Minisql总体设计报告

叶沐阳 3210104095

实验完成说明

单人完成五个模块的代码，均通过test测试；在main中进行测试，能完成验收群中测试文件的全部指令；完成bonus1，实现了clock replacer。证明截图和代码如下（本实验在ubuntu虚拟机中完成）：

通过总测试：

```
ymy@ymy-VirtualBox:~/minisql-master/build$ make minisql_test
[ 10%] Built target glogbase
[ 12%] Built target glog
[ 74%] Built target zSql
[ 78%] Built target gtest
[100%] Built target minisql_test
ymy@ymy-VirtualBox:~/minisql-master/build$ ./test/minisql_test
[==========] Running 19 tests from 9 test suites.
[----------] Global test environment set-up.
[----------] 1 test from BufferPoolManagerTest
[ RUN      ] BufferPoolManagerTest.BinaryDataTest
[       OK ] BufferPoolManagerTest.BinaryDataTest (0 ms)
[----------] 1 test from BufferPoolManagerTest (0 ms total)

[----------] 1 test from LRUReplacerTest
[ RUN      ] LRUReplacerTest.SampleTest
[       OK ] LRUReplacerTest.SampleTest (0 ms)
[----------] 1 test from LRUReplacerTest (0 ms total)

[----------] 3 tests from CatalogTest
[ RUN      ] CatalogTest.CatalogMetaTest
```

```
[----------] 1 test from TableHeapTest
[ RUN      ] TableHeapTest.TableHeapSampleTest
[       OK ] TableHeapTest.TableHeapSampleTest (648 ms)
[----------] 1 test from TableHeapTest (648 ms total)

[----------] Global test environment tear-down
[==========] 19 tests from 9 test suites ran. (14885 ms total)
[  PASSED  ] 19 tests.
ymy@ymy-VirtualBox:~/minisql-master/build$
```

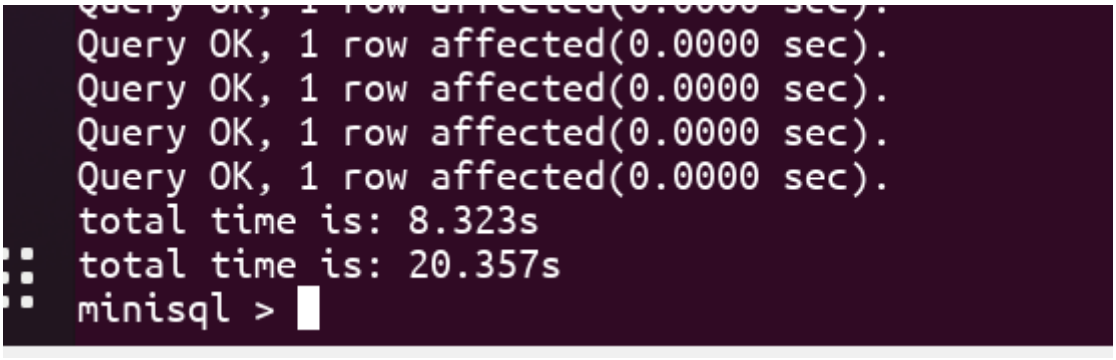
验收时运行的测试指令如下：

```
//sql.txt文件中，execfile指令执行
create database db0;
create database db1;
create database db2;
show databases;
use db0;
create table account(
  id int,
  name char(16),
  balance float,
  primary key(id)
);
```

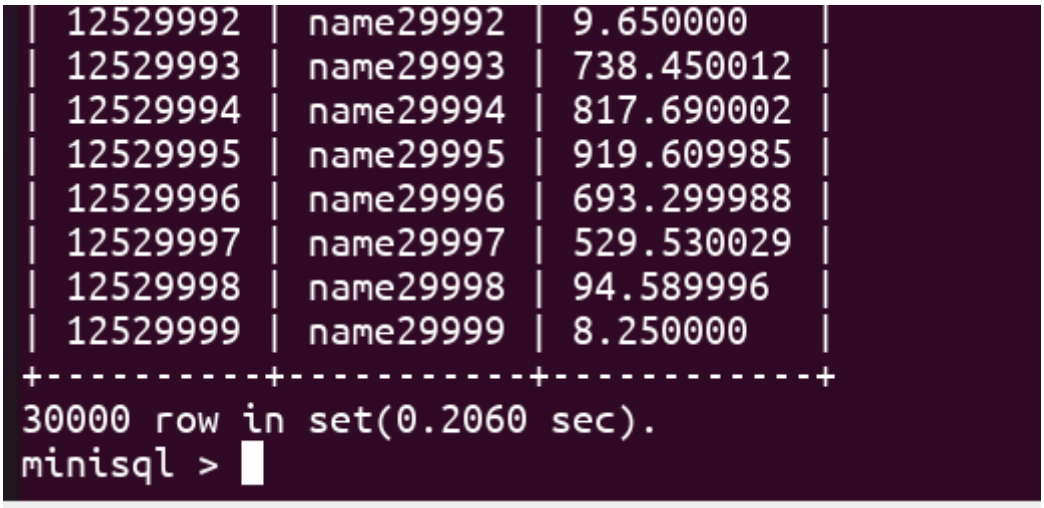
```
execfile "account00.txt";
execfile "account01.txt";
execfile "account02.txt";

//手动输入测试
select * from account;//验证插入30000条成功
select * from account where name = "name26789";//select功能
select name, balance from account where balance > 800 and id <= 12529999;//投影，范围查找

delete from account where balance <>9.65;//不等于条件
select * from account;//删除后，只剩下一条记录
insert into account values(12525600, "name25600", 880.67);//插入功能
insert into account values(12525600, "name25600", 880.67);//主键约束，插入失败
update account set name = "name000" where where id=12525600;//更新功能

create index idx01 on account(name);//建立索引
show indexes;
drop index idx01;//删除索引

delete from account;//删除所有行

show tables;
drop table account;//删除table
show tables;
```

执行截图：

执行 execfile "sql.txt"; 完成30000条指令插入，总耗时20s。



select检查，输出30000条指令。



```
select * from account where name = "name26789";执行seq_scan检索，select功能正常。
```

```
minisql > select * from account where name = "name26789";
[INFO] Sql syntax parse ok!
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12526789 | name26789 | 665.830017 |
+----------+-----------+------------+
1 row in set(0.0910 sec).
minisql >
```

```
select id,name where balance > 800 and id <= 12529999;//投影，范围查找
```

在30000行数据中总共找到5898行满足以上条件，输出了对应的id,name字段。

```
| 12529978 | name29978 |
| 12529984 | name29984 |
| 12529987 | name29987 |
| 12529990 | name29990 |
| 12529994 | name29994 |
| 12529995 | name29995 |
+----------+-----------+
5898 row in set(0.1440 sec).
minisql >
```

```
delete from account where balance <>9.65;//删除功能和不等于检索
```

执行后，删除了除了balance = 9.65的所有行，共移除了29999行。利用select语句检查，只剩下一行。

```
minisql > delete from account where balance <>9.65;
[INFO] Sql syntax parse ok!
Query OK, 29999 row affected(0.5670 sec).
minisql > select * from account;
[INFO] Sql syntax parse ok!
+----------+-----------+----------+
| id       | name      | balance  |
+----------+-----------+----------+
| 12529992 | name29992 | 9.650000 |
+----------+-----------+----------+
1 row in set(0.0000 sec).
minisql >
```

```
insert into account values(12525600, "name25600", 880.67);//插入功能
```

插入功能正常。执行该插入指令两次，第二次返回duplicate primary key的错误信息，0 row affected，主键约束生效，无法插入id相同的两行数据。

select 语句检查，发现插入成功。

```
minisql > insert into account values(12525600, "name25600", 880.67);
[INFO] Sql syntax parse ok!
Query OK, 1 row affected(0.0000 sec).
minisql > insert into account values(12525600, "name25600", 880.67);
[INFO] Sql syntax parse ok!
I20230628 15:49:41.438935  3949 insert_executor.cpp:39] duplicate primary key
Query OK, 0 row affected(0.0000 sec).
minisql > select * from account;
[INFO] Sql syntax parse ok!
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12529992 | name29992 | 9.650000   |
| 12525600 | name25600 | 880.669983 |
+----------+-----------+------------+
2 row in set(0.0000 sec).
minisql >
```

```
update account set name = "name000" where where id=12525600;//更新指令
```

输入update指令，对指定行进行更新；输出更新前后的行信息，发现更新成功。

```
minisql > select * from account;
[INFO] Sql syntax parse ok!
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12529992 | name29992 | 9.650000   |
| 12525600 | name25600 | 880.669983 |
+----------+-----------+------------+
2 row in set(0.0000 sec).
minisql > update account set name="name000" where id=12525600;
[INFO] Sql syntax parse ok!
Query OK, 1 row affected(0.0010 sec).
minisql > select * from account;
[INFO] Sql syntax parse ok!
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12529992 | name29992 | 9.650000   |
| 12525600 | name000   | 880.669983 |
+----------+-----------+------------+
2 row in set(0.0000 sec).
minisql > ^A
```

```
create index idx01 on account(name);//在name列上建立索引。
show indexes;
drop index idx01;//删除索引
```

利用show indews指令检查建立索引、删除索引是否成功。

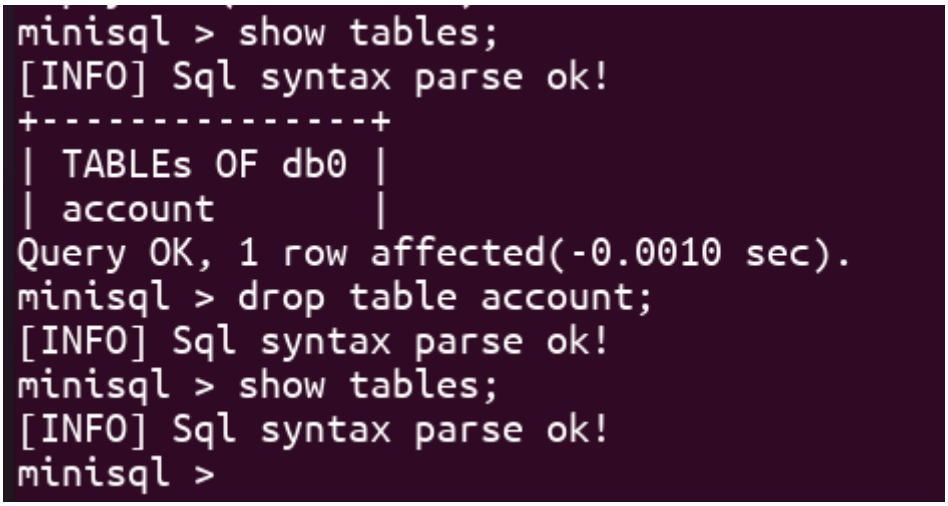
```
minisql > show indexes;
[INFO] Sql syntax parse ok!
+----------------------+
| Indexes              |
| account.PK OF account |
Query OK, 1 row affected(-0.0010 sec).
minisql > create index idx01 on account(name);
[INFO] Sql syntax parse ok!
minisql > show indexes;
[INFO] Sql syntax parse ok!
+----------------------+
| Indexes              |
| account.idx01        |
| account.PK OF account |
Query OK, 2 row affected(-0.0010 sec).
minisql > drop index idx01;
[INFO] Sql syntax parse ok!
minisql > show indexes;
[INFO] Sql syntax parse ok!
+----------------------+
| Indexes              |
| account.PK OF account |
Query OK, 1 row affected(-0.0010 sec).
minisql >
```

```
delete from account;//删除所有行
```

删除所有行并检查。

```
Query OK, 1 row affected(-0.0010 sec).
minisql > select * from account;
[INFO] Sql syntax parse ok!
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12529992 | name29992 | 9.650000   |
| 12525600 | name25600 | 880.669983 |
+----------+-----------+------------+
2 row in set(0.0000 sec).
minisql > delete from account;
[INFO] Sql syntax parse ok!
Query OK, 2 row affected(0.0000 sec).
minisql > select * from account;
[INFO] Sql syntax parse ok!
Empty set(0.0000 sec).
minisql >
```

```
drop table account;//删除table，并用show tables指令检查。
show tables;
```



Bonus 1说明：

在lru_replacer.h/cpp中实现一种新的缓冲区替换算法Clock Replacer。

具体思路如下：

（1）为每个Replacer中的页面设置一个usebit访问位（初始为1），再将内存中的页面都通过链接指针链接成一个循环队列。

（2）当需要淘汰（victim函数）一个页面时，从index指针开始循环访问链表，检查指向的页面访问位usebit：如果是0，选择此页换出；如果是1，将它置0，暂不换出，继续检查下一个页面。若第一轮扫描中所有页面都是1，则将这些页面的访问位依次置为0，再进行第二轮扫描，第二轮扫描中一定会有访问位为0的页面，因此clock算法选择一个淘汰页面最多会经过两轮扫描。

具体代码如下：

在lru_replacer.h中加入继承类ClockReplacer和ClockListNode结构体。

```
struct ClockListNode{
  ClockListNode* next;
  ClockListNode* prev;//链表的下个点和上一个点
  frame_id_t frame_id;//储存的frame_id
  bool usebit;//是否使用
};

class ClockReplacer : public Replacer{
  public:
  explicit ClockReplacer(size_t num_pages);
  ~ClockReplacer() override;

  bool Victim(frame_id_t *frame_id) override;
  void Pin(frame_id_t frame_id) override;
  void Unpin(frame_id_t frame_id) override;
  size_t Size() override;

private:
  size_t size;
  unordered_map<frame_id_t, ClockListNode*> Hash;
```

```
  ClockListNode* Head;//frame_id + use bit
  ClockListNode* index;
};

#endif  // MINISQL_LRU_REPLACER_H
```

lru_replacer.cpp中加入具体的类实现代码：

```
//clockreplacer
ClockReplacer::ClockReplacer(size_t num_pages){
  size = 0;
  Hash.clear();
  Head = new ClockListNode();//dummy node
  Head->next = Head->prev = nullptr;
  index = nullptr;
}

ClockReplacer::~ClockReplacer(){
  ClockListNode *node = Head->next;
  delete Head;
  while(size){//删除掉所有的点
    Head = node;
    node = node->next;
    delete Head;
    size--;
  }
}

bool ClockReplacer::Victim(frame_id_t *frame_id){
  if(!size) return false;
  while(index->usebit){
    index->usebit ^= 1;//找到一个可以替换的点
    index = index->next;//不断找下一个
  }
  *frame_id = index->frame_id;
  Hash.erase(*frame_id);
  size--;
  ClockListNode* DeleteNode = nullptr;
  // cout << size << endl;
  if(size){
    index->prev->next = index->next;
    index->next->prev = index->prev;
    DeleteNode = index;
    index = index->next;
  }else{
    DeleteNode = index;
    index = nullptr;
  }
  delete DeleteNode;
  return true;
}

void ClockReplacer::Pin(frame_id_t frame_id){
  if(!Hash.count(frame_id)) return;//如果不在其中就不做任何作用
  ClockListNode* node = Hash[frame_id];
```

```
  Hash.erase(node->frame_id);
  size--;
  ClockListNode* DeleteNode = nullptr;
  if(size){
    node->prev->next = node->next;
    node->next->prev = node->prev;
    DeleteNode = node;
    if(index == node) index = node->next;//相等时需要移动index
  }else{
    DeleteNode = node;
    index = nullptr;
  }
  delete DeleteNode;
}

void ClockReplacer::Unpin(frame_id_t frame_id){
  if(Hash.count(frame_id)){
    Hash[frame_id]->usebit = 1;
    return;//如果在其中就设置一下使用位
  }
  ClockListNode* newnode = new ClockListNode();
  newnode->frame_id = frame_id;
  newnode->usebit = true;//刚刚使用过
  if(!size){
    Head->next = newnode;
    Head->next->prev = Head->next;
    Head->next->next = Head->next;//自己的前后都是自己
    index = newnode;//初始化index
  }
  else{//设置newnode为index的前一个点
    newnode->next = index;//下一个点
    newnode->prev = index->prev;//最后一个点
    newnode->next->prev = newnode;
    newnode->prev->next = newnode;
    Head->next = newnode;//change the first point
  }
  Hash[frame_id] = Head->next;
  size++;
}

size_t ClockReplacer::Size() {
  return size;
}
```

可以在BufferPoolManager选择具体的replacer实现结构，来切换使用ClockReplacer。

```
BufferPoolManager::BufferPoolManager(size_t pool_size, DiskManager *disk_manager)
    : pool_size_(pool_size), disk_manager_(disk_manager) {
  pages_ = new Page[pool_size_];
  replacer_ = new LRUReplacer(pool_size_);
  //replacer_ = new ClockReplacer(pool_size_);

  for (size_t i = 0; i < pool_size_; i++) {
    free_list_.emplace_back(i);
  }
}
```

实验目标

1. 设计并实现一个精简型单用户SQL引擎MiniSQL，允许用户通过字符界面输入SQL语句实现基本的增删改查操作，并能够通过索引来优化性能。
2. 通过对MiniSQL的设计与实现，提高学生的系统编程能力，加深对数据库管理系统底层设计的理解。

系统需求：

1. 数据类型：要求支持三种基本数据类型：`integer`，`char(n)`，`float`。
2. 表定义：一个表可以定义多达32个属性，各属性可以指定是否为`unique`，支持单属性的主键定义。
3. 索引定义：对于表的主属性自动建立B+树索引，对于声明为`unique`的属性也需要建立B+树索引。
4. 数据操作: 可以通过`and`或`or`连接的多个条件进行查询，支持等值查询和区间查询。支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。
5. 在工程实现上，使用源代码管理工具（如Git）进行代码管理，代码提交历史和每次提交的信息清晰明确；同时编写的代码应符合代码规范，具有良好的代码风格。

模块实现

模块一 Disk Manager

模块需求：

Database File 是存储数据库中所有数据的文件，其主要由记录（Record）数据、索引（Index）数据和目录（Catalog）数据组成。在本实验中，采取共享表空间的设计方式，即将所有的数据和索引放在同一个文件中。Disk Manager负责DB File中数据页的分配和回收，以及数据页中数据的读取和写入。

对应实验：#1 DISK AND BUFFER POOL MANAGER

模块实现：

Bitmap

位图页BitmapPage需要实现对页面的分配、回收和释放。采用bitset结构登记页面是否存在；用page_allocated变量记录已分配页面数量，保证不会超过bitmap容量；next_free_page记录下一个空位。

```
private:
 /** The space occupied by all members of the class should be equal to the PageSize */
 [[maybe_unused]] uint32_t page_allocated_{0};
 [[maybe_unused]] uint32_t next_free_page_{0};//初始化为0

 //[[maybe_unused]] unsigned char bytes[MAX_CHARS];
  std::bitset<8 * MAX_CHARS> bytes;
};
```

DiskManager

把一个位图页加一段连续的数据页看成数据库文件中的一个分区（Extent），再通过一个额外的元信息页来记录这些分区的信息。

DiskMetaPage是数据库文件中的第0个数据页，它维护了分区相关的信息，如分区的数量num_extents、每个分区中已经分配的页的数量num_allocated_pages和各个分区中page数量extent_used_page_[]。

```
public:
 uint32_t num_allocated_pages_{0};
 uint32_t num_extents_{0};  // each ex
 uint32_t extent_used_page_[0];
};
```

分配：AllocatePage()

如果当前分区已满，新增一个分区，新建位图页面，分配新页面；如果已达到最大分区数，无法新建，返回INVALID。

如果当前分区还未满，分配新页面，读取该分区的bitmap并修改相关属性。

释放：DeAllocatePage(logical_page_id)

读取该分区的bitmap（即extent_meta_page），释放页面；读取整个disk的metapage，修改相关属性。

模块二 Buffer Pool Manager

模块需求：

Buffer Manager 负责缓冲区的管理，主要功能包括：

1. 根据需要，从磁盘中读取指定的数据页到缓冲区中或将缓冲区中的数据页转储（Flush）到磁盘；
2. 实现缓冲区的替换算法，当缓冲区满时选择合适的数据页进行替换；
3. 记录缓冲区中各页的状态，如是否是脏页（Dirty Page）、是否被锁定（Pin）等；
4. 提供缓冲区页的锁定功能，被锁定的页将不允许替换。。

对应实验：#1 DISK AND BUFFER POOL MANAGER。

模块实现：

Replacer

Buffer Pool Replacer负责跟踪Buffer Pool中数据页的使用情况，并在Buffer Pool没有空闲页时决定替换哪一个数据页。在抽象类Replacer的接口下，本实验实现了LRUReplacer/Clock Replacer两种替换策略。

LRU实现思路：

新建结构体ListNode，采用链表的形式，将数据页串联。在LRUReplacer类中记录链表头（虚节点）、尾和长度。

```
struct ListNode{
  ListNode* next;
  ListNode* prev;//链表的下个点和上一个点
  frame_id_t frame_id;//储存的frame_id
};
```

```
private:
  unordered_map<frame_id_t, ListNode*> Hash;//unordered_map记录frame中的位置
  ListNode *Head, *Tail;
  size_t size;
  size_t MAX_SIZE;//缓冲区最大容量
  // add your own private member variables here
};
```

替换：删除最前面的点；Pin：将指定数据页从链表中删除；Unpin：将数据页插入链表尾部。

Clock Replacer实现详见报告开头bonus 1说明。

BufferPoolManager

所有内存页面都由Page对象表示，每个Page对象都包含了一段连续的内存空间data和相关的页面信息（标识符page_id，固定页面的线程数pin_count，是否脏页is_dirty_）。

在BufferPoolManager类中设计了page_table(页表)记录page_id和frame_id的映射关系；replacer调用对应策略查找替换；free_list提供替换名单。

```
private:
 size_t pool_size_;                              // number of pages in buffer pool
 Page *pages_;                                   // array of pages
 DiskManager *disk_manager_;                     // pointer to the disk manager.
 unordered_map<page_id_t, frame_id_t> page_table_;  // to keep track of pages
 Replacer *replacer_;                            // to find an unpinned page for replacement
 list<frame_id_t> free_list_;                    // to find a free page for replacement
 recursive_mutex latch_;                         // to protect shared data structure
};
//pages_[frame_id]存储了对应的page 对应关系在page_table_中
```

获取页面FetchPage：

在页表中搜索请求的页 P。如果 P 存在，固定并立即返回。如果 P 不存在，从free_list中找到替换页面R；若free_list为空，用replacer的victim函数获得替换页面R。

如果 R是脏页，将其写回磁盘。从页表中删除 R 并插入 P，更新 P 的元数据，从磁盘读取页面内容，然后返回指向 P 的指针。

分配新页面NewPage：

如果缓冲池中的所有页面都固定，则返回 nullptr。从free_list中找到可替换页面P；若free_list为空，用replacer的victim函数获得替换页面P。始终先从免费列表中选择。

更新 `P` 的`meta`属性，将`data`清除并将 `P` 添加到页表中。返回指向 `P` 的指针。

模块三 Record Manager

模块需求

Record Manager 负责管理数据表中记录。所有的记录以堆表（Table Heap）的形式进行组织。Record Manager 的主要功能包括：记录的插入、删除与查找操作，并对外提供相应的接口。其中查找操作返回的是符合条件记录的起始迭代器，对迭代器的迭代访问操作由执行器（Executor）进行。

对应实验：#2 RECORD MANAGER。

模块实现

序列化

多采用宏定义MACH_WRITE_INT32(Buf, Data)等完成，注意schema、column序列化、反序列化开头处要进行对应的SCHEMA_MAGIC_NUM、COLUMN_MAGIC_NUM验证。

Table Heap

对于数据表中的每一行记录，都有一个唯一标识符RowId与之对应，存储了该RowId对应记录所在数据页的page_id，和对应的是数据页中的第几条记录。

堆表是由多个数据页构成的链表，每个数据页中包含一条或多条记录。堆表中所有的记录都是无序存储的。

插入：沿着TablePage构成的链表依次查找，直到找到第一个能够容纳该记录的TablePage（First Fit 策略）。当需要从堆表中删除指定RowId对应的记录时，通过打上Delete Mask来标记记录被删除，在之后某个时间段再从物理意义上真正删除该记录。

更新：如果TablePage能够容纳下更新后的数据，直接更新；如果不能够容纳下，先删除原数据，再插入新数据。

模块四 Index Manager

模块说明

Index Manager 负责数据表索引的实现和管理，包括：索引的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。在本实验中，实现了一个基于磁盘的B+树动态索引结构。

对应实验：#3 INDEX MANAGER。

模块实现：

插入：超过叶节点，如果超过max_size大小，分裂split；递归处理父节点。

代码如下：

```
bool BPlusTree::InsertIntoLeaf(GenericKey *key, const RowId &value, Transaction
*transaction) {
  //注意插入一定不会是某个叶节点的第一个，所以插入后除非分裂，父节点的key不会变化
  Page* fth_page = this->FindLeafPage(key);//被插入的叶页面
  LeafPage* tmp_leaf_page = reinterpret_cast<LeafPage*>(fth_page->GetData());//被
插入的叶节点
  int old_size = tmp_leaf_page->GetSize();
  int new_size = tmp_leaf_page->Insert(key, value, processor_);
  if(new_size==old_size){//重复
    fth_page->WUnlatch();
    buffer_pool_manager_->UnpinPage(fth_page->GetPageId(), false);
    return false;
  }
  else if(new_size < leaf_max_size_){//没有分裂
    fth_page->WUnlatch();
    buffer_pool_manager_->UnpinPage(fth_page->GetPageId(), true);//修改了
    return true;
  }else{
    //分裂
```

```
    LeafPage* sibling_leaf_node = Split(tmp_leaf_page,transaction);//注意unpin和
WULATCH
    sibling_leaf_node->SetNextPageId(tmp_leaf_page->GetNextPageId());//继承了tmp后
半部分的元素
    tmp_leaf_page->SetNextPageId(sibling_leaf_node->GetPageId());
    //处理父节点的key
    GenericKey *new_key = sibling_leaf_node->KeyAt(0);
    InsertIntoParent(tmp_leaf_page, new_key, sibling_leaf_node, transaction);
    fth_page->WUnlatch();
    buffer_pool_manager_->UnpinPage(fth_page->GetPageId(), true);
    buffer_pool_manager_->UnpinPage(sibling_leaf_node->GetPageId(), true);
    return true;
  }

}

BPlusTreeInternalPage *BPlusTree::Split(InternalPage *node, Transaction
*transaction) {
  page_id_t new_page_id;
  Page* new_page = buffer_pool_manager_->NewPage(new_page_id);
  InternalPage* new_node = reinterpret_cast<InternalPage*>(new_page->GetData());

  new_node->Init(new_page_id, node->GetParentPageId(), node-
>GetKeySize(),internal_max_size_);
  node->MoveHalfTo(new_node, buffer_pool_manager_);//此时new_node的
keyat(0)!=INVALID，而是保存了VALUE(0)的最小值

  return new_node;
}

BPlusTreeLeafPage *BPlusTree::Split(LeafPage *node, Transaction *transaction) {
  page_id_t new_page_id;
  Page* new_page = buffer_pool_manager_->NewPage(new_page_id);//申请新页面
  LeafPage* new_node = reinterpret_cast<LeafPage*>(new_page->GetData());//新页面是
叶节点

  new_node->Init(new_page_id,node->GetParentPageId(),node-
>GetKeySize(),leaf_max_size_);

  node->MoveHalfTo(new_node);//移动后半节点
  return new_node;
}

void BPlusTree::InsertIntoParent(BPlusTreePage *old_node, GenericKey *key,
BPlusTreePage *new_node,
                                 Transaction *transaction) {
  //如果分裂的是根节点
  if(old_node->IsRootPage()){
    Page* new_page = buffer_pool_manager_->NewPage(root_page_id_);//生成新根节点

    InternalPage *new_root = reinterpret_cast<InternalPage *>(new_page-
>GetData());
    new_root-
>Init(root_page_id_,INVALID_PAGE_ID,processor_.GetKeySize(),internal_max_size_);
    new_root->PopulateNewRoot(old_node->GetPageId(),key,new_node->GetPageId());
```

```
    old_node->SetParentPageId(root_page_id_);
    new_node->SetParentPageId(root_page_id_);
    buffer_pool_manager_->UnpinPage(new_page->GetPageId(), true);
    UpdateRootPageId(0);//更新
    return;
  }
  //如果不是根节点,插入在父节点上
  Page* parent_page = buffer_pool_manager_->FetchPage(old_node-
>GetParentPageId());
  InternalPage *parent_node = reinterpret_cast<InternalPage *>(parent_page-
>GetData());
  int new_size = parent_node->InsertNodeAfter(old_node->GetPageId(),key,
new_node->GetPageId());//在old key后加上new key
  if (new_size < internal_max_size_) //中间节点不分裂
  {
    buffer_pool_manager_->UnpinPage(parent_page->GetPageId(), true);
  }else{
    //父节点分裂
    auto parent_new_sibling_node = Split(parent_node,transaction);
    GenericKey* new_key = parent_new_sibling_node->KeyAt(0);//注意此时KAY(0)保存了
VALUE(0)中最小值,即new_node最小值
    InsertIntoParent(parent_node, new_key, parent_new_sibling_node,
transaction);

    buffer_pool_manager_->UnpinPage(parent_page->GetPageId(), true);
    buffer_pool_manager_->UnpinPage(parent_new_sibling_node->GetPageId(), true);
  }
  return;
}
```

删除:

如果叶节点大小小于min_size，进入CoalesceOrRedistribute()函数判断。如果和兄弟节点合并后小于max_size，合并节点，调用Coalesce()，递归处理父节点；反之则重分配节点，调用Redistribute()。

```
void BPlusTree::Remove(const GenericKey *key, Transaction *transaction) {
  if(IsEmpty())
    return;
  Page* leaf_page = FindLeafPage(key);//找到页面
  LeafPage *node = reinterpret_cast<LeafPage *>(leaf_page->GetData());
  int org_size = node->GetSize();
  int new_size = node->RemoveAndDeleteRecord(key, processor_);
  if (org_size == new_size) //如果不存在key
  {
    leaf_page->WUnlatch();
    buffer_pool_manager_->UnpinPage(leaf_page->GetPageId(), false);
    return;
  }
  else
  {
    leaf_page->WUnlatch();
    if(CoalesceOrRedistribute(node, transaction))//如果该node确认要删除
    {
      buffer_pool_manager_->DeletePage(node->GetPageId());
    }
```

```
    buffer_pool_manager_->UnpinPage(leaf_page->GetPageId(), true);
  }
  return ;

}

//判断合并or分配，返回node是否应该被删除
template <typename N>
bool BPlusTree::CoalesceOrRedistribute(N *&node, Transaction *transaction) {
  //如果是根节点
  if(node->IsRootPage())
    return AdjustRoot(node);//返回是否应该被删除
  //如果删除后大于等于最小size
  if (node->GetSize() >= node->GetMinSize())
    return false;
  //一般情况判断(非根节点删除后小于min_size)
  Page* parent_page = buffer_pool_manager_->FetchPage(node-
>GetParentPageId());//父节点
  InternalPage* parent_node = reinterpret_cast<InternalPage *>(parent_page-
>GetData());

  int index = parent_node->ValueIndex(node->GetPageId());//对应index
  int sibling_index;
  if(index == 0)
    sibling_index = 1;
  else
    sibling_index = index -1;//确保存在兄弟节点

  Page* sibling_page = buffer_pool_manager_->FetchPage(parent_node-
>ValueAt(sibling_index));
  N* sibling_node = reinterpret_cast<N*>(sibling_page->GetData());
  //sibling_page->Wlatch();?
  //重新分配
  if (node->GetSize() + sibling_node->GetSize() >= node->GetMaxSize())
  {
    Redistribute(sibling_node, node, index);

    buffer_pool_manager_->UnpinPage(parent_node->GetPageId(), true);
    sibling_page->WUnlatch();
    buffer_pool_manager_->UnpinPage(sibling_page->GetPageId(), true);

    return false;
  }
  //合并
  else
  {
    bool if_should_delete = Coalesce(sibling_node, node, parent_node, index);//返
回父节点是否应该被删除
    if(if_should_delete)
    {
      buffer_pool_manager_->DeletePage(parent_node->GetPageId());
    }
    buffer_pool_manager_->UnpinPage(parent_node->GetPageId(), true);
    sibling_page->WUnlatch();
    buffer_pool_manager_->UnpinPage(sibling_page->GetPageId(), true);
```

```
    return true;
  }
  return false;
}

bool BPlusTree::Coalesce(LeafPage *&neighbor_node, LeafPage *&node, InternalPage
*&parent, int index,
                         Transaction *transaction) {
  int delete_index= index;
  if (index == 0)
  {
    delete_index = 1;
    std::swap(node, neighbor_node);
  }//sibling在前 node在后面

  node->MoveAllTo(neighbor_node);
  buffer_pool_manager_->UnpinPage(neighbor_node->GetPageId(), true);

  parent->Remove(delete_index);
  return CoalesceOrRedistribute(parent);//递归父亲节点
}

bool BPlusTree::Coalesce(InternalPage *&neighbor_node, InternalPage *&node,
InternalPage *&parent, int index,
                         Transaction *transaction) {
  int delete_index= index;
  if (index == 0)
  {
    delete_index = 1;
    std::swap(node, neighbor_node);
  }//sibling在前 node在后面

  GenericKey* middle_key = parent->KeyAt(delete_index);//node value(0)中的最小值,记
录在父节点中
  node->MoveAllTo(neighbor_node,middle_key,buffer_pool_manager_);
  buffer_pool_manager_->UnpinPage(neighbor_node->GetPageId(), true);

  parent->Remove(delete_index);
  return CoalesceOrRedistribute(parent);//递归父亲节点
}

void BPlusTree::Redistribute(LeafPage *neighbor_node, LeafPage *node, int index)
{//index是node的
  Page* parent_page = buffer_pool_manager_->FetchPage(node->GetParentPageId());
  InternalPage* parent_node = reinterpret_cast<InternalPage*>(parent_page-
>GetData());
  //node在前面，neighbor第一个前移
  if (index == 0)
  {
    neighbor_node->MoveFirstToEndOf(node);
    parent_node->SetKeyAt(1, neighbor_node->KeyAt(0));
  }
  //node在后，neighbor_node最后一个后移
  else
  {
```

```
    neighbor_node->MoveLastToFrontOf(node);
    parent_node->SetKeyAt(index, node->KeyAt(0));
  }
  buffer_pool_manager_->UnpinPage(parent_node->GetPageId(), true);//修改了parent

}
void BPlusTree::Redistribute(InternalPage *neighbor_node, InternalPage *node,
int index) {
  Page* parent_page = buffer_pool_manager_->FetchPage(node->GetParentPageId());
  InternalPage* parent_node = reinterpret_cast<InternalPage*>(parent_page-
>GetData());
  //node在前面，neighbor第一个前移
  if (index == 0)
  {
    neighbor_node->MoveFirstToEndOf(node,parent_node-
>KeyAt(1),buffer_pool_manager_);
    parent_node->SetKeyAt(1, neighbor_node->KeyAt(0));
  }
  //node在后，neighbor_node最后一个后移
  else
  {
    neighbor_node->MoveLastToFrontOf(node,parent_node-
>KeyAt(index),buffer_pool_manager_);
    parent_node->SetKeyAt(index, node->KeyAt(0));
  }
  buffer_pool_manager_->UnpinPage(parent_node->GetPageId(), true);


}

bool BPlusTree::AdjustRoot(BPlusTreePage *old_root_node) {
   //如果root是中间节点而且只剩下一个孩子
  if (!old_root_node->IsLeafPage() && old_root_node->GetSize() == 1)
  {
    InternalPage *root_node = reinterpret_cast<InternalPage*>(old_root_node);
    Page* child_page = buffer_pool_manager_->FetchPage(root_node->ValueAt(0));//
唯一的孩子做根节点

    BPlusTreePage *child_node = reinterpret_cast<BPlusTreePage*>(child_page-
>GetData());
    child_node->SetParentPageId(INVALID_PAGE_ID);
    this->root_page_id_ = child_node->GetPageId();
    UpdateRootPageId(0);//更新
    buffer_pool_manager_->UnpinPage(child_page->GetPageId(), true);
    return true;
  }
  if(old_root_node->IsLeafPage() && old_root_node->GetSize() == 0){//如果是叶子节点
且没有孩子，要删除old_root_node
    return true;
  }
  return false;
}
```

模块五 Catalog Manager

模块说明

Catalog Manager 负责管理数据库的所有模式信息，包括：

1. 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
2. 表中每个字段的定义信息，包括字段类型、是否唯一等。
3. 数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

对应实验：#4 CATALOG MANAGER。

模块实现

数据库中定义的表和索引在内存中以TableInfo和IndexInfo的形式表现，其维护了与之对应的表或索引的元信息和操作对象。主要实现CatalogMeta、IndexMetadata、TableMetadata的序列化和反序列化，调用模块二中实现的序列化函数。

CatalogManager

实现该类，完成对TableInfo和IndexInfo的管理。

```
private:
 [[maybe_unused]] BufferPoolManager *buffer_pool_manager_;
 [[maybe_unused]] LockManager *lock_manager_;
 [[maybe_unused]] LogManager *log_manager_;
 CatalogMeta *catalog_meta_;
 std::atomic<table_id_t> next_table_id_;
 std::atomic<index_id_t> next_index_id_;
 // map for tables
 std::unordered_map<std::string, table_id_t> table_names_;
 std::unordered_map<table_id_t, TableInfo *> tables_;
 // map for indexes: table_name->index_name->indexes
 std::unordered_map<std::string, std::unordered_map<std::string, index_id_t>> index_names_;
 std::unordered_map<index_id_t, IndexInfo *> indexes_;
};
```

初始化时，如果不是初次创建，需要从bufferpoolmanager中读取页面，反序列化，创建Table和Index。

CreateTable具体流程：

如果已经存在，返回错误信息。生成table_id，即最后一个table_id+1。生成table_heap，table_meta，获取新页面生成table_meta_page，生成table_info，更新catalog中table相关属性。

创建Index具体流程：

检查table是否存在，返回错误信息。生成index_id。生成key_map，index_meta，获取新页面生成index_meta_page，生成index_info，更新catalog中index相关属性。

模块六 Planner and Executor

模块说明

Executor（执行器）的主要功能是遍历Planner生成的计划树，将树上的 PlanNode 替换成对应的 Executor，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行。Executor采用的是火山模型，提供迭代器接口，每次调用时会返回一个元组和相应的 RID，直到执行完成。

对应实验：#5 PLANNER AND EXECUTOR。

模块实现

execute_engine

实现Table和Index的相关操作（Create，Drop，Show）。

Create Table具体流程：

获得Table_name，遍历ast子节点，找到主键节点并记录主键元素。遍历子节点，获得各个column信息（type,unique,nullable...）并生成column对象合集，遍历完成后生成schema对象。最后，创建主键和unique元素上的索引。

executor

完成了delete、insert、seq_scan、index_scan、update操作的executor执行器。其中，最复杂的是index_scan_executor。具体流程如下：

利用深度搜索算法从根节点开始遍历plan语义树，获得所有and谓语条件的Expression节点。对每个expression，遍历所有indexes进行配对，如果是改条件的index，利用Scankey函数获得所有满足条件的row集合，和之前获得的row集合取交集。最后，如果不是谓词中所有的列上都有索引，对获得的结果再进行seq_scan筛选。

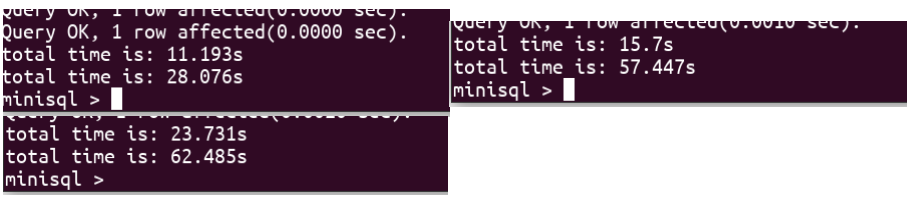
性能测试

执行数据库文件`sql1.txt，sql2.txt，sql3.txt`，分三次共向表中插入100000条记录。

文件内容如下：

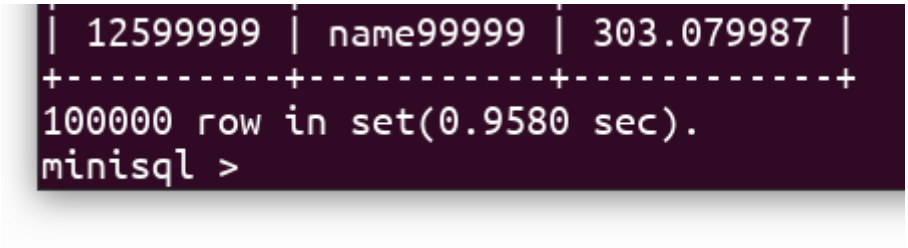
```
//sql1.txt
create database db;
use db;
create table account(
  id int,
  name char(16) unique,
  balance float,
  primary key(id)
);
execfile "account00.txt";
execfile "account01.txt";
execfile "account02.txt";
//sql2.txt
execfile "account03.txt";
execfile "account04.txt";
execfile "account05.txt";
execfile "account06.txt";
//sql3.txt
execfile "account07.txt";
execfile "account08.txt";
execfile "account09.txt";
```

测试用时：



sql1.txt用时28s，sql2.txt用时57s，sql3.txt用时62s。共用时约3分钟。

运行select * from account; 共检索出100000rows。插入成功。



反思：运行速度上存在较大进步空间，可能由于采用ubuntu虚拟机执行，执行速度较慢。