# Minisql-Report

## 3210105321 郑浩博

Row::SerializeTo(*buf, *schema)

实现思路

Row::DeserializeFrom(*buf, *schema)

实现思路

Row::GetSerializedSize(*schema)

实现思路

Column::SerializeTo(*buf)

实现思路

Column::DeserializeFrom(*buf, *&column)

实现思路

Column::GetSerializedSize()

实现思路

Schema::SerializeTo(*buf)

实现思路

Schema::DeserializeFrom(*buf, *&schema)

实现思路

Schema::GetSerializedSize()

实现思路

Table Heap

TableHeap::InsertTuple(&row, *txn)

实现思路

TableHeap::UpdateTuple(&row, &rid, *txn)

实现思路

TableHeap::ApplyDelete(&rid, *txn)

实现思路

TableHeap::GetTuple(*row, *txn)

实现思路

TableHeap::Begin()

实现思路

TableHeap::End()

实现思路

TableIterator::operator++()

实现思路

TableIterator::operator++(int)

实现思路

第三模块-INDEX MANAGER

BPLUSTREELEAFPAGE

Init(page_id, parent_id, key_size, max_size)

实现思路:

KeyIndex(key, KM)

实现思路:

Insert(key, value, KM)

实现思路:

MoveHalfTo(recipient)

实现思路:

Lookup(key, value, KM)

实现思路:

RemoveAndDeleteRecord(key, KM)

实现思路:

MoveAllTo(recipient)

实现思路:

MoveFirstToEndOf(recipient)

实现思路:

CopyLastFrom(key, value)

实现思路:

MoveLastToFrontOf(recipient)

实现思路：

CopyFirstFrom(key, value)

实现思路：

CopyNFrom(src, size)

实现思路：

BPLUSTREEINTERNALPAGE

Init(page_id, parent_id, key_size, max_size)

实现思路：

PopulateNewRoot(old_value, new_key, new_value)

实现思路：

InsertNodeAfter(old_value, new_key, new_value)

实现思路：

MoveHalfTo(recipient, buffer_pool_manager)

实现思路：

Lookup(key, value, KM)

实现思路：

Remove(index)

实现思路：

RemoveAndReturnOnlyChild()

实现思路：

MoveAllTo(recipient)

实现思路：

MoveFirstToEndOf(recipient)

实现思路：

CopyLastFrom(key, value)

实现思路：

MoveLastToFrontOf(recipient)

实现思路：

CopyFirstFrom(key, value)

实现思路：

CopyNFrom(src, size, buffer_pool_manager)

实现思路：

BPLUSTREE

BPlusTree(index_id, buffer_pool_manager, KM, leaf_max_size, internal_max_size)

实现思路：

Destroy(current_page_id)

实现思路：

DestroyAllLeaf(page)

实现思路：

DestroyInternalPage(page)

实现思路：

GetValue(key, result, transaction)

实现思路：

Insert(key, value, transaction)

实现思路：

StartNewTree(key, value)

实现思路：

InsertIntoLeaf(key, value, transaction)

实现思路：

Split(node, transaction)

实现思路：

Split(node, transaction)(leaf)

实现思路：

InsertIntoParent(old_node, key, new_node, transaction)

实现思路：

Remove(key, transaction)

实现思路：

CoalesceOrRedistribute(node, transaction)

实现思路：

Coalesce(neighbor_node, node, parent, index, transaction)

实现思路：

Coalesce(neighbor_node, node, parent, index, transaction)（internal）

实现思路：

Redistribute(neighbor_node, node, index)(leaf)

实现思路：

Redistribute(neighbor_node, node, index)(internal)

实现思路：

AdjustRoot(old_root_node)

实现思路：

Begin()

实现思路：

Begin(key)

实现思路：

End()

实现思路：

FindLeafPage(key, page_id, leftMost)

实现思路：

UpdateRootPageId(insert_record)

实现思路：

第四模块-CATALOG MANAGER

CATALOG

GetSerializedSize()

实现思路：

CatalogManager(buffer_pool_manager, lock_manager, log_manager, init)

实现思路：

CreateIndex(table_name, index_name, index_keys, txn, index_info, index_type)

实现思路：

GetIndex(table_name, index_name, index_info)

实现思路：

GetTableIndexes(table_name, indexes)

实现思路：

CreateTable(table_name, schema, txn, table_info)

实现思路：

GetTable(table_name, table_info)

实现思路：

GetTables(tables)

实现思路：

DropTable(table_name)

实现思路：

DropIndex(table_name, index_name)

实现思路：

DropIndex(table_name, index_name)

实现思路：

FlushCatalogMetaPage()

实现思路：

LoadTable(table_id, page_id)

实现思路：

LoadIndex(index_id, page_id)

实现思路：

第五模块-PLANNER AND EXECUTOR

ExecuteEngine

ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteUseDatabase(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteShowTables(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteDropTable(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteDropIndex(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteExecfile(pSyntaxNode ast, ExecuteContext *context)

实现思路:

ExecuteQuit(pSyntaxNode ast, ExecuteContext *context)

实现思路:

SeqScanExecutor

SeqScanExecutor::Init()

实现思路:

SeqScanExecutor::Next(Row *row, RowId *rid)

实现思路:

IndexScanExecutor

IndexScanExecutor::Init()

实现思路:

IndexScanExecutor::Next(Row *row, RowId *rid)

实现思路:

InsertExecutor

InsertExecutor::Init()

实现思路:

InsertExecutor::Next([[maybe_unused]] Row *row, RowId *rid)

实现思路:

DeleteExecutor

DeleteExecutor::Init()

实现思路:

DeleteExecutor::Next([[maybe_unused]] Row *row, RowId *rid)

实现思路:

UpdateExecutor

UpdateExecutor::Init()

实现思路:

UpdateExecutor::Next([[maybe_unused]] Row *row, RowId *rid)

实现思路:

UpdateExecutor::GenerateUpdatedTuple(const Row &src_row)

实现思路:

**五、验收验证-正确性测试、性能测试**

Test全部通过

性能测试

# 一、实验完成情况总体概况

- 完成bonus： clock-replacer
- 验收全部通过，性能测试、正确性全部通过

## bonus部分代码：

在"buffer/lru_replacer.cpp“中新增

```
/*下面是clock lru 的实现*/

ClockReplacer::ClockReplacer(size_t num_pages) :
capacity(num_pages),victim_num(0),clock_pointer(0){
  struct PageState page;
  page.isPin= true;
  page.recently_visited=true;
  for(int i=0;i<num_pages;i++){
    clock_vec.push_back(page);
  }
}

ClockReplacer::~ClockReplacer() = default;

// the least recently visited id should be returned by pointe frame_id
bool ClockReplacer::Victim(frame_id_t *frame_id)
{

  while(victim_num>0){
    //如果最近被访问，将状态改为false，然后跳过
    if (clock_vec[clock_pointer].recently_visited) {
      clock_vec[clock_pointer].recently_visited = false;
      clock_pointer=(clock_pointer+1)%(capacity);
      continue;
    }
    //如果被pin，直接跳过
    if (clock_vec[clock_pointer].isPin) {
      clock_pointer=(clock_pointer+1)%(capacity);
      continue;
    }
    //满足ispin==false 且 recently_visited==false
    clock_vec[clock_pointer].isPin = true;
    *frame_id =clock_pointer;
    victim_num--;
    clock_pointer=(clock_pointer+1)%(capacity);
    return true;
  }
  return false;//没有可以victim的了
}
```

```
void ClockReplacer::Pin(frame_id_t frame_id)
{
  if (clock_vec[frame_id].isPin==false) {
    clock_vec[frame_id].isPin = true;
    victim_num--;
  }
}

void ClockReplacer::Unpin(frame_id_t frame_id)
{

  if (clock_vec[frame_id].isPin) {
    clock_vec[frame_id].isPin = false;
    clock_vec[frame_id].recently_visited = true;
    victim_num++;
  }
}

//这里size表示里面还可以替换出来的数量！！
size_t ClockReplacer::Size()
{
  return victim_num;
}
```

在"buffer/lru_replacer.h"中新增

```
class ClockReplacer : public Replacer
{
 public:
  /**
   * Create a new LRUReplacer.
   * @param num_pages the maximum number of pages the LRUReplacer will be
required to store
   */
  explicit ClockReplacer(size_t num_pages);

  /**
   * Destroys the LRUReplacer.
   */
  ~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:
  struct PageState {
    bool recently_visited;
    bool isPin;
```

```
  };
  size_t capacity;    // The clock size
  size_t victim_num;  //number of pages that can be victim (not be pined )
  size_t clock_pointer;  //clock的指针
  std::vector<PageState> clock_vec;//下标为frame_id
};
```

- minisql合作人俞博：负责1、2、3、4报告撰写、测试编写
- 本人负责1、2、5、报告撰写、测试编写
- 贡献一致，而且由于后期才合作，我们都单独完成了1、2

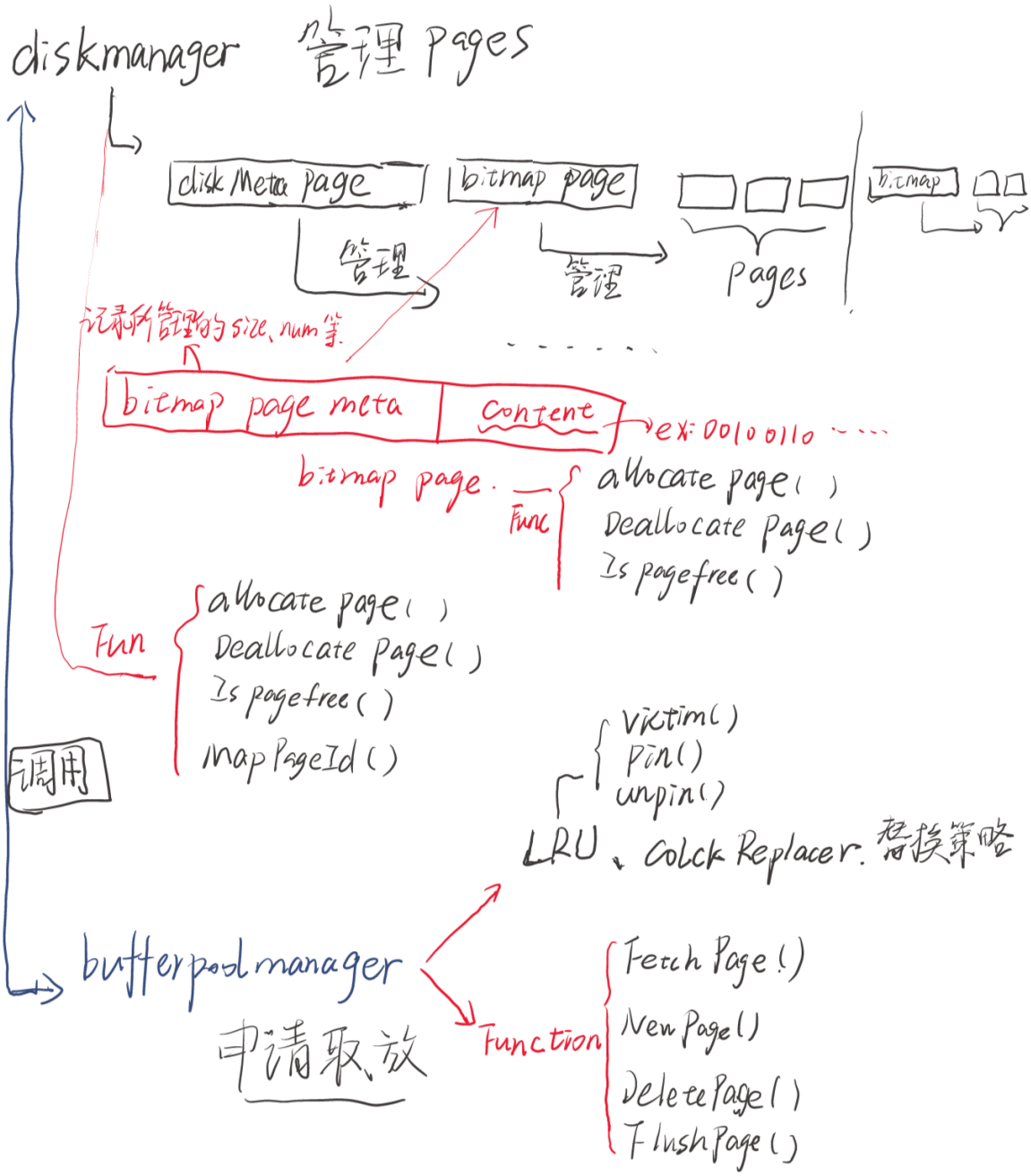
# 二、实验要求

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。

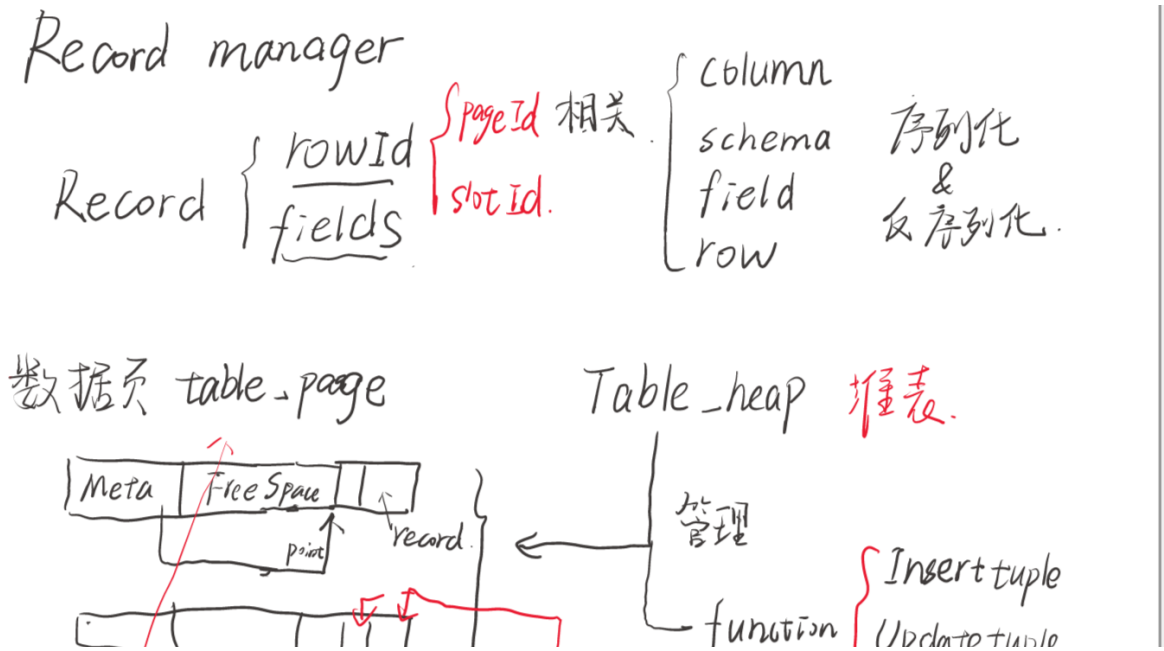
# 三、系统架构

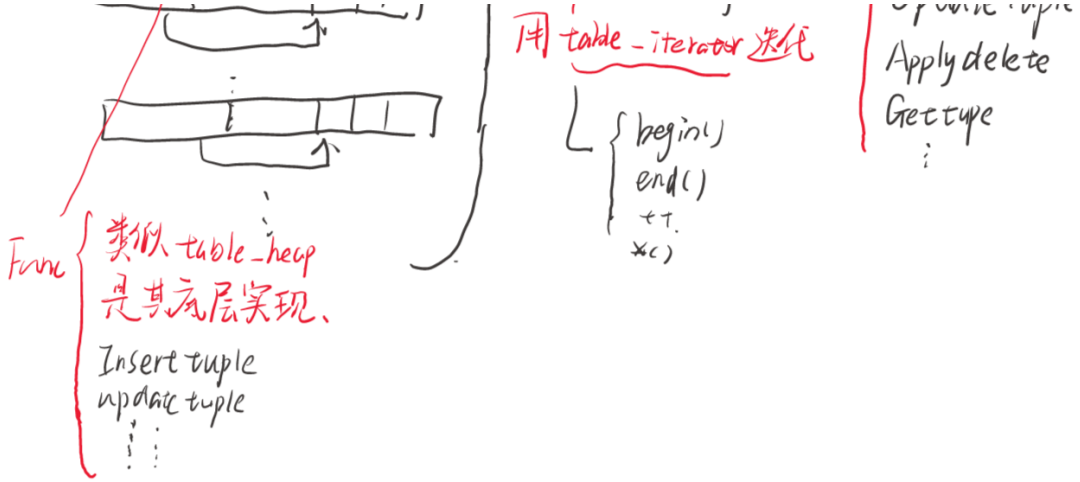
## 3.1 系统各模块调用关系

第一模块：

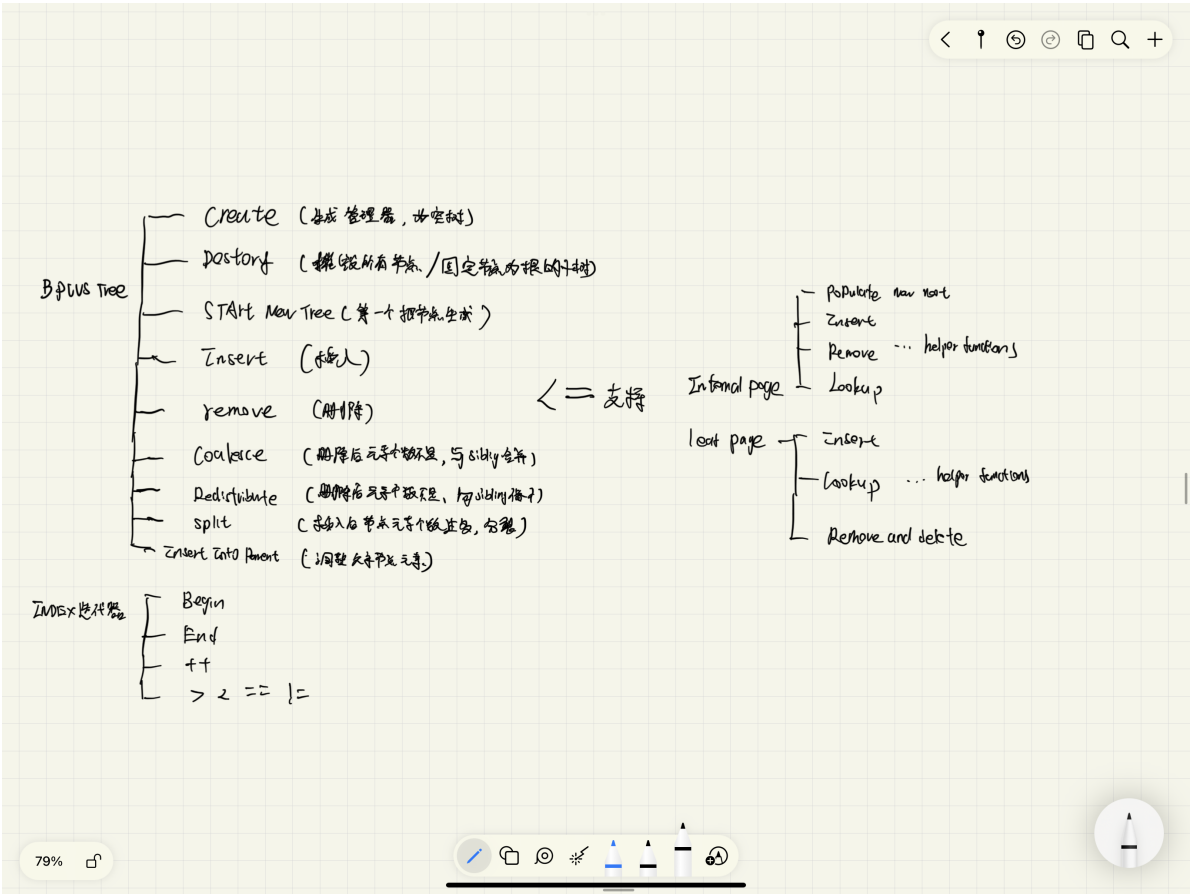

diskmanager 管理 pages
disk Meta page
bitmap page
管理
管理
Pages
bitmap
记录所管理的 size、num 等.
bitmap page meta
content
ex: 0010 0110 ……
bitmap page.
Func
allocate page( )
Deallocate Page( )
Is pagefree( )
Fun
allocate page( )
Deallocate Page( )
Is pagefree( )
MapPageId( )
调用
victim( )
Pin( )
unpin( )
LRU、Colck Replacer. 替换策略
bufferpoolmanager
申请取放
Function
Fetch Page( )
New Page( )
Delete Page( )
Flush Page( )

第二模块：

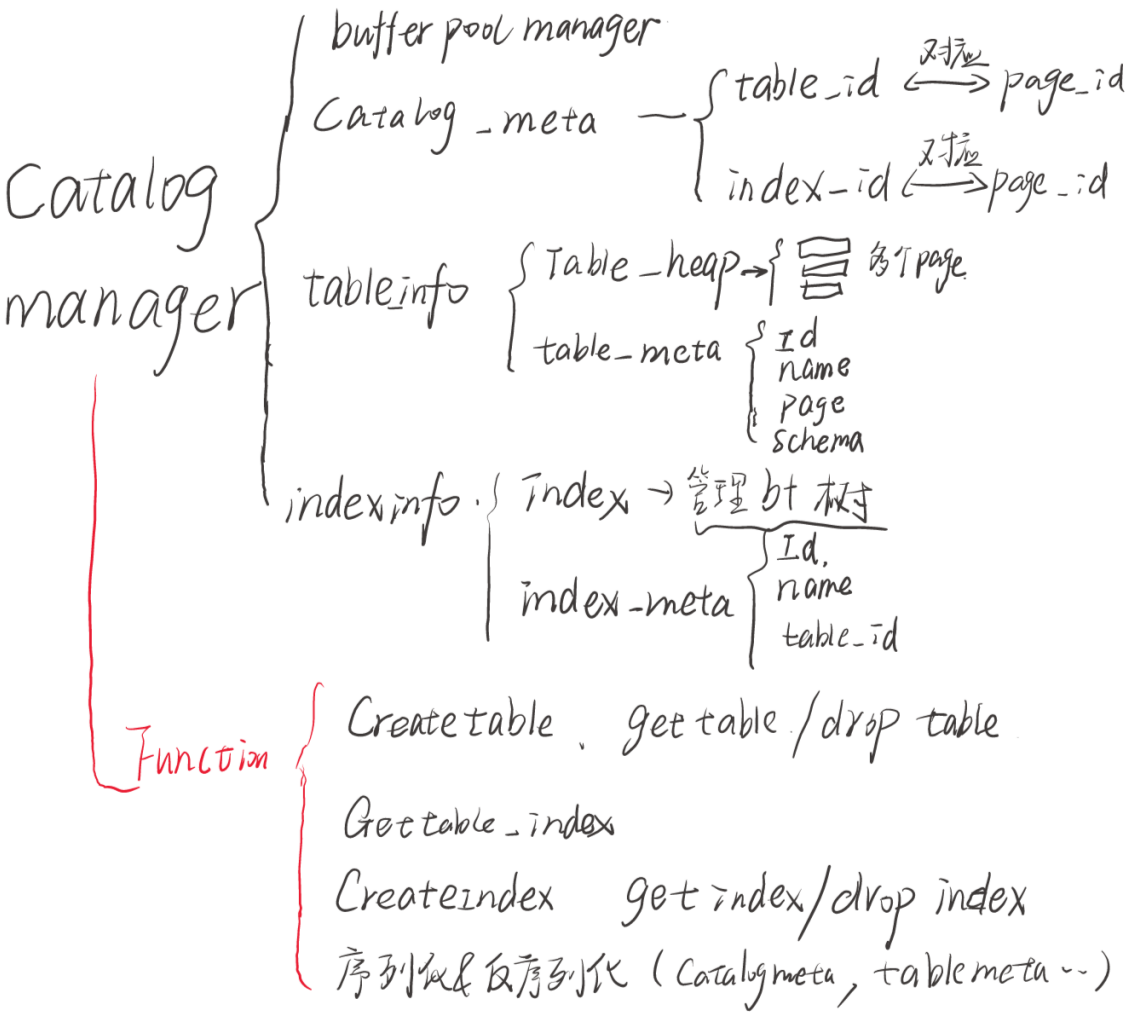

Record manager
Record
rowId
fields
pageId 相关
slotId.
column
schema
field
row
序列化
&
反序列化.
数据页 table_page
Table_heap 堆表.
Meta
Free Space
point
record
管理
function
Insert tuple


用 table_iterator 迭代
begin()
end()
++
*()
Apply delete
GetType
Func
类似 table_heap
是其底层实现。
Insert tuple
update tuple

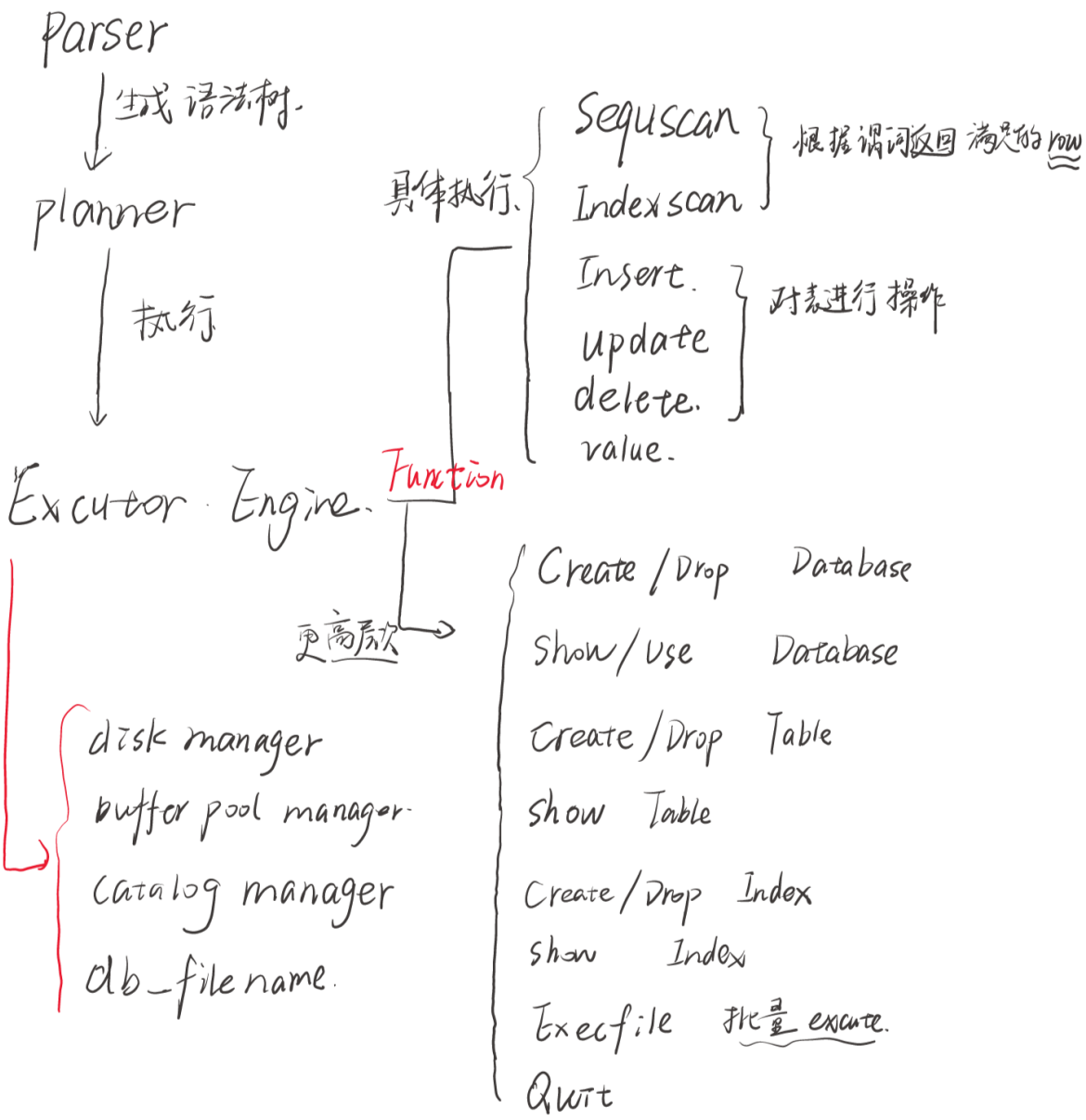
第三模块：


B plus Tree
Create (生成 管理器，为空树)
Destory (删除所有节点，/固定输入为根的子树)
START New Tree (算一个根节点生成)
Insert (插入)
remove (删除)
Coalesce (删除后元素个数不足，与sibly合并)
Redistribute (删除后元素个数不足，与sibling借)
split (插入后节点元素个数溢出，分裂)
Insert into Parent (向父节点元素)
<二 支持
Internal page
Populate new root
Insert
Remove ... helper functions
Lookup
leaf page
Insert
Lookup ... helper functions
Remove and delete
INDEX 迭代器
Begin
End
++
>
<
==
!=
79%

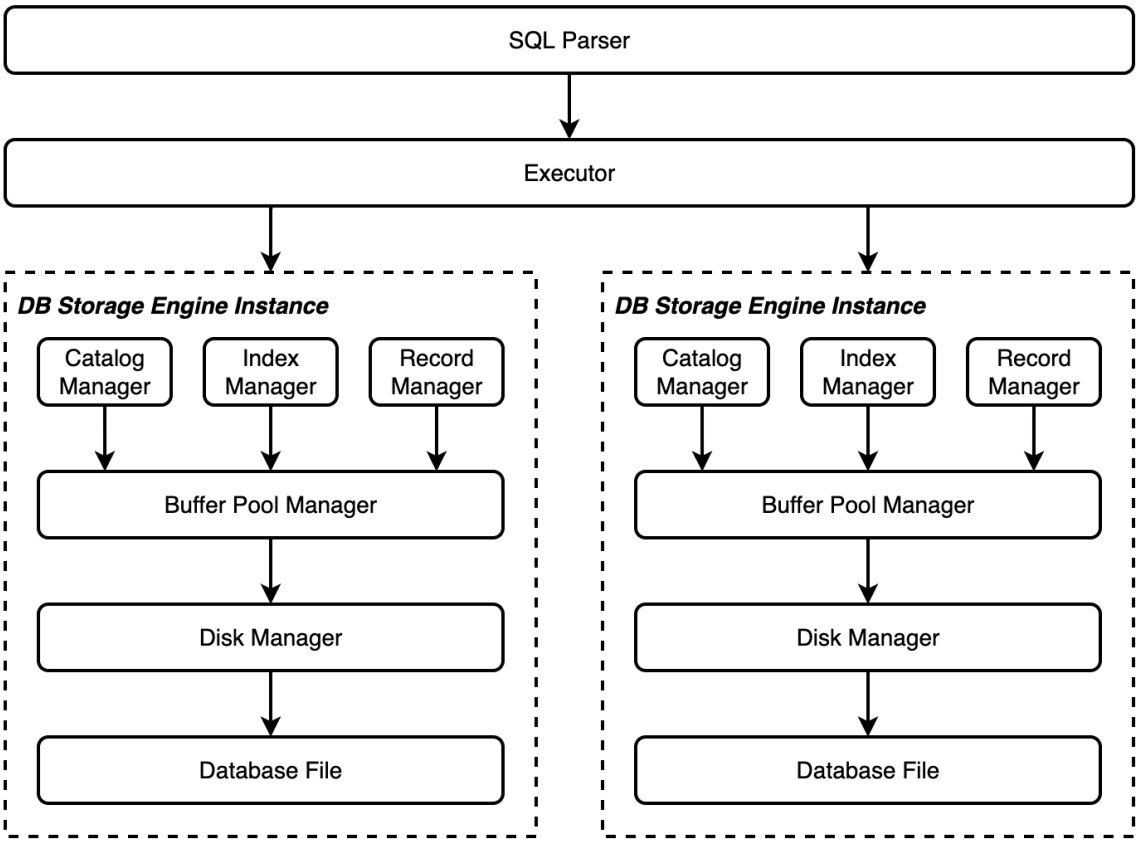
第四模块：


Catalog manager
buffer pool manager
Catalog_meta
table_id
对应
page_id
index_id
对应
page_id
tableinfo
Table_heap
多个page
table_meta
Id
name
page
schema
indexinfo
Index
管理bt树
index_meta
Id.
name
table_id
Function
Createtable
Gettable/droptable
Gettable_index
Createindex
Getindex/dropindex
序列化&反序列化（Catalogmeta，tablemeta…）

第五模块：


Parser
生成语法树
planner
执行
Excutor Engine
具体执行
Sequscan
Indexscan
根据谓词返回满足的row
Insert
update
delete
value
对表进行操作
Function
更高层次
disk manager
buffer pool manager
Catalog manager
db_file name
Create / Drop Database
Show / Use Database
Create / Drop Table
Show Table
Create / Drop Index
Show Index
Execfile 批量 excute
Quit

## 3.2 系统总架构




# 四、各模块具体实现

## 第一模块-Disk AND BUFFER POOL MANAGER

### BITMAP PAGE

#### AllocatePage(&page_offset)

- 分配一个空闲页，并通过 `page_offset` 返回所分配的空闲页位于该段中的下标（从 `0` 开始）；

**实现思路：**

- 如果bitmap已满，返回false
- 否则通过next_free_page寻找到空闲页位置
- 通过位运算更新bitmap中的信息（0代表空闲，1代表非空闲）
- 更新next_free_page和page_allocated

```
/**
 * TODO: Student Implement
 */
// allocate a empty page and return the index of the page if succeeded
template <size_t PageSize>
bool BitmapPage<PageSize>::AllocatePage(uint32_t &page_offset)
{
  // if the bit map is already full
  if (this->next_free_page_ >= MAX_CHARS * sizeof(char) * 8)
  {
    return false;
  }
```

```
  // calculate the page_offset, allocate the page
  page_offset = this->next_free_page_;
  int byte_offset = this->next_free_page_ / (sizeof(char) * 8);
  int bit_offset = this->next_free_page_ % (sizeof(char) * 8);
  unsigned char page_mask = ((unsigned char)128) >> bit_offset;
  bytes[byte_offset] = bytes[byte_offset] | page_mask;
  // update the next_free_page and page_allocated
  uint32_t i;
  for (i = this->next_free_page_ + 1; i < MAX_CHARS * 8; i++)
  {
    if (IsPageFree(i))
      break;
  }
  this->next_free_page_ = i;
  this->page_allocated_++;
  return true;
}
```

## DeallocatePage(page_offset)

- 回收已经被分配的页；

**实现思路**

- 通过page_offset寻找出对应page在bitmap中的状态位
- 如果已经空闲，返回false
- 否则置空，更新next_free_page_ 和 page_allocated_

```
/**
 * TODO: Student Implement
 */
// reset the bit, and return false if the bit is reset already
template <size_t PageSize>
bool BitmapPage<PageSize>::DeAllocatePage(uint32_t page_offset)
{
  // get the state of the page
  int byte_offset = page_offset / (sizeof(char) * 8);
  int bit_offset = page_offset % (sizeof(char) * 8);
  unsigned char page_mask = ((unsigned char)128) >> bit_offset;
  int state = bytes[byte_offset] & page_mask;
  // return false if it's reset
  if (state == 0)
  {
    return false;
  }
  // change to the state
  bytes[byte_offset] = bytes[byte_offset] & (~page_mask);
  // update next_free_page, page_allocated
  this->next_free_page_ = this->next_free_page_ > page_offset ? page_offset :
this->next_free_page_;
  this->page_allocated_--;
  return true;
}
```

### IsPageFree(page_offset) const

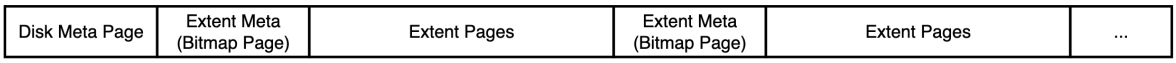
- 判断给定的页是否是空闲（未分配）的。

**实现思路**

- 为上一个函数的一部分
- 位运算来取出该页状态，0返回true，1返回false

```
/**
 * TODO: Student Implement
 */
template <size_t PageSize>
bool BitmapPage<PageSize>::IsPageFree(uint32_t page_offset) const
{
  // extract the state of the page
  int byte_offset = page_offset / (sizeof(char) * 8);
  int bit_offset = page_offset % (sizeof(char) * 8);
  unsigned char page_mask = ((unsigned char)128 >> bit_offset);
  int state = bytes[byte_offset] & page_mask;
  // examine the state
  return state == 0 ? true : false;
}
```

## DISK MANAGER（通过disk meta pag来管理物理存储的数据页）




### AllocatePage()

- 从磁盘中分配一个空闲页，并返回空闲页的**逻辑页号**；

**实现思路**

- 寻找到空闲分区
- 如果空闲分区存在，把该分区的bitmap读出，改变其状态；更新meta page中的数据
- 如果不存在，则需要开辟新分区，并初始化该分区的bitmap
- 返回逻辑页号（去掉bitmap和meta page的页号）

```
/**
 * TODO: Student Implement
 */
page_id_t DiskManager::AllocatePage()
{
  // Find the extent that is not full yet
  size_t PageNum = BitmapPage<PAGE_SIZE>::GetMaxSupportedSize();
  DiskFileMetaPage *MetaPage = reinterpret_cast<DiskFileMetaPage *>(this-
>meta_data_);
  uint32_t extent_id = 0;
  for (extent_id = 0; extent_id < MetaPage->num_extents_; extent_id++)
  {
    if (MetaPage->extent_used_page_[extent_id] != PageNum)
    {
      break;
```

```
    }
  }
  // update the num_allocated_pages
  MetaPage->num_allocated_pages_++;
  uint32_t page_offset;
  // If extent exists, then allocate the page using bitmap api
  if (extent_id < MetaPage->num_extents_)
  {
    char temp[PAGE_SIZE];
    ReadPhysicalPage((PageNum + 1) * extent_id + 1, temp);
    reinterpret_cast<BitmapPage<PAGE_SIZE> *>(temp)->AllocatePage(page_offset);
    MetaPage->extent_used_page_[extent_id]++;
    WritePhysicalPage((PageNum + 1) * extent_id + 1, temp);
  }
  // If not exists, then new a extent
  else
  {
    MetaPage->num_extents_++;
    MetaPage->extent_used_page_[extent_id] = 1;
    BitmapPage<PAGE_SIZE> *newMap = new BitmapPage<PAGE_SIZE>();
    newMap->AllocatePage(page_offset);
    WritePhysicalPage((PageNum + 1) * extent_id + 1, reinterpret_cast<char *>
(newMap));
  }
  // We need return the page_id that is transparent to user(not the real
physical id)
  return extent_id * PageNum + page_offset;
}
```

## DeallocatePage(logical_page_id)

- 释放磁盘中**逻辑页号**对应的物理页。

**实现思路**

- 把page_id对应的分区的bitmap提出
- 判断是否可deallocate，不行返回false
- 修改meta page与bitmap，把bitmap存回磁盘

```
/**
 * TODO: Student Implement
 */
void DiskManager::DeAllocatePage(page_id_t logical_page_id)
{
  // extract the page out
  size_t PageNum = BitmapPage<PAGE_SIZE>::GetMaxSupportedSize();
  uint32_t extent_id = logical_page_id / PageNum;
  uint32_t page_offset = logical_page_id % PageNum;
  char temp[PAGE_SIZE];
  ReadPhysicalPage((PageNum + 1) * extent_id + 1, temp);
  // deallocate it, return if fails
  bool result = true;
  result = reinterpret_cast<BitmapPage<PAGE_SIZE> *>(temp)-
>DeAllocatePage(page_offset);
  if (result == false)
```

```
  {
    return;
  }
  // update the num_allocated_pages_ and extent_used_page_
  WritePhysicalPage((PageNum + 1) * extent_id + 1, temp);
  DiskFileMetaPage *MetaPage = reinterpret_cast<DiskFileMetaPage *>(this-
>meta_data_);
  MetaPage->num_allocated_pages_--;
  MetaPage->extent_used_page_[extent_id]--;
  return;
}
```

## IsPageFree(logical_page_id)

- 判断该**逻辑页号**对应的数据页是否空闲。

**实现思路**

- 和bitmap的差不多

```
/**
 * TODO: Student Implement
 */
bool DiskManager::IsPageFree(page_id_t logical_page_id)
{
  // extract the page out
  size_t PageNum = BitmapPage<PAGE_SIZE>::GetMaxSupportedSize();
  uint32_t extent_id = logical_page_id / PageNum;
  uint32_t page_offset = logical_page_id % PageNum;
  char temp[PAGE_SIZE];
  ReadPhysicalPage((PageNum + 1) * extent_id + 1, temp);
  // determine whether the page is free or not
  return reinterpret_cast<BitmapPage<PAGE_SIZE> *>(temp)-
>IsPageFree(page_offset);
}
```

## MapPageId(logical_page_id)

- 可根据需要实现。在`DiskManager`类的私有成员中，该函数可以用于将逻辑页号转换成物理页号。

**实现思路**

- 这块的逻辑在之前的函数中也实现了

```
/**
 * TODO: Student Implement
 */
page_id_t DiskManager::MapPageId(page_id_t logical_page_id)
{
  size_t PageNum = BitmapPage<PAGE_SIZE>::GetMaxSupportedSize();
  return (PageNum + 1) * logical_page_id / PageNum + 1 + logical_page_id %
PageNum;
}
```

# LRU

- LRU这个模块是吧pageid和buffer中的frameid动态绑定起来，方便buffer_pool_manager迅速通过pageid来获取到buffer中的page

## Victim(*frame_id)

- 替换（即删除）与所有被跟踪的页相比最近最少被访问的页，将其页帧号（即数据页在Buffer Pool的Page数组中的下标）存储在输出参数 `frame_id` 中输出并返回 `true`，如果当前没有可以替换的元素则返回 `false`；

**实现思路**

- 先判断一下buffer中还有没有可用的frame，假如没有，则意味着buffer刚初始化，还未导入任何物理页，则不允许调用victim
- 否则把frame_id从list表和hash表中删除

```
/**
 * TODO: Student Implement
 */
// the least recently visited id should be returned by pointe frame_id
bool LRUReplacer::Victim(frame_id_t *frame_id)
{
  // first we need to judge whther there are enough frmae_ids to be deleted
  if ((this->lru_list_).size() == 0)
  {
    return false;
  }
  // then delete the frame_id from lru_list_ and lru_hash_
  *frame_id = (this->lru_list_).back();
  (this->lru_hash_).erase(*frame_id);
  (this->lru_list_).pop_back();
  return true;
}
```

## Pin(frame_id)

- 把该frame_id钉住，不让其他进程获取该frame_id，即从list和hash表中删去

**实现思路**

```
/**
 * TODO: Student Implement
 */
void LRUReplacer::Pin(frame_id_t frame_id)
{
  // delete from the list and the hash table, so that no other threads can pin
or delete it
  if (this->lru_hash_.count(frame_id) != 0)
  {
    auto it = (this->lru_hash_)[frame_id];
    (this->lru_list_).erase(it);
    (this->lru_hash_).erase(frame_id);
  }
}
```

### Unpin(frame_id)

- 把frame_id加回list和hash中，其他进程也可访问

**实现思路**

```
/**
 * TODO: Student Implement
 */
void LRUReplacer::Unpin(frame_id_t frame_id)
{
  // determine whther the capacity is full and whether the frame is already in
the buffer pool
  if ((this->lru_hash_).count(frame_id) != 0 || (this->lru_list_).size() ==
this->capacity)
  {
    return;
  }
  // insert the new page in the front of the list and map it into the hash table
  (this->lru_list_).push_front(frame_id);
  auto it = (this->lru_list_).begin();
  (this->lru_hash_).emplace(frame_id, it);
  return;
}
```

## BUFFER POOL MANAGER

### FetchPage(page_id)

- 根据逻辑页号获取对应的数据页，如果该数据页不在内存中，则需要从磁盘中进行读取；

**实现思路**

- 找现在buffer中是否有page_id，如果有把他取出，并调用lru
- 如果没有，需要从磁盘中获取。先尝试加载到空余的buffer中
- 如果buffer都满，则通过lru记载道最近最少的那个buffer中
- 如果buffer里原来的page是脏的，写入磁盘，否则不做处理

```
/**
```

```
 * TODO: Student Implement
 */
// Here稍微理一下，diskmanager有个metadata，后面跟着的是bitmap+数据页，这层是物理磁盘的
// 然后往上，有Bufferpool。bufferpool里的pages是缓存的物理数据，通过page_table完成pageid
和frameid（也就是bufferpages里的id）完成映射
// freelist里存的应该就是pages中还能用的frameid，只有满了才有必要更新缓存区
// 通过lru的管理，可以把最老的frameid绑成新的物理数据页。当然，lru本身只是告诉我们哪个frame是
最老的，实际替换是buffermanager做的。
Page *BufferPoolManager::FetchPage(page_id_t page_id)
{
  // 1.     Search the page table for the requested page (P).
  // 1.1    If P exists, pin it and return it immediately.
  frame_id_t frame_id;
  if ((this->page_table_).count(page_id) != 0)
  {
    frame_id = (this->page_table_)[page_id];
    // Pin the frame so that the unpin will make the frame exist at the front
    replacer_->Pin(frame_id);
    (this->pages_)[frame_id].pin_count_++;
    return this->pages_ + frame_id;
  }
  // 1.2    If P does not exist, find a replacement page (R) from either the
free list or the replacer.
  //        Note that pages are always found from the free list first.
  if ((this->free_list_).size() != 0)
  {
    frame_id = (this->free_list_).front();
    (this->free_list_).pop_front();
    (this->page_table_)[page_id] = frame_id;
    this->disk_manager_->ReadPage(page_id, (this->pages_)[frame_id].data_);
    // update page info
    (this->pages_)[frame_id].pin_count_ = 1;
    (this->pages_)[frame_id].page_id_ = page_id;
    (this->replacer_)->Pin(frame_id);
    return this->pages_ + frame_id;
  }
  else
  {
    // find from replacer
    if (!(this->replacer_)->Victim(&frame_id))
      return nullptr;
    (this->replacer_)->Pin(frame_id);
    // 2.     If R is dirty, write it back to the disk.
    if ((this->pages_)[frame_id].IsDirty())
    {
      (this->disk_manager_)->WritePage((this->pages_)[frame_id].GetPageId(),
(this->pages_)[frame_id].GetData());
      // 3.     Delete R from the page table and insert P.
      (this->page_table_).erase((this->pages_)[frame_id].GetPageId());
      (this->page_table_)[page_id] = frame_id;
      (this->disk_manager_)->ReadPage(page_id, (this->pages_)[frame_id].data_);
      // 4.     Update P's metadata, read in the page content from disk, and
then return a pointer to P.
      (this->pages_)[frame_id].pin_count_ = 1;
      (this->pages_)[frame_id].page_id_ = page_id;
```

```
      return this->pages_ + frame_id;
    }
  }
}
```

## NewPage(&page_id)

- 分配一个新的数据页，并将逻辑页号于`page_id`中返回；

**实现思路**

- 如果buffer还有空余，则用空余buffer接受新分配的数据页。否则用lru接收
- lru接受需要把之前的脏页面存储入磁盘，并且pin现在的buffer
- 如果所有的buffer都pin了，则返回空指针
- 更新meta data

```
**
 * TODO: Student Implement
 */
Page *BufferPoolManager::NewPage(page_id_t &page_id)
{
  // 0.   Make sure you call AllocatePage!
  // 1.   If all the pages in the buffer pool are pinned, return nullptr.
  frame_id_t frame_id;
  // 2.   Pick a victim page P from either the free list or the replacer. Always
pick from the free list first.
  if ((this->free_list_).size() > 0)
  {
    frame_id = (this->free_list_).front();
    (this->free_list_).pop_front();
    page_id = this->disk_manager_->AllocatePage();
  }
  else
  {
    if (!(this->replacer_)->Victim(&frame_id))
    {
      return nullptr;
    }
    // (this->replacer_)->Pin(frame_id);
    page_id = this->disk_manager_->AllocatePage();
    (this->page_table_).erase((this->pages_)[frame_id].page_id_);
    if ((this->pages_)[frame_id].IsDirty())
    {
      (this->pages_)[frame_id].is_dirty_ = false;
      this->disk_manager_->WritePage((this->pages_)[frame_id].page_id_, (this-
>pages_)[frame_id].data_);
    }
  }
  (this->replacer_)->Pin(frame_id);
  // 3.   Update P's metadata, zero out memory and add P to the page table.
  (this->pages_)[frame_id].pin_count_ = 1;
  (this->pages_)[frame_id].page_id_ = page_id;
  // 4.   Set the page ID output parameter. Return a pointer to P.
  (this->page_table_)[page_id] = frame_id;
  return this->pages_ + frame_id;
```

```
}
```

## UnpinPage(page_id, is_dirty)

- 取消固定一个数据页；

**实现思路**

- 把该数据页设为脏页，并且递减pin_count
- lru的frame_id同时unpin

```
/**
 * TODO: Student Implement
 */
bool BufferPoolManager::UnpinPage(page_id_t page_id, bool is_dirty)
{
  if ((this->page_table_).count(page_id) == 0)
    return false;
  frame_id_t frame_id = (this->page_table_)[page_id];
  (this->pages_)[frame_id].pin_count_--;
  (this->pages_)[frame_id].is_dirty_ = is_dirty;
  (this->replacer_)->Unpin(frame_id);
  return true;
}
```

## FlushPage(page_id)

- 将数据页转储到磁盘中；

**实现思路**

- 存储的时候锁住
- 调用writepage把它存进磁盘

```
/**
 * TODO: Student Implement
 */
bool BufferPoolManager::FlushPage(page_id_t page_id)
{
  this->latch_.lock();
  if ((this->page_table_).count(page_id) == 0)
    return false;
  frame_id_t frame_id = (this->page_table_)[page_id];
  (this->disk_manager_)->WritePage(page_id, (this->pages_)[frame_id].data_);
  this->latch_.unlock();
  return true;
}
```

## DeletePage(page_id)

- 释放一个数据页；

**实现思路**

- 如果page不存在，返回true
- 如果page存在，且目前被pin，返回false
- 没被pin，则reset buffer，让磁盘中的数据页也被释放，更新meta data

```
**
 * TODO: Student Implement
 */
bool BufferPoolManager::DeletePage(page_id_t page_id)
{
  // 0.   Make sure you call DeallocatePage!
  // 1.   Search the page table for the requested page (P).
  // 1.   If P does not exist, return true.
  // 2.   If P exists, but has a non-zero pin-count, return false. Someone is
using the page.
  // 3.   Otherwise, P can be deleted. Remove P from the page table, reset its
metadata and return it to the free list.
  size_t frame_id;
  for (frame_id = 0; frame_id < pool_size_; frame_id++)
  {
    if ((this->pages_)[frame_id].page_id_ == page_id)
    {
      break;
    }
  }
  if (frame_id < pool_size_)
  {
    if ((this->pages_)[frame_id].pin_count_ != 0)
    {
      return false;
    }
  }
  else
  {
    return true;
  }
  this->DeallocatePage(page_id);
  (this->pages_)[frame_id].page_id_ = INVALID_PAGE_ID;
  (this->page_table_).erase(page_id);
  (this->pages_)[frame_id].ResetMemory();
  (this->free_list_).push_back(frame_id);
  return true;
}
```

# 第二模块-Record Manager

## Serialize

## Row::SerializeTo(*buf, *schema)

**实现思路**

- 没写null bitmap，过不了测试
- 每个fields_进行SerializeTo
- 通过指针相减返回大小

```
/**
 * TODO: Student Implement
 */
uint32_t Row::SerializeTo(char *buf, Schema *schema) const
{
  ASSERT(schema != nullptr, "Invalid schema before serialize.");
  ASSERT(schema->GetColumnCount() == fields_.size(), "Fields size do not match
schema's column size.");
  // replace with your code here
  // First to change all the pointers in the fields to the corresponding type
  char *buf_for_each = buf;
  // uint32_t head_size = ((this->fields_).size() / 8 + 1) + sizeof(int);
  // *reinterpret_cast<int *>(buf_for_each) = head_size;
  // buf_for_each += sizeof(int);
  // char *null_bit_map = buf_for_each;
  for (int i = 0; i < (this->fields_).size(); i++)
  {
    buf_for_each += (this->fields_)[i]->SerializeTo(buf_for_each);
    // if ((this->fields_)[i]->IsNull())
    // {
    //   // 0 is not null and 1 is null
    //   null_bit_map[i / 8] = null_bit_map[i / 8] | ((unsigned char)128 >> (i %
8));
    // }
  }
  // Then calculate the size of Row
  return buf_for_each - buf;
}
```

## Row::DeserializeFrom(*buf, *schema)

**实现思路**

- DeserializeFrom+vector push_back

```
uint32_t Row::DeserializeFrom(char *buf, Schema *schema)
{
  ASSERT(schema != nullptr, "Invalid schema before serialize.");
  ASSERT(fields_.empty(), "Non empty field in row.");
  // replace with your code here
  uint32_t len = schema->GetColumnCount();
  // char *head = buf;
  // uint32_t head_size = *reinterpret_cast<int *>(head);
  // char *buf_for_each = buf + head_size;
  // char *null_bit_map = buf + sizeof(int);
  char *buf_for_each = buf;
  for (int i = 0; i < (int)len; i++)
```

```
  {
    // read each field from buf and store into a field
    Field *f;
    // 1 is null; 0 is not null
    buf_for_each += f->DeserializeFrom(buf_for_each, schema->GetColumn(i)->GetType(), &f, false);
    // if (null_bit_map[i / 8] & ((unsigned char)128 >> (i % 8)))
    // {
    //   f->SetIsNull(true);
    // }
    (this->fields_).push_back(f);
  }
  return buf_for_each - buf;
}
```

## Row::GetSerializedSize(*schema)

**实现思路**

- 指针相减返回size

```
uint32_t Row::GetSerializedSize(Schema *schema) const
{
  ASSERT(schema != nullptr, "Invalid schema before serialize.");
  ASSERT(schema->GetColumnCount() == fields_.size(), "Fields size do not match schema's column size.");
  // replace with your code here
  uint32_t size = 0;
  // uint32_t head_size = ((this->fields_).size() / 8 + 1) + sizeof(int);
  for (int i = 0; i < (int)schema->GetColumnCount(); i++)
  {
    size += (this->fields_)[i]->GetSerializedSize();
  }
  // return size + head_size;
  return size;
}
```

## Column::SerializeTo(*buf)

**实现思路**

```
/**
 * TODO: Student Implement
 */
uint32_t Column::SerializeTo(char *buf) const
{
  // replace with your code here
  char *pos = buf;
  // name is a varchar, so we have to use a length
  uint32_t name_len = (this->name_).size();
  memcpy(pos, &name_len, sizeof(uint32_t));
  pos += sizeof(uint32_t);
  (this->name_).copy(pos, name_len, 0);
  pos += name_len;
  memcpy(pos, &(this->type_), sizeof(TypeId));
```

```
  pos += sizeof(TypeId);
  memcpy(pos, &(this->len_), sizeof(uint32_t)); // len_, for varrchar
  pos += sizeof(uint32_t);
  memcpy(pos, &(this->table_ind_), sizeof(uint32_t)); // table_ind_, index
  pos += sizeof(uint32_t);
  memcpy(pos, &(this->nullable_), sizeof(bool));
  pos += sizeof(bool);
  memcpy(pos, &(this->unique_), sizeof(bool));
  pos += sizeof(bool);
  return pos - buf;
  return 0;
}
```

## Column::DeserializeFrom(*buf, *&column)

### 实现思路

```
/**
 * TODO: Student Implement
 */
uint32_t Column::DeserializeFrom(char *buf, Column *&column)
{
  // replace with your code here
  char *pos = buf;
  uint32_t name_len = MACH_READ_FROM(uint32_t, pos);
  pos += sizeof(uint32_t);
  std::string name;
  name.append(pos, name_len);
  TypeId type = MACH_READ_FROM(TypeId, pos);
  pos += sizeof(TypeId);
  uint32_t len = MACH_READ_FROM(uint32_t, pos);
  pos += sizeof(uint32_t);
  uint32_t table_id = MACH_READ_FROM(uint32_t, pos);
  pos += sizeof(uint32_t);
  bool null_able = MACH_READ_FROM(bool, pos);
  pos += sizeof(bool);
  bool unique = MACH_READ_FROM(bool, pos);
  pos += sizeof(bool);
  column = new Column(name, type, len, table_id, null_able, unique);
  return pos - buf;
}
```

## Column::GetSerializedSize()

### 实现思路

```
/**
 * TODO: Student Implement
 */
uint32_t Column::GetSerializedSize() const
{
  // replace with your code here
  return (uint32_t)(sizeof(uint32_t) * 3 + sizeof(TypeId) + 2 * sizeof(bool));
}
```

## Schema::SerializeTo(*buf)

### 实现思路

```
/**
 * TODO: Student Implement
 */
uint32_t Schema::SerializeTo(char *buf) const
{
  // replace with your code here
  char *buf_for_each = buf;
  for (int i = 0; i < int((this->columns_).size()); i++)
  {
    buf_for_each += (this->columns_)[i]->SerializeTo(buf_for_each);
  }
  return buf_for_each - buf;
}
```

## Schema::DeserializeFrom(*buf, *&schema)

### 实现思路

```
uint32_t Schema::DeserializeFrom(char *buf, Schema *&schema)
{
  // replace with your code here
  char *buf_for_each = buf;
  std::vector<Column *> columns;
  Column *column;
  for (int i = 0; i < schema->GetColumnCount(); i++)
  {
    buf_for_each += Column::DeserializeFrom(buf_for_each, column);
    columns.push_back(column);
  }
  schema = new Schema(columns, true);
  return buf_for_each - buf;
}
```

## Schema::GetSerializedSize()

### 实现思路

```
uint32_t Schema::GetSerializedSize() const
{
  // replace with your code here
  uint32_t size = 0;
  for (int i = 0; i < int((this->columns_).size()); i++)
  {
    size += (this->columns_)[i]->GetSerializedSize();
  }
  return size;
}
```

# Table Heap

## TableHeap::InsertTuple(&row, *txn)

- 向堆表中插入一条记录，插入记录后生成的`RowId`需要通过`row`对象返回（即`row.rid_`）；

**实现思路**

- 先检查row的大小是否大于一个page的最大大小，如果过大，返回false
- 通过buffer_pool_manager_ 来拿取 first_page_id_的page
- 不断检查双向链表中的page是否可以成功插入tuple，如果成功插入，UnpinPage，并设置为脏页
- 如果双向链表中的所有page都已满，需要new一个page来存储tuple，并把page加入双向链表中

```
/**
 * TODO: Student Implement
 * buffer_pool_manager掌管从内存与磁盘中的读取数据
 * 所有的被用过的数据页通过链表连在一起进行管理
 */
bool TableHeap::InsertTuple(Row &row, Transaction *txn)
{
  // first to examine whther the row is suitable to store in one page
  uint32_t row_size = row.GetSerializedSize(this->schema_);
  if (row_size > TablePage::SIZE_MAX_ROW)
  {
    return false;
  }
  // then to examine whether the pages in the dual-side list can hold the row
  TablePage *page_to_check = reinterpret_cast<TablePage *>(this-
>buffer_pool_manager_->FetchPage(this->first_page_id_));
  this->buffer_pool_manager_->UnpinPage(page_to_check->GetPageId(), false);
  while (1)
  {
    if (page_to_check->InsertTuple(row, schema_, txn, lock_manager_,
log_manager_))
    {
      this->buffer_pool_manager_->UnpinPage(page_to_check->GetPageId(), true);
// The page is dirty
      return true;
    }
    page_id_t NextPageId = page_to_check->GetNextPageId();
    if (NextPageId == INVALID_PAGE_ID) // all the page in the dual-side list
have been checked
    {
      break;
    }
    page_to_check = reinterpret_cast<TablePage *>(this->buffer_pool_manager_-
>FetchPage(NextPageId));
    this->buffer_pool_manager_->UnpinPage(page_to_check->GetPageId(), false);
  }
  // Because all the pages are not available, we have to new a page
  int new_page_id = INVALID_PAGE_ID;
  // the last page in the list haven't been pined
  this->buffer_pool_manager_->UnpinPage(page_to_check->GetPageId(), true);
  TablePage *New_Page = reinterpret_cast<TablePage *>(this-
>buffer_pool_manager_->NewPage(new_page_id));
```

```
  // this->buffer_pool_manager_->UnpinPage(New_Page->GetPageId(), false);
  if (!New_Page)
    return false;
  // connect to the dual-side list
  New_Page->Init(new_page_id, page_to_check->GetPageId(), this->log_manager_,
txn);
  New_Page->InsertTuple(row, this->schema_, txn, this->lock_manager_, this-
>log_manager_);
  buffer_pool_manager_->UnpinPage(New_Page->GetPageId(), true); // this is dirty
  page_to_check->SetNextPageId(new_page_id);
  buffer_pool_manager_->UnpinPage(page_to_check->GetPageId(), true); // this is
dirty
  return true;
}
```

## TableHeap::UpdateTuple(&row, &rid, *txn)

- 将 `RowId` 为 `rid` 的记录 `old_row` 替换成新的记录 `new_row`，并将 `new_row` 的 `RowId` 通过 `new_row.rid_` 返回;

**实现思路**

- 先寻找page_id的page，如果没找到，返回false
- 检查是否可以直接插入，插入成功unpin后返回true
- 插入失败，检查page剩余的空间加上当前元组的大小是否能容下更新后的元组
- 如果不能容下， 删除当前元组，将更新后元组insert
- 如果能容下，则代表slotnumber无效或者tuple已经被删除，返回false

```
/**
 * TODO: Student Implement
 */
bool TableHeap::UpdateTuple(const Row &row, const RowId &rid, Transaction *txn)
{
  // Find the page which contains the tuple.
  auto page = reinterpret_cast<TablePage *>(buffer_pool_manager_-
>FetchPage(rid.GetPageId()));
  // If the page could not be found, then abort the transaction.
  if (page == nullptr)
  {
    return false;
  }
  // Otherwise, mark the tuple as deleted.
  page->WLatch();
  Row *old_row = new Row(rid);
  if (!page->UpdateTuple(row, old_row, this->schema_, txn, lock_manager_,
log_manager_))
  {
    if (page->GetFreeSpaceRemaining() + page->GetTupleSize(old_row-
>GetRowId().GetSlotNum()) < row.GetSerializedSize(this->schema_))
    {
      // If there is not enough space to update, we need to update via delete
followed by an insert (not enough space).
      this->MarkDelete(rid, txn);
      this->ApplyDelete(rid, txn);
      Row _row = row;
```

```
      bool result = this->InsertTuple(_row, txn);
      ASSERT(result, "Insert in update failed");
      this->buffer_pool_manager_->UnpinPage(page->GetTablePageId(), result);
      page->WUnlatch();
      return result;
    }
    else
    {
      // if the slotnumber is invalid or the tuple is deleted
      this->buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
      page->WUnlatch();
      return false;
    }
  }
  page->WUnlatch();
  buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
  return true;
}
```

## TableHeap::ApplyDelete(&rid, *txn)

- 从物理意义上删除这条记录;

**实现思路**

- 找到page后调用page的方法删除

```
/**
 * TODO: Student Implement
 */
void TableHeap::ApplyDelete(const RowId &rid, Transaction *txn)
{
  // Step1: Find the page which contains the tuple.
  auto page = reinterpret_cast<TablePage *>(buffer_pool_manager_-
>FetchPage(rid.GetPageId()));
  assert(page != nullptr);
  // Step2: Delete the tuple from the page.
  page->ApplyDelete(rid, txn, this->log_manager_);
  this->buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
}
```

## TableHeap::GetTuple(*row, *txn)

- `获取`RowId`为`row->rid_`的记录;

**实现思路**

- 找到page后调用page的方法获取元组

```
/**
 * TODO: Student Implement
 */
bool TableHeap::GetTuple(Row *row, Transaction *txn)
{
```

```
  auto page = reinterpret_cast<TablePage *>(buffer_pool_manager_->FetchPage(row->GetRowId().GetPageId()));
  if (page == nullptr)
  {
    return false;
  }
  bool result = page->GetTuple(row, this->schema_, txn, this->lock_manager_);
  this->buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
  return result;
}
```

## TableHeap::Begin()

- 获取堆表的首迭代器；

**实现思路**

- 从第一个page开始寻找，找到能成功返回tupleRid的就生成TableIterator返回

```
/**
 * TODO: Student Implement
 */
TableIterator TableHeap::Begin(Transaction *txn)
{
  auto page = reinterpret_cast<TablePage *>(this->buffer_pool_manager_->FetchPage(this->first_page_id_));
  RowId rid;
  // in case rows have been deleted
  while (page->GetTablePageId() != INVALID_PAGE_ID)
  {
    this->buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
    if (page->GetFirstTupleRid(&rid))
    {
      break;
    }
    page = reinterpret_cast<TablePage *>(this->buffer_pool_manager_->FetchPage(page->GetNextPageId()));
  }
  return TableIterator(rid, this);
}
```

## TableHeap::End()

- `TableHeap::End()`：获取堆表的尾迭代器；

**实现思路**

```
/**
 * TODO: Student Implement
 */
TableIterator TableHeap::End()
{
  RowId rid = RowId(INVALID_ROWID);
  return TableIterator(rid, this);
}
```

## TableIterator::operator++()

-

**实现思路**

- 比较合理的是，tableiterator有tableheap和当前指向的row两个成员
- tableheap方便其继续移动，row指向当前的元组
- 前置++需要判断这页是否已经到底
- 如果到底需要在后续页的记录中进行滑动

```
// ++iter
TableIterator &TableIterator::operator++()
{
  // first to fetch the page that contains the row
  auto page = reinterpret_cast<TablePage *>((this->table_heap_-
>buffer_pool_manager_)->FetchPage((this->row_->GetRowId()).GetPageId()));
  // examine whether the row is the last row in the page
  RowId NextId = INVALID_ROWID;
  if (!page->GetNextTupleRid(this->row_->GetRowId(), &NextId))
  {
    // the row is the last row in the page, we have to find new row in other
page
    // In case the page in the list have been deleted
    while (page->GetNextPageId() != INVALID_PAGE_ID)
    {
      (this->table_heap_->buffer_pool_manager_)->UnpinPage(page->GetPageId(),
false);
      page = reinterpret_cast<TablePage *>((this->table_heap_-
>buffer_pool_manager_)->FetchPage(page->GetNextPageId()));
      if (page->GetNextPageId() != INVALID_PAGE_ID)
        break;
    }
  }
  // delete the row_ and make it point to the new row
  delete this->row_;
  this->row_ = new Row(NextId);
  this->table_heap_->buffer_pool_manager_->UnpinPage(NextId.GetPageId(), false);
  return *this;
}
```

## TableIterator::operator++(int)

**实现思路**

- 调用前置++，保存初始状态返回

```
// iter++
TableIterator TableIterator::operator++(int)
{
  TableIterator old(*this);
  ++(*(this));
  return TableIterator(old);
}
```

# 第三模块-INDEX MANAGER

## BPLUSTREELEAFPAGE

### Init(page_id, parent_id, key_size, max_size)

- 初始化设置该叶子页

**实现思路：**

- 根据传入的参数，设置相应的信息
- 如果max_size为0，设置为LEAF_PAGE_SIZE

```
/**
 * TODO: Student Implement
 */
/**
 * Init method after creating a new leaf page
 * Including set page type, set current size to zero, set page id/parent id, set
 * next page id and set max size
 * 未初始化next_page_id
 */
void LeafPage::Init(page_id_t page_id, page_id_t parent_id, int key_size, int
max_size)
{
  this->SetPageType(IndexPageType::LEAF_PAGE);
  this->SetSize(0);
  this->SetPageId(page_id);
  this->SetParentPageId(parent_id);
  this->SetMaxSize(max_size);
  this->SetNextPageId(INVALID_PAGE_ID);
  this->SetKeySize(key_size);
  this->next_page_id_ = INVALID_PAGE_ID;
  if (max_size == 0)
  {
    int size = LEAF_PAGE_SIZE;
    this->SetMaxSize(size);
  }
}
```

### KeyIndex(key, KM)

- 找到节点中第一个大于等于key的位置

**实现思路：**

- 二分搜索法，如果能找到，即返回，否则返回当前size

```
/**
 * TODO: Student Implement
 */
/**
 * Helper method to find the first index i so that pairs_[i].first >= key
 * NOTE: This method is only used when generating index iterator
 */
```

```
int LeafPage::KeyIndex(const GenericKey *key, const KeyManager &KM)
{
  int right = this->GetSize() - 1, center, left = 0;
  while (left <= right)
  {
    center = (left + right) / 2;
    if (KM.CompareKeys(key, this->KeyAt(center)) > 0)
    {
      if (center == this->GetSize() - 1 || KM.CompareKeys(key, this-
>KeyAt(center + 1)) <= 0)
      {
        return center + 1;
      }
      left = center + 1;
    }
    else
    {
      if (center == 0 || (KM.CompareKeys(key, this->KeyAt(center - 1)) > 0))
      {
        return center;
      }
      right = center - 1;
    }
  }
  return this->GetSize();
}
```

## Insert(key, value, KM)

- 把传入的key-value对插入合适的位置，返回插入后的size

**实现思路:**

- 通过调用KeyIndex方法，来寻找到合适的位置，执行插入
- 如果已经在B+树中，不执行操作

```
/*****************************************************************************
 * INSERTION
 *****************************************************************************/
/*
 * Insert key & value pair into leaf page ordered by key
 * @return page size after insertion
 */
int LeafPage::Insert(GenericKey *key, const RowId &value, const KeyManager &KM)
{
  int pos = KeyIndex(key, KM);
  // the key is already in the page
  // std::cout << "pos" << pos << "size" << GetSize() << std::endl;
  if (pos != GetSize() && KM.CompareKeys(key, this->KeyAt(pos)) == 0)
    return GetSize();
  for (int i = GetSize(); i > pos; i--)
  {
    // printf("in loop\n");
    this->SetKeyAt(i, this->KeyAt(i - 1));
    this->SetValueAt(i, this->ValueAt(i - 1));
```

```
  }
  this->SetKeyAt(pos, key);
  this->SetValueAt(pos, value);
  this->IncreaseSize(1);
  return this->GetSize();
}
```

## MoveHalfTo(recipient)

- 将一般的键值对移动到recipient页里

**实现思路：**

- 调用CopyNfrom函数来帮助执行

```
/*
 * Remove half of key & value pairs from this page to "recipient" page
 */
void LeafPage::MoveHalfTo(LeafPage *recipient)
{
  int size = this->GetSize();
  recipient->CopyNFrom(pairs_off + (size - size / 2) * pair_size, size / 2);
  this->IncreaseSize(-size / 2);
}
```

## Lookup(key, value, KM)

- 查找是否key已经在叶子中，如果确实存在则把它存到value中，并返回true，否则返回false

**实现思路：**

- 通过KeyIndex函数来获取位置，并用位置的key通过KM的比较函数来比较是否已经存在
- 如果并不存在，则存储相应位置的value到参数value中

```
/*****************************************************************************
 * LOOKUP
 *****************************************************************************/
/*
 * For the given key, check to see whether it exists in the leaf page. If it
 * does, then store its corresponding value in input "value" and return true.
 * If the key does not exist, then return false
 */
bool LeafPage::Lookup(const GenericKey *key, RowId &value, const KeyManager &KM)
{
  // std::cout << "LookUp-leaf" << std::endl;
  int pos = this->KeyIndex(key, KM);
  // std::cout << pos << endl;
  if (pos == GetSize())
  {
    return false;
  }
  if (KM.CompareKeys(key, this->KeyAt(pos)) == 0)
  {
    // std::cout << "not exist " << pos << std::endl;
    value = this->ValueAt(pos);
```

```
    return true;
  }
  return false;
}
```

## RemoveAndDeleteRecord(key, KM)

- 查找并删除记录

**实现思路：**

- 先遍历叶子页查找需要删除的key是否存在
- 如果存在，执行删除
- 否则立刻返回
- 返回值为删除后的page size

```
/*****************************************************************************
 * REMOVE
 *****************************************************************************/
/*
 * First look through leaf page to see whether delete key exist or not. If
 * existed, perform deletion, otherwise return immediately.
 * NOTE: store key&value pair continuously after deletion
 * @return  page size after deletion
 */
int LeafPage::RemoveAndDeleteRecord(const GenericKey *key, const KeyManager &KM)
{
  RowId rid;
  if (!this->Lookup(key, rid, KM))
  {
    return -1;
  }
  int pos = this->KeyIndex(key, KM);
  for (int i = pos; i < this->GetSize(); i++)
  {
    this->SetKeyAt(i, this->KeyAt(i + 1));
    this->SetValueAt(i, this->ValueAt(i + 1));
  }
  this->IncreaseSize(-1);
  return this->GetSize();
}
```

## MoveAllTo(recipient)

- 把该节点所有内容移到recipient中，在B+树的coalesce中会用到

**实现思路：**

- 通过CopyNFrom函数实现
- recipient的NextPageId需要更新

```
/*****************************************************************************
 * MERGE
 *****************************************************************************/
/*
 * Remove all key & value pairs from this page to "recipient" page. Don't forget
 * to update the next_page id in the sibling page
 */
void LeafPage::MoveAllTo(LeafPage *recipient)
{
  recipient->CopyNFrom(pairs_off, this->GetSize());
  recipient->SetNextPageId(this->GetNextPageId());
  this->IncreaseSize(-this->GetSize());
}
```

## MoveFirstToEndOf(recipient)

- 把第一个键值对移动到recipient的最后，用于B+树redistribute的时候

**实现思路：**

- 通过recipient的CopyLastFrom方法实现
- 在移动过后，本节点的键值对也需要跟着移动

```
/*****************************************************************************
 * REDISTRIBUTE
 *****************************************************************************/
/*
 * Remove the first key & value pair from this page to "recipient" page.
 *
 */
void LeafPage::MoveFirstToEndOf(LeafPage *recipient)
{
  recipient->CopyLastFrom(this->KeyAt(0), this->ValueAt(0));
  for (int i = 0; i < this->GetSize() - 1; i++)
  {
    this->SetKeyAt(i, this->KeyAt(i + 1));
    this->SetValueAt(i, this->ValueAt(i + 1));
  }
  this->IncreaseSize(-1);
}
```

## CopyLastFrom(key, value)

- 把传入的key-value对拷贝到当前节点的最后

**实现思路：**

- 通过SetKeyAt与SetValueAt方法即可实现

```
/*
 * Copy the item into the end of my item list. (Append item to my array)
 */
void LeafPage::CopyLastFrom(GenericKey *key, const RowId value)
{
  int size = this->GetSize();
  this->SetKeyAt(size, key);
  this->SetValueAt(size, value);
  this->IncreaseSize(1);
}
```

## MoveLastToFrontOf(recipient)

- 将最后一个键值对移动到recipient的最前面，在redistribute中会用到

**实现思路：**

- 通过recipient的CopyFirstFrom方法可以简单做到

```
/*
 * Remove the last key & value pair from this page to "recipient" page.
 */
void LeafPage::MoveLastToFrontOf(LeafPage *recipient)
{
  recipient->CopyFirstFrom(this->KeyAt(this->GetSize() - 1), this->ValueAt(this-
>GetSize() - 1));
  this->IncreaseSize(-1);
}
```

## CopyFirstFrom(key, value)

- 把传入键值对放到当前节点最前面

**实现思路：**

- 先移动后续键值对
- 再SetKeyAt(0)与SetValueAt(0)

```
/*
 * Insert item at the front of my items. Move items accordingly.
 *
 */
void LeafPage::CopyFirstFrom(GenericKey *key, const RowId value)
{
  for (int i = GetSize(); i > 0; i--)
  {
    this->SetKeyAt(i, this->KeyAt(i - 1));
    this->SetValueAt(i, this->ValueAt(i - 1));
  }
  this->SetKeyAt(0, key);
  this->SetValueAt(0, value);
  this->IncreaseSize(1);
}
```

### CopyNFrom(src, size)

- 把来自src，单位为size的内容拷贝至当前节点

**实现思路：**

- 简单调用PariCopy即可，PairCopy函数是memcpy函数的封装

```
/*
 * Copy starting from items, and copy {size} number of elements into me.
 */
void LeafPage::CopyNFrom(void *src, int size)
{
  this->PairCopy(this->data_ + pair_size * this->GetSize(), src, size);
  this->IncreaseSize(size);
}
```

## BPLUSTREEINTERNALPAGE

### Init(page_id, parent_id, key_size, max_size)

- 初始化设置该内部节点

**实现思路：**

- 根据传入的参数，设置相应的信息

```
/**
 * TODO: Student Implement
 */
/*****************************************************************************
 * HELPER METHODS AND UTILITIES
 *****************************************************************************/
/*
 * Init method after creating a new internal page
 * Including set page type, set current size, set page id, set parent id and set
 * max page size
 */
void InternalPage::Init(page_id_t page_id, page_id_t parent_id, int key_size,
int max_size)
{
  // the size should be 0 at first
  this->SetSize(0);
  this->SetPageId(page_id);
  this->SetParentPageId(parent_id);
  this->SetMaxSize(max_size);
  this->SetPageType(IndexPageType::INTERNAL_PAGE);
  this->SetKeySize(key_size);
}
```

## PopulateNewRoot(old_value, new_key, new_value)

- 生成一个新的root，包含old_value, new_key和new_value
- 只在b+树的InsertIntoParent方法中被调用

**实现思路：**

- 简单SetValueAt与SetKeyAt即可

```
/*****************************************************************************
 * INSERTION
 *****************************************************************************/
/*
 * Populate new root page with old_value + new_key & new_value
 * When the insertion cause overflow from leaf page all the way upto the root
 * page, you should create a new root page and populate its elements.
 * NOTE: This method is only called within InsertIntoParent()(b_plus_tree.cpp)
 */
void InternalPage::PopulateNewRoot(const page_id_t &old_value, GenericKey
*new_key, const page_id_t &new_value)
{
  this->SetValueAt(0, old_value);
  this->SetKeyAt(1, new_key);
  this->SetValueAt(1, new_value);
  this->SetSize(2);
}
```

## InsertNodeAfter(old_value, new_key, new_value)

- 把new_key和new_value的键值对插在old_value之后

**实现思路：**

- 通过顺序遍历来寻找到位置，执行插入

```
/*
 * Insert new_key & new_value pair right after the pair with its value ==
 * old_value
 * @return:  new size after insertion
 */
int InternalPage::InsertNodeAfter(const page_id_t &old_value, GenericKey
*new_key, const page_id_t &new_value)
{
  int pos = ValueIndex(old_value);
  if (pos == -1)
  {
    ASSERT(false, "old value doesn't exist");
    return -1;
  }
  pos++;
  int i;
  for (i = this->GetSize() - 1; i >= pos; i--)
  {
    this->SetKeyAt(i + 1, this->KeyAt(i));
    this->SetValueAt(i + 1, this->ValueAt(i));
  }
```

```
  this->SetKeyAt(pos, new_key); // 插入新节点
  this->SetValueAt(pos, new_value);
  this->IncreaseSize(1);
  return this->GetSize();

}
```

## MoveHalfTo(recipient, buffer_pool_manager)

- 将一半的键值对移动到recipient页里

**实现思路：**

- 调用CopyNfrom函数来帮助执行

```
/*****************************************************************************
 * SPLIT
 *****************************************************************************/
/*
 * Remove half of key & value pairs from this page to "recipient" page
 * buffer_pool_manager 是干嘛的？传给CopyNFrom()用于Fetch数据页
 */
void InternalPage::MoveHalfTo(InternalPage *recipient, BufferPoolManager
*buffer_pool_manager)
{
//  int size = this->GetSize();
//  recipient->CopyNFrom(pairs_off + INTERNAL_PAIR * (size - size / 2), int(size
/ 2), buffer_pool_manager);
//  IncreaseSize(-int(size / 2));
  //fix bug ,date 6.24. 2:32
  int index=GetMinSize();
  SetSize(index);
  recipient->CopyNFrom(PairPtrAt(index),GetMaxSize()-index,buffer_pool_manager);
}
```

## Lookup(key, value, KM)

- 找到并返回指向包含参数key的child page，并返回它的pageid

**实现思路：**

- 二分查找，类似leafpage中的pageindex函数

```
/*****************************************************************************
 * LOOKUP
 *****************************************************************************/
/*
 * Find and return the child pointer(page_id) which points to the child page
 * that contains input "key"
 * Start the search from the second key(the first key should always be invalid)
 * 用了二分查找
 */
page_id_t InternalPage::Lookup(const GenericKey *key, const KeyManager &KM)
{
  int max = this->GetSize() - 1, low = 1, center = 0; // 从第二个节点开始查找
  while (low <= max)
```

```
  {
    center = (low + max) / 2; // 二分法加速查找
    if (KM.CompareKeys(key, this->KeyAt(center)) < 0)
    { // 小于当前中间值
      if (center == 1 || KM.CompareKeys(key, this->KeyAt(center - 1)) >= 0)
      { // 大于前一个值
        return this->ValueAt(center - 1);
      }
      else
        max = center - 1;
    }
    else
    { // 大于当前键值
      if (center == max || KM.CompareKeys(key, this->KeyAt(center + 1)) < 0)
      { // 小于后一个键值
        return this->ValueAt(center);
      }
      else
        low = center + 1;
    }
  }
  return ValueAt(center);
}
```

## Remove(index)

- 把位于index的键值对删掉

**实现思路:**

- 代码即思路

```
/*****************************************************************************
 * REMOVE
 *****************************************************************************/
/*
 * Remove the key & value pair in internal page according to input index(a.k.a
 * array offset)
 * NOTE: store key&value pair continuously after deletion
 */
void InternalPage::Remove(int index)
{
  for (int i = index + 1; i < this->GetSize(); i++)
  {
    this->SetKeyAt(i - 1, this->KeyAt(i));
    this->SetValueAt(i - 1, this->ValueAt(i));
  }
  this->IncreaseSize(-1);
}
```

## RemoveAndReturnOnlyChild()

- 字面意思

**实现思路：**

- 代码即思路

```
/*
 * Remove the only key & value pair in internal page and return the value
 * NOTE: only call this method within AdjustRoot()(in b_plus_tree.cpp)
 */
page_id_t InternalPage::RemoveAndReturnOnlyChild()
{
  page_id_t val = ValueAt(0);
  this->SetSize(0);
  return val;
}
```

## MoveAllTo(recipient)

- 把该节点所有内容移到recipient中，在B+树的coalesce中会用到

**实现思路：**

- 通过CopyNFrom函数实现
- recipient的NextPageId需要更新

```
/*****************************************************************************
 * MERGE
 *****************************************************************************/
/*
 * Remove all key & value pairs from this page to "recipient" page.
 * The middle_key is the separation key you should get from the parent. You need
 * to make sure the middle key is added to the recipient to maintain the
invariant.
 * You also need to use BufferPoolManager to persist changes to the parent page
id for those
 * pages that are moved to the recipient
 */
void InternalPage::MoveAllTo(InternalPage *recipient, GenericKey *middle_key,
BufferPoolManager *buffer_pool_manager)
{
  // middle key stored in 0 pos? maybe not
  recipient->CopyLastFrom(middle_key, *reinterpret_cast<page_id_t *>(pairs_off +
val_off), buffer_pool_manager);
  recipient->CopyNFrom(pairs_off + pair_size, GetSize() - 1,
buffer_pool_manager);
  this->SetSize(0);
}
```

## MoveFirstToEndOf(recipient)

- 把第一个键值对移动到recipient的最后，用于B+树redistribute的时候

**实现思路：**

- 通过recipient的CopyLastFrom方法实现
- 在移动过后，本节点的键值对也需要跟着移动

```
/*****************************************************************************
 * REDISTRIBUTE
 *****************************************************************************/
/*
 * Remove the first key & value pair from this page to tail of "recipient" page.
 *
 * The middle_key is the separation key you should get from the parent. You need
 * to make sure the middle key is added to the recipient to maintain the
invariant.
 * You also need to use BufferPoolManager to persist changes to the parent page
id for those
 * pages that are moved to the recipient
 */
void InternalPage::MoveFirstToEndOf(InternalPage *recipient, GenericKey
*middle_key,
                                    BufferPoolManager *buffer_pool_manager)
{
  recipient->CopyLastFrom(middle_key, *reinterpret_cast<page_id_t *>(pairs_off +
val_off), buffer_pool_manager);
  this->Remove(0);
}
```

## CopyLastFrom(key, value)

- 把传入的key-value对拷贝到当前节点的最后

**实现思路：**

- 通过SetKeyAt与SetValueAt方法即可实现

```
/* Append an entry at the end.
 * Since it is an internal page, the moved entry(page)'s parent needs to be
updated.
 * So I need to 'adopt' it by changing its parent page id, which needs to be
persisted with BufferPoolManger
 */
void InternalPage::CopyLastFrom(GenericKey *key, const page_id_t value,
BufferPoolManager *buffer_pool_manager)
{
  Page *page = buffer_pool_manager->FetchPage(value);
  // Find the page that holds the new entry
  BPlusTreePage *bptp = reinterpret_cast<BPlusTreePage *>(page->GetData());
  bptp->SetParentPageId(GetPageId());
  buffer_pool_manager->UnpinPage(value, true);
  int pos = this->GetSize();
  this->SetKeyAt(pos, key);
  this->SetValueAt(pos, value);
```

```
  IncreaseSize(1);
}
```

## MoveLastToFrontOf(recipient)

- 将最后一个键值对移动到recipient的最前面，在redistribute中会用到

**实现思路：**

- 通过recipient的CopyFirstFrom方法可以简单做到

```
/*
 * Remove the last key & value pair from this page to head of "recipient" page.
 * You need to handle the original dummy key properly, e.g. updating recipient's
array to position the middle_key
 at the
 * right place.
 * You also need to use BufferPoolManager to persist changes to the parent page
id for those pages that are
 * moved to the recipient
 */
void InternalPage::MoveLastToFrontOf(InternalPage *recipient, GenericKey
*middle_key,
                                     BufferPoolManager *buffer_pool_manager)
{
  int last = GetSize() - 1;
  recipient->SetKeyAt(0, middle_key);
  // first is zero which is not used
  recipient->CopyFirstFrom(this->ValueAt(last - 1), buffer_pool_manager);
  this->Remove(last);
}
```

## CopyFirstFrom(key, value)

- 把传入键值对放到当前节点最前面

**实现思路：**

- 先移动后续键值对
- 再SetKeyAt(0)与SetValueAt(0)

```
/* Append an entry at the beginning.
 * Since it is an internal page, the moved entry(page)'s parent needs to be
updated.
 * So I need to 'adopt' it by changing its parent page id, which needs to be
persisted with BufferPoolManger
 */
void InternalPage::CopyFirstFrom(const page_id_t value, BufferPoolManager
*buffer_pool_manager)
{
  Page *page = buffer_pool_manager->FetchPage(value);
  BPlusTreePage *bptp = reinterpret_cast<BPlusTreePage *>(page->GetData());
  bptp->SetParentPageId(this->GetPageId());
  buffer_pool_manager->UnpinPage(value, true);
  for (int i = this->GetSize() - 1; i >= 0; i--)
```

```
  {
    this->SetKeyAt(i + 1, this->KeyAt(i));
    this->SetValueAt(i + 1, this->ValueAt(i));
  }
  this->SetValueAt(0, value);
  this->IncreaseSize(1);
}
```

## CopyNFrom(src, size, buffer_pool_manager)

- 把来自src，单位为size的内容拷贝至当前节点

**实现思路：**

- 简单调用PariCopy即可，PairCopy函数是memcpy函数的封装
- 不过还需要把所有的childpage的parent设置为recipient

```
/* Copy entries into me, starting from {items} and copy {size} entries.
 * Since it is an internal page, for all entries (pages) moved, their parents
page now changes to me.
 * So I need to 'adopt' them by changing their parent page id, which needs to be
persisted with BufferPoolManger
 * 不做size检测
 */
void InternalPage::CopyNFrom(void *src, int size, BufferPoolManager
*buffer_pool_manager)
{
  // copy src into this
  this->PairCopy(pairs_off + this->GetSize() * INTERNAL_PAIR, src, size);
  for (int i = 0; i < size; i++)
  {
    // change the parent page id
    Page *child_page = buffer_pool_manager->FetchPage(ValueAt(this->GetSize() +
i));
    BPlusTreePage *BNode = reinterpret_cast<BPlusTreePage *>(child_page-
>GetData());
    BNode->SetParentPageId(this->GetPageId());
    buffer_pool_manager->UnpinPage(child_page->GetPageId(), true);
  }
  this->IncreaseSize(size);
}
```

## BPLUSTREE

## BPlusTree(index_id, buffer_pool_manager, KM, leaf_max_size, internal_max_size)

- 构造函数

**实现思路：**

- 如果传入的leaf_max_size是0，需要自己设置

```
/**
 * TODO: Student Implement
 */
BPlusTree::BPlusTree(index_id_t index_id, BufferPoolManager
*buffer_pool_manager, const KeyManager &KM,
                     int leaf_max_size, int internal_max_size)
    : index_id_(index_id),
      buffer_pool_manager_(buffer_pool_manager),
      processor_(KM),
      leaf_max_size_(leaf_max_size),
      internal_max_size_(internal_max_size)
{
  Page *page = buffer_pool_manager_->FetchPage(TREE_INDEX_META);

  IndexRootsPage *indexRootsPage = reinterpret_cast<IndexRootsPage *>(page-
>GetData());
  indexRootsPage->GetRootId(index_id, &root_page_id_); // 获取根节点的 id
  if (leaf_max_size == 0)
  { // undefined
    leaf_max_size_ = (PAGE_SIZE - LEAF_PAGE_HEADER_SIZE) / (KM.GetKeySize() +
sizeof(RowId)) - 1;
    internal_max_size_ = leaf_max_size_ + 1;
  }
  buffer_pool_manager_->UnpinPage(TREE_INDEX_META, false);
  buffer_pool_manager_->UnpinPage(root_page_id_, false);
}
```

## Destroy(current_page_id)

- 删除current_page_id所指向的page与他的所有后代page

**实现思路：**

- 如果它是leaf，则需要删除它的parent中的对应值
- 如果是internalpage，调用函数delete internalpage

```
void BPlusTree::Destroy(page_id_t current_page_id)
{
  if (current_page_id == INVALID_PAGE_ID)
  { // delete all
    BPlusTreePage *page = reinterpret_cast<BPlusTreePage *>
(buffer_pool_manager_->FetchPage(root_page_id_)->GetData());
    if (page->IsLeafPage())
    {
      DestroyAllLeaf(page);
    }
    else
    {
      DestroyInternalPage(page);
    }
    return;
```

```
  }
  BPlusTreePage *cur_page =
      reinterpret_cast<BPlusTreePage *>(buffer_pool_manager_-
>FetchPage(current_page_id)->GetData());
  if (cur_page->IsLeafPage())
  {
    if (cur_page->GetPageId() != INVALID_PAGE_ID)
    {
      InternalPage *parent =
          reinterpret_cast<InternalPage *>(buffer_pool_manager_-
>FetchPage(cur_page->GetParentPageId())->GetData());
      int index = parent->ValueIndex(current_page_id);
      parent->Remove(index);
      if (index)
      {
        auto *leftSibling =
            reinterpret_cast<LeafPage *>(buffer_pool_manager_->FetchPage(parent-
>ValueAt(index - 1))->GetData());
        leftSibling->SetPageType(IndexPageType::LEAF_PAGE);
        // because remove will make the index move left by 1
        leftSibling->SetNextPageId(parent->ValueAt(index));
      }
      else if (parent->GetSize() == 0)
        Destroy(parent->GetPageId());

      this->buffer_pool_manager_->DeletePage(current_page_id);
    }
  }
  else
  {
    DestroyInternalPage(cur_page);
  }
}
```

## DestroyAllLeaf(page)

- 把所有的leaf全部删除

**实现思路：**

- 顺序遍历，并调用Destroy函数

```
void BPlusTree::DestroyAllLeaf(BPlusTreePage *page)
{
  LeafPage *leafPage = reinterpret_cast<LeafPage *>(page);
  while (leafPage->GetNextPageId() != INVALID_PAGE_ID)
  { // 将整块的叶子顺序删除
    LeafPage *nextPage =
        reinterpret_cast<LeafPage *>(buffer_pool_manager_->FetchPage(leafPage-
>GetNextPageId())->GetData());
    nextPage->SetPageType(IndexPageType::LEAF_PAGE);
    Destroy(leafPage->GetPageId());
    leafPage = nextPage;
  }
  Destroy(leafPage->GetPageId());
}
```

## DestroyInternalPage(page)

- 删除内部节点

**实现思路:**

- 把所有的childpage的parent设置为INVALID_PAGE_ID，因此destroy函数中他们也会被删掉

```
void BPlusTree::DestroyInternalPage(BPlusTreePage *page)
{
  InternalPage *internalPage = reinterpret_cast<InternalPage *>(page);
  int size = internalPage->GetSize();
  for (int i = 0; i < size; i++)
  {
    auto *nextPage =
        reinterpret_cast<BPlusTreePage *>(buffer_pool_manager_-
>FetchPage(internalPage->ValueAt(i))->GetData());
    // This is important!!! ortherwise the parent will be visited again in
Destory
    nextPage->SetParentPageId(INVALID_PAGE_ID);
    Destroy(internalPage->ValueAt(i));
  }
  Destroy(internalPage->GetPageId());
}
```

## GetValue(key, result, transaction)

- 找到key对应的record，并存入result中

**实现思路:**

- 先通过findleafpage找到对应的leaf
- 然后判断leaf中是否有key
- 如果有，把record存入result中，返回true
- 如果无，返回false

```
/*****************************************************************************
 * SEARCH
 *****************************************************************************/
/*
```

```
 * Return the only value that associated with input key
 * This method is used for point query
 * @return : true means key exists
 */
bool BPlusTree::GetValue(const GenericKey *key, std::vector<RowId> &result,
Transaction *transaction)
{
  if (this->IsEmpty())
  {
    return false;
  }
  Page *page = this->FindLeafPage(key, root_page_id_, false);
  LeafPage *leaf_page = reinterpret_cast<LeafPage *>(page->GetData());
  RowId row;
  // Row row_;
  ASSERT(leaf_page != nullptr, "leaf_page is null!");
  // (this->processor_).DeserializeToKey(key, row_, processor_.key_schema_);
  // std::cout << "Getvalue" << (row_.GetField(0))->toString() <<
(row_.GetField(1))->toString() << std::endl;
  bool res = leaf_page->Lookup(key, row, this->processor_);
  // std::cout << "find" << res << std::endl;
  // std::cout << row.GetPageId() << " " << row.GetSlotNum() << std::endl;
  this->buffer_pool_manager_->UnpinPage(page->GetPageId(), false); // 没有对该页进
行修改，不是脏页
  if(res)
    result.push_back(row);
  return res;
}
```

## Insert(key, value, transaction)

- 把key-value对插入b+树中

**实现思路：**

- 如果当前树是空的，则stratnewtree
- 否则寻找到对应的leafpage，并执行插入

```
/*****************************************************************************
 * INSERTION
 *****************************************************************************/
/*
 * Insert constant key & value pair into b+ tree
 * if current tree is empty, start new tree, update root page id and insert
 * entry, otherwise insert into leaf page.
 * @return: since we only support unique key, if user try to insert duplicate
 * keys return false, otherwise return true.
 */
bool BPlusTree::Insert(GenericKey *key, const RowId &value, Transaction
*transaction)
{
  if (this->IsEmpty())
  {
    this->StartNewTree(key, value);
    return true;
```

```
  }
  // printf("hello\n");

  LeafPage *page = reinterpret_cast<LeafPage *>(this->FindLeafPage(key, this->root_page_id_, false));
  RowId new_value;
  if (page->Lookup(key, new_value, this->processor_))
  {
    buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
    return false;
  }
  page->Insert(key, value, this->processor_);
  if (page->GetSize() > page->GetMaxSize())
  {
    LeafPage *newleaf = this->Split(page, transaction);
    // for we have inserted the key into the page, and we are not sure where the key is
    this->InsertIntoParent(page, newleaf->KeyAt(0), newleaf, transaction);
  }
  buffer_pool_manager_->UnpinPage(page->GetPageId(), true);
  return true;
}
```

## StartNewTree(key, value)

- 生成一颗新树

**实现思路:**

- 一定要把该类型设置成LEAF_PAGE
- 并且需要再rootspage源页中更新index信息

```
/*
 * Insert constant key & value pair into an empty tree
 * User needs to first ask for new page from buffer pool manager(NOTICE: throw
 * an "out of memory" exception if returned value is nullptr), then update b+
 * tree's root page id and insert entry directly into leaf page.
 */
void BPlusTree::StartNewTree(GenericKey *key, const RowId &value)
{
  // insert
  Page *page = this->buffer_pool_manager_->NewPage(root_page_id_);
  this->UpdateRootPageId(1);
  if (page == NULL)
    throw runtime_error("out of memory");

  LeafPage *root_node = reinterpret_cast<LeafPage *>(page->GetData());
  root_node->Init(page->GetPageId(), INVALID_PAGE_ID, processor_.GetKeySize(), this->leaf_max_size_);
  root_node->Insert(key, value, this->processor_);
  root_node->SetPageType(IndexPageType::LEAF_PAGE);
  buffer_pool_manager_->UnpinPage(root_page_id_, true);
  this->index_id_++;

}
```

## InsertIntoLeaf(key, value, transaction)

- 插入叶子中

**实现思路：**

- 直接insert即可

```
/*
 * Insert constant key & value pair into leaf page
 * User needs to first find the right leaf page as insertion target, then look
 * through leaf page to see whether insert key exist or not. If exist, return
 * immediately, otherwise insert entry. Remember to deal with split if
necessary.
 * @return: since we only support unique key, if user try to insert duplicate
 * keys return false, otherwise return true.
 */

bool BPlusTree::InsertIntoLeaf(GenericKey *key, const RowId &value, Transaction
*transaction)
{
  return this->Insert(key, value, transaction);
}
```

## Split(node, transaction)

- 把当前page的一半内容挪到新page中去

**实现思路：**

- new_page+movehalfto

```
/*
 * Split input page and return newly created page.
 * Using template N to represent either internal page or leaf page.
 * User needs to first ask for new page from buffer pool manager(NOTICE: throw
 * an "out of memory" exception if returned value is nullptr), then move half
 * of key & value pairs from input page to newly created page
 */

BPlusTreeInternalPage *BPlusTree::Split(InternalPage *node, Transaction
*transaction)
{
  page_id_t page_Id;
  InternalPage *newLeaf = reinterpret_cast<InternalPage *>(this-
>buffer_pool_manager_->NewPage(page_Id));
  if (newLeaf == NULL)
  {
    throw runtime_error("out of memory");
  }
  newLeaf->Init(page_Id, node->GetParentPageId(), node->GetKeySize(), node-
>GetMaxSize());
  node->MoveHalfTo(newLeaf, this->buffer_pool_manager_);
  return newLeaf;
}
```

## Split(node, transaction)(leaf)

- 把当前page的一半内容挪到新page中去

**实现思路：**

- new_page+movehalfto

```
// 将一个叶子节点分成两半（没有排序）
BPlusTreeLeafPage *BPlusTree::Split(LeafPage *node, Transaction *transaction)
{
  page_id_t page_Id;
  LeafPage *newLeaf = reinterpret_cast<LeafPage *>(this->buffer_pool_manager_-
>NewPage(page_Id));
  if (newLeaf == NULL)
  { // 建立新页失败
    throw runtime_error("out of memory");
  }
  newLeaf->Init(page_Id, node->GetParentPageId(), processor_.GetKeySize(), node-
>GetMaxSize());
  node->MoveHalfTo(newLeaf); // 各一半
  newLeaf->SetNextPageId(node->GetNextPageId());
  node->SetNextPageId(newLeaf->GetPageId());
  return newLeaf;
}
```

## InsertIntoParent(old_node, key, new_node, transaction)

- 节点满了，需要插入parent中去

**实现思路：**

- 如果是root，直接插即可
- 如果不是root，需要判断parent的大小，递归进行插入

```
/*
 * Insert key & value pair into internal page after split
 * @param   old_node      input page from split() method
 * @param   key
 * @param   new_node      returned page from split() method
 * User needs to first find the parent page of old_node, parent node must be
 * adjusted to take info of new_node into account. Remember to deal with split
 * recursively if necessary.
 */
 //内部节点插入一个新的节点
void BPlusTree::InsertIntoParent(BPlusTreePage *old_node, GenericKey *key,
BPlusTreePage *new_node,
                                 Transaction *transaction)
{
  if (old_node->GetParentPageId() == INVALID_PAGE_ID)
  {
    page_id_t root_Id;
    InternalPage *newroot = reinterpret_cast<InternalPage *>(this-
>buffer_pool_manager_->NewPage(root_Id));
    if (newroot == NULL)
    {
```

```
      throw runtime_error("out of mem");
    }
    newroot->Init(root_Id, INVALID_PAGE_ID, old_node->GetKeySize(),
internal_max_size_);
    root_page_id_ = newroot->GetPageId();
    this->UpdateRootPageId(0);
    newroot->PopulateNewRoot(old_node->GetPageId(), key, new_node->GetPageId());
    old_node->SetParentPageId(root_Id);
    new_node->SetParentPageId(root_Id);

    buffer_pool_manager_->UnpinPage(root_Id, true);
    buffer_pool_manager_->UnpinPage(old_node->GetPageId(), true);
    buffer_pool_manager_->UnpinPage(new_node->GetPageId(), true);
    return;
  }
  // this is not root
  InternalPage *internalPage =
      reinterpret_cast<InternalPage *>(this->buffer_pool_manager_-
>FetchPage(old_node->GetParentPageId()));
  int index = internalPage->ValueIndex(old_node->GetPageId());
  internalPage->InsertNodeAfter(old_node->GetPageId(), key, new_node-
>GetPageId());
  new_node->SetParentPageId(internalPage->GetPageId());
  if (internalPage->GetSize() > internalPage->GetMaxSize())
  {
    auto *newNode = this->Split(internalPage, transaction);
    GenericKey *temp;
    for (int i = 0; i < internalPage->GetSize(); i++) {
      BPlusTreePage *page1 =
          reinterpret_cast<BPlusTreePage *>(buffer_pool_manager_-
>FetchPage(internalPage->ValueAt(i)));
      page1->SetParentPageId(internalPage->GetPageId());
      buffer_pool_manager_->UnpinPage(page1->GetPageId(), false);
    }
    for (int i = 0; i < newNode->GetSize(); i++) {
      BPlusTreePage *page1 =
          reinterpret_cast<BPlusTreePage *>(buffer_pool_manager_-
>FetchPage(newNode->ValueAt(i)));
      page1->SetParentPageId(newNode->GetPageId());
      buffer_pool_manager_->UnpinPage(page1->GetPageId(), false);
    }
    GenericKey *new_key = newNode->KeyAt(0);
    InsertIntoParent(internalPage, new_key, newNode, transaction);
  }
  buffer_pool_manager_->UnpinPage(new_node->GetPageId(), true);
  buffer_pool_manager_->UnpinPage(old_node->GetPageId(), true);
  buffer_pool_manager_->UnpinPage(internalPage->GetPageId(), true);
}
```

### Remove(key, transaction)

- 从b+树种删除key，并从右边进行维护

**实现思路：**

- 如果删除后过小，需要进行coalesceandredistribute

```
/*****************************************************************************
 * REMOVE
 *****************************************************************************/
/*
 * Delete key & value pair associated with input key
 * If current tree is empty, return immediately.
 * If not, User needs to first find the right leaf page as deletion target, then
 * delete entry from leaf page. Remember to deal with redistribute or merge if
 * necessary.
 */
// 从b+树中删除 key，并从右边进行维护
void BPlusTree::Remove(const GenericKey *key, Transaction *transaction)
{
  vector<RowId> result;
  if (this->GetValue(key, result, transaction) == false)
    return;
  LeafPage *leafPage = reinterpret_cast<LeafPage *>(FindLeafPage(key,
root_page_id_, false));
  int index = leafPage->RemoveAndDeleteRecord(key, this->processor_);
  if (index != leafPage->GetSize())
  {
    CoalesceOrRedistribute(leafPage, transaction);
  }
  buffer_pool_manager_->UnpinPage(leafPage->GetPageId(), true);
}
```

## CoalesceOrRedistribute(node, transaction)

- 叶子节点内部元素数量太少时的融合与重新排布

**实现思路：**

- 如果sibling能存下两个叶子节点的所有元素，则coalesce
- 否则redistribute

```
/* todo
 * User needs to first find the sibling of input page. If sibling's size + input
 * page's size > page's max size, then redistribute. Otherwise, merge.
 * Using template N to represent either internal page or leaf page.
 * @return: true means target leaf page should be deleted, false means no
 * deletion happens
 */
// 删除之后的合并或重分配的维护
template <typename N>
bool BPlusTree::CoalesceOrRedistribute(N *&node, Transaction *transaction)
{
  if (node->IsRootPage())
  {
    return this->AdjustRoot(node);
  }
  if ((node->IsLeafPage() && node->GetSize() >= node->GetMinSize()) ||
```

```
      (!node->IsLeafPage() && node->GetSize() > node->GetMinSize()))
    return false; // 上述情况删除后没有影响，无需合并
  page_id_t parent_page = node->GetParentPageId();
  InternalPage *parentNode = reinterpret_cast<InternalPage *>(
      this->buffer_pool_manager_->FetchPage(parent_page)->GetData()); // 获取父亲
节点 if (parentNode == nullptr)
  {
    throw runtime_error("all page are pinned during CoalesceOrRedistribute");
  }
  // 获取当前节点在父节点中的下标位置
  int index = parentNode->ValueIndex(node->GetPageId());
  int sibling_Id;
  if (index == 0)
  {
    sibling_Id = parentNode->ValueAt(index + 1); // 此时只能找右边的兄弟
  }
  else
  {
    sibling_Id = parentNode->ValueAt(index - 1); // 左兄弟
  }
  auto *sibling_Page = reinterpret_cast<N *>(this->buffer_pool_manager_-
>FetchPage(sibling_Id)->GetData());
  if (sibling_Page == nullptr)
  {
    throw runtime_error("all page are pinned while CoalesceOrRedistribute");
  }
  if (sibling_Page->GetSize() + node->GetSize() > node->GetMaxSize())
  {
    this->Redistribute(sibling_Page, node, index);
    this->buffer_pool_manager_->UnpinPage(parent_page, true);
    return false;
  } // 进行合并
  if (index == 0)
  {
    this->Coalesce(node, sibling_Page, parentNode, index, transaction);
    this->buffer_pool_manager_->UnpinPage(parent_page, true);
    return false; // 此时node没有被删除
  }
  else
    this->Coalesce(sibling_Page, node, parentNode, index, transaction);
  this->buffer_pool_manager_->UnpinPage(parent_page, true);
  return true;
}
```

## Coalesce(neighbor_node, node, parent, index, transaction)

- 融合node与neighbor_node

**实现思路：**

- 依据注释

```
/*
 * Move all the key & value pairs from one page to its sibling page, and notify
 * buffer pool manager to delete this page. Parent page must be adjusted to
 * take info of deletion into account. Remember to deal with coalesce or
```

```
 * redistribute recursively if necessary.
 * Using template N to represent either internal page or leaf page.
 * @param   neighbor_node      sibling page of input "node"
 * @param   node               input from method coalesceOrRedistribute()
 * @param   parent             parent page of input "node"
 * @return  true means parent node should be deleted, false means no deletion
happened
 */
// 两个节点融合，删除右边那一页，不用管index，传入已经不同了
bool BPlusTree::Coalesce(LeafPage *&neighbor_node, LeafPage *&node, InternalPage
*&parent, int index,
                         Transaction *transaction)
{
  node->MoveAllTo(neighbor_node);
  neighbor_node->SetNextPageId(node->GetNextPageId());      // 更新链表
  buffer_pool_manager_->UnpinPage(node->GetPageId(), true); // 删除前先解锁
  buffer_pool_manager_->DeletePage(node->GetPageId());      // 删除右边页
  int reIndex = index == 0 ? 1 : index;                     // 需要删除点的下标
  parent->Remove(reIndex);                                  // 从父亲节点中删除右边
的节点
  buffer_pool_manager_->UnpinPage(neighbor_node->GetPageId(), true);
  return this->CoalesceOrRedistribute(parent, transaction); // 递归维护 parent 节点
}
```

## Coalesce(neighbor_node, node, parent, index, transaction)（internal)

- 融合node与neighbor_node

**实现思路:**

- 依据注释

```
bool BPlusTree::Coalesce(InternalPage *&neighbor_node, InternalPage *&node,
InternalPage *&parent, int index,
                         Transaction *transaction)
{
  node->MoveAllTo(node, parent->KeyAt(index), buffer_pool_manager_);
  buffer_pool_manager_->UnpinPage(node->GetPageId(), true); // 删除前先解锁
  buffer_pool_manager_->DeletePage(node->GetPageId());      // 删除右边页
  int reIndex = index == 0 ? 1 : index;                     // 需要删除点的下标
  parent->Remove(reIndex);                                  // 从父亲节点中删除右边
的节点
  buffer_pool_manager_->UnpinPage(neighbor_node->GetPageId(), true);
  return this->CoalesceOrRedistribute(parent, transaction); // 递归维护 parent 节点
  return false;
}
```

## Redistribute(neighbor_node, node, index)(leaf)

- 向sibling借一个元素过来，并更改父亲节点的元素

**实现思路：**

- 依据注释

```
/*
 * Redistribute key & value pairs from one page to its sibling page. If index ==
 * 0, move sibling page's first key & value pair into end of input "node",
 * otherwise move sibling page's last key & value pair into head of input
 * "node".
 * Using template N to represent either internal page or leaf page.
 * @param   neighbor_node      sibling page of input "node"
 * @param   node               input from method coalesceOrRedistribute()
 */
// 当删除后需要从兄弟借节点时
void BPlusTree::Redistribute(LeafPage *neighbor_node, LeafPage *node, int index)
{
  // 获取父亲节点，在更新子节点后也更新父节点
  InternalPage *parent =
      reinterpret_cast<InternalPage *>(buffer_pool_manager_->FetchPage(node-
>GetParentPageId())->GetData());
  if (parent == NULL)
  {
    throw runtime_error("all page are pinned during Redistribute");
  }
  if (index == 0)
  {
    neighbor_node->MoveFirstToEndOf(node);             // 把第一个节点移动到node的末
尾
    parent->SetValueAt(1, neighbor_node->GetPageId()); // 更新第二个指针
    parent->SetKeyAt(1, neighbor_node->KeyAt(0));      // 更新第一个键值
  }
  else
  {
    neighbor_node->MoveLastToFrontOf(node);        // 把最后一个节点移动到node的头
    parent->SetValueAt(index, node->GetPageId()); // 更新父亲节点中node的头
    parent->SetKeyAt(index, node->KeyAt(0));
  }
  buffer_pool_manager_->UnpinPage(parent->GetPageId(), true); // 释放父亲节点
  buffer_pool_manager_->UnpinPage(neighbor_node->GetPageId(), true);
  buffer_pool_manager_->UnpinPage(node->GetPageId(), true); // 释放并更新内存
}
```

## Redistribute(neighbor_node, node, index)(internal)

- 向sibling借一个元素过来，并更改父亲节点的元素

**实现思路：**

- 依据注释

```
void BPlusTree::Redistribute(InternalPage *neighbor_node, InternalPage *node,
int index)
{
  // 获取父亲节点，在更新子节点后也更新父节点
  InternalPage *parent =
```

```
      reinterpret_cast<InternalPage *>(buffer_pool_manager_->FetchPage(node->GetParentPageId())->GetData());
  if (parent == NULL)
  {
    throw runtime_error("all page are pinned during Redistribute");
  }
  if (index == 0)
  {
    neighbor_node->MoveFirstToEndOf(node, parent->KeyAt(index), buffer_pool_manager_);
    parent->SetValueAt(1, neighbor_node->GetPageId()); // 更新第二个指针
    parent->SetKeyAt(1, neighbor_node->KeyAt(0));      // 更新第一个键值
  }
  else
  {
    neighbor_node->MoveLastToFrontOf(node, parent->KeyAt(index), buffer_pool_manager_);
    parent->SetValueAt(index, node->GetPageId()); // 更新父亲节点中node的头
    parent->SetKeyAt(index, node->KeyAt(0));
  }
  buffer_pool_manager_->UnpinPage(parent->GetPageId(), true); // 释放父亲节点
  buffer_pool_manager_->UnpinPage(neighbor_node->GetPageId(), true);
  buffer_pool_manager_->UnpinPage(node->GetPageId(), true); // 释放并更新内存
}
```

## AdjustRoot(old_root_node)

- rootpage有也需要调整，不过rootpage的最小元素个数为1，返回是否需要被删除

**实现思路:**

- 代码与注释非常清楚

```
/*
 * Update root page if necessary
 * NOTE: size of root page can be less than min size and this method is only
 * called within coalesceOrRedistribute() method
 * case 1: when you delete the last element in root page, but root page still
 * has one last child
 * case 2: when you delete the last element in whole b+ tree
 * @return : true means root page should be deleted, false means no deletion
 * happened
 */
bool BPlusTree::AdjustRoot(BPlusTreePage *old_root_node)
{
  if (old_root_node->IsLeafPage())
  {
    if (old_root_node->GetSize() == 0)
    { // 删除这个节点
      root_page_id_ = INVALID_PAGE_ID;
      UpdateRootPageId(0);
      return true;
    }
    else
      return false; // 不用调整
  }
```

```
  if (old_root_node->GetSize() == 1)
  {                                                                    // 需要进行融合
    auto *page = reinterpret_cast<InternalPage *>(old_root_node); // 类型转换
    root_page_id_ = page->ValueAt(0);
    this->UpdateRootPageId(0); // 只有一个孩子，则这个孩子直接成为新的根
    auto *newRoot = reinterpret_cast<InternalPage *>(buffer_pool_manager_-
>FetchPage(root_page_id_)->GetData());
    if (page == nullptr)
    {
      throw runtime_error("all pages are pinned while AdjustRoot");
    }
    newRoot->SetParentPageId(INVALID_PAGE_ID);
    buffer_pool_manager_->UnpinPage(root_page_id_, true);
    return true;
  }
  return false;
}
```

## Begin()

- 迭代器的begin，指向第一个元素

**实现思路：**

- 代码即思路

```
/*****************************************************************************
 * INDEX ITERATOR
 *****************************************************************************/
/*
 * Input parameter is void, find the left most leaf page first, then construct
 * index iterator
 * @return : index iterator
 */
IndexIterator BPlusTree::Begin() {
  GenericKey *key = processor_.InitKey();
  auto left_page = FindLeafPage(key, 0, true);
  auto *leaf = reinterpret_cast<LeafPage *>(left_page->GetData());
  page_id_t current_page_id = leaf->GetPageId();
  buffer_pool_manager_->UnpinPage(current_page_id, false);
  return IndexIterator(current_page_id, buffer_pool_manager_, 0);
}
```

## Begin(key)

- 迭代器的begin，指向包含key的元素

**实现思路：**

- 代码即思路

```
/*
 * Input parameter is low-key, find the leaf page that contains the input key
 * first, then construct index iterator
 * @return : index iterator
 */
IndexIterator BPlusTree::Begin(const GenericKey *key) {
  auto left_page = FindLeafPage(key);
  auto *leaf = reinterpret_cast<LeafPage *>(left_page->GetData());
  page_id_t current_page_id = leaf->GetPageId();
  buffer_pool_manager_->UnpinPage(current_page_id, false);
  return IndexIterator(current_page_id, buffer_pool_manager_, leaf-
>KeyIndex(key, processor_));
}
```

## End()

- 返回内容为空的迭代器

**实现思路：**

- 略

```
/*
 * Input parameter is void, construct an index iterator representing the end
 * of the key/value pair in the leaf node
 * @return : index iterator
 */
IndexIterator BPlusTree::End() { return IndexIterator(INVALID_PAGE_ID,
buffer_pool_manager_, 0); }
```

## FindLeafPage(key, page_id, leftMost)

- 找到包含key的leafpage
- 如果leftMost为true，找到最左边的leafpage

**实现思路：**

- 通过key比较，一层一层向下迭代，直到找到leafpage

```
/*****************************************************************************
 * UTILITIES AND DEBUG
 *****************************************************************************/
/*
 * Find leaf page containing particular key, if leftMost flag == true, find
 * the left most leaf page
 * Note: the leaf page is pinned, you need to unpin it after use.
 */
Page *BPlusTree::FindLeafPage(const GenericKey *key, page_id_t page_id, bool
leftMost)
{
  Page *curr_page = this->buffer_pool_manager_->FetchPage(this->root_page_id_);
  // Unpin after each update
  this->buffer_pool_manager_->UnpinPage(root_page_id_, false);
  BPlusTreePage *curr_node = reinterpret_cast<BPlusTreePage *>(curr_page-
>GetData());
```

```
  // Row row;
  // processor_.DeserializeToKey(key, row, processor_.key_schema_);
  // printf("%s", (row.GetField(0))->toString());
  // std::cout << "root size in find:" << curr_node->GetSize() << std::endl;
  while (!curr_node->IsLeafPage())
  {
    InternalPage *internal_node = reinterpret_cast<InternalPage *>(curr_node);
    page_id_t child_page_id;
    if (leftMost == true)
    {
      // if it is left most, no need to find key, just return the left most page
      child_page_id = internal_node->ValueAt(0);
    }
    else
    {
      // else, we have to find the key
      // printf("hhh\n");
      child_page_id = internal_node->Lookup(key, this->processor_);
      // printf("done %d\n", child_page_id);
    }
    Page *child_page = this->buffer_pool_manager_->FetchPage(child_page_id);
    ASSERT(child_page != nullptr, "child page is null!");
    BPlusTreePage *child_node = reinterpret_cast<BPlusTreePage *>(child_page-
>GetData());
    this->buffer_pool_manager_->UnpinPage(child_page_id, false);
    // 交换
    curr_page = child_page;
    curr_node = child_node;
  }
  // std::cout << "find leaf" << std::endl;
  return curr_page;
}
```

## UpdateRootPageId(insert_record)

- IndexRootsPage中存储的index_id需要在b+树构造时更新

**实现思路:**

- 代码即思路

```
/*
 * Update/Insert root page id in header page(where page_id = 0, header_page is
 * defined under include/page/header_page.h)
 * Call this method everytime root page id is changed.
 * @parameter: insert_record      default value is false. When set to true,
 * insert a record <index_name, current_page_id> into header page instead of
 * updating it.
 */
void BPlusTree::UpdateRootPageId(int insert_record)
{
  auto *IndexPage = reinterpret_cast<IndexRootsPage *>(buffer_pool_manager_-
>FetchPage(INDEX_ROOTS_PAGE_ID));
  if (insert_record == 0) {
    IndexPage->Update(this->index_id_, this->root_page_id_);
  } else {
```

```
    IndexPage->Insert(this->index_id_, this->root_page_id_);
  }
  buffer_pool_manager_->UnpinPage(INDEX_ROOTS_PAGE_ID, true);
}
```

# 第四模块-CATALOG MANAGER

## CATALOG

### GetSerializedSize()

- catalog序列化的大小

**实现思路：**

- table_mata_pages_ 和index_meat_pages_里的invalidpage需要去除
- 除此之外，还有magic_num，table_nums和index_nums需要计入

```
/**
 * TODO: Student Implement
 */
uint32_t CatalogMeta::GetSerializedSize() const
{
  // ASSERT(false, "Not Implemented yet");
  // in case the page is invalid
  uint32_t len = table_meta_pages_.size();
  for (auto it : table_meta_pages_)
  {
    if (it.second == INVALID_PAGE_ID)
      len--;
  }
  len += index_meta_pages_.size();
  for (auto it : index_meta_pages_)
  {
    if (it.second == INVALID_PAGE_ID)
      len--;
  }
  len = len * (sizeof(uint32_t) + sizeof(int)) + 3 * sizeof(uint32_t);
  return len;
}
```

### CatalogManager(buffer_pool_manager, lock_manager, log_manager, init)

- 构造函数

**实现思路：**

- 如果是init，只需new catalogmeta
- 否则需要把所有的table和index信息导入catlogmanager之中

```
/**
 * TODO: Student Implement
 */
CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager,
LockManager *lock_manager,
```

```
                                LogManager *log_manager, bool init)
    : buffer_pool_manager_(buffer_pool_manager), lock_manager_(lock_manager),
log_manager_(log_manager)
{
  //    ASSERT(false, "Not Implemented yet");
  if (init)
  {
    this->catalog_meta_ = CatalogMeta::NewInstance();
    this->next_table_id_ = 0;
    this->next_index_id_ = 0;
  }
  else
  {
    // else, we can just load the meta page
    Page *catalogMetaPage = buffer_pool_manager-
>FetchPage(CATALOG_META_PAGE_ID);
    catalog_meta_ = CatalogMeta::DeserializeFrom(catalogMetaPage->GetData());

    next_table_id_ = catalog_meta_->GetNextTableId();
    next_index_id_ = catalog_meta_->GetNextIndexId();

    for (auto it = catalog_meta_->table_meta_pages_.begin(); it !=
catalog_meta_->table_meta_pages_.end(); it++)
    {
      if (it->second != INVALID_PAGE_ID)
      {
        LoadTable(it->first, it->second);
      }
    }

    for (auto it = catalog_meta_->index_meta_pages_.begin(); it !=
catalog_meta_->index_meta_pages_.end(); it++)
    {
      if (it->second != INVALID_PAGE_ID)
      {
        LoadIndex(it->first, it->second);
      }
    }

    buffer_pool_manager->UnpinPage(CATALOG_META_PAGE_ID, false);
  }
}
```

## CreateIndex(table_name, index_name, index_keys, txn, index_info, index_type)

- 通过catalogmanager创建新的index

**实现思路:**

- 只要不是已经存在，就继续创建
- catalog_meta_ 中的index_meta_pages_需要更新
- index_info需要初始化
- index_names_ 和indexes_需要记录创建的table信息

```cpp
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::CreateIndex(const std::string &table_name, const string
&index_name,
                                    const std::vector<std::string> &index_keys,
Transaction *txn,
                                    IndexInfo *&index_info, const string
&index_type)
{
  // ASSERT(false, "Not Implemented yet");
  // index_keys里存的是index是以哪几个column来建立索引的
  TableInfo *table_info;
  if (GetTable(table_name, table_info) != DB_SUCCESS)
    return DB_TABLE_NOT_EXIST;

  if (index_names_[table_name].count(index_name) != 0)
  {
    return DB_INDEX_ALREADY_EXIST;
  }

  // whether the columns of the index exists
  Schema *schema = table_info->GetSchema();
  vector<uint32_t> col_indexes;
  for (const auto &s : index_keys)
  {
    uint32_t column_index;
    if (schema->GetColumnIndex(s, column_index) != DB_SUCCESS)
      return DB_COLUMN_NAME_NOT_EXIST;
    col_indexes.push_back(column_index);
  }

  index_info = IndexInfo::Create();
  index_id_t index_id = this->next_index_id_;
  this->next_index_id_++;
  // create Metadata
  IndexMetadata *index_meta_data = IndexMetadata::Create(index_id, index_name,
table_info->GetTableId(), col_indexes);
  index_info->Init(index_meta_data, table_info, buffer_pool_manager_);
  page_id_t page_id;
  catalog_meta_->index_meta_pages_[index_id + 1] = INVALID_PAGE_ID;
  Page *page = buffer_pool_manager_->NewPage(page_id);
  if (page == nullptr)
    return DB_FAILED;

  index_meta_data->SerializeTo(page->GetData());
  catalog_meta_->index_meta_pages_[index_id] = page_id;
  buffer_pool_manager_->UnpinPage(page_id, false);

  indexes_.emplace(index_id, index_info);
  index_names_[table_name][index_name] = index_id;
  if (index_meta_data != nullptr)
    return DB_SUCCESS;
  return DB_FAILED;
```

```
}
```

## GetIndex(table_name, index_name, index_info)

- 通过index_name获取相应的index

**实现思路：**

- 只要table_names_ 里与tables_里都存在table_name对应的属性
- index_names_ 里存在index_name相应的属性
- 把indexes_ 中存的index_info赋值给参数index_info即可

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::GetIndex(const std::string &table_name, const
std::string &index_name,
                                 IndexInfo *&index_info) const
{
  //find table
  if(index_names_.find(table_name)==index_names_.end())
    return DB_TABLE_NOT_EXIST;

  //find index
  auto indname_id=index_names_.find(table_name)->second;
  if(indname_id.find(index_name)==indname_id.end())
    return DB_INDEX_NOT_FOUND;

  //have found and return index_info
  index_id_t index_id=indname_id[index_name];
  index_info=indexes_.find(index_id)->second;

  return DB_SUCCESS;
}
```

## GetTableIndexes(table_name, indexes)

- 通过catalogmanager把所有的表插回indexed中

**实现思路：**

- 遍历indexes_

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::GetTableIndexes(const std::string &table_name,
std::vector<IndexInfo *> &indexes) const
{
  // ASSERT(false, "Not Implemented yet");
  if (index_names_.size() == 0)
    return DB_SUCCESS;
  for (auto iter = index_names_.at(table_name).begin(); iter !=
index_names_.at(table_name).end(); iter++)
  {
```

```
    indexes.push_back(indexes_.at(iter->second));
  }
  return DB_SUCCESS;
}
```

## CreateTable(table_name, schema, txn, table_info)

- 通过catalogmanager创建新的表

**实现思路：**

- 只要不是已经存在，就继续创建
- catalog_meta_ 中的table_meta_pages_需要更新
- table_info需要初始化
- table_names_ 和tables_需要记录创建的table信息

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema
*schema,
                                    Transaction *txn, TableInfo *&table_info)
{
  // ASSERT(false, "Not Implemented yet");
  if (table_names_.count(table_name) != 0)
    return DB_TABLE_ALREADY_EXIST;

  table_id_t table_id = this->next_table_id_;
  this->next_table_id_++;
  page_id_t new_table_page_id;
  Page *new_table_page = buffer_pool_manager_->NewPage(new_table_page_id);
  catalog_meta_->table_meta_pages_[table_id] = new_table_page_id;
  catalog_meta_->table_meta_pages_[next_table_id_] = INVALID_PAGE_ID;
  TableMetadata *table_meta_data = TableMetadata::Create(table_id, table_name,
new_table_page_id, schema);
  TableHeap *table_heap = TableHeap::Create(buffer_pool_manager_, schema, txn,
log_manager_, lock_manager_);

  // 简单来说，就是通过几种方法建立table
  table_meta_data->SerializeTo(new_table_page->GetData());
  table_info=table_info->Create();
  table_info->Init(table_meta_data, table_heap);
  buffer_pool_manager_->UnpinPage(new_table_page_id, true);

  table_names_.emplace(table_name, new_table_page_id);
  tables_.emplace(new_table_page_id, table_info);

  if (table_meta_data != nullptr && table_heap != nullptr)
    return DB_SUCCESS;
  return DB_FAILED;
}
```

## GetTable(table_name, table_info)

- 通过table_name获取相应的table

**实现思路：**

- 只要table_names_ 里与tables_里都存在table_name对应的属性
- 把tables_ 中存的table_info赋值给参数tablel_info即可

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::GetTable(const string &table_name, TableInfo
*&table_info)
{
  // ASSERT(false, "Not Implemented yet");
  if ((this->table_names_).count(table_name) == 0)
    return DB_TABLE_NOT_EXIST;
  page_id_t table_page_id = this->table_names_[table_name];
  if ((this->tables_).count(table_page_id) == 0)
    return DB_FAILED;
  table_info = (this->tables_)[table_page_id];
  return DB_SUCCESS;
}
```

## GetTables(tables)

- 通过catalogmanager把所有的表插回tables中

**实现思路：**

- 遍历tables_

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::GetTables(vector<TableInfo *> &tables) const
{
  // ASSERT(false, "Not Implemented yet");
  auto iter = tables_.begin();
  while (iter != tables_.end())
  {
    tables.push_back(iter->second);
    iter++;
  }
  return DB_SUCCESS;
}
```

## DropTable(table_name)

- 通过catalogmanager删除table

**实现思路：**

- 通过第二部分的tableheap来删除table

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::DropTable(const string &table_name)
{
  // ASSERT(false, "Not Implemented yet");
  if (table_names_.count(table_name) == 0)
  {
    return DB_TABLE_NOT_EXIST;
  }
  table_id_t tableId = table_names_[table_name];
  tables_[tableId]->GetTableHeap()->DeleteTable();
  buffer_pool_manager_->DeletePage(catalog_meta_->table_meta_pages_[tableId]);

  catalog_meta_->table_meta_pages_.erase(tableId);
  table_names_.erase(table_name);
  tables_.erase(tableId);
  return DB_SUCCESS;

}
```

## DropIndex(table_name, index_name)

- 通过catalogmanager删除index

**实现思路：**

- 通过第三部分的b+树来删除index

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::DropIndex(const string &table_name, const string
&index_name)
{
  if (table_names_.find(table_name) == table_names_.end()) return
DB_TABLE_NOT_EXIST;
  // ASSERT(false, "Not Implemented yet");
  if (index_names_[table_name].count(index_name) == 0)
  {
    return DB_INDEX_NOT_FOUND;
  }

  index_id_t indexId = index_names_[table_name][index_name];
  buffer_pool_manager_->DeletePage(catalog_meta_->index_meta_pages_[indexId]);

  catalog_meta_->index_meta_pages_.erase(indexId);
  indexes_.erase(indexId);
  index_names_[table_name].erase(index_name);

  return DB_SUCCESS;
```

```
}
```

## DropIndex(table_name, index_name)

- 通过catalogmanager删除index

**实现思路：**

- 通过第三部分的b+树来删除index

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::DropIndex(const string &table_name, const string
&index_name)
{
  if (table_names_.find(table_name) == table_names_.end()) return
DB_TABLE_NOT_EXIST;
  // ASSERT(false, "Not Implemented yet");
  if (index_names_[table_name].count(index_name) == 0)
  {
    return DB_INDEX_NOT_FOUND;
  }

  index_id_t indexId = index_names_[table_name][index_name];
  buffer_pool_manager_->DeletePage(catalog_meta_->index_meta_pages_[indexId]);

  catalog_meta_->index_meta_pages_.erase(indexId);
  indexes_.erase(indexId);
  index_names_[table_name].erase(index_name);

  return DB_SUCCESS;

}
```

## FlushCatalogMetaPage()

- 把catalog

**实现思路：**

- buffer_pool_manager把所有的table_meta_ 都先unpin为脏页
- 最后存储catalog_meta_page

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::FlushCatalogMetaPage() const
{
  // ASSERT(false, "Not Implemented yet");
  // flush catalog meta page
  Page *page = buffer_pool_manager_->FetchPage(CATALOG_META_PAGE_ID);
  this->catalog_meta_->SerializeTo(page->GetData());
  buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, true);
```

```
  // flush all the table pages
  std::vector<TableInfo *> tables;
  this->GetTables(tables);
  for (auto table : tables)
  {
    Page *page = this->buffer_pool_manager_->FetchPage(table->GetRootPageId());
    table->table_meta_->SerializeTo(page->GetData());
    this->buffer_pool_manager_->UnpinPage(table->GetRootPageId(), true);
  }
  buffer_pool_manager_->FlushPage(CATALOG_META_PAGE_ID);
  return DB_SUCCESS;
}
```

## LoadTable(table_id, page_id)

- 加载对应的table

**实现思路:**

- 注释即思路

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::LoadTable(const table_id_t table_id, const page_id_t
page_id)
{
  try{ //yt's
    //init
    page_id_t meta_page_id=0;
    Page* meta_page=nullptr;
    page_id_t table_page_id=0;
    Page* table_page=nullptr;
    string table_name_="";
    TableMetadata* table_meta_=nullptr;
    TableHeap* table_heap_=nullptr;
    TableSchema* schema_=nullptr;
    TableInfo* table_info=nullptr;
    //init table info
    table_info=table_info->Create();
    //load table meta page
    meta_page_id=page_id;
    meta_page=buffer_pool_manager_->FetchPage(meta_page_id);
    //LOad table meta
    table_meta_->DeserializeFrom(meta_page->GetData(),table_meta_);
    //table init
    ASSERT(table_id==table_meta_->GetTableId(),"Load wrong table");
    table_name_=table_meta_->GetTableName();
    table_page_id=table_meta_->GetFirstPageId();
    schema_=table_meta_->GetSchema();
    table_heap_=table_heap_-
>Create(buffer_pool_manager_,table_page_id,schema_,nullptr,nullptr);
    //table info
    table_info->Init(table_meta_,table_heap_);
    //table meta
```

```
    table_names_[table_name_]=table_id;
    tables_[table_id]=table_info;
    return DB_SUCCESS;
  }catch(exception e){
    return DB_FAILED;
  }
}
```

## LoadIndex(index_id, page_id)

- 加载对应的index

**实现思路：**

- 注释即思路

```
/**
 * TODO: Student Implement
 */
dberr_t CatalogManager::LoadIndex(const index_id_t index_id, const page_id_t
page_id)
{
  try{
    string table_name="";
    string index_name="";
    page_id_t meta_page_id=0;
    std::vector<std::uint32_t> key_map{};

    meta_page_id=page_id;
    Page* meta_page=buffer_pool_manager_->FetchPage(meta_page_id);
    //deserial index meta
    IndexMetadata* index_meta_= nullptr;
    IndexMetadata::DeserializeFrom(meta_page->GetData(),index_meta_);
    //get index name from index meta
    index_name=index_meta_->GetIndexName();
    //get table id from index meta
    table_id_t table_id=index_meta_->GetTableId();

    //table info
    TableInfo* table_info_=tables_[table_id];
    table_name=table_info_->GetTableName();

    //index info
    IndexInfo* index_info=index_info->Create();
    index_info->Init(index_meta_,table_info_,buffer_pool_manager_);
    //table meta
    indexes_[index_id]=index_info;
    index_names_[table_name][index_name]=index_id;
    return DB_SUCCESS;
  }catch(exception e){
    return DB_FAILED;
  }
}
```

# 第五模块-PLANNER AND EXECUTOR

## ExecuteEngine

### ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext *context)

- 创建数据库

**实现思路：**

- 获取名字db_name后直接new一个DBStorageEngine即可

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteCreateDatabase(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateDatabase" << std::endl;
#endif
  string db_name(ast->child_->val_);
  if(dbs_.find(db_name)!=dbs_.end())
    return DB_ALREADY_EXIST;
  dbs_[db_name]=new DBStorageEngine(db_name);
  cout << "Database " << db_name << " successfully created." << endl;
  return DB_SUCCESS;
}
```

### ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext *context)

- 删除数据库

**实现思路：**

- 获取名字db_name后直接删除
- 同时如果当前正在使用这个数据库，把当前使用设为空

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteDropDatabase(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteDropDatabase" << std::endl;
#endif
  string db_name(ast->child_->val_);
  if(dbs_.find(db_name)==dbs_.end())
    return DB_NOT_EXIST;
  cout << "Database " << db_name << " successfully dropped." << endl;
  delete dbs_[db_name];
  dbs_.erase(db_name);
  if(current_db_==db_name)
    current_db_="";
  return DB_SUCCESS;
```

```
}
```

## ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext *context)

- 展示所有的数据库

**实现思路:**

- 调用ResultWriter类格式化输出所有数据库

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteShowDatabases(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteShowDatabases" << std::endl;
#endif
  std::stringstream sstream;
  ResultWriter Writer(sstream);
  vector<int>width_vec;
  width_vec.push_back(10);
  //先放一个10
  for(auto pair : dbs_)
    width_vec[0]=max((int)(pair.first.length()),width_vec[0]);

  Writer.Divider(width_vec);
  Writer.BeginRow();
  Writer.WriteHeaderCell("Database",width_vec[0]);//长度为width_vec【0】
  Writer.EndRow();
  Writer.Divider(width_vec);

  for(auto pair:dbs_){
    Writer.BeginRow();
    Writer.WriteCell(pair.first,width_vec[0]);
    Writer.EndRow();
  }
  Writer.Divider(width_vec);
  cout<<Writer.stream_.rdbuf();
  return DB_SUCCESS;
}
```

## ExecuteUseDatabase(pSyntaxNode ast, ExecuteContext *context)

- 设置当前使用的数据库

**实现思路:**

- current_db_=db_name即可

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteUseDatabase(pSyntaxNode ast, ExecuteContext
*context) {
```

```
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteUseDatabase" << std::endl;
#endif
  string db_name(ast->child_->val_);
  if(dbs_.find(db_name)==dbs_.end())
    return DB_NOT_EXIST;
  cout << "Use database " << db_name << endl;
  current_db_=db_name;
  return DB_SUCCESS;
}
```

## ExecuteShowTables(pSyntaxNode ast, ExecuteContext *context)

- 展示所有表

**实现思路:**

- 通过context遍历所有表
- 调用ResultWriter类格式化输出所有数据表

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteShowTables(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteShowTables" << std::endl;
#endif
  //未选当前使用db
  if(current_db_.empty())
    return DB_FAILED;
  string Head="Tables at database "+current_db_;

  vector<TableInfo*>table_info_vec;
  table_info_vec.clear();
  context->GetCatalog()->GetTables(table_info_vec);

  std::stringstream sstream;
  ResultWriter Writer(sstream);
  vector<int>width_vec;
  width_vec.push_back(Head.length());
  //把表头的长度放进去
  for(auto itr:table_info_vec)
    width_vec[0]=max((int)itr->GetTableName().length(),width_vec[0]);
  Writer.Divider(width_vec);
  Writer.BeginRow();
  Writer.WriteHeaderCell(Head,width_vec[0]);
  Writer.EndRow();
  Writer.Divider(width_vec);

  for(auto itr:table_info_vec){
    Writer.BeginRow();
    Writer.WriteCell(itr->GetTableName(),width_vec[0]);
    Writer.EndRow();
  }
```

```
  Writer.Divider(width_vec);
  std::cout<<Writer.stream_.rdbuf();
  return DB_SUCCESS;
}
```

## ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context)

- 创建表

**实现思路：**

- 根据context建立表
- 同时将primarykey的index和unique的index建立

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateTable" << std::endl;
#endif
  //未选当前使用db
  if(current_db_.empty())
    return DB_FAILED;

  string table_name(ast->child_->val_);
  vector<string>primary_key_vec;
  vector<string>unique_key_vec;
  auto cur_node=ast->child_->next_->child_;
  while(cur_node!= nullptr){
    //得到主键循环
    if(cur_node->type_==kNodeColumnList&&string(cur_node->val_)=="primary keys")
{
      auto pri_node=cur_node->child_;
      while(pri_node!= nullptr){
        primary_key_vec.push_back(string(pri_node->val_));
        pri_node=pri_node->next_;
      }
    }
    cur_node=cur_node->next_;
  }
  cur_node=ast->child_->next_->child_;//回到原来的node

  // 获取columns
  uint32_t index_cnt=0;
  vector<Column*>columns_vec;
  while(cur_node!= nullptr&&cur_node->type_==kNodeColumnDefinition){
    //是否是unique?
    bool is_unique=(cur_node->val_!=nullptr&&string(cur_node->val_)=="unique");
    auto child_node=cur_node->child_;
    string col_name(child_node->val_);
    Column *column_new;
    string type_str(child_node->next_->val_);
    if(type_str=="int")
```

```
      column_new=new Column(col_name,kTypeInt,index_cnt,true,is_unique);
    if(type_str=="float")
      column_new=new Column(col_name,kTypeFloat,index_cnt,true,is_unique);
    if(type_str=="char"){
      string num(child_node->next_->child_->val_);
      //一个一个看是不是数字,不然stoi（num）会报错
      for(auto digit:num)
        if(!isdigit(digit))
          return DB_FAILED;
      //不能是负数！！！
      if(stoi(num)<0)
        return DB_FAILED;
      column_new=new
Column(col_name,kTypeChar,stoi(num),index_cnt,true,is_unique);
    }
    if(is_unique){
      unique_key_vec.push_back(col_name);
    }
    columns_vec.push_back(column_new);
    cur_node=cur_node->next_;
    index_cnt++;
  }
  Schema *schema=new Schema(columns_vec);
  TableInfo *table_info;
  auto err=context->GetCatalog()->CreateTable(table_name,schema,context-
>GetTransaction(),table_info);
  if(err!=DB_SUCCESS)
    return err;

  if(primary_key_vec.size()!=0) {
    IndexInfo *index_info;
    err=context->GetCatalog()->CreateIndex(table_name,
"AUTO_PK_IDX_ON_"+table_name, primary_key_vec, context->GetTransaction(),
index_info, "bptree");
    if(err!=DB_SUCCESS)
      return err;
  }
  for(auto cur_unique_key:unique_key_vec){
    string name = "UNIQUE_"+cur_unique_key;
    name+="_ON_"+table_name;
    IndexInfo *index_info;
    err=context->GetCatalog()->CreateIndex(table_name,name,unique_key_vec,
context->GetTransaction(), index_info, "bptree");
    if(err!=DB_SUCCESS)
      return err;
  }
  cout<<"Table "<<table_name<<" successfully created."<<endl;
  return DB_SUCCESS;

}
```

## ExecuteDropTable(pSyntaxNode ast, ExecuteContext *context)

- 删除表

**实现思路：**

- 将表删除
- 同时将所有在其上的index删除

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteDropTable(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteDropTable" << std::endl;
#endif
  if(current_db_.empty())
    return DB_FAILED;
  string table_name(ast->child_->val_);

  auto err=context->GetCatalog()->DropTable(table_name);
  if(err!=DB_SUCCESS)
    return err;

  vector<IndexInfo*>index_info_vec_;
  context->GetCatalog()->GetTableIndexes(table_name,index_info_vec_);
  for(auto index_info_:index_info_vec_){
    err=context->GetCatalog()->DropIndex(table_name,index_info_-
>GetIndexName());
    if(err!=DB_SUCCESS)
      return err;
  }

  return DB_SUCCESS;
}
```

## ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext *context)

- 展示所有索引

**实现思路：**

- 遍历索引（context），格式化输出（ResultWriter）

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteShowIndexes(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteShowIndexes" << std::endl;
#endif
  if(current_db_.empty())
    return DB_FAILED;
```

```
  dberr_t err;
  vector<TableInfo*>table_info_vec_;{
    err=context->GetCatalog()->GetTables(table_info_vec_);
    if(err!=DB_SUCCESS)
      return err;
  }

  vector<IndexInfo*>index_info_vec_;
  for(auto table_info_:table_info_vec_){
    err=context->GetCatalog()->GetTableIndexes(table_info_-
>GetTableName(),index_info_vec_);
    if(err!=DB_SUCCESS)
      return err;
  }

  std::stringstream sstream;
  ResultWriter Writer(sstream);
  vector<int>width_vec_;
  //先放5（index的len）
  width_vec_.push_back(5);
  for(auto index_info_:index_info_vec_)
    width_vec_[0]=max((int)index_info_->GetIndexName().length(),width_vec_[0]);
  Writer.Divider(width_vec_);
  Writer.BeginRow();
  Writer.WriteHeaderCell("Index",width_vec_[0]);
  Writer.EndRow();
  Writer.Divider(width_vec_);

  for(auto index_info_:index_info_vec_){
    Writer.BeginRow();
    Writer.WriteCell(index_info_->GetIndexName(),width_vec_[0]);
    Writer.EndRow();
  }

  Writer.Divider(width_vec_);
  std::cout<<Writer.stream_.rdbuf();

  return DB_SUCCESS;
}
```

## ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context)

- 创建index

**实现思路：**

- 解析context的到index的column，在表上建立新的index

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateIndex" << std::endl;
#endif
```

```
  string index_name(ast->child_->val_);
  string table_name(ast->child_->next_->val_);
  vector<string> column_names;
  for (auto column_itr = ast->child_->next_->next_->child_; column_itr !=
nullptr; column_itr = column_itr->next_)
    column_names.push_back(column_itr->val_);
  string type="bptree";
  if (ast->child_->next_->next_->next_ != nullptr)
    type = string(ast->child_->next_->next_->next_->child_->val_);


  TableInfo * table_info_;
  IndexInfo * index_info_;
  auto err = context->GetCatalog()->GetTable(table_name, table_info_);
  if (err != DB_SUCCESS)
    return err;
  err = context->GetCatalog()-
>CreateIndex(table_name,index_name,column_names,context-
>GetTransaction(),index_info_,type);
  if (err != DB_SUCCESS)
    return err;

  for (auto row =  table_info_->GetTableHeap()->Begin(context-
>GetTransaction()); row != table_info_->GetTableHeap()->End(); ++row) {
    auto rid = (*row).GetRowId();
    vector<Field> field_vec_;
    for (auto column : index_info_->GetIndexKeySchema()->GetColumns())
      field_vec_.push_back(*(*row).GetField(column->GetTableInd()));
    Row key(field_vec_);
    err = index_info_->GetIndex()->InsertEntry(key,rid,context-
>GetTransaction());
    if (err != DB_SUCCESS)
      return err;
  }
  return DB_SUCCESS;
}
```

## ExecuteDropIndex(pSyntaxNode ast, ExecuteContext *context)

- 通过context删除index

**实现思路：**

- 得到index信息后直接删除即可

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteDropIndex(pSyntaxNode ast, ExecuteContext
*context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteDropIndex" << std::endl;
#endif
  if(current_db_.empty())
    return DB_FAILED;
```

```
  string index_name(ast->child_->val_);
  vector<TableInfo*>table_info_vec_;
  auto res=DB_INDEX_NOT_FOUND;
  context->GetCatalog()->GetTables(table_info_vec_);
  for(auto table_info_:table_info_vec_){
    dberr_t err=context->GetCatalog()->DropIndex(table_info_-
>GetTableName(),index_name);
    if(err!=DB_SUCCESS)
      return err;
  }
  return DB_SUCCESS;
}
```

## ExecuteExecfile(pSyntaxNode ast, ExecuteContext *context)

- 执行sql脚本文件

**实现思路:**

- 通过循环调用Execute来实现对文件中多条命令的执行
- 同时记录执行时间
- 注意要将所有命令一条一条读进来，；为每条命令结束符

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteExecfile(pSyntaxNode ast, ExecuteContext *context)
{
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteExecfile" << std::endl;
#endif
  string file_name(ast->child_->val_);
  ifstream fp(file_name, ios::in);
  tcount_ = 0;
  exec_= true;
  cmd_vec_.resize(0);
  int i = 0;
  char char0;
  char cmd[1024];//1024Byte buffer size
  memset(cmd, 0, 1024);
  if (fp.is_open()) {
    while (fp.get(char0)) {
      cmd[i] = char0;
      i++;
      if (char0 == ';') {
        //这一命令结束了。读入\n
        fp.get(char0);
        cmd_vec_.emplace_back(cmd);

        YY_BUFFER_STATE bp = yy_scan_string(cmd);
        if (bp == nullptr) {
          LOG(ERROR) << "Failed to create yy buffer state." << std::endl;
          exit(1);
        }
        yy_switch_to_buffer(bp);
```

```
        MinisqlParserInit();
        yyparse();
        if (MinisqlParserGetError())
          cout<< MinisqlParserGetErrorMessage()<<endl;
        auto res = Execute(MinisqlGetParserRootNode());
        MinisqlParserFinish();
        yy_delete_buffer(bp);
        yylex_destroy();
        ExecuteInformation(res);
        if (res == DB_QUIT) {
          break;
        }
        memset(cmd, 0, 1024);
        i = 0;//i reset , cmd reset
      }
    }
    fp.close();
  }
  itr_ = cmd_vec_.begin();
  return DB_SUCCESS;
  //时间的输出最后再exec函数中
}
```

### ExecuteQuit(pSyntaxNode ast, ExecuteContext *context)

- 退出minisql

**实现思路:**

- 情况当前db
- 返回DB_QUIT

```
/**
 * TODO: Student Implement
 */
dberr_t ExecuteEngine::ExecuteQuit(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteQuit" << std::endl;
#endif
  current_db_ = "";
  return DB_QUIT;
}
```

## SeqScanExecutor

### SeqScanExecutor::Init()

- 初始化SeqScanExecutor

**实现思路:**

- 通过AbstractExecutor获取成员信息

```
void SeqScanExecutor::Init() {
    table_info_ = TableInfo::Create();;
    exec_ctx_->GetCatalog()->GetTable(plan_->table_name_, table_info_);
    itr_ = table_info_->GetTableHeap()->Begin(exec_ctx_->GetTransaction());
}
```

### SeqScanExecutor::Next(Row *row, RowId *rid)

- 返回扫描到的符合条件的row

**实现思路:**

- 注释即思路

```
bool SeqScanExecutor::Next(Row *row, RowId *rid) {
    while(itr_!=table_info_->GetTableHeap()->End() ){
      //如果是空谓词，直接全都返回；或者谓词判定后该row符合条件，也返回
      if(!plan_->filter_predicate_||Field(kTypeInt, 1).CompareEquals(plan_-
>filter_predicate_->Evaluate(&(*itr_)))){
        //只输出所需的field构成的row！！！！！
        //这里在seq的测试中测不出来，反应在了delete的测试中
        vector<Field> out_fields;
        out_fields.clear();
        uint32_t column_index;
        for( auto column_p : plan_->OutputSchema()->GetColumns() ){
          table_info_->GetSchema()->GetColumnIndex(column_p-
>GetName(),column_index);
          out_fields.push_back(*(*itr_).GetField(column_index));
        }
        *row = Row(out_fields);
        row->SetRowId((*itr_).GetRowId());
        //*rid = (itr_)->GetRowId(); fix bug when find bug in #2
        rid = new RowId(row->GetRowId());
        itr_++;
        return true;
      }
      itr_++;
    }
    //遍历结束
    return false;
}
```

## IndexScanExecutor

### IndexScanExecutor::Init()

- 初始化IndexScanExecutor

**实现思路：**

- 直接在初始化时候找到所有符合的row

```
void IndexScanExecutor::Init() {
  map<uint32_t, pair<string, Field>> map0;
  map<uint32_t, uint32_t> map1;
  FindIndexColumn(plan_->GetPredicate(), map0);
  FindIndexColumn(plan_->indexes_, map1);
  //insert the first then go for cycle
  auto it = map1.begin();
  vector<Field> key_field;
  key_field.push_back(map0.at(it->second).second);
  (plan_->indexes_)[0]->GetIndex()->ScanKey(Row(key_field), rid_vec_, nullptr,
(map0.at(it->second)).first);
  for(it++; it != map1.end(); it++){
    vector<RowId> res;
    vector<Field> key_field_;
    key_field_.push_back((map0.at(it->second)).second);
    (plan_->indexes_)[it->first]->GetIndex()->ScanKey(Row(key_field_), res,
nullptr, (map0.at(it->second)).first);
    sort(res.begin(), res.end(), ridcomp);
    auto itr = set_intersection(res.begin(), res.end(),rid_vec_.begin(),
rid_vec_.end(), rid_vec_.begin(), ridcomp);
    rid_vec_.resize(itr - rid_vec_.begin());
  }
}
```

## IndexScanExecutor::Next(Row *row, RowId *rid)

- 返回扫描到的符合条件的row

**实现思路：**

- 直接返回已经存在vector中的row

```
bool IndexScanExecutor::Next(Row *row, RowId *rid) {
  if(count < rid_vec_.size()){
    *rid = rid_vec_[count];
    TableInfo *table_info_=TableInfo::Create();
    GetExecutorContext()->GetCatalog()->GetTable(plan_->GetTableName(),
table_info_);
    row->SetRowId(rid_vec_[count]);
    row->GetFields().clear();
    table_info_->GetTableHeap()->GetTuple(row, exec_ctx_->GetTransaction());
    count++;
    return true;
  }
  else
    return false;

}
```

# InsertExecutor

## InsertExecutor::Init()

- 初始化InsertExecutor

**实现思路：**

- 通过AbstractExecutor获取成员信息

```
void InsertExecutor::Init() {
  //init child_executor_
  child_executor_->Init();
  // init table_info_
  table_info_ = TableInfo::Create();
  exec_ctx_->GetCatalog()->GetTable(plan_->table_name_,table_info_);
  // init index_info_vec_
  exec_ctx_->GetCatalog()->GetTableIndexes(table_info_-
>GetTableName(),index_info_vec_);
}
```

## InsertExecutor::Next([[maybe_unused]] Row *row, RowId *rid)

- 执行插入语句！

**实现思路：**

- 获取要插入的row插入表中
- 同时根据row的信息在index中插入key

```
bool InsertExecutor::Next([[maybe_unused]] Row *row, RowId *rid) {
  RowId child_rid;
  Row child_row;
  if(!child_executor_->Next(&child_row,&child_rid))return false;
  for(auto it = index_info_vec_.begin();it!=index_info_vec_.end();it++){//第一次遍
历，查看所有的index，是否有unique的冲突
    vector<Field> fields;
    auto keySchema = (*it)->GetIndexKeySchema();
    for(int i=0;i<keySchema->GetColumnCount();i++){
      Field* field = child_row.GetField(keySchema->GetColumn(i)-
>GetTableInd());//先获取列id，然后找到field，从而去建立scankey
      fields.push_back(*field);
    }
    vector<RowId>res;
    Row scankey(fields);
    if(DB_SUCCESS==(*it)->GetIndex()->ScanKey(scankey,res, exec_ctx_-
>GetTransaction()) ){
      //duplicate
      cout<<"Duplicate record, unique conflict when insert\n";
      return false;
    }
  }
  table_info_->GetTableHeap()->InsertTuple(child_row, nullptr);
  for(auto it = index_info_vec_.begin();it!=index_info_vec_.end();it++){//遍历，更
新所有index
```

```
    auto keySchema = (*it)->GetIndexKeySchema();
    vector<Field>fields;
    for(int i=0;i<keySchema->GetColumnCount();i++){//获取索引
      Field* field = child_row.GetField(keySchema->GetColumn(i)-
>GetTableInd());//先获取列id，然后找到field，从而去建立key
      fields.push_back(*field);
    }
    Row key_to_insert(fields);
    (*it)->GetIndex()->InsertEntry(key_to_insert,child_row.GetRowId(),exec_ctx_-
>GetTransaction());
  }
  *rid = child_row.GetRowId();//返回rid
  return true;
}
```

# DeleteExecutor

## DeleteExecutor::Init()

- 初始化DeleteExecutor

**实现思路：**

- 通过AbstractExecutor获取成员信息

```
void DeleteExecutor::Init() {
  //same with insert!
  //init child_executor_
  child_executor_->Init();
  // init table_info_
  table_info_ = TableInfo::Create();
  exec_ctx_->GetCatalog()->GetTable(plan_->table_name_,table_info_);
  // init index_info_vec_
  exec_ctx_->GetCatalog()->GetTableIndexes(table_info_-
>GetTableName(),index_info_vec_);
}
```

## DeleteExecutor::Next([[maybe_unused]] Row *row, RowId *rid)

- 执行删除语句！

**实现思路：**

- 和插入类似
- 获取要插入的row在表中删除
- 同时根据row的信息删除index中的key

```
bool DeleteExecutor::Next([[maybe_unused]] Row *row, RowId *rid) {
  RowId child_rid;
  Row child_row;
  if(!child_executor_->Next(&child_row,&child_rid))return false;
  if(!table_info_->GetTableHeap()->MarkDelete(child_row.GetRowId(), exec_ctx_-
>GetTransaction()))return false;
  for(auto it = index_info_vec_.begin();it!=index_info_vec_.end();it++){//遍历所有
index，删除其中的child构成的key
```

```
    auto keySchema = (*it)->GetIndexKeySchema();
    vector<Field> fields;
    for( auto col : keySchema->GetColumns() ){
      uint32_t col_index;
      table_info_->GetSchema()->GetColumnIndex(col->GetName(),col_index);
      fields.push_back(*(child_row.GetField(col_index)));
    }
    Row key_to_insert(fields);
    (*it)->GetIndex()->RemoveEntry(key_to_insert,child_row.GetRowId(),exec_ctx_-
>GetTransaction());
  }
  return true;
}
```

# UpdateExecutor

## UpdateExecutor::Init()

- 初始化UpdateExecutor

**实现思路:**

- 通过AbstractExecutor获取成员信息

```
void UpdateExecutor::Init() {
  //init child_executor_
  child_executor_->Init();
  // init table_info_
  table_info_ = TableInfo::Create();
  exec_ctx_->GetCatalog()->GetTable(plan_->table_name_,table_info_);
  // init index_info_vec_
  exec_ctx_->GetCatalog()->GetTableIndexes(table_info_-
>GetTableName(),index_info_vec_);
}
```

## UpdateExecutor::Next([[maybe_unused]] Row *row, RowId *rid)

- 执行更新语句!

**实现思路:**

- 获取要插入的row,在表中更新
- 同时根据row的信息更新index中的key

```
bool UpdateExecutor::Next([[maybe_unused]] Row *row, RowId *rid) {
  Row childRow;
  RowId childRowId;
  if(!child_executor_->Next(&childRow,&childRowId))return false;
    Row newrow = GenerateUpdatedTuple(childRow);

    table_info_->GetTableHeap()->UpdateTuple(newrow,childRow.GetRowId(),
exec_ctx_->GetTransaction());
    //更新所有index，删除key后插入key
    for(auto itr = index_info_vec_.begin();itr!=index_info_vec_.end();itr++){
      auto keySchema = (*itr)->GetIndexKeySchema();
```

```
      vector<Field>oldFields,newFields;
      for( auto col : keySchema->GetColumns() ){
        uint32_t col_index;
        table_info_->GetSchema()->GetColumnIndex(col->GetName(),col_index);
        oldFields.push_back(*(childRow.GetField(col_index)));
        newFields.push_back(*(newrow.GetField(col_index)));
      }
      Row oldkey(oldFields),newkey(newFields);
      (*itr)->GetIndex()->RemoveEntry(oldkey,childRowId, exec_ctx_-
>GetTransaction());
      (*itr)->GetIndex()->InsertEntry(newkey,newrow.GetRowId(), exec_ctx_-
>GetTransaction());
    }
    return true;
}
```

### UpdateExecutor::GenerateUpdatedTuple(const Row &src_row)

- 生成更新的row

**实现思路：**

- 返回newrow

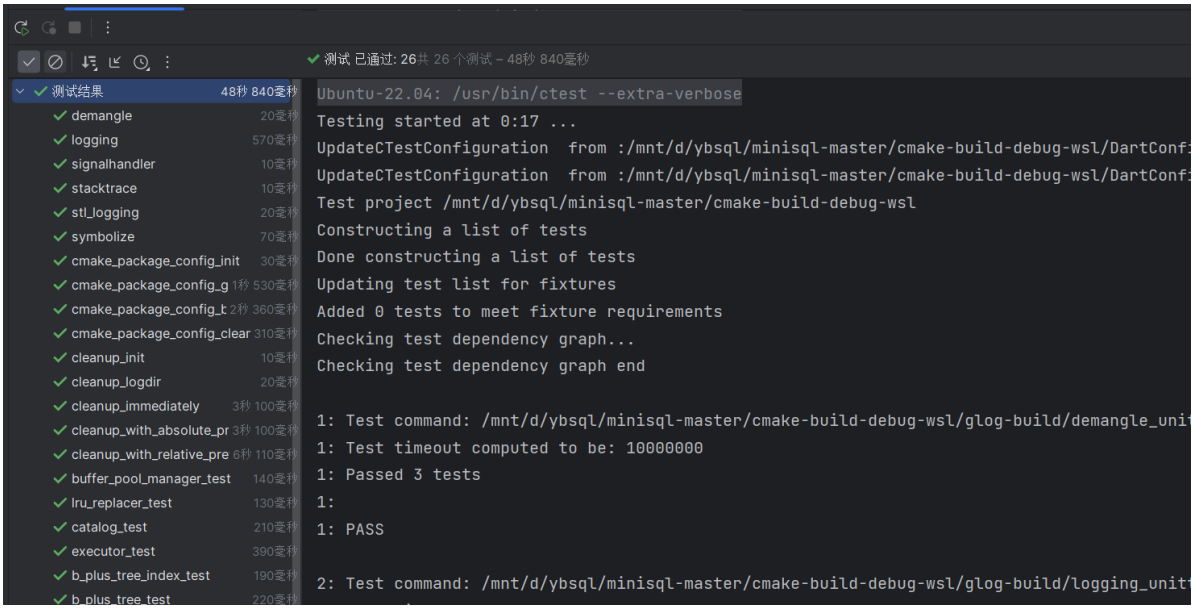
```
Row UpdateExecutor::GenerateUpdatedTuple(const Row &src_row) {
  Row newrow(src_row);

  for(auto itr = plan_->update_attrs_.begin();itr!=plan_-
>update_attrs_.end();itr++){
    Field newField = (*itr).second->Evaluate(&src_row);
    newrow.GetField((*itr).first)->operator=(newField);
  }
  return newrow;
}
```

至此，左右模块均已分析完毕

# 五、验收验证-正确性测试、性能测试

## Test全部通过




# 性能测试

测试代码

```
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 "../../sql_gen/account00.txt";
execfile "../../sql_gen/account01.txt";
execfile "../../sql_gen/account02.txt";
select * from account;

select * from account where id = 12502345;
select * from account where balance = 181.259995;
select * from account where name = "name26789";
select * from account where id <> 12509999;
select * from account where balance <> 86.269997;
select * from account where name <> "name09999";

select id, name from account where balance >= 990 and balance < 3000;
select name, balance from account where balance > 1000 and id <= 12529999;
select * from account where id < 12515000 and name > "name14500";
select * from account where id < 12500200 and name < "name00100";

insert into account values(12509999,"name99999",8.1);

create index idx01 on account(name);
select * from account where name = "name26789";
select * from account where name = "name45678";
```

```
select * from account where id < 12500200 and name < "name00100";
delete from account where name = "name25678";
insert into account values(12525678, "name25678", 880.67);
drop index idx01;

update account set id =12529999 where name = "name29999";

delete from account where balance = 123123.123;
delete from account;
drop table account;
```

测试中途截图：

```
| 12529984 | name29984 | 988.630005 |
| 12529985 | name29985 | 474.440002 |
| 12529986 | name29986 | 568.690002 |
| 12529987 | name29987 | 833.039978 |
| 12529988 | name29988 | 491.420013 |
| 12529989 | name29989 | 592.419983 |
| 12529997 | name29997 | 529.530029 |
| 12529998 | name29998 | 94.589996  |
| 12529999 | name29999 | 8.250000   |
+----------+-----------+------------+
30000 row in set(0.2120 sec).
minisql >
```

```
minisql > select * from account where id = 12502345;
[INFO] Sql syntax parse ok!
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12502345 | name02345 | 433.829987 |
+----------+-----------+------------+
1 row in set(0.0000 sec).
```

```
minisql > select * from account where balance = 181.259995;
[INFO] Sql syntax parse ok!
+----------+-----------+------------+
| id       | name      | balance    |
+----------+-----------+------------+
| 12509995 | name09995 | 181.259995 |
| 12518676 | name18676 | 181.259995 |
+----------+-----------+------------+
2 row in set(0.0730 sec).
minisql >
```

```
minisql > create index idx01 on account(name);
[INFO] Sql syntax parse ok!
minisql >
```

```
minisql > select * from account where name = "name26789";
[INFO] Sql syntax parse ok!
+----------+----------+------------+
| id       | name     | balance    |
+----------+----------+------------+
| 12526789 | name26789 | 665.830017 |
+----------+----------+------------+
1 row in set(0.0000 sec).
minisql > drop index idx01;
[INFO] Sql syntax parse ok!
minisql >
```

```
minisql > update account set id =12529999 where name = "name29999";
[INFO] Sql syntax parse ok!
Query OK, 1 row affected(0.0790 sec).
minisql >
```

```
minisql > delete from account where balance = 123123.123;
[INFO] Sql syntax parse ok!
Query OK, 0 row affected(0.0740 sec).
minisql >
```

```
minisql > delete from account;
[INFO] Sql syntax parse ok!
Query OK, 30000 row affected(1.5400 sec).
minisql >
```