**数据库系统实现**

**——聚集存储**

2018051409001 黄俊勋

**目录**

[第一章 项目介绍 1](#_Toc50313020)

[1.1 项目名称 1](#_Toc50313021)

[1.2 项目详细功能描述 1](#_Toc50313022)

[1.3 项目完成情况 1](#_Toc50313023)

[第二章 项目需求分析与设计 2](#_Toc50313024)

[2.1 项目需求分析 2](#_Toc50313025)

[2.2 record结构分析与设计 2](#_Toc50313026)

[2.2.1 record的结构 2](#_Toc50313027)

[2.2.2 记录在内存中的表示 2](#_Toc50313028)

[2.2.3 Record类 3](#_Toc50313029)

[2.3 block结构分析与设计 3](#_Toc50313030)

[2.3.1 block的结构 3](#_Toc50313031)

[2.3.1 Block类 4](#_Toc50313032)

[2.4 file结构分析与设计 4](#_Toc50313033)

[2.4.1 file的结构 5](#_Toc50313034)

[2.5 Schema结构分析与设计 5](#_Toc50313035)

[2.6 Table类分析与设计 5](#_Toc50313036)

[2.6.1 插入记录 5](#_Toc50313037)

[2.6.2 删除记录 6](#_Toc50313038)

[2.6.3 更新记录 6](#_Toc50313039)

[2.6.3 枚举记录 6](#_Toc50313040)

[第三章 项目需求设计与实现 7](#_Toc50313041)

[3.1 记录的枚举 7](#_Toc50313042)

[3.1.1 block的迭代器 7](#_Toc50313043)

[3.1.2 record的迭代器 8](#_Toc50313044)

[3.2 记录的插入 10](#_Toc50313045)

[3.2.1 block定位 10](#_Toc50313046)

[3.2.2 record插入 11](#_Toc50313047)

[3.2.3分裂block 13](#_Toc50313048)

[3.3 记录的删除 15](#_Toc50313049)

[3.3.1 block定位 15](#_Toc50313050)

[3.3.2 record删除 16](#_Toc50313051)

[3.3.3 block删除 18](#_Toc50313052)

[3.4 记录的修改 18](#_Toc50313053)

[第四章 项目测试 20](#_Toc50313054)

[4.1 插入、枚举功能测试 20](#_Toc50313055)

[4.2 删除功能测试 21](#_Toc50313056)

[4.2 修改功能测试 23](#_Toc50313057)

[第五章 项目总结与展望 25](#_Toc50313058)

[5.1 项目总结 25](#_Toc50313059)

[5.2 项目展望 25](#_Toc50313060)

第一章 项目介绍

1.1 项目名称

聚集存储实现

1.2 项目详细功能描述

根据磁盘管理的知识设计并实现一个磁盘存储系统：

1. 插入记录
2. 修改记录
3. 删除记录
4. 枚举记录

1.3 项目完成情况

完成了记录的插入、修改、删除、枚举，实现了数据的聚集存储。

第二章 项目需求分析与设计

2.1 项目需求分析

记录（record）存储在数据块（block）中，数据块（block）存储在文件（file）中，通过对记录的多个操作，实现一个聚集存储系统，因此，我们需要设计record、block和file的结构。

此外，对于一张关系表来说，在进行记录进行操作时，我们需要获知整个表的整体结构，才能进行操作，因此，我们还设计表的schema。

对记录的插入、删除、修改、枚举的操作，我们需要对外提供接口，因此，我们选择在一个Table类中实现。

2.2 record结构分析与设计

一条记录有多个变长或定长的字段，每个字段值描述该对象的一个属性或特征。此外，一条记录有一个主键，该主键的位置在表的schema中定义。

2.2.1 record的结构

模仿InnoDB的设计思路，一个自解释的record结构，分为4个部分：

1. 记录长度
2. 字段偏移量数组，逆序，从header开始到各字段头部的偏移量
3. Header，类型
4. 各字段顺序摆放

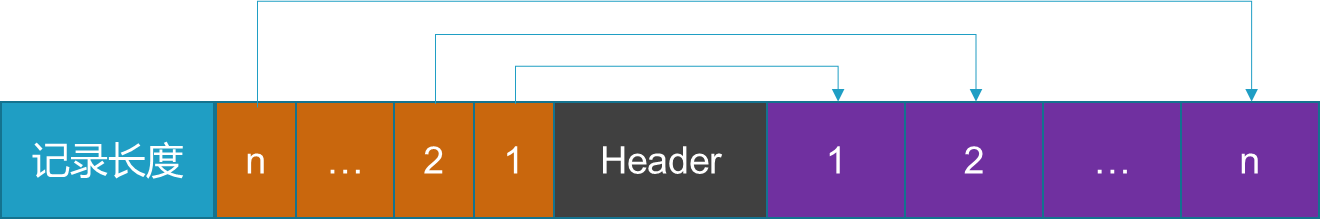


图1·record的结构

2.2.2 记录在内存中的表示

一条记录有多个字段，在内存中，将记录的字段存储在iovec数组中。

2.2.3 Record类

在Record类中根据record的结构定义相关字段，并提供如下方法：

1. 关联buffer
2. 向buffer里写各个域
3. 从buffer获取/引用各字段
4. 从buffer引用特定字段
5. 获得记录总长度
6. 获取记录字段个数

2.3 block结构分析与设计

Block主要存放着record，要求block内可以对record进行存储并排序。

2.3.1 block的结构

Block按照4B/8B对齐，主要有以下字段：

1. Common Header

Block的通用头部，包含表id、blockid、下一个block的blockid、block类型、垃圾链表等字段。

1. Data Header

数据block的头部，包含数据block的一些特有字段。

1. Data

Block存放数据的空间，即存放record，而且空间向下挤压。

1. Free Space

Block的空闲空间。

1. Slots

对record进行排序索引的数组，数组的每一个元素存储着对应record的偏移量，而且空间向上挤压。

1. Trailer

Block的校验字段。

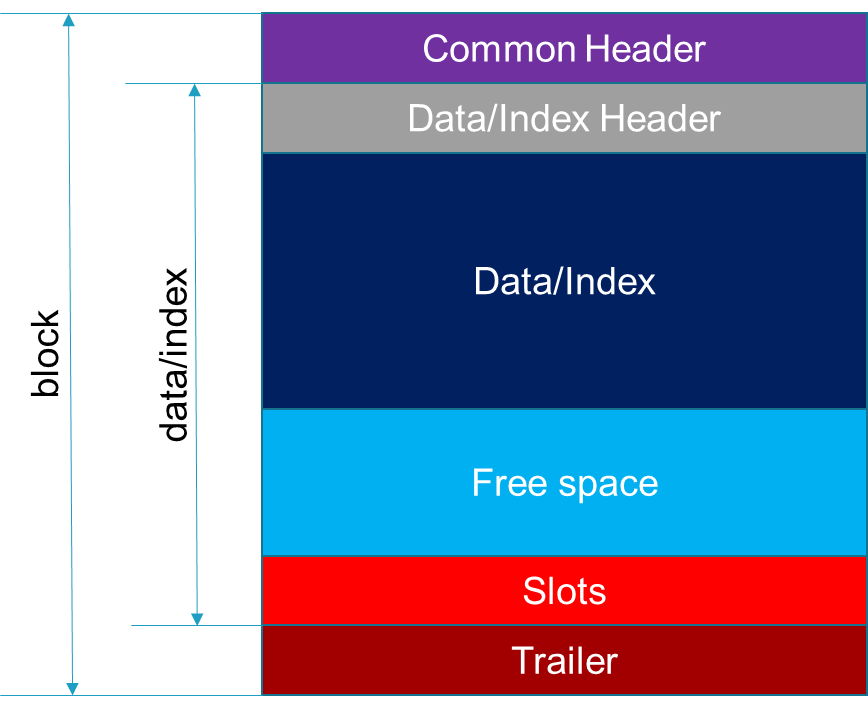


图2·block的结构

2.3.1 Block类

在Block类中根据block的结构定义相关字段，并提供如下方法：

1. 关联/消buffer
2. 获取/设置各个字段的方法
3. 插入/删除record的方法
4. ……

2.4 file结构分析与设计

一个文件中存储着多个block，根据聚集存储，需要把各个block串成链表，而且，各个block的主键范围必须不重复而且有序。

因为文件里有多个block，我们需要知道这个文件里有多少个block，有哪些block是无效的，哪些block的有用的，哪个block是block链表是第一个block，因此，我们引入了root数据块来存储这些信息。Root数据块是文件的第一个数据块，在读其他block之前，需要先读root获取相关信息。此外，对于block链表来说，root数据块相当于头指针。



图3·block链表的结构

2.4.1 file的结构

根据block链表的结构，我们可以如下设计file的结构：



图4·file的结构

2.5 Schema结构分析与设计

表的schema定义了表的基本结构以及其他信息：

1. 文件的路径
2. 域的个数
3. 表类型
4. 主键的位置
5. 行数
6. 各域的描述（域名、字段类型等）
7. ……

以上schema的信息也以数据块（metablock）进行存储，但是，与datablock不在同一个文件中，不再同一个表空间中。

2.6 Table类分析与设计

在Table类中，我们实现记录的插入、删除、修改、枚举功能。

2.6.1 插入记录

插入的记录的各个字段存储在iovec数组中，记录的header字段单独存储在一个字符串上，因此，插入记录的方法：

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

2.6.2 删除记录

删除记录时，我们只要知道主键，就可以定位到对应的记录，因此，删除记录的方法只需要一个主键的参数，用iovec结构体表示：

int remove(struct iovec \*keyField);

2.6.3 更新记录

更新记录时，相当于把原有记录删除，并把新的记录插入，也就是插入、删除记录的二合一：

int update(

struct iovec \*keyField,

const unsigned char \*header,

struct iovec \*record,

int iovcnt);

2.6.3 枚举记录

枚举记录时，需要实现一个迭代器。因为有File→blcok→record的层次结构，因此，我们需要两个迭代器，一个是blcok在file中的迭代器，一个是record在block中的迭代器。

//record的迭代器

struct iterator;

// block的迭代器

struct blockIter;

第三章 项目需求设计与实现

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

3.1 记录的枚举

因为block在file中以链表的形式串联在一起，所以，为了方便枚举block，需要一个block的迭代器。又因为record在block中，枚举record需要先枚举block，确定block之后，还需要在block中枚举record，因此，为了方便枚举，还需要一个record的迭代器。

3.1.1 block的迭代器

在文件中，每一个block都有自己唯一的标识即blockid，因此，在迭代器中，我们将以blockid来区分不同的block。

在重载++运算符时，因为每一个block都有nextid字段，表示下一个block的blockid，因此，可以获取nextid字段来得到下一个blcok的blcokid，进而实现++的运算。

在重载==和!=运算符时，只需要根据blockid是否相同来确定是否相等。

在实现解引用时，只需要把blockid对应的block（读到buffer）并返回即可。

**struct** **blockIter**

{

**private**:

unsigned int blockid; *// block位置*

Table &table;

DataBlock block;

**public**:

friend **struct** **iterator**;

**public**:

blockIter(unsigned int bid, Table &itable)

: blockid(bid)

, table(itable)

{}

blockIter(**const** blockIter &o)

: blockid(o.blockid)

, table(o.table)

{}

~blockIter() {}

unsigned int getBlockid() { **return** blockid; }

blockIter &operator=(**const** blockIter &o)

{

blockid = o.blockid;

table = o.table;

**return** \*this;

}

blockIter &operator++() *// 前缀*

{

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

table.relationInfo->file.read(

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

block.attach(table.buffer\_);

**if** (blockid != -1) blockid = block.getNextid();

**return** \*this;

}

blockIter operator++(int) *// 后缀*

{

blockIter tmp(\*this);

operator++();

**return** tmp;

}

bool operator==(**const** blockIter &rhs) **const**

{

**return** blockid == rhs.blockid;

}

bool operator!=(**const** blockIter &rhs) **const**

{

**return** blockid != rhs.blockid;

}

DataBlock &operator\*()

{

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

table.relationInfo->file.read(

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

block.attach(table.buffer\_);

**return** block;

}

};

3.1.2 record的迭代器

在block中，slots数组给每一个record进行排序，因此，在迭代器中，我们将以record在数组的位置（即下标sloti）来区分不同的record。

因为record是在block中，因此，在record中需要包含block的成员，在这里为blcok的迭代器（迭代器嵌套）。

在重载++运算符时，因为每一个record都有对应的数值下标sloti，sloti加1即可实现++运算。

在重载==和!=运算符时，只需要根据sloti是否相同来确定是否相等。

在实现解引用时，只需要把sloti对应的record返回即可。

**struct** **iterator**

{

**private**:

unsigned short sloti; *// slots[]索引*

unsigned short slotmax;

*// Table &table;*

blockIter &blockit;

Record record;

**public**:

iterator(unsigned short si, blockIter &iblockit)

: sloti{si}

, blockit(iblockit)

{

slotmax = (\*blockit).getSlotsNum() - 1;

}

iterator(**const** iterator &o)

: sloti(o.sloti)

, blockit(o.blockit)

, slotmax(o.slotmax)

{}

iterator &operator=(**const** iterator &o)

{

sloti = o.sloti;

blockit = o.blockit;

slotmax = o.slotmax;

**return** \*this;

}

iterator &operator++() *// 前缀*

{

**if** (sloti <= slotmax) sloti++;

**return** \*this;

}

iterator operator++(int) *// 后缀*

{

iterator tmp(\*this);

operator++();

**return** tmp;

}

bool operator==(**const** iterator &rhs) **const**

{

**return** sloti == rhs.sloti && blockit.blockid == rhs.blockit.blockid;

}

bool operator!=(**const** iterator &rhs) **const**

{

**return** sloti != rhs.sloti || blockit.blockid != rhs.blockit.blockid;

}

unsigned short getSlotid() { **return** sloti; }

Record &operator\*()

{

DataBlock block = \*blockit;

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

*// ++blockit;*

*// block = \*blockit;*

*// }*

unsigned short reoff = block.getSlot(sloti);

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

**return** record;

}

};

3.2 记录的插入

3.2.1 block定位

根据聚集存储的要求，在block链表中，block间的主键范围不重复而且有序，因此，在插入record的时候，我们需要先定位插入到哪一个block中。



图5·block的定位

对于block来说，主键范围就是block中记录的最大最小值。很显然，实际的主键没有+∞和-∞，但是，我们发现-∞的范围位于第一个block，+∞的范围位于最后一个block，因此，我们可以通过block的位置来进行这两种情况的特判。如图所示，如果要插入record的主键大于b而小于c，则插入的block应该是Block2；如果插入的record的主键大于e，且Block4是最后一个block，则插入的block应该就是Block4；如果插入的record的主键小于a，且Block1是第一个block，则插入的block应该就是Block1。

*//插入block的定位*

**for** (**auto** bit1 = blockBegin(), bit2 = ++blockBegin(); bit1 != blockEnd();

++bit1, ++bit2) {

*//最后一个block特殊情况处理，表示[a,+∞]的情况*

**if** (bit2 == blockEnd()) {

data = \*bit1;

**break**;

}

*//获取连续两个block的主键范围的左边界*

data = \*bit1;

**if** (data.getSlotsNum() == 0) **continue**;

iovec key1, key2;

Record rec = \*iterator(0, bit2);

rec.specialRef(

key2, key); *//获取key2，即第二个block的主键范围的左边界*

data = \*bit1;

rec =

\*iterator(0, bit1); *//获取key1，即第一个block的主键范围的左边界*

rec.specialRef(key1, key);

*//如果插入block的主键在key1和key2的范围内*

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

keyField.iov\_base,

key2.iov\_base,

keyField.iov\_len,

key2.iov\_len) &&

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

key1.iov\_base,

keyField.iov\_base,

key1.iov\_len,

keyField.iov\_len))) {

data = \*bit1;

**break**;

}

*//第一个block特殊情况处理,表示[-∞,a]的情况*

**else** **if** (

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

keyField.iov\_base,

key1.iov\_base,

keyField.iov\_len,

key1.iov\_len) &&

bit1 == blockBegin()) {

data = \*bit1;

**break**;

}

}

3.2.2 record插入

record插入的具体过程隐藏在Block类的插入记录的方法中，主要过程就是先判断空间，然后插入，再进行相关的调整。

bool DataBlock::allocate(

**const** unsigned char \*header,

**struct** **iovec** \*iov,

int iovcnt)

{

*// 判断是否有空间*

unsigned short length = getFreeLength();

**if** (length == 0) **return** false;

*// 判断能否分配*

std::pair<size\_t, size\_t> ret = Record::size(iov, iovcnt);

length -= 2; *// 一个slot占2字节*

**if** (ret.first > length) {

int usedspace = getUsedspace();

**if** (ret.first < INITIAL\_FREE\_SPACE\_SIZE - usedspace - 2) {

rewrite();

length = getFreeLength();

**if** (length < 2) **return** false;

length -= 2;

**if** (ret.first > length) **return** false;

} **else**

**return** false;

}

*// 写入记录*

Record record;

unsigned short oldf = getFreespace();

record.attach(buffer\_ + oldf, length);

unsigned short pos = (unsigned short) record.set(iov, iovcnt, header);

*// 调整usedspace*

int usedspace = getUsedspace();

usedspace += pos;

usedspace += 2;

setUsedspace(usedspace);

*// 调整freespace*

setFreespace(pos + oldf);

*// 写slot*

unsigned short slots = getSlotsNum();

setSlotsNum(slots + 1); *// 增加slots数目*

setSlot(slots, oldf); *// 第slots个*

*// slots未排序，同时需要setChecksum*

**return** true;

}

因此，在Table类中，我们直接调用这个方法即可实现record的插入。

*//插入*

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

**if** (!ret) {

splitDataBlock(data.blockid()); *//插入失败，分裂block*

ret = insert(header, record, iovcnt);

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

**return** S\_OK;

}

插入record后，根据blcok的设计，我们需要调整slots数组，使得record有序。在进行slots排序的时候，我们仅仅需要把整个slots数组读到一个vector容器中，然后，编写自己的排序函数（或类），进行排序，然后再写回去即可。

*// 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]);

排序类中，我们实现了对给定偏移量对应的record进行排序。

**struct** **Compare**

{

**private**:

FieldInfo &fin;

unsigned int key;

Table &table;

**public**:

Compare(FieldInfo &f, unsigned int k, Table &itable)

: fin(f)

, key(k)

, table(itable)

{}

bool operator()(**const** unsigned short &x, **const** unsigned short &y) **const**

{

*//根据x, y偏移量，引用两条记录；*

Record rx, ry;

rx.attach(table.buffer\_ + x, Block::BLOCK\_SIZE);

ry.attach(table.buffer\_ + y, Block::BLOCK\_SIZE);

iovec keyx, keyy;

rx.specialRef(keyx, key);

ry.specialRef(keyy, key);

**return** fin.type->compare(

keyx.iov\_base, keyy.iov\_base, keyx.iov\_len, keyy.iov\_len);

}

};

完成对slots的排序后，我们就完成了对record的插入。

3.2.3分裂block

因为block的大小有限，当所定位的blcok插满了后，我们不得不对这个block进行分裂，来满足我们插入record的需求。

分裂block的时候，我们把一半的数据（record）放在新的blcok，把剩下的一半留在原本的block。

与此同时，我们需要调整blcok链表，通过调整原block和新block的nextid字段，把新block串联在block链表上。

int Table::splitDataBlock(int blockid)

{

*//原block*

int nextid;

DataBlock block;

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

relationInfo->file.read(offset, (char \*) buffer\_, Block::BLOCK\_SIZE);

block.attach(buffer\_);

nextid = block.getNextid();

*//分裂block，block1为原来的block，block2为新的block*

DataBlock block1, block2;

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

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

block1.attach(db1);

block1.clear(block.blockid());

block1.setNextid(++DataBlockCnt); *//调整nextid字段*

block2.attach(db2);

block2.clear(DataBlockCnt);

block2.setNextid(nextid); *//调整nextid字段*

*//数据对半劈开，分别放到block1和block2*

unsigned short slotsNum = block.getSlotsNum();

**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);

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

free(iov);

}

**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);

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

free(iov);

}

*//写block*

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

relationInfo->file.write(offset, (**const** char \*) db1, Block::BLOCK\_SIZE);

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

relationInfo->file.write(offset, (**const** char \*) db2, Block::BLOCK\_SIZE);

*//更新root*

int ret = writeRoot();

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

**return** S\_OK;

}

3.3 记录的删除

3.3.1 block定位

因为record存储在block中，block串成了链表，因此，在删除record的时候，还需要先对要删除record所在的block进行定位。



图6·block的定位

对于block来说，主键范围就是block中记录的最大最小值。因此，当我们要定位所要删除记录所在的block时，我们只需要逐个判断删除的record的主键是否大于当前blcok中记录的最小值、小于当前blcok中记录的最大值。

*// block定位*

**auto** bit = blockBegin();

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

data = \*bit;

**if** (data.getSlotsNum() == 0) **continue**;

iovec keyFront, keyBack;

Record recFront = front(bit);

recFront.specialRef(keyFront, key); *//当前block的最小值*

Record reckBack = back(bit);

reckBack.specialRef(keyBack, key); *//当前block的最大值*

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

keyField->iov\_base,

keyFront.iov\_base,

keyField->iov\_len,

keyFront.iov\_len) &&

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

keyBack.iov\_base,

keyField->iov\_base,

keyBack.iov\_len,

keyField->iov\_len)) {

blockid = data.blockid(); *//确定删除记录所在的block*

**break**;

}

}

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

3.3.2 record删除

record删除的具体过程隐藏在Block类的删除记录的方法中：

int DataBlock::recDelete(**struct** **iovec** \*keyField, RelationInfo \*relationInfo)

{

unsigned int key = relationInfo->key;

unsigned short slotsNum = getSlotsNum();

int deleteindex = -1;

std::vector<unsigned short> slotsv;

*//逐个record进行判断*

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

unsigned short recOffset = getSlot(index);

Record record;

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

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

record.specialRef(field, key);

*//如果当前record的主键就是要删除record的主键*

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

field.iov\_base,

keyField->iov\_base,

field.iov\_len,

keyField->iov\_len)) &&

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

keyField->iov\_base,

field.iov\_base,

keyField->iov\_len,

field.iov\_len))) {

deleteindex = index; *//确定了要删除record在block里的位置*

*// 调整usedspace*

int usedspace = getUsedspace();

int recSize = ((int) record.length() + Record::ALIGN\_SIZE - 1) /

Record::ALIGN\_SIZE \* Record::ALIGN\_SIZE;

usedspace -= recSize;

usedspace -= 2;

setUsedspace(usedspace);

**continue**; *//不把删除的record的slots添加到vector容器中*

}

slotsv.push\_back(recOffset); *//把slots加到vector容器中*

}

**if** (deleteindex != -1) {

*// 调整slots*

setSlotsNum(--slotsNum);

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

setSlot(index, slotsv[index]); *//写slots*

}

**return** deleteindex; *//返回删除record在block中的位置，-1代表失败*

}

在这里，我们对删除的record采用标记删除的策略，把要删除的record在slots数组中删除，但是，record本身并不删除。

但是，这样子会出现大量无效的record填充在block中的情况，因此，在这里，采用usedspace字段来记录blcok中有效占用空间的大小，INITIAL\_FREE\_SPACE\_SIZE定义了初始空闲空间大小。在插入记录时，如果出现了block剩余空间不足以插入新的record，但是，通过INITIAL\_FREE\_SPACE\_SIZE和usedspace计算得知实际上有空间插入新的record，只是空间被无效record占用的时候，我们对当前block进行一次刷新重写，挤压掉无效的record。

bool DataBlock::allocate(

**const** unsigned char \*header,

**struct** **iovec** \*iov,

int iovcnt)

{

*// 判断是否有空间*

unsigned short length = getFreeLength();

**if** (length == 0) **return** false;

*// 判断能否分配*

std::pair<size\_t, size\_t> ret = Record::size(iov, iovcnt);

length -= 2; *// 一个slot占2字节*

**if** (ret.first > length) {

int usedspace = getUsedspace();

**if** (ret.first < INITIAL\_FREE\_SPACE\_SIZE - usedspace - 2) {

rewrite();

length = getFreeLength();

**if** (length < 2) **return** false;

length -= 2;

**if** (ret.first > length) **return** false;

} **else**

**return** false;

}

……

}

因此，在Table类中，我们直接调用这个方法即可实现record的删除。

*//删除record*

readBlock(blockid);

ret = data.recDelete(keyField, relationInfo);

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

3.3.3 block删除

如果一个blcok里的所有record都删除了，那么这个block应该也要删除掉。在删除blcok时，关键是对block链表进行调整。

通过调整要删除block的前一个blcok的nextid字段，把要删除的block从block链表上删除。

*//如果blcok的record全部都被删除，则删除block*

**if** (data.getSlotsNum() == 0) {

int nextid = data.getNextid();

**if** (blockid = head\_) *//如果要删除的block是第一个blcok*

{

head\_ = nextid; *//调整block链表的链头*

writeRoot(head\_);

} **else** {

**auto** it = blockBegin();

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

DataBlock block = \*it;

*//得到要删除block的前一个block*

**if** (block.getNextid() == blockid)

{

*//调整nextid字段*

block.setNextid(nextid);

writeBlock();

**break**;

}

}

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

}

}

3.4 记录的修改

记录的修改相当于把要修改的记录删除，然后插入修改后的记录，因此，就是记录的删除加上记录的插入：

int Table::update(

**struct** **iovec** \*keyField,

**const** unsigned char \*header,

**struct** **iovec** \*record,

int iovcnt)

{

int ret;

ret = remove(keyField);

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

ret = insert(header, record, iovcnt);

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

**return** S\_OK;

}

第四章 项目测试

4.1 插入、枚举功能测试

向关系表中逆序插入10000个record，主键是long long类型的id号，即从1000到1的顺序插入。插入后，枚举record，判断记录是否是从1排到10000。

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

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

*//插入*

**for** (int i = 10000; 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"

"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);

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

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

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

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

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

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

outputfile << std::endl;

}

*//枚举*

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;

*//判断id字段是否符合预期的结果*

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);

}

}

table.close("tablee.dat");

outputfile.close();

}

测试结果表明，10000个record插入了344个block。

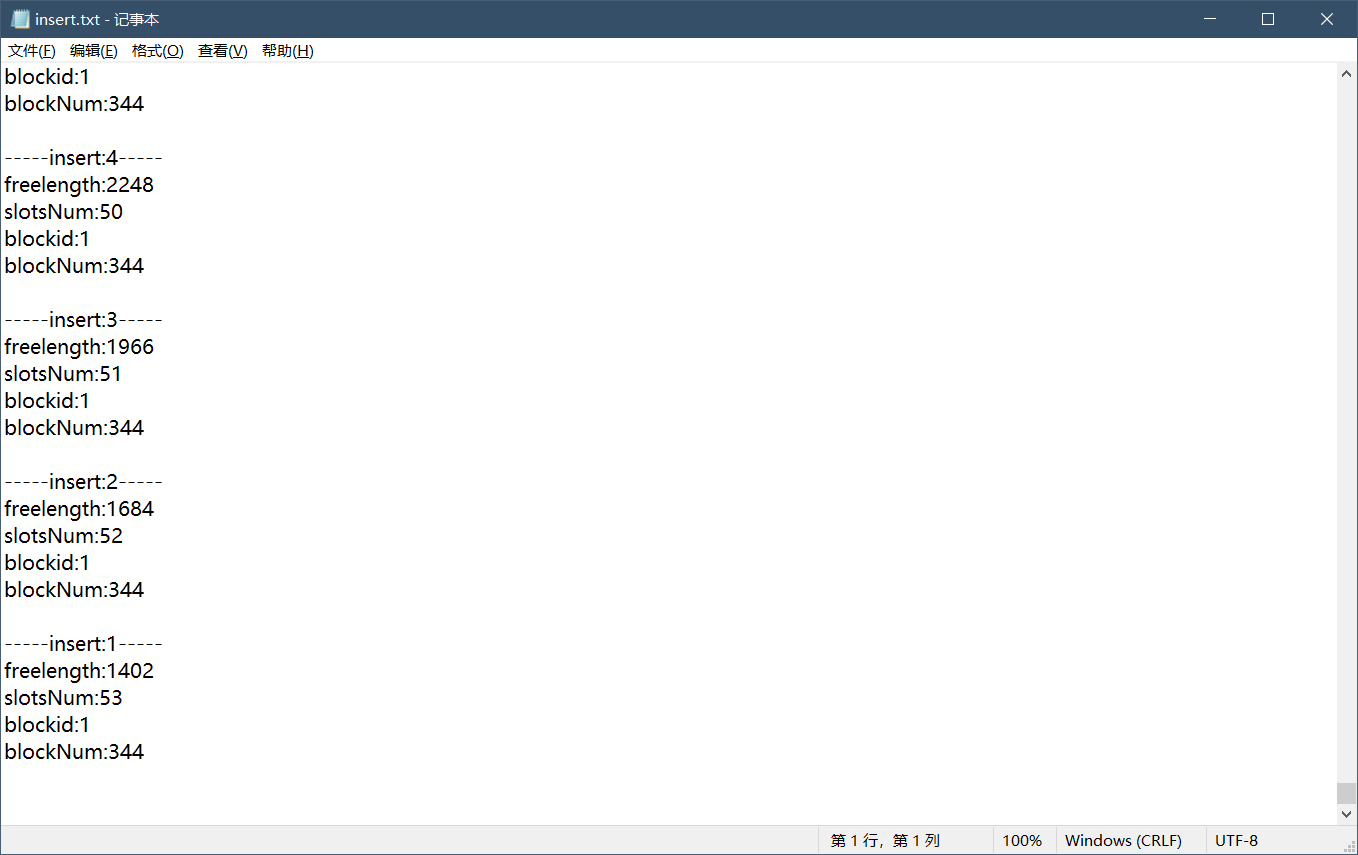


图7·输出的insert.txt文件

4.2 删除功能测试

在插入10000条record后，删除8999条record，删除的主键从1到8999。对于这种顺序删除，每删除一条record，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 < 9000; 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);

*//删除后，判断第一个记录是不是i+1*

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

DataBlock 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);

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

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

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

outputfile << std::endl;

}

table.close("tablee.dat");

outputfile.close();

}

测试结果表明，代码通过了所有的测试。

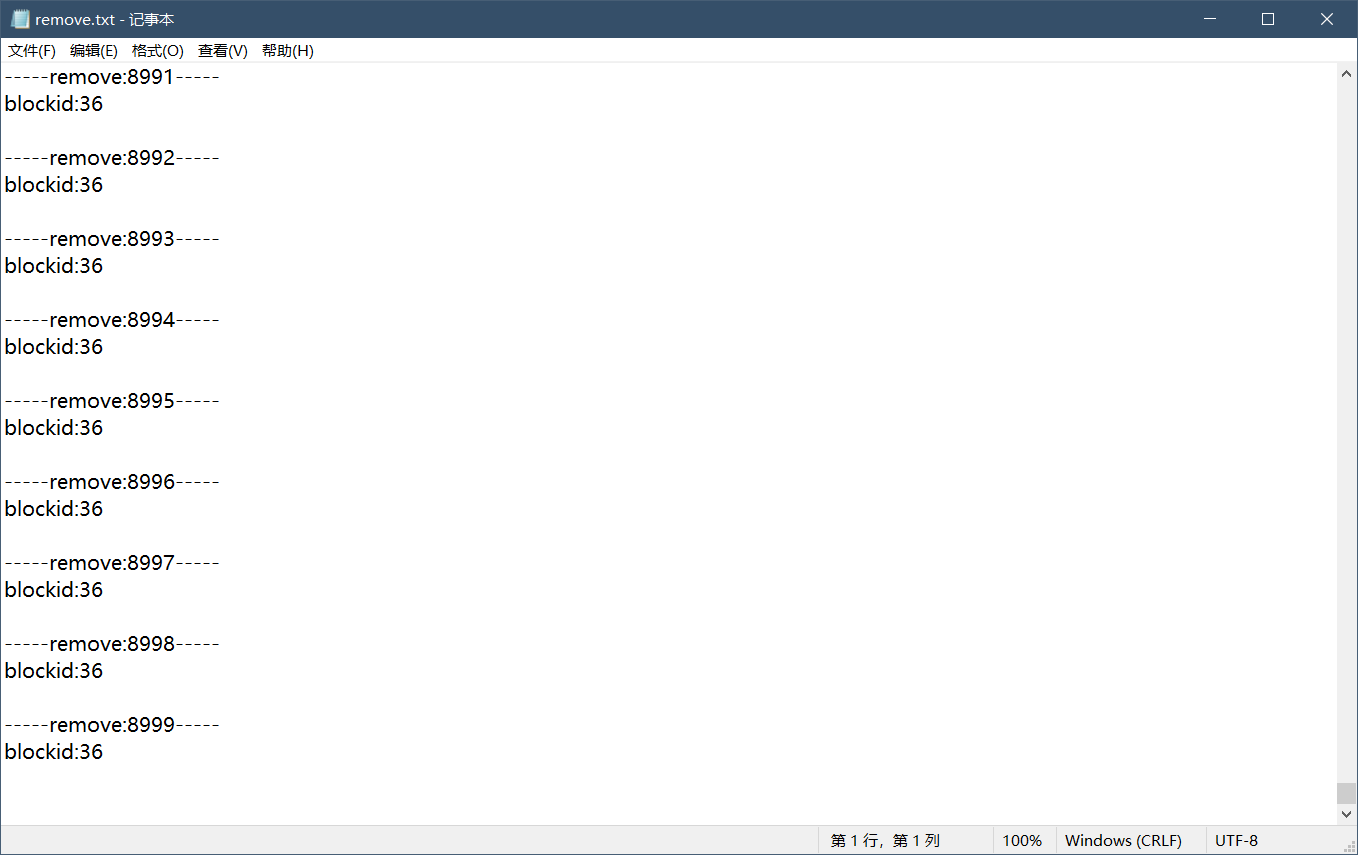


图8·输出的remove.txt文件

4.2 修改功能测试

在删除了8999条record记录后，把第9000条记录的主键修改为3，电话号码也进行修改。因为主键修改为3后，这条记录应该成为这个表的第一条记录，因此，判断表的第一个记录的主键是否是3，电话号码是否为新的电话号码。

SECTION("updata")

{

iovec field;

long long keyid = 9000;

field.iov\_base = &keyid;

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

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

long long id = 3;

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

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

char \*phone = "13318181238";

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

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

char \*name = "Junix";

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

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

unsigned char header = 0x84;

Table table;

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

REQUIRE(ret == S\_OK);

ret = table.initial();

REQUIRE(ret == S\_OK);

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

REQUIRE(ret == S\_OK);

*//表的第一个记录*

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

DataBlock 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 == 3); *//判断主键*

record.specialRef(Field, 1);

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

REQUIRE(

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

0); *//判断电话号码*

table.close("tablee.dat");

}

测试结果表明，代码通过了测试。

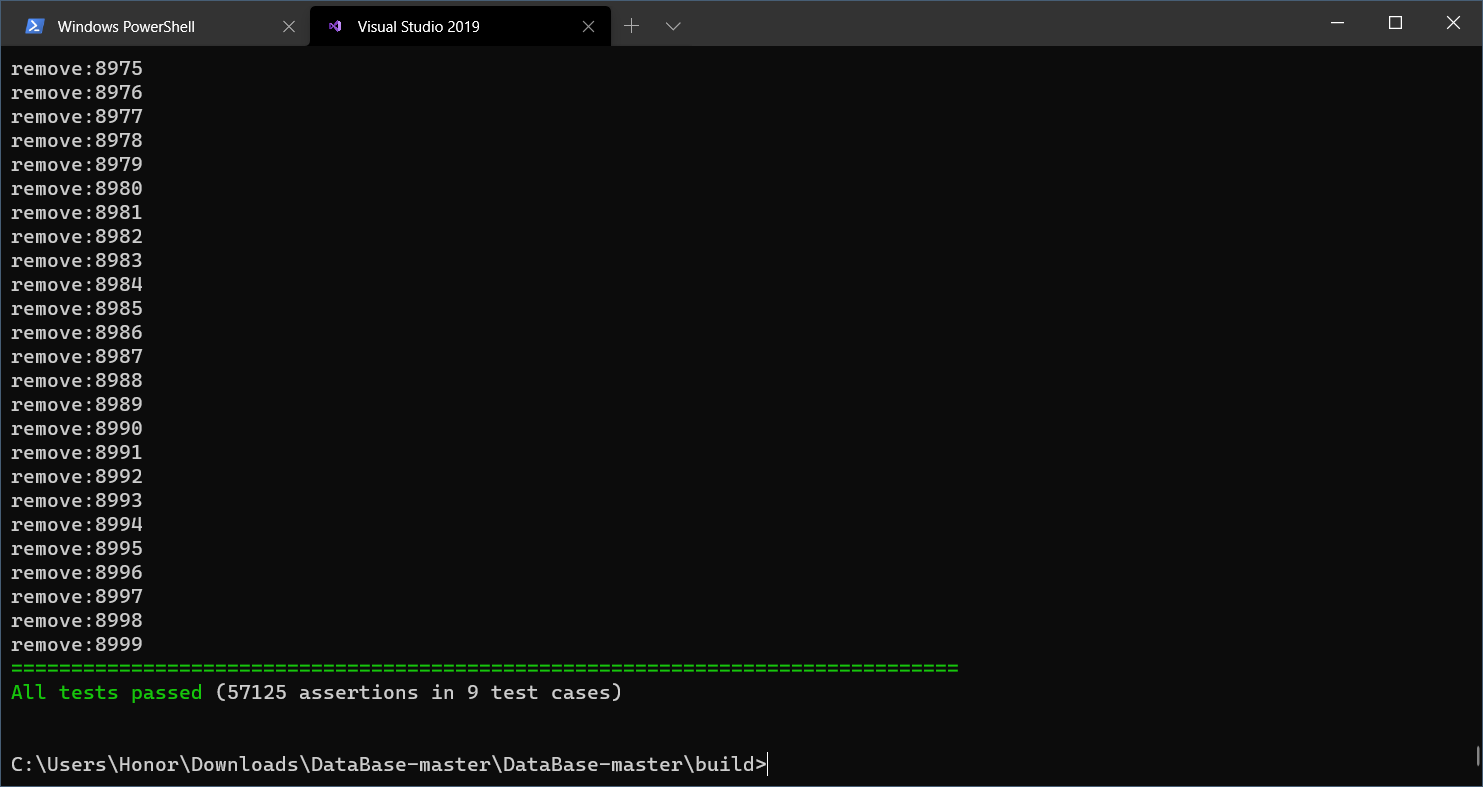


图9·代码通过所有测试

第五章 项目总结与展望

5.1 项目总结

本项目根据磁盘管理的知识设计并实现一个磁盘存储系统，包含记录的插入、修改、删除、枚举。

5.2 项目展望

本项目在定位block的时候，采取了线性逐个比较的方式，可以进一步引进二分查找的思想，加速block的定位过程。

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。