << "-----" << std::endl;

std::cout << "remove:" << i << std::endl;

outputfile << std::endl;

}

outputfile.close();

table.close("tablee.dat");

}

测试结果表明，代码通过了所有的测试。

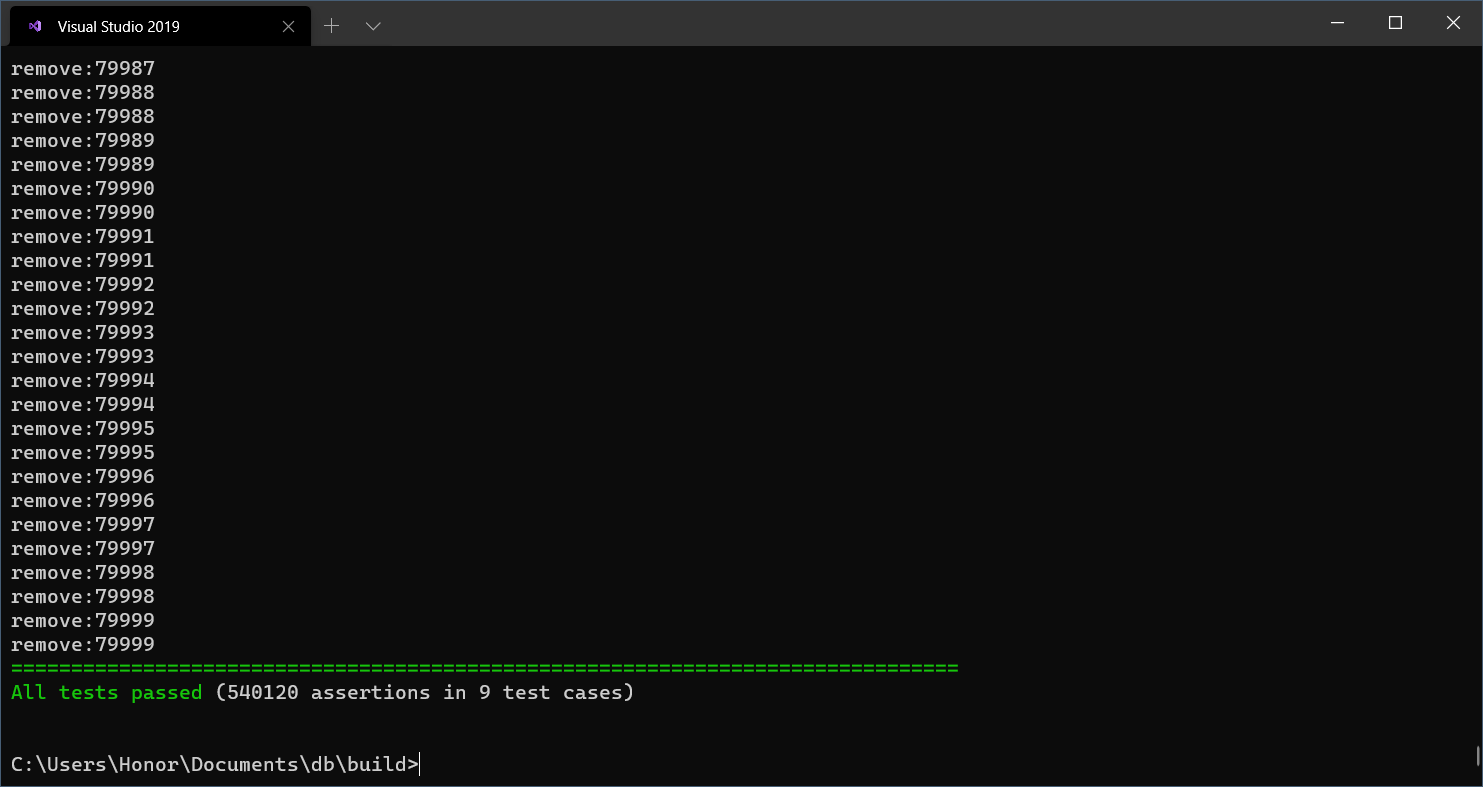


图8·代码通过所有测试

第五章 项目总结与展望

5.1 项目总结

本项目在聚集存储的基础上，根据b+tree的知识设计并实现了一个磁盘存储的索引，包含了索引条目的查找、插入、删除。

本项目还重新修改聚集存储的实现，将b+tree的结构嵌套在聚集存储的block链表结构之上，加速了记录的插入、删除、修改。

5.2 项目展望

本项目在处理b+tree节点的约束条件的时候仍然存在一定的问题，可以进行进一步的设计优化。

此外，本项目部分代码存在累赘的现象，可以进行进一步的重构优化。