浙江大學
ZHEJIANG UNIVERSITY

数据库系统

minisql 个人报告

3220103422 刘韬

2024 年 6 月 14 日

目录

个人完成部分

Lab1 及 bonus，Lab2，Lab3，系统整体功能调试。完善了 Lab2，Lab3 的测试用例。

1 DISK AND BUFFER POOL MANAGER

1.1 Disk Manager summary

为了管理数据库中的文件，我们需要实现一个对磁盘进行读写的类 DiskManager，它的功能是对数据中的 page（基本数据单元）进行 allocation 和 de allocation，执行从磁盘中进行读写 page 的操作。除了存储数据的数据页，我们还需要一些元数据来帮助和加速数据页的管理，于是位图页就自然的出现了。我们将一个位图页和它所管理的一段连续的数据页称作数据库文件中的一个分区（extent）。为了提高存储的容量，我们使用一个额外的元信息页来存储各个分区的信息。这种套娃的方式可以使磁盘文件能够维护更多的数据页信息。下面我们依次实现 DiskManager 中的各个部分，并对其进行测试。

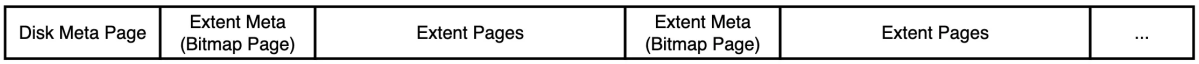

图 1: Disk data

1.2 Bitmap page

首先是位图页的实现。位图页的大小与数据页的大小一致，如2所示，位图页由 Bitmap Page Meta 和 Bitmap Content 两部分组成。Bitmap Page Meta 就是位图页的元数据，含当前已经分配的页的数量（page_allocated_）以及下一个空闲的数据页 (next_free_page_)。Bitmap Content 使用 char 数组来记录数据，其容量大小是 `MAX_CHARS = PageSize - 2 * sizeof(uint32_t)`，每个 bit 的含义是对应的页是否被分配，一个 char 的大小是 8bits，位图页对外提供分配、回收数据页和判断数据页是否可用的接口，内部实现是对上述数组的管理，以下是各个接口的具体实现。

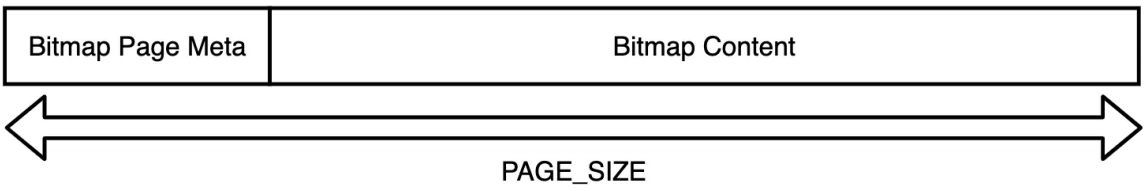

图 2: Bitmap page

AllocatePage 设置相应的next_free_page，并更新free_index来利用离散空间，返回一个 bool 值来说明是否分配成功，并将分配的页号返回。

```
template <size_t PageSize>
bool BitmapPage<PageSize>::AllocatePage(uint32_t &page_offset) {
  if(IsPageFree(next_free_page_)){
    bytes[next_free_page_ / 8] |= (1 << (next_free_page_ % 8)); // 把指定位设为 1
    page_offset = next_free_page_;
    page_allocated_++;
    uint32_t free_index = 0;
    while (!IsPageFree(free_index) && free_index < GetMaxSupportedSize() - 1)
    {
      free_index++;
    }
    next_free_page_ = free_index;
    return true;
  }
  return false;
}
```

DeAllocatePage DeAllocatePage 与 AllocatePage 相对，将输入的 page_offset 释放，更新 bitmap page 的信息，并更新 next_free_page。

```
template <size_t PageSize>
bool BitmapPage<PageSize>::DeAllocatePage(uint32_t page_offset) {
  if(IsPageFree(page_offset)){
    return false;
  }
  bytes[page_offset / 8] &= ~(1 << (page_offset % 8)); // 把指定位设为 0
  page_allocated_--;
  if(page_offset < next_free_page_){
    next_free_page_ = page_offset;
  }
  return true;
}
```

IsPageFree 访问位图页信息，找到对应的 bit 位，根据信息判断是否空闲。需要注意的是应当判断 page_offset 的有效性。

```
template <size_t PageSize>
bool BitmapPage<PageSize>::IsPageFree(uint32_t page_offset) const {
```

```
  if (page_offset >= GetMaxSupportedSize())
  {
    LOG(ERROR) << "page_offset is out of range";
    return false;
  }
  return IsPageFreeLow(page_offset / 8, page_offset % 8);
}

template <size_t PageSize>
bool BitmapPage<PageSize>::IsPageFreeLow(uint32_t byte_index, uint8_t bit_index)
↪  const {
  return (bytes[byte_index] & (1 << bit_index)) == 0;
}
```

1.3 Disk Manager

DiskManager 具有向磁盘读写物理页，分配和回收页的功能。为了更好地管理数据，我们在前面设置了元数据页，但对于上层的程序而言，数据管理器提供的应当是连续的数据页，即逻辑页号。因此我们需要在这里对逻辑页和物理页进行映射，使得上层能够无感知地获得数据页。映射示意图3

物理页号	0	1	2	3	4	5	6	...
职责	磁盘元数据	位图页	数据页	数据页	数据页	位图页	数据页	
逻辑页号	/	/	0	1	2		3	

图 3: phy-log

MapPageId 对输入的逻辑页号返回其物理页号，按照图3的映射完成。这个函数将在物理页的读写中被调用。

```
page_id_t DiskManager::MapPageId(page_id_t logical_page_id) {
  return 1 + logical_page_id / BITMAP_SIZE * (BITMAP_SIZE + 1) + 1 + logical_page_id
  ↪  % BITMAP_SIZE;
}
```

IsPageFree 根据逻辑页号找出该数据页对应的位图页的物理页号，调用位图页的函数来判断是否 free, 需要考虑计算出的物理页号是否有效，如果是无效的超出已经分配的范围，那直接返回 true 即可。

```cpp
bool DiskManager::IsPageFree(page_id_t logical_page_id) {
  BitmapPage<PAGE_SIZE>* bitmap_page = new BitmapPage<PAGE_SIZE>();
  uint32_t extent_id = logical_page_id / BITMAP_SIZE;
  uint32_t page_offset = logical_page_id % BITMAP_SIZE;
  page_id_t bitmap_page_id = 1 + extent_id * (BITMAP_SIZE + 1);
  // if the extent_id is larger than the number of extents, then the page is free
  if (extent_id >= reinterpret_cast<DiskFileMetaPage*>(meta_data_)->num_extents_)
  {
    return true;
  }
  ReadPhysicalPage(bitmap_page_id,reinterpret_cast<char*>(bitmap_page));
  bool is_free = bitmap_page->IsPageFree(page_offset);
  delete bitmap_page;
  return is_free;
}
```

AllocatePage 这是磁盘管理器的分配页实现，思路是先找到一个空闲的页，在磁盘元数据页的定义中我们保留了已经分配的页数，总共的 extent 数以及每个 extent 中已经被使用的页数。那么在读取磁盘中的元数据页后，我们通过访问每个 extent 中已经被使用页数的数组可以找到目前已经分配好的 extent 中是否存在空闲页，然后调用位图页的 allocate page 函数进行分配。如果目前所有的页都已经被分配，说明我们需要再新建一个 extent，然后分配页。同时需要更新元数据页中的数据，并写回磁盘，具体实现如下：

```cpp
page_id_t DiskManager::AllocatePage() {
  // read meta page
  DiskFileMetaPage* meta_page = reinterpret_cast<DiskFileMetaPage*>(meta_data_);
  if(meta_page->num_allocated_pages_ == MAX_VALID_PAGE_ID){
    return INVALID_PAGE_ID;
  }
  // find the first free page
  // if there is free page in existed page
  for(uint32_t i = 0;i < meta_page->num_extents_;i++){
    if(meta_page->GetExtentUsedPage(i) < BITMAP_SIZE){
      BitmapPage<PAGE_SIZE>* bitmap_page = new BitmapPage<PAGE_SIZE>();
      ReadPhysicalPage(1 + i * (BITMAP_SIZE +
      ↪ 1),reinterpret_cast<char*>(bitmap_page));
      uint32_t page_offset;
      bool res = bitmap_page->AllocatePage(page_offset);
      ASSERT(res,"Allocate page failed");
      if(res){
        meta_page->num_allocated_pages_++;
```

```
        meta_page->extent_used_page_[i]++;
        WritePhysicalPage(1 + i * (BITMAP_SIZE +
        ↪  1),reinterpret_cast<char*>(bitmap_page));
        return i * BITMAP_SIZE + page_offset;
      }
    }
  }
  // create a new extent
  BitmapPage<PAGE_SIZE>* bitmap_page = new BitmapPage<PAGE_SIZE>();
  uint32_t page_offset = 0;
  bitmap_page->AllocatePage(page_offset);
  WritePhysicalPage(1 + meta_page->num_extents_ * (BITMAP_SIZE +
  ↪  1),reinterpret_cast<char*>(bitmap_page));
  meta_page->num_allocated_pages_++;
  meta_page->extent_used_page_[meta_page->num_extents_]++;
  meta_page->num_extents_++;
  return (meta_page->num_extents_ - 1) * BITMAP_SIZE + page_offset;
}
```

DeAllocatePage 这部分要做的事情是找到相应的数据页，调用 Bitmap page 的 DeAllocatePage，更新元数据，写回磁盘，具体实现如下：

```
void DiskManager::DeAllocatePage(page_id_t logical_page_id) {
  uint32_t extent_id = logical_page_id / BITMAP_SIZE;
  page_id_t bitmap_page_id = 1 + extent_id * (BITMAP_SIZE + 1);

  DiskFileMetaPage* meta_page = reinterpret_cast<DiskFileMetaPage*>(meta_data_);
  BitmapPage<PAGE_SIZE>* bitmap_page = new BitmapPage<PAGE_SIZE>();

  ReadPhysicalPage(bitmap_page_id,reinterpret_cast<char*>(bitmap_page));
  uint32_t page_offset = logical_page_id % BITMAP_SIZE;
  bool res = bitmap_page->DeAllocatePage(page_offset);
  if(res){
    meta_page->num_allocated_pages_--;
    meta_page->extent_used_page_[extent_id]--;
    WritePhysicalPage(bitmap_page_id,reinterpret_cast<char*>(bitmap_page));
  }else{
    LOG(ERROR) << "Deallocate page failed." << logical_page_id;
  }
}
```

1.4 Buffer pool manager

Buffer pool manager 是提供给上层调用，用于数据读写管理的接口。功能需求是

1. `Page *FetchPage(page_id_t 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；
2. `BufferPoolManager::NewPage(&page_id)`: 分配一个新的数据页，并将逻辑页号于`page_id`中返回；
3. `BufferPoolManager::UnpinPage(page_id, is_dirty)`: 取消固定一个数据页；
4. `BufferPoolManager::FlushPage(page_id)`: 将数据页转储到磁盘中；
5. `BufferPoolManager::DeletePage(page_id)`: 释放一个数据页；

1.4.1 Buffer pool replacement algorithm

为了保证在 buffer pool 没有空闲页的情况下可以正常的使用，我们需要一个替换单元，这里使用 LRU 替换算法来决定哪一页被替换，同时还实现了 Bonus: Clock Replacer 并为其实现测试用例。以下是两种替换算法的实现：

Least Recently Used Replacer LRU 替换算法是将最近最少被访问的元素作为被替代的对象，因此实现时我们可以通过一个有序列表，将每次使用后的都换到列表的头部，那么列表的尾部就是我们所需要的对象。我们使用了 list 来存保证有序性，用 unordered_map 来加快查询。

Victim 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；

```
bool LRUReplacer::Victim(frame_id_t *frame_id) {
  if (lru_list_.empty()) {
    return false;
  }
  *frame_id = lru_list_.back();
  lru_list_.pop_back();
  lru_list_map_.erase(*frame_id);
  return true;
}
```

Pin 将数据页固定使之不能被 Replacer 替换，即从 lru_list_ 中移除该数据页对应的页帧。

```
void LRUReplacer::Pin(frame_id_t frame_id) {
  auto it = lru_list_map_.find(frame_id);
  if(it != lru_list_map_.end()){
    lru_list_.erase(it->second);
    lru_list_map_.erase(it);
  }
}
```

UnPin 将数据页解除固定，放入 lru_list_ 中，使之可以在必要时被 Replacer 替换掉。

```
void LRUReplacer::Unpin(frame_id_t frame_id) {
  auto it = lru_list_map_.find(frame_id);
  if(it == lru_list_map_.end()){
    lru_list_.push_front(frame_id);
    lru_list_map_[frame_id] = lru_list_.begin();
  }
}
```

Clock Replacer 这种替换方式采用标记，对每个元素进行标记，然后每次选择替换时进行一次遍历，对找到的每个元素查看标记，如果标记是 1 就变为 0，如果标记是 0 就替换出，每次查询是上次结束的位置开始，对新入队的元素标记是 1。类的成员变量有 list 用于存储替换页和 map 用于存储标记状态。

Victim 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；

```
bool CLOCKReplacer::Victim(frame_id_t *frame_id) {
  // if the clock_list is empty, return false
  if (clock_list.empty()) {
    return false;
  }
  // find the first frame that is not pinned
  while (clock_status[clock_list.front()] == 1) {
    clock_list.push_back(clock_list.front());
    clock_list.pop_front();
    clock_status[clock_list.back()] = 0;
```

```
  }
  // return the frame_id
  *frame_id = clock_list.front();
  clock_list.pop_front();
  clock_status.erase(*frame_id);
  return true;
}
```

Pin 将数据页固定使之不能被 Replacer 替换，即从 lru_list_ 中移除该数据页对应的页帧。

```
void CLOCKReplacer::Pin(frame_id_t frame_id) {
  // if the frame is in the clock_list, then remove it from the clock_list
  if (clock_status.find(frame_id) != clock_status.end()) {
    clock_list.remove(frame_id);
    clock_status.erase(frame_id);
  }
}

```

1.4.2 buffer pool manager implementation

UnPin 将数据页解除固定，放入 lru_list_ 中，使之可以在必要时被 Replacer 替换掉。

```
void CLOCKReplacer::Unpin(frame_id_t frame_id) {
  // if the frame is not in the clock_list, then add it to the clock_list
  if (clock_status.find(frame_id) == clock_status.end()) {
    clock_list.push_back(frame_id);
    clock_status[frame_id] = 1;
  }else{
    clock_status[frame_id] = 1;
  }
}
```

FetchPage 首先尝试在 page table 中找到要求的 page(P)，如果存在就固定它并立刻返回这个 Page；如果不存在则需要从 free list（优先）或者 replacer 中找到一个替换页(R)。如果 R 是脏页，就应当先写回磁盘保证数据的一致性，然后把 R 从 page table 中删掉并且插入 P，并更新 P 的元数据，从磁盘中读出数据即可。

```cpp
Page *BufferPoolManager::FetchPage(page_id_t page_id) {
  // 1.     Search the page table for the requested page (P).
  frame_id_t frame_id = INVALID_FRAME_ID;
  if(page_table_.find(page_id) != page_table_.end()) {
    // 1.1    If P exists, pin it and return it immediately.
    frame_id = page_table_[page_id];
    pages_[frame_id].pin_count_++;
    replacer_->Pin(frame_id);
    return &pages_[frame_id];
  }else{
    // 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(!free_list_.empty()){
      frame_id = free_list_.front();
      free_list_.pop_front();
    }else{
      // get from replacer
      if(!replacer_->Victim(&frame_id)){
        return nullptr;
      }
    }
    // 2.     If R is dirty, write it back to the disk.
    if(pages_[frame_id].IsDirty()){
      FlushPage(pages_[frame_id].GetPageId());
    }
    // 3.     Delete R from the page table and insert P.
    page_table_.erase(pages_[frame_id].GetPageId());
    page_table_[page_id] = frame_id;
    // 4.     Update P's metadata, read in the page content from disk, and then
    ↪  return a pointer to P.
    pages_[frame_id].pin_count_ = 1;
    pages_[frame_id].page_id_ = page_id;
    pages_[frame_id].ResetMemory();
    disk_manager_->ReadPage(page_id,pages_[frame_id].GetData());
    replacer_->Pin(frame_id);
    return &pages_[frame_id];
  }
}
```

NewPage 如果已经没有空闲页（free list 和 replacer 都没有了），就返回空指针；如果还有就取出一个，然后更新元数据，清空页面的原数据，固定并返回 page。

```cpp
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.
  if(free_list_.empty() && replacer_->Size() == 0){
    return nullptr;
  }
  // 2.   Pick a victim page P from either the free list or the replacer. Always pick from the free list first.
  frame_id_t frame_id = INVALID_FRAME_ID;
  if(!free_list_.empty()){
    frame_id = free_list_.front();
    free_list_.pop_front();
  }else{
    if(!replacer_->Victim(&frame_id)){
      return nullptr;
    }
  }
  // if the page get from replacer, we should erase it and get space for the new page.
  page_table_.erase(pages_[frame_id].GetPageId());
  // 3.   Update P's metadata, zero out memory and add P to the page table.
  page_id = AllocatePage();
  pages_[frame_id].page_id_ = page_id;
  pages_[frame_id].pin_count_ = 1;
  pages_[frame_id].is_dirty_ = false;
  pages_[frame_id].ResetMemory();
  page_table_[page_id] = frame_id;
  replacer_->Pin(frame_id);
  // 4.   Set the page ID output parameter. Return a pointer to P.
  return &pages_[frame_id];
}
```

DeletePage 找到需要删除的 page，如果不存在返回 true，如果存在但是仍然被固定，返回 false，除此以外这个 page 可以被删除，从 table 中删除，reset 数据，放回 free list。

```cpp
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.
  if(page_table_.find(page_id) == page_table_.end()){
    return true;
```

```
  }else{
  // 2.   If P exists, but has a non-zero pin-count, return false. Someone is using
  ↪  the page.
    frame_id_t frame_id = page_table_[page_id];
    if(pages_[frame_id].GetPinCount() != 0){
      return false;
    }else{
  // 3.   Otherwise, P can be deleted. Remove P from the page table, reset its
  ↪  metadata and return it to the free list.
      page_table_.erase(page_id);
      DeallocatePage(page_id);
      pages_[frame_id].ResetMemory();
      pages_[frame_id].page_id_ = INVALID_PAGE_ID;
      pages_[frame_id].pin_count_ = 0;
      pages_[frame_id].is_dirty_ = false;
      free_list_.push_back(frame_id);
      replacer_->Pin(frame_id);
      return true;
    }
  }
}
```

UnpinPage 在 pagetable 中找到这个页，对是否要写回磁盘进行标记，然后将 pin_count-1，如果 pin 数为 0 就把它从 replacer 中 unpin 掉。

```
bool BufferPoolManager::UnpinPage(page_id_t page_id, bool is_dirty) {
  if (page_table_.find(page_id) == page_table_.end()){
    return true;
  }
  frame_id_t frame_id = page_table_[page_id];
  if(pages_[frame_id].pin_count_ > 0){
    pages_[frame_id].pin_count_ --;
  }
  pages_[frame_id].is_dirty_ |= is_dirty;
  if(pages_[frame_id].pin_count_ == 0){
    replacer_->Unpin(frame_id);
  }
  return true;
}
```

FlushPage 找到 page，将其写回磁盘，无论是否 pin。这是为了保证磁盘数据的一致性。

```
bool BufferPoolManager::FlushPage(page_id_t page_id) {
  if(page_table_.find(page_id) == page_table_.end()){
    return false;
  }else{
    frame_id_t frame_id = page_table_[page_id];
    disk_manager_->WritePage(page_id,pages_[frame_id].GetData());
    pages_[frame_id].is_dirty_ = false;
    return true;
  }
}
```

1.5 test

使用所给的测试用例进行测试，增加了 clock replacer 的测试，其测试与 LRU 测试相同。测试结果如下:

本节的测试全都通过，包括 bonus 的测试。

```
└> ./test/disk_manager_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from DiskManagerTest
[ RUN      ] DiskManagerTest.BitMapPageTest
[       OK ] DiskManagerTest.BitMapPageTest (162 ms)
[ RUN      ] DiskManagerTest.FreePageAllocationTest
[       OK ] DiskManagerTest.FreePageAllocationTest (24502 ms)
[----------] 2 tests from DiskManagerTest (24664 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (24664 ms total)
[  PASSED  ] 2 tests.
```

图 4: test1

```
└> ./test/lru_replacer_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from LRUReplacerTest
[ RUN      ] LRUReplacerTest.SampleTest
[       OK ] LRUReplacerTest.SampleTest (0 ms)
[----------] 1 test from LRUReplacerTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

图 5: test2

```
└> ./test/clock_replacer_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from CLOCKReplacerTest
[ RUN      ] CLOCKReplacerTest.SampleTest
[       OK ] CLOCKReplacerTest.SampleTest (0 ms)
[----------] 1 test from CLOCKReplacerTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

图 6: test3

```
└> ./test/buffer_pool_manager_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from BufferPoolManagerTest
[ RUN      ] BufferPoolManagerTest.BinaryDataTest
[       OK ] BufferPoolManagerTest.BinaryDataTest (7 ms)
[----------] 1 test from BufferPoolManagerTest (7 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (7 ms total)
[  PASSED  ] 1 test.
```

图 7: test4

2 RECORD MANAGER

2.1 Record design

我们的记录包含以下几个概念：

- **Column**：用于定义和表示数据表中的某一个字段，即包含了这个字段的字段名、字段类型、是否唯一等等；
- **Schema**：用于表示一个数据表或是一个索引的结构。一个 Schema 由一个或多个的 Column 构成；
- **Field**：它对应于一条记录中某一个字段的数据信息，如存储数据的数据类型，是否是空，存储数据的值等等；
- **Row**：用于表示一条记录，即一行数据，由多个 Field 组成；

这是我们数据在数据库中的组织方式，但是为了将其持久化地存储在硬盘中，我们需要使用序列化和反序列化地方法，将其转化为字节流。下面的章节介绍了我们如何进行序列化和反序列化。

2.2 Record Persistence

2.2.1 Column

Column 的结构是 |COLUMN_MAGIC_NUM 4 | name_ length 4 | name.strlen() | | type_ 4 | len_ 4 | table_ind_ 4 | nullable_ 1 | unique_ 1 |

Serialize 逐个写入每个字段，然后将其转化为字节流。

```
uint32_t Column::SerializeTo(char *buf) const {
  uint32_t offset = 0;
  memcpy(buf + offset, &COLUMN_MAGIC_NUM, sizeof(COLUMN_MAGIC_NUM));
  offset += sizeof(COLUMN_MAGIC_NUM);
  uint32_t name_len = name_.length();
  memcpy(buf + offset, &name_len, sizeof(name_len));
  offset += sizeof(name_len);
  memcpy(buf + offset, name_.c_str(), name_len);
  offset += name_len;
  memcpy(buf + offset, &type_, sizeof(type_));
  offset += sizeof(type_);
  memcpy(buf + offset, &len_, sizeof(len_));
  offset += sizeof(len_);
  memcpy(buf + offset, &table_ind_, sizeof(table_ind_));
```

```
  offset += sizeof(table_ind_);
  memcpy(buf + offset, &nullable_, sizeof(nullable_));
  offset += sizeof(nullable_);
  memcpy(buf + offset, &unique_, sizeof(unique_));
  offset += sizeof(unique_);
  return offset;
}
```

Deserialize 逐个读取每个字段，然后将其转化为 Column。

```
uint32_t Column::DeserializeFrom(char *buf, Column *&column) {
  if (column != nullptr) {
    LOG(WARNING) << "Pointer to column is not null in column deserialize." <<
    ↪  std::endl;
  }
  uint32_t offset = 0;
  uint32_t magic_num;
  memcpy(&magic_num, buf + offset, sizeof(magic_num));
  offset += sizeof(magic_num);
  ASSERT(magic_num == COLUMN_MAGIC_NUM, "Invalid magic number.");
  if (magic_num != COLUMN_MAGIC_NUM) {
    return 0;
  }
  uint32_t name_len;
  memcpy(&name_len, buf + offset, sizeof(name_len));
  offset += sizeof(name_len);
  std::string name(buf + offset, name_len);
  offset += name_len;
  TypeId type;
  memcpy(&type, buf + offset, sizeof(type));
  offset += sizeof(type);
  uint32_t len;
  memcpy(&len, buf + offset, sizeof(len));
  offset += sizeof(len);
  uint32_t table_ind;
  memcpy(&table_ind, buf + offset, sizeof(table_ind));
  offset += sizeof(table_ind);
  bool nullable;
  memcpy(&nullable, buf + offset, sizeof(nullable));
  offset += sizeof(nullable);
  bool unique;
  memcpy(&unique, buf + offset, sizeof(unique));
  offset += sizeof(unique);
```

```
  if (type == TypeId::kTypeChar) {
    column = new Column(name, type, len, table_ind, nullable, unique);
  } else {
    column = new Column(name, type, table_ind, nullable, unique);
  }
  return offset;
}
```

2.2.2 Schema

Schema 的结构是 SCHEMA_MAGIC_NUM 4 | column size 4 | column1 | column2 | ... ||

Serialize 逐个写入每个字段，然后将其转化为字节流。

```
uint32_t Schema::SerializeTo(char *buf) const {
  uint32_t offset = 0;
  memcpy(buf + offset, &SCHEMA_MAGIC_NUM, sizeof(SCHEMA_MAGIC_NUM));
  offset += sizeof(SCHEMA_MAGIC_NUM);
  uint32_t column_size = columns_.size();
  memcpy(buf + offset, &column_size, sizeof(column_size));
  offset += sizeof(column_size);
  for (auto column : columns_) {
    offset += column->SerializeTo(buf + offset);
  }
  return offset;
}
```

Deserialize 逐个读取每个字段，然后将其转化为 Schema。

```
uint32_t Schema::DeserializeFrom(char *buf, Schema *&schema) {
  uint32_t offset = 0;
  uint32_t magic_num;
  memcpy(&magic_num, buf + offset, sizeof(magic_num));
  offset += sizeof(magic_num);
  ASSERT(magic_num == SCHEMA_MAGIC_NUM, "Invalid magic number.");
  if (magic_num != SCHEMA_MAGIC_NUM) {
    return 0;
  }
  uint32_t column_size;
  memcpy(&column_size, buf + offset, sizeof(column_size));
```

```
  offset += sizeof(column_size);
  std::vector<Column *> columns;
  for (uint32_t i = 0; i < column_size; i++) {
    Column *temp = nullptr;
    offset += Column::DeserializeFrom(buf + offset, temp);
    columns.emplace_back(temp);
  }
  schema = new Schema(columns, true);
  return offset;
}
```

2.2.3 Row

对于 Row 类型对象的序列化，对于为 null 的 Field，可以通过位图的方式标记 (即 Null Bitmaps)；对于 Row 类型对象的反序列化，在反序列化每一个 Field 时，反序列化出来的 Field 的内存都由该 Row 对象维护。

Serialize 写入长度 num，写入位图页，然后写入每一个 Field。

```
uint32_t Schema::SerializeTo(char *buf) const {
  // | SCHEMA_MAGIC_NUM 4 | column size 4 | column1 | column2 | ... |
  uint32_t offset = 0;
  memcpy(buf + offset, &SCHEMA_MAGIC_NUM, sizeof(SCHEMA_MAGIC_NUM));
  offset += sizeof(SCHEMA_MAGIC_NUM);
  uint32_t column_size = columns_.size();
  memcpy(buf + offset, &column_size, sizeof(column_size));
  offset += sizeof(column_size);
  for (auto column : columns_) {
    offset += column->SerializeTo(buf + offset);
  }
  return offset;
}
```

Deserialize 读取长度 num，读取位图页，然后读取每一个 Field。

```
uint32_t Schema::DeserializeFrom(char *buf, Schema *&schema) {
  uint32_t offset = 0;
  uint32_t magic_num;
  memcpy(&magic_num, buf + offset, sizeof(magic_num));
  offset += sizeof(magic_num);
  ASSERT(magic_num == SCHEMA_MAGIC_NUM, "Invalid magic number.");
```

```
  if (magic_num != SCHEMA_MAGIC_NUM) {
    return 0;
  }
  uint32_t column_size;
  memcpy(&column_size, buf + offset, sizeof(column_size));
  offset += sizeof(column_size);
  std::vector<Column *> columns;
  for (uint32_t i = 0; i < column_size; i++) {
    Column *temp = nullptr;
    offset += Column::DeserializeFrom(buf + offset, temp);
    columns.emplace_back(temp);
  }
  schema = new Schema(columns, true);
  return offset;
}
```

2.3 table heap

对于数据表中的每一行记录，都有一个唯一标识符 RowId 与之对应。RowId 同时具有逻辑和物理意义，在物理意义上，它是一个 64 位整数，是每行记录的唯一标识；RowId 的作用主要体现在两个方面：一是在索引中，叶结点中存储的键值对是索引键 Key 到 RowId 的映射，通过索引键 Key，沿着索引查找，我们能够得到该索引键对应记录的 RowId，也就能够在堆表中定位到该记录；二是在堆表中，借助 RowId 中存储的逻辑信息（page_id 和 slot_num），可以快速地定位到其对应的记录位于物理文件的哪个位置。而在逻辑意义上，它的高 32 位存储的是该 RowId 对应记录所在数据页的 page_id，低 32 位存储的是该 RowId 在 page_id 对应的数据页中对应的是第几条记录。我们可以通过 RowId 的逻辑信息，得到该 Row 的数据。

我们需要一个 table heap 来存储数据表中的记录，table heap 的结构如下：堆表中的每个数据页都由表头（Table Page Header）、空闲空间（Free Space）和已经插入的数据（Inserted Tuples）三部分组成，表头在页中从左往右扩展，记录了 PrevPageId、NextPageId、FreeSpacePointer 以及每条记录在 TablePage 中的偏移和长度；插入的记录在页中从右向左扩展，每次插入记录时会将 FreeSpacePointer 的位置向左移动。

2.3.1 table heap implementation

实现堆表的插入、删除、查询等操作，具体实现如下：

InsertTuple 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_）；我们首先判断插入的记录是否能够放入当前页中，如果超出当前页的空间限制，我们需要将记录插入到下一个页中。

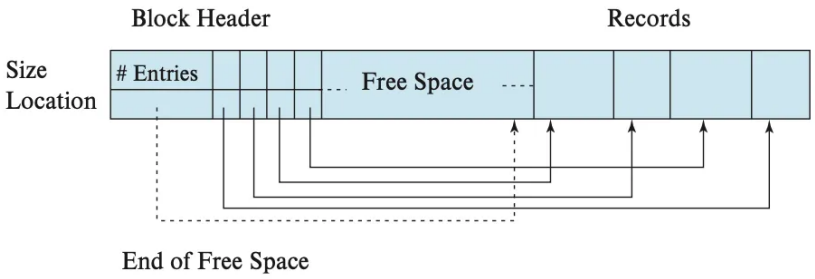

Figure 13.6 Slotted-page structure.

图 8: Table Page

```cpp
bool TableHeap::InsertTuple(Row &row, Txn *txn) {
  if(row.GetSerializedSize(schema_) >= PAGE_SIZE) return false;
  auto page = reinterpret_cast<TablePage
  ↪  *>(buffer_pool_manager_->FetchPage(GetFirstPageId()));
  if (page == nullptr) {
    return false;
  }
  while(!page->InsertTuple(row, schema_, txn, lock_manager_, log_manager_)){
    buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
    page_id_t next_page_id = page->GetNextPageId();
    if(next_page_id == INVALID_PAGE_ID){
      auto new_page = reinterpret_cast<TablePage
      ↪  *>(buffer_pool_manager_->NewPage(next_page_id));
      page->SetNextPageId(next_page_id);
      new_page->Init(next_page_id, page->GetTablePageId(), log_manager_, txn);
      buffer_pool_manager_->UnpinPage(page->GetPageId(), true);
      page = new_page;
    }else{
      auto next_page = reinterpret_cast<TablePage
      ↪  *>(buffer_pool_manager_->FetchPage(next_page_id));
      buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
      page = next_page;
    }
  }
  buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
  return true;
}
```

ApplyDelete 删除记录是采用先标记后统一执行输出的方式进行删除，这里实现的是执行删除的部分。

```
void TableHeap::ApplyDelete(const RowId &rid, Txn *txn) {
  // Step1: Find the page which contains the tuple.
  auto page = reinterpret_cast<TablePage
  ↪  *>(buffer_pool_manager_->FetchPage(rid.GetPageId()));
  ASSERT(page != nullptr, "The page could not be found.");
  // Step2: Delete the tuple from the page.
  page->ApplyDelete(rid, txn, log_manager_);
  buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
}
```

GetTuple 根据 Row 找到 page，然后根据 page 读出数据

```
bool TableHeap::GetTuple(Row *row, Txn *txn) {
  page_id_t page_id = row->GetRowId().GetPageId();
  auto page = reinterpret_cast<TablePage
  ↪  *>(buffer_pool_manager_->FetchPage(page_id));
  if (page == nullptr) {
    return false;
  }
  bool result = page->GetTuple(row, schema_, txn, lock_manager_);
  buffer_pool_manager_->UnpinPage(page_id, false);
  return result;
}
```

UpdateTuple 这一部分要根据 page 的 UpdateTuple 的返回值来判读，我在原来 page 的函数基础上修改了返回值，如果 slot_num 大于 tuple_count，说明 slot_num 是无效的，返回 1；如果 tuple_size 是删除的，返回 2；如果没有足够的空间更新，返回 3；如果成功，返回 0。修改后的函数如下：

```
uint32_t TablePage::UpdateTuple(Row &new_row, Row *old_row, Schema *schema, Txn *txn,
↪  LockManager *lock_manager,
LogManager *log_manager) {
  ASSERT(old_row != nullptr && old_row->GetRowId().Get() != INVALID_ROWID.Get(),
  ↪  "invalid old row.");
  uint32_t serialized_size = new_row.GetSerializedSize(schema);
  ASSERT(serialized_size > 0, "Can not have empty row.");
  uint32_t slot_num = old_row->GetRowId().GetSlotNum();
  // If the slot number is invalid, abort.
  if (slot_num >= GetTupleCount()) {
    return 1;
  }
```

```
  uint32_t tuple_size = GetTupleSize(slot_num);
  // If the tuple is deleted, abort.
  if (IsDeleted(tuple_size)) {
    return 2;
  }
  // If there is not enough space to update, we need to update via delete followed by
  ↪  an insert (not enough space).
  if (GetFreeSpaceRemaining() + tuple_size < serialized_size) {
    return 3;
  }
  ...
  // if success
  return 0;
}
```

在上面的三种失败情况后，如果是情况 3，我们需要先删除再插入，下面是具体的实现：

```
bool TableHeap::UpdateTuple(Row &row, const RowId &rid, Txn *txn) {
  // Find the page which contains the tuple.
  auto page = reinterpret_cast<TablePage
  ↪  *>(buffer_pool_manager_->FetchPage(rid.GetPageId()));
  if (page == nullptr) {
    return false;
  }
  Row old_row(rid);
  // Update the tuple in the page.
  int update_res = page->UpdateTuple(row, &old_row, schema_, txn, lock_manager_,
  ↪  log_manager_);
  if(update_res != 0){
    if(update_res == 3){
      if(InsertTuple(row, txn)){
        MarkDelete(rid, txn);
        buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
        return true;
      }else{
        buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
        return false;
      }
    }else{
      buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
      return false;
    }
  }
```

```
25    else{
26      row.SetRowId(rid);
27      buffer_pool_manager_->UnpinPage(page->GetTablePageId(), true);
28      return true;
29    }
30  }
```

2.3.2 table heap iterator implementation

实现迭代器，用于遍历 table heap 中的所有记录。在上层中提供 begin 和 end 函数，返回迭代器，然后通过迭代器的 next 函数来遍历所有记录。迭代器的成员变量是`TableHeap *table_heap_;RowId rid_;`，其中 table_heap 是指向 table heap 的指针，rid 是当前记录的 RowId。下面给出迭代器的实现：

TableIterator ctor 构造函数，初始化迭代器的成员变量。

```
TableIterator::TableIterator(TableHeap *table_heap, RowId rid, Txn *txn) {
  table_heap_ = table_heap;
  rid_ = rid;
  txn_ = txn;
}
```

TableIterator copy ctor 拷贝构造函数，用于拷贝迭代器。

```
TableIterator::TableIterator(const TableIterator &other) {
  table_heap_ = other.table_heap_;
  rid_ = other.rid_;
  txn_ = other.txn_;
}
```

TableIterator dtor 析构函数，用于释放迭代器的资源。

```
TableIterator::~TableIterator() {
  table_heap_ = nullptr;
  rid_ = RowId();
  txn_ = nullptr;
}
```

operator== == 运算符，用于判断两个迭代器是否相等。

```
bool TableIterator::operator==(const TableIterator &itr) const {
  return this->rid_ == itr.rid_;
}
```

operator!= != 运算符，用于判断两个迭代器是否不相等。

```
bool TableIterator::operator!=(const TableIterator &itr) const {
  return !(*this == itr);
}
```

operator* * 运算符，用于获取迭代器指向的记录。根据 RowwId 找到 page，然后根据 page 读出数据，返回 Row。

```
const Row &TableIterator::operator*() {
  auto page = reinterpret_cast<TablePage
  ↪  *>(table_heap_->buffer_pool_manager_->FetchPage(rid_.GetPageId()));
  if (page == nullptr) {
    table_heap_->buffer_pool_manager_->UnpinPage(rid_.GetPageId(), false);
    Row *row = new Row(INVALID_ROWID);
    return *row;
  }
  Row *row = new Row(rid_);
  bool res = page->GetTuple(row, table_heap_->schema_, txn_,
  ↪  table_heap_->lock_manager_);
  if(res){
    table_heap_->buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
    return *row;
  }else{
    table_heap_->buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
    Row *row = new Row(INVALID_ROWID);
    return *row;
  }
}
```

operator-> -> 运算符，用于获取迭代器指向的记录，返回 Row 的指针。

```
Row *TableIterator::operator->() {
  auto page = reinterpret_cast<TablePage
  ↪  *>(table_heap_->buffer_pool_manager_->FetchPage(rid_.GetPageId()));
  if (page == nullptr) {
    table_heap_->buffer_pool_manager_->UnpinPage(rid_.GetPageId(), false);
    return nullptr;
  }
  Row *row = new Row(rid_);
  bool res = page->GetTuple(row, table_heap_->schema_, txn_,
  ↪  table_heap_->lock_manager_);
  if(res){
    table_heap_->buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
    return row;
  }else{
    table_heap_->buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
    return nullptr;
  }
}
```

operator= = 运算符，用于赋值和初始化。

```
TableIterator &TableIterator::operator=(const TableIterator &itr) noexcept {
  table_heap_ = itr.table_heap_;
  rid_ = itr.rid_;
  txn_ = itr.txn_;
  return *this;
}
```

operator++//++iter ++ 运算符，用于迭代器自增，从 tableheap 中找到下一个记录，需要注意的是可能存在下一条记录在下一页的情况。

```
TableIterator &TableIterator::operator++() {
  auto page = reinterpret_cast<TablePage
  ↪  *>(table_heap_->buffer_pool_manager_->FetchPage(rid_.GetPageId()));
  if (page == nullptr) {
    table_heap_->buffer_pool_manager_->UnpinPage(rid_.GetPageId(), false);
    this->table_heap_ = nullptr;
    this->rid_ = RowId();
    this->txn_ = nullptr;
    return *this;
  }
```

```
  bool res = page->GetNextTupleRid(rid_, &rid_);
  if(res){
    // 不用翻下一页
    table_heap_->buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
    return *this;
  }else{
    // 翻下一页
    page_id_t next_page_id = page->GetNextPageId();
    if(next_page_id == INVALID_PAGE_ID){
      this->table_heap_ = nullptr;
      this->rid_ = RowId();
      this->txn_ = nullptr;
      return *this;
    }else{
      auto next_page = reinterpret_cast<TablePage
      ↪  *>(table_heap_->buffer_pool_manager_->FetchPage(next_page_id));
      table_heap_->buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
      page = next_page;
      page->GetFirstTupleRid(&rid_);
    }
  }
  table_heap_->buffer_pool_manager_->UnpinPage(page->GetPageId(), false);
  return *this;
}
```

operator++(int)//iter++　++ 运算符，用于迭代器自增，从 tableheap 中找到下一个记录，需要注意的是可能存在下一条记录在下一页的情况。

```
TableIterator TableIterator::operator++(int) {
  TableIterator p(*this);
  ++(*this);
  return TableIterator{p};
}
```

2.3.3 table heap iterator

这是迭代器给上层的接口，包括 begin 和 end 函数，用于返回迭代器。

begin　返回一个指向第一条记录的迭代器，使用 page 实现的 GetFirstTuple 函数。如果没找到，返回一个空迭代器。

```
TableIterator TableHeap::Begin(Txn *txn) {
  auto page = reinterpret_cast<TablePage
  ↪  *>(buffer_pool_manager_->FetchPage(GetFirstPageId()));
  if (page == nullptr) {
    buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
    return TableIterator(this, RowId(), nullptr);
  }
  RowId rid;
  if(page->GetFirstTupleRid(&rid)){
    buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
    return TableIterator(this, rid, txn);
  }else{
    buffer_pool_manager_->UnpinPage(page->GetTablePageId(), false);
    return TableIterator(this, RowId(), nullptr);
  }
}
```

end 返回 end 迭代器，应当是最后一个的下一个，即返回一个空迭代器。

```
TableIterator TableHeap::End() { return TableIterator(nullptr, RowId(), nullptr); }
```

2.4 test

这部分框架提供的测试用例太少了，只对 insert tuple 和 get tuple 进行了测试，为了全面地测试我们实现的 table heap 和 table heap iterator，我们自己编写了测试用例，测试了 iterator，update tuple and free 测试用例如下：

> 这部分代码是自己添加的测试代码，完善了框架的测试，希望能够加入到下一年的测试框架中。

```
    static string db_file_name = "table_heap_test.db";
using Fields = std::vector<Field>;

TEST(TableHeapTest, TableHeapSampleTest) {
  // init testing instance
  vector<RowId> row_ids;
  remove(db_file_name.c_str());
  auto disk_mgr_ = new DiskManager(db_file_name);
```

```
9   auto bpm_ = new BufferPoolManager(DEFAULT_BUFFER_POOL_SIZE, disk_mgr_);
10  const int row_nums = 10000;
11  // create schema
12  std::vector<Column *> columns = {new Column("id", TypeId::kTypeInt, 0, false,
    ↪  false),
13                                   new Column("name", TypeId::kTypeChar, 64, 1, true,
                                     ↪  false),
14                                   new Column("account", TypeId::kTypeFloat, 2, true,
                                     ↪  false)};
15  auto schema = std::make_shared<Schema>(columns);
16  // create rows
17  std::unordered_map<int64_t, Fields *> row_values;
18  uint32_t size = 0;
19  TableHeap *table_heap = TableHeap::Create(bpm_, schema.get(), nullptr, nullptr,
    ↪  nullptr);
20  for (int i = 0; i < row_nums; i++) {
21    int32_t len = RandomUtils::RandomInt(0, 64);
22    char *characters = new char[len];
23    RandomUtils::RandomString(characters, len);
24    Fields *fields =
25        new Fields{Field(TypeId::kTypeInt, i), Field(TypeId::kTypeChar,
          ↪  const_cast<char *>(characters), len, true),
26                   Field(TypeId::kTypeFloat, RandomUtils::RandomFloat(-999.f,
                     ↪  999.f))};
27    Row row(*fields);
28    ASSERT_TRUE(table_heap->InsertTuple(row, nullptr));
29    if (row_values.find(row.GetRowId().Get()) != row_values.end()) {
30      std::cout << row.GetRowId().Get() << std::endl;
31      ASSERT_TRUE(false);
32    } else {
33      row_values.emplace(row.GetRowId().Get(), fields);
34      row_ids.push_back(row.GetRowId());
35      size++;
36    }
37    delete[] characters;
38  }
39  ASSERT_EQ(row_nums, row_values.size());
40  ASSERT_EQ(row_nums, size);
41  for (auto row_kv : row_values) {
42    size--;
43    Row row(RowId(row_kv.first));
44    table_heap->GetTuple(&row, nullptr);
45    ASSERT_EQ(schema.get()->GetColumnCount(), row.GetFields().size());
46    for (size_t j = 0; j < schema.get()->GetColumnCount(); j++) {
```

```
47      ASSERT_EQ(CmpBool::kTrue,
        ↪  row.GetField(j)->CompareEquals(row_kv.second->at(j)));
48    }
49    // free spaces
50    delete row_kv.second;
51  }
52  // test iterator
53  ASSERT_EQ(size, 0);
54  int count = 0;
55  for(auto iter = table_heap->Begin(nullptr); iter != table_heap->End(); ++iter){
56    Row row(iter->GetRowId());
57    table_heap->GetTuple(&row, nullptr);
58    ASSERT_EQ(schema.get()->GetColumnCount(), row.GetFields().size());
59    ASSERT_EQ(schema.get()->GetColumnCount(), iter->GetFields().size());
60    for(size_t j = 0; j < schema.get()->GetColumnCount(); j++){
61      ASSERT_EQ(CmpBool::kTrue, row.GetField(j)->CompareEquals(*iter->GetField(j)));
62    }
63    count++;
64  }
65  ASSERT_EQ(count, row_nums);
66
67  // test update
68  std::unordered_map<int64_t, Fields *> new_row_values;
69  std::set<page_id_t> used_pages;
70  for(int i = 0; i < row_nums; i++){
71    int32_t len = RandomUtils::RandomInt(0, 64);
72    char *characters = new char[len];
73    RandomUtils::RandomString(characters, len);
74    Fields *fields = new Fields{Field(TypeId::kTypeInt, i), Field(TypeId::kTypeChar,
      ↪  const_cast<char *>(characters), len, true),
75                                Field(TypeId::kTypeFloat,
                                  ↪  RandomUtils::RandomFloat(-999.f, 999.f))};
76    Row row(*fields);
77    ASSERT_EQ(true, table_heap->UpdateTuple(row, row_ids.at(i), nullptr));
78    ASSERT_EQ(false, row.GetRowId().GetPageId()== INVALID_PAGE_ID);
79    new_row_values[row.GetRowId().Get()] = fields;
80    delete[] characters;
81  }
82  // check the update result
83  for(auto row_kv2 : new_row_values){
84    Row row(RowId(row_kv2.first));
85    ASSERT_EQ(true, table_heap->GetTuple(&row, nullptr));
86    used_pages.insert(row.GetRowId().GetPageId());
87    ASSERT_EQ(schema.get()->GetColumnCount(), row.GetFields().size());
```

```
88      for(size_t j = 0; j < schema.get()->GetColumnCount(); j++){
89        ASSERT_EQ(CmpBool::kTrue,
          ↪  row.GetField(j)->CompareEquals(row_kv2.second->at(j)));
90      }
91      table_heap->ApplyDelete(row.GetRowId(), nullptr);
92      ASSERT_EQ(false, table_heap->GetTuple(&row, nullptr));
93      delete row_kv2.second;
94    }
95    // test free
96    table_heap->FreeTableHeap();
97    auto *disk = new DiskManager(db_file_name);
98    auto *bpm = new BufferPoolManager(100, disk);
99    ASSERT_EQ(true, bpm->IsPageFree(*used_pages.begin()));
100   for(auto page_id : used_pages){
101     ASSERT_EQ(true, bpm->IsPageFree(page_id));
102   }
103   delete disk;
104   delete bpm;
105   delete disk_mgr_;
106   delete bpm_;
107 }
```

本节的 test 全部通过。

3 INDEX MANAGER

3.1 summary

Index Manager 负责数据表索引的实现和管理，包括：索引的创建和删除，索引键的等值查找，索引键的范围查找（返回对应的迭代器），以及插入和删除键值等操作，并对外提供相应的接口。为了提高搜索的效率，我们实现基于磁盘的 B+ 树动态索引结构。首先完成 b+ 树数据结构的实现，然后建立在此基础上建立索引。

3.2 BPlusTree

首先我们要实现 B+ 树的数据结构，在这里是基于磁盘的数据页实现的，因此我们需要实现 BPlusTreeInternal 和 BPlusTreeLeaf，为 BPlusTree 的实现提供接口。

3.2.1 BPlusTreePage

我们知道，b+ 树存在两种节点，一种是内部节点，一种是叶子节点。内部节点存储的是索引键和指向子节点的指针，叶子节点存储的是索引键和指向记录的指针。这二

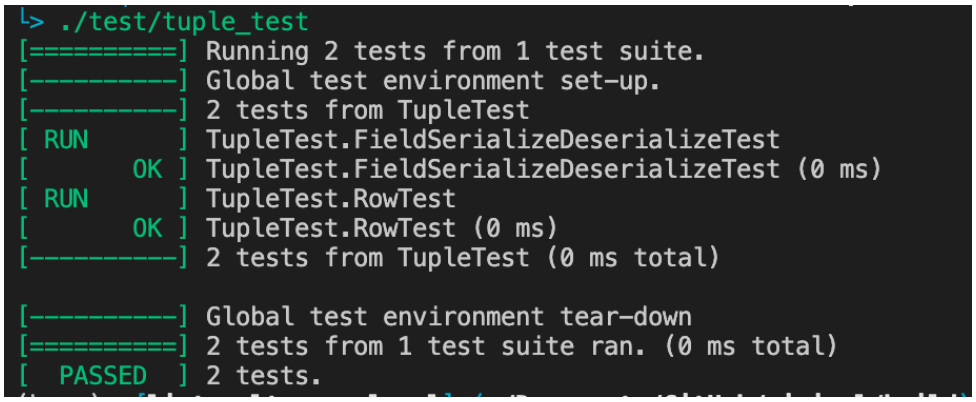

图 9: Tuple Test

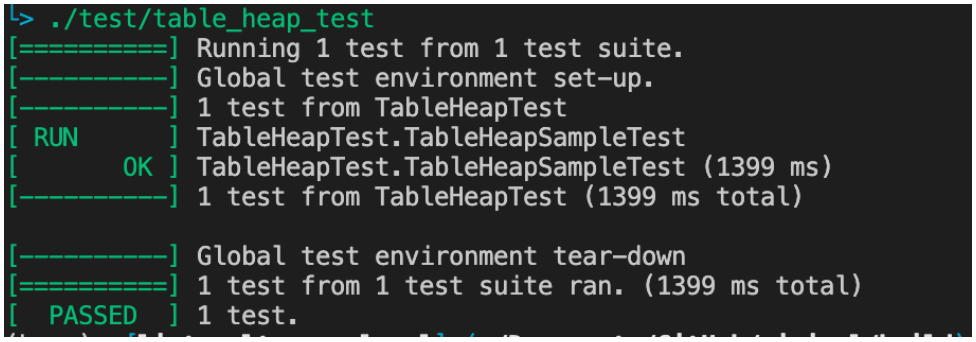

图 10: Table Heap Test

者具有部分相同的成员变量，因此设计上可以通过一个共同父类继承得到两个子类，这里的`BPlusTreepage`是`BPlusTreeInternalPage`和`BPlusTreeLeafPage`的父类。包含了中间结点和叶子结点的共同成员变量。

- **page_type_**: 用于标识当前页是叶子页还是中间页。
- **key_size_**: 索引键的大小。
- **lsn_**: 数据页的日志序列号，该模块中不会用到；
- **size_**: 当前结点中存储 Key-Value 键值对的数量；
- **max_size_**: 当前结点中最多可以存储的 Key-Value 键值对的数量；
- **parent_page_id_**: 父结点对应数据页的 page_id;
- **page_id_**: 当前数据页的 page_id;

这个类的实现较为简单，主要是一些 getter 和 setter 函数，需要注意的点就是最小 size 应当是最大 size 的一半，这部分实现较为简单，这里就不再赘述。

3.2.2 BPlusTreeInternalPage

内部节点的实现，继承自 BPlusTreePage，内部节点存储的是索引键和指向子节点的指针，因此需要实现插入、删除、查找等操作，除此以外，BPlusTree 还会做 merge 和 split 操作，需要一些将子节点的 key 和 value 移动到另一个节点的操作。以下是各个接口的具体实现：

Init 初始化 page_id, parent_page_id, page_type, key_size, max_size, size, lsn 等成员变量。

```
void InternalPage::Init(page_id_t page_id, page_id_t parent_id, int key_size, int
↪  max_size) {
  SetPageType(IndexPageType::INTERNAL_PAGE);
  SetSize(0);
  SetPageId(page_id);
  SetParentPageId(parent_id);
  SetKeySize(key_size);
  SetMaxSize(max_size);
  SetLSN(INVALID_LSN);
}
```

Search

Lookup 这里要使用二分查找来提高搜索的速度，这里调用了 KeyManager 的 CompareKeys 函数来比较两个 key 的大小，我们应当是无感知地获得比较的结果。

```
page_id_t InternalPage::Lookup(const GenericKey *key, const KeyManager &KM) {
  int left = 1, right = GetSize() - 1;
  while (left <= right)
  {
    int mid = (left + right) / 2;
    if (KM.CompareKeys(KeyAt(mid), key) <= 0)
      left = mid + 1;
    else
      right = mid - 1;
  }
  return ValueAt(right);
}
```

Insert

实现插入操作需要使用的几个函数。

PopulateNewRoot 给建立的新的根节点进行初始化，将两个子节点的 page_id 和 key_id 插入到新的根节点中。

```
void InternalPage::PopulateNewRoot(const page_id_t &old_value, GenericKey *new_key,
↪  const page_id_t &new_value) {
  SetSize(2);
  SetValueAt(0, old_value);
  SetKeyAt(1, new_key);
  SetValueAt(1, new_value);
}
```

InsertNodeAfter 在给定的位置后插入一个新的 key 和 value，这里需要将原来的 key 和 value 向后移动一个位置，然后插入新的 key 和 value，返回当前的 size。

```
int InternalPage::InsertNodeAfter(const page_id_t &old_value, GenericKey *new_key,
↪  const page_id_t &new_value) {
  int index = ValueIndex(old_value);
  if (index == -1)
    return -1;
  int size = GetSize();
  for (int i = size; i > index + 1; --i)
  {
    PairCopy(pairs_off + i * pair_size, pairs_off + (i - 1) * pair_size);
  }
  SetKeyAt(index + 1, new_key);
  SetValueAt(index + 1, new_value);
  IncreaseSize(1);
  return size + 1;
}
```

split

MoveHalfTo 将当前节点的一半 key 和 value 移动到另一个节点，这个函数调用了 CopyNfrom 这个复制函数，将当前节点的一半 key 和 value 复制到另一个节点中，然后将当前节点的一半 key 和 value 删除。

```
void InternalPage::MoveHalfTo(InternalPage *recipient, BufferPoolManager
↪  *buffer_pool_manager) {
  int size = GetSize();
  int left_size = size / 2, right_size = size - left_size;
  recipient->CopyNFrom(pairs_off + left_size * pair_size, right_size,
  ↪  buffer_pool_manager);
```

```
  IncreaseSize(-right_size);
}
```

CopyNFrom 向自身拷贝另一个节点的一部分 key 和 value，这里调用了 PairCopy 函数，将另一个节点的 key 和 value 复制到自身中。此时是中间结点还要对儿子节点的 parent 进行更新。

```
void InternalPage::CopyNFrom(void *src, int size, BufferPoolManager
↪  *buffer_pool_manager) {
  // copy entries
  int old_size = GetSize();
  PairCopy(pairs_off + old_size * pair_size, src, size);
  IncreaseSize(size);
  // update parent page id and get persisted
  for(int i = old_size; i < size; ++i)
  {
    page_id_t page_id = ValueAt(i);
    auto *page = buffer_pool_manager->FetchPage(page_id);
    ASSERT(page != nullptr, "page is nullptr");
    auto *internal_page = reinterpret_cast<BPlusTreePage *>(page->GetData());
    internal_page->SetParentPageId(GetPageId());
    buffer_pool_manager->UnpinPage(page_id, true);
  }
}
```

Remove

Remove 在中间结点中删除特定的 key 和 value。

```
void InternalPage::Remove(int index) {
  int size = GetSize();
  for (int i = index; i < size - 1; ++i)
  {
    PairCopy(pairs_off + i * pair_size, pairs_off + (i + 1) * pair_size);
  }
  IncreaseSize(-1);
}
```

RemoveAndReturnOnlyChild 删除当前节点的 key 和 value，返回当前节点的唯一的子节点的 page_id。这个函数只会在 AdjustRoot() 中调用。

```
page_id_t InternalPage::RemoveAndReturnOnlyChild() {
  page_id_t value = ValueAt(0);
  IncreaseSize(-1);
  return value;
}
```

merge

MoveAllTo 将自身的全部结点移动到另一个接收结点中，需要更新子节点的 parent_page_id。

```
void InternalPage::MoveAllTo(InternalPage *recipient, GenericKey *middle_key,
↪  BufferPoolManager *buffer_pool_manager) {
  int size = GetSize(), recipient_size = recipient->GetSize();
  recipient->CopyNFrom(pairs_off, size, buffer_pool_manager);
  recipient->SetKeyAt(recipient_size, middle_key);
  // update parent page id and get persisted
  for(int i = 0; i < size; ++i)
  {
    page_id_t page_id = ValueAt(i);
    auto *page = buffer_pool_manager->FetchPage(page_id);
    ASSERT(page != nullptr, "page is nullptr");
    auto *internal_page = reinterpret_cast<BPlusTreePage *>(page->GetData());
    internal_page->SetParentPageId(recipient->GetPageId());
    buffer_pool_manager->UnpinPage(page_id, true);
  }
  SetSize(0);
}
```

Redistribute

MoveFirstToEndOf 将另一个节点的第一个 key 和 value 移动到自身的最后一个位置。

```
void InternalPage::MoveFirstToEndOf(InternalPage *recipient, GenericKey *middle_key,
↪  BufferPoolManager *buffer_pool_manager) {
  recipient->CopyLastFrom(middle_key, ValueAt(0),buffer_pool_manager);
  Remove(0);
}
```

CopyLastFrom 将另一个节点的最后一个 key 和 value 复制到自身的最后一个位置。

```
void InternalPage::CopyLastFrom(GenericKey *key, const page_id_t value,
↪  BufferPoolManager *buffer_pool_manager) {
  int size = GetSize();
  SetKeyAt(size, key);
  SetValueAt(size, value);
  IncreaseSize(1);
  auto *page = buffer_pool_manager->FetchPage(value);
  ASSERT(page != nullptr, "page is nullptr");
  auto *internal_page = reinterpret_cast<BPlusTreePage *>(page->GetData());
  internal_page->SetParentPageId(GetPageId());
  buffer_pool_manager->UnpinPage(value, true);
}
```

MoveLastToFrontOf 将另一个节点的最后一个 key 和 value 移动到自身的第一个位置。

```
void InternalPage::MoveLastToFrontOf(InternalPage *recipient, GenericKey
↪  *middle_key,BufferPoolManager *buffer_pool_manager) {
  recipient->CopyFirstFrom(ValueAt(GetSize() - 1), buffer_pool_manager);
  recipient->SetKeyAt(0, KeyAt(GetSize() - 1));
  recipient->SetKeyAt(1, middle_key);
  Remove(GetSize() - 1);
}
```

CopyFirstFrom 将另一个节点的第一个 key 和 value 复制到自身的第一个位置，修改儿子节点的 parent_page_id。

```
void InternalPage::CopyFirstFrom(const page_id_t value, BufferPoolManager
↪  *buffer_pool_manager) {
  int size = GetSize();
  for(int i = size;i > 0;i--){
    PairCopy(pairs_off + i * pair_size, pairs_off + (i - 1) * pair_size);
  }
  SetValueAt(0, value);
  IncreaseSize(1);
  auto *page = buffer_pool_manager->FetchPage(value);
  ASSERT(page != nullptr, "page is nullptr");
  auto *internal_page = reinterpret_cast<BPlusTreePage *>(page->GetData());
  internal_page->SetParentPageId(GetPageId());
```

```
  buffer_pool_manager->UnpinPage(value, true);
}
```

3.2.3 BPlusTreeLeafPage

叶子节点的实现，继承自 BPlusTreePage，叶子节点存储的是索引键和指向记录的指针，大体的实现与内部节点类似，只是叶子节点不需要存储子节点的 page_id，而需要存储 next_page_id，用于在范围查找时返回下一个叶子节点的 page_id。以下是各个接口的具体实现：

Init 初始化 page_id, parent_page_id, page_type, key_size, max_size, size, lsn 等成员变量。

```
void LeafPage::Init(page_id_t page_id, page_id_t parent_id, int key_size, int
↪  max_size) {
  SetPageType(IndexPageType::LEAF_PAGE);
  SetKeySize(key_size);
  SetLSN(INVALID_LSN);
  SetSize(0);
  SetPageId(page_id);
  SetParentPageId(parent_id);
  SetMaxSize(max_size);
  SetNextPageId(INVALID_PAGE_ID);
}
```

Search

KeyIndex 在叶子节点中查找特定的 key，返回对应的 value。

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

Lookup 对给定的 key，查找是否存在。

```
bool LeafPage::Lookup(const GenericKey *key, RowId &value, const KeyManager &KM) {
  int index = KeyIndex(key, KM);
  if (index < GetSize() && KM.CompareKeys(KeyAt(index), key) == 0)
  {
    value = ValueAt(index);
    return true;
  }
  return false;
}
```

Insert 在叶子节点中插入特定的 key 和 value

```
int LeafPage::Insert(GenericKey *key, const RowId &value, const KeyManager &KM) {
  int index = KeyIndex(key, KM);
  if (index < GetSize() && KM.CompareKeys(KeyAt(index), key) == 0) {
    return GetSize();
  }
  int size = GetSize();
  for (int i = size; i > index; i--)
  {
    PairCopy(pairs_off + i * pair_size, pairs_off + (i - 1) * pair_size);
  }
  SetKeyAt(index, key);
  SetValueAt(index, value);
  SetSize(size + 1);
  return size + 1;
}
```

split

MoveHalfTo 将当前节点的一半 key 和 value 移动到另一个节点，这个函数调用了 CopyNfrom 这个复制函数，将当前节点的一半 key 和 value 复制到另一个节点中，然后将当前节点的一半 key 和 value 删除。

```
void LeafPage::MoveHalfTo(LeafPage *recipient) {
  int size = GetSize();
  int half = size / 2;
  recipient->CopyNFrom(pairs_off + half * pair_size, size - half);
```

```
  IncreaseSize(-size + half);
}
```

CopyNFrom 向自身拷贝另一个节点的一部分 key 和 value，这里调用了 PairCopy 函数，将另一个节点的 key 和 value 复制到自身中。

```
void LeafPage::CopyNFrom(void *src, int size) {
  PairCopy(pairs_off + GetSize() * pair_size, src, size);
  IncreaseSize(size);
}
```

Remove

RemoveAndDeleteRecord 在叶子节点中删除特定的 key 和 value。

```
int LeafPage::RemoveAndDeleteRecord(const GenericKey *key, const KeyManager &KM) {
  int index = KeyIndex(key, KM);
  if (index < GetSize() && KM.CompareKeys(KeyAt(index), key) == 0)
  {
    int size = GetSize();
    for (int i = index; i < size - 1; i++)
    {
      PairCopy(pairs_off + i * pair_size, pairs_off + (i + 1) * pair_size);
    }
    IncreaseSize(-1);
  }
  return GetSize();
}
```

merge

MoveAllTo 将自身的全部结点移动到另一个接收结点中，需要更新 next_page_id。

```
void LeafPage::MoveAllTo(LeafPage *recipient) {
  recipient->CopyNFrom(pairs_off, GetSize());
  recipient->SetNextPageId(GetNextPageId());
  SetSize(0);
}
```

Redistribute

MoveFirstToEndOf 将另一个节点的第一个 key 和 value 移动到自身的最后一个位置。

```
void LeafPage::MoveFirstToEndOf(LeafPage *recipient) {
  int size = GetSize();
  recipient->CopyLastFrom(KeyAt(0), ValueAt(0));
  for (int i = 0; i < size - 1; i++)
  {
    PairCopy(pairs_off + i * pair_size, pairs_off + (i + 1) * pair_size);
  }
  IncreaseSize(-1);
}
```

CopyLastFrom 将另一个节点的最后一个 key 和 value 复制到自身的最后一个位置。

```
void LeafPage::CopyLastFrom(GenericKey *key, const RowId value) {
  int size = GetSize();
  SetKeyAt(size, key);
  SetValueAt(size, value);
  IncreaseSize(1);
}
```

MoveLastToFrontOf 将另一个节点的最后一个 key 和 value 移动到自身的第一个位置。

```
void LeafPage::MoveLastToFrontOf(LeafPage *recipient) {
  int size = GetSize();
  recipient->CopyFirstFrom(KeyAt(size - 1), ValueAt(size - 1));
  IncreaseSize(-1);
}
```

CopyFirstFrom 将另一个节点的第一个 key 和 value 复制到自身的第一个位置。

```
void LeafPage::CopyFirstFrom(GenericKey *key, const RowId value) {
  int size = GetSize();
  for (int i = size; i > 0; i--)
  {
    PairCopy(pairs_off + i * pair_size, pairs_off + (i - 1) * pair_size);
```

```
  }
  SetKeyAt(0, key);
  SetValueAt(0, value);
  IncreaseSize(1);
}
```

3.2.4 BPlusTree implementation

在前面的部分，我们已经完成了叶子结点和中间结点的实现，接下来我们要实现 BPlusTree 的接口，完成 B+ 树的创建、插入、删除、查找和释放等操作。这里 BPlusTree 的成员变量包括：

- index_id_t index_id_;
- page_id_t root_page_id_INVALID_PAGE_ID;
- BufferPoolManager *buffer_pool_manager_;
- KeyManager processor_;
- int leaf_max_size_; 即 ((PAGE_SIZE - LEAF_PAGE_HEADER_SIZE) / (processor_.GetKeySize() + sizeof(RowId)) - 1)
- int internal_max_size_; 即 ((PAGE_SIZE - INTERNAL_PAGE_HEADER_SIZE) / (processor_.GetKeySize() + sizeof(page_id_t)) - 1)

Init 首先创建一个 BPlusTree 的实例，初始化一些成员变量，包括 key_size, internal_page_size, leaf_page_size, root_page_id, buffer_pool_manager 等。除此以外，我们还需要修改元数据页即 IndexRootsPage，将 root_page_id 写入到元数据页中，调用了`UpdateRootPageId`函数。

```
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) {
    root_page_id_ = INVALID_PAGE_ID;
    Page *page = buffer_pool_manager_->FetchPage(INDEX_ROOTS_PAGE_ID);
    IndexRootsPage *index_roots_page = reinterpret_cast<IndexRootsPage
    *>(page->GetData());
```

```
    if (!index_roots_page->GetRootId(index_id, &root_page_id_)) {
      root_page_id_ = INVALID_PAGE_ID;
      UpdateRootPageId(1);
    }
    buffer_pool_manager_->UnpinPage(INDEX_ROOTS_PAGE_ID, false);
    leaf_max_size_ = LEAF_PAGE_SIZE;
    internal_max_size_ = INTERNAL_PAGE_SIZE;
}
```

UpdateRootPageId 修改 index_roots_page 中的当前 index 的 root_page_id，用于索引查找 b+ 树的根节点。

```
void BPlusTree::UpdateRootPageId(int insert_record) {
  auto* page = buffer_pool_manager_->FetchPage(INDEX_ROOTS_PAGE_ID);
  auto* index_roots_page = reinterpret_cast<IndexRootsPage*>(page->GetData());
  if (insert_record) {
    index_roots_page->Insert(index_id_, root_page_id_);
  } else {
    index_roots_page->Update(index_id_, root_page_id_);
  }
  buffer_pool_manager_->UnpinPage(INDEX_ROOTS_PAGE_ID, true);
}
```

destroy

> 我认为这里的 destroy 设计默认值为 INVALID_PAGE_ID 时实际上要求是删除全树这件事情十分不合理，这样的表意不明确，在文档里面也没有提及，造成了许多的困扰。

销毁以传入的结点为根结点的 b+ 树，这里的销毁是递归的，首先销毁子节点，然后销毁自身，最后释放自身的 page。如果是默认情况就删除全树，还要更新元数据页中的 root_page_id。

```
void BPlusTree::Destroy(page_id_t current_page_id) {
 if(current_page_id == INVALID_PAGE_ID) {
    current_page_id = root_page_id_;
  }
    Page* page = buffer_pool_manager_->FetchPage(current_page_id);
```

```
    BPlusTreePage* node = reinterpret_cast<BPlusTreePage*>(page->GetData());
    if(node->IsLeafPage()) {
      buffer_pool_manager_->DeletePage(current_page_id);
      buffer_pool_manager_->UnpinPage(current_page_id, true);
    }else{
      InternalPage* internal = reinterpret_cast<InternalPage*>(node);
      for(int i = 0; i < internal->GetSize(); i++) {
        Destroy(internal->ValueAt(i));
      }
      buffer_pool_manager_->DeletePage(current_page_id);
      buffer_pool_manager_->UnpinPage(current_page_id, true);
  }
  if(current_page_id == root_page_id_) {
    auto head = buffer_pool_manager_->FetchPage(INDEX_ROOTS_PAGE_ID);
    IndexRootsPage* index_roots_page =
    ↪  reinterpret_cast<IndexRootsPage*>(head->GetData());
    index_roots_page->Delete(index_id_);
    buffer_pool_manager_->UnpinPage(INDEX_ROOTS_PAGE_ID, true);
  }
}
```

IsEmpty 判断 b+ 树是否为空，如果根结点 page_id 为 INVALID_PAGE_ID，那么 b+ 树为空。

```
bool BPlusTree::IsEmpty() const {
  return root_page_id_ == INVALID_PAGE_ID;
}
```

FindLeafPage 这是一个重要的函数，用于查找特定的 key 所在的叶子结点，在后面的插入、删除、查询中都会用到。

```
Page *BPlusTree::FindLeafPage(const GenericKey *key, page_id_t page_id, bool
↪  leftMost) {
  if(page_id == INVALID_PAGE_ID){
    return FindLeafPage(key, root_page_id_, leftMost);
  }
  Page *page = buffer_pool_manager_->FetchPage(page_id);
  auto *node = reinterpret_cast<BPlusTreePage *>(page->GetData());
  if(node->IsLeafPage()) {
    return page;
  }else{
```

```
    auto *internal_node = reinterpret_cast<InternalPage *>(node);
    page_id_t child_page_id = leftMost ? internal_node->ValueAt(0) :
    ↪  internal_node->Lookup(key, processor_);
    buffer_pool_manager_->UnpinPage(page_id, false);
    return FindLeafPage(key, child_page_id, leftMost);
  }
  return nullptr;
}
```

Search 查找特定的 key，返回对应的 value，这里调用了 FindLeafPage 函数，找到对应的叶子结点，然后调用叶子结点的 Lookup 函数，返回对应的 value。

```
bool BPlusTree::GetValue(const GenericKey *key, std::vector<RowId> &result, Txn
↪  *transaction) {
  if(IsEmpty()) {
    return false;
  }
  LeafPage *leaf_node = reinterpret_cast<LeafPage *>(FindLeafPage(key, root_page_id_,
  ↪  false)->GetData());
  RowId value;
  bool find = leaf_node->Lookup(key, value, processor_);
  if(find) {
    result.push_back(value);
  }
  buffer_pool_manager_->UnpinPage(leaf_node->GetPageId(), false);
  return find;
}
```

Insertion

Insert 插入一个 key-value 键值对，如果是个空树，那么直接插入到根结点中，否则调用插入到叶子结点的函数。返回值代表是否插入重复 key，我们是不允许插入重复 key 的。

```
bool BPlusTree::Insert(GenericKey *key, const RowId &value, Txn *transaction) {
  if(IsEmpty()) {
    StartNewTree(key, value);
    return true;
  }
  return InsertIntoLeaf(key, value, transaction);
}
```

StartNewTree 如果是一个空树，那么直接插入到根结点中，这里需要创建一个新的叶子结点，然后将新的叶子结点的 page_id 写入到元数据页中。

```
void BPlusTree::StartNewTree(GenericKey *key, const RowId &value) {
  Page *page = buffer_pool_manager_->NewPage(root_page_id_);
  ASSERT(page != nullptr, "out of memory");
  LeafPage *leaf_page = reinterpret_cast<LeafPage *>(page->GetData());
  leaf_page->Init(root_page_id_, INVALID_PAGE_ID, processor_.GetKeySize(),
  ↪  leaf_max_size_);
  leaf_page->Insert(key, value, processor_);
  buffer_pool_manager_->UnpinPage(root_page_id_, true);
  UpdateRootPageId(0);
}
```

InsertIntoLeaf 插入一个 key-value 键值对到叶子结点中，如果叶子结点的 size 超过了最大值，那么需要进行 split 操作，将一半的 key 和 value 移动到另一个新的叶子结点中，然后将新的 key 和 value 插入到叶子结点中，然后将新的叶子插到 parent 中。

```
bool BPlusTree::InsertIntoLeaf(GenericKey *key, const RowId &value, Txn *transaction)
↪  {
  static int count = 0;
  Page *leaf_page = FindLeafPage(key, root_page_id_, false);
  LeafPage *leaf_node = reinterpret_cast<LeafPage *>(leaf_page->GetData());
  RowId receive_value;
  if(leaf_node->Lookup(key, receive_value, processor_)) {
    cout << "Duplicate key" << endl;
    buffer_pool_manager_->UnpinPage(leaf_node->GetPageId(), false);
    return false;
  }
  if(leaf_node->GetSize() < leaf_node->GetMaxSize()) {
    leaf_node->Insert(key, value, processor_);
    buffer_pool_manager_->UnpinPage(leaf_node->GetPageId(), true);
  }else{
    // deal with split
    LeafPage *new_leaf_node = Split(leaf_node, transaction);
    if(processor_.CompareKeys(key, new_leaf_node->KeyAt(0)) < 0) {
      leaf_node->Insert(key, value, processor_);
    }else{
      new_leaf_node->Insert(key, value, processor_);
    }
    InsertIntoParent(leaf_node, new_leaf_node->KeyAt(0), new_leaf_node, transaction);
    buffer_pool_manager_->UnpinPage(leaf_node->GetPageId(), true);
    buffer_pool_manager_->UnpinPage(new_leaf_node->GetPageId(), true);
```

```
  }
  return true;
}
```

Split 这一部分包含叶子结点和中间结点的 split 操作，首先创建一个新的叶子结点或者中间结点，然后将当前结点的一半 key 和 value 移动到新的结点中，然后返回新的结点。叶子结点需要更新 next_page_id，中间结点需要更新子节点的 parent_page_id。这里的实现依赖于前面结点页的实现，

```
BPlusTreeInternalPage *BPlusTree::Split(InternalPage *node, Txn *transaction) {
  page_id_t new_page_id;
  Page *page = buffer_pool_manager_->NewPage(new_page_id);
  ASSERT(page != nullptr, "out of memory");
  InternalPage *new_internal_page = reinterpret_cast<InternalPage
  ↪  *>(page->GetData());
  new_internal_page->Init(new_page_id, node->GetParentPageId(),
  ↪  processor_.GetKeySize(), internal_max_size_);
  node->MoveHalfTo(new_internal_page, buffer_pool_manager_);
  return new_internal_page;
}

BPlusTreeLeafPage *BPlusTree::Split(LeafPage *node, Txn *transaction) {
  page_id_t new_page_id;
  Page *page = buffer_pool_manager_->NewPage(new_page_id);
  ASSERT(page != nullptr, "out of memory");
  LeafPage *new_leaf_page = reinterpret_cast<LeafPage *>(page->GetData());
  new_leaf_page->Init(new_page_id, node->GetParentPageId(), processor_.GetKeySize(),
  ↪  leaf_max_size_);
  node->MoveHalfTo(new_leaf_page);
  new_leaf_page->SetNextPageId(node->GetNextPageId());
  node->SetNextPageId(new_page_id);
  return new_leaf_page;
}
```

InsertIntoParent 插入一个 key 和 value 到父节点中，如果此时的结点已经是根节点了，那么就要新建一个根节点作为父节点，然后将原来的根节点和新的结点插入到新的根节点中，然后更新根节点的 page_id。否则就可以根据父节点的 size 判断是否可以插入，如果不能插入就需要进行 split 操作，然后递归调用 InsertIntoParent 函数，直到插入成功。

```
void BPlusTree::InsertIntoParent(BPlusTreePage *old_node, GenericKey *key,
↪  BPlusTreePage *new_node, Txn *transaction) {
  if(old_node->IsRootPage()) {
    page_id_t new_page_id;
    Page *page = buffer_pool_manager_->NewPage(new_page_id);
    ASSERT(page != nullptr, "out of memory");
    InternalPage *new_root_node = reinterpret_cast<InternalPage *>(page->GetData());
    new_root_node->Init(new_page_id, INVALID_PAGE_ID, processor_.GetKeySize(),
    ↪  internal_max_size_);
    new_root_node->PopulateNewRoot(old_node->GetPageId(), key,
    ↪  new_node->GetPageId());
    old_node->SetParentPageId(new_page_id);
    new_node->SetParentPageId(new_page_id);
    root_page_id_ = new_page_id;
    UpdateRootPageId(0);
    buffer_pool_manager_->UnpinPage(new_page_id, true);
  }else{
    page_id_t page_id = old_node->GetParentPageId();
    Page *page = buffer_pool_manager_->FetchPage(page_id);
    InternalPage *parent_node = reinterpret_cast<InternalPage *>(page->GetData());
    if(parent_node->GetSize() < parent_node->GetMaxSize()) {
      parent_node->InsertNodeAfter(old_node->GetPageId(), key,
      ↪  new_node->GetPageId());
      new_node->SetParentPageId(page_id);
      buffer_pool_manager_->UnpinPage(page_id, true);
    }else{
      InternalPage *new_internal_node = Split(parent_node, transaction);
      if(processor_.CompareKeys(key, new_internal_node->KeyAt(0)) < 0) {
        new_node->SetParentPageId(page_id);
        parent_node->InsertNodeAfter(old_node->GetPageId(), key,
        ↪  new_node->GetPageId());
      }else{
        new_node->SetParentPageId(new_internal_node->GetPageId());
        new_internal_node->InsertNodeAfter(old_node->GetPageId(), key,
        ↪  new_node->GetPageId());
      }
      InsertIntoParent(parent_node, new_internal_node->KeyAt(0), new_internal_node,
      ↪  transaction);
      buffer_pool_manager_->UnpinPage(parent_node->GetPageId(), true);
      buffer_pool_manager_->UnpinPage(new_internal_node->GetPageId(), true);
    }
  }
}
```

Deletion

插入的过程中需要分裂，删除的过程中则需要合并和重分配。下面是各个接口的具体实现：

Remove 如果是空树的话，直接返回 false，否则找到 leaf 后调用 RemoveAndDeleteRecord 函数，如果删除后叶子结点的 size 小于最小值，那么需要进行合并或者重分配操作。

```
void BPlusTree::Remove(const GenericKey *key, Txn *transaction) {
  if(IsEmpty()) {
    return;
  }
  Page* page = FindLeafPage(key, root_page_id_, false);
  LeafPage *leaf_node = reinterpret_cast<LeafPage *>(page->GetData());
  int new_size = leaf_node->RemoveAndDeleteRecord(key, processor_);
  if(new_size < leaf_node->GetMinSize()) {
    if(CoalesceOrRedistribute(leaf_node, transaction)) {
      buffer_pool_manager_->UnpinPage(leaf_node->GetPageId(), false);
      buffer_pool_manager_->DeletePage(leaf_node->GetPageId());
    }else{
      buffer_pool_manager_->UnpinPage(leaf_node->GetPageId(), true);
    }
  }else{
    buffer_pool_manager_->UnpinPage(leaf_node->GetPageId(), true);
  }
}
```

CoalesceOrRedistribute 这里使用模板实现了合并和重分配操作，如果叶子结点的 size 小于最小值，那么需要进行合并或者重分配操作。合并的条件是兄弟节点与自身的 size 之和小于最大值，我们的实现逻辑是优先实现重分配，如果重分配失败，那么就进行合并操作，合并操作需要递归调用 CoalesceOrRedistribute 函数，直到合并成功。这里返回值的意义是要删除的叶子节点所在的 page 是否要删除，即是否发生了合并。

```
  template <typename N>
bool BPlusTree::CoalesceOrRedistribute(N *&node, Txn *transaction) {
  if (node->IsRootPage()) {
    return AdjustRoot(node);
  }
  auto parent = reinterpret_cast<InternalPage
  ↪  *>(buffer_pool_manager_->FetchPage(node->GetParentPageId())->GetData());
  int index = parent->ValueIndex(node->GetPageId());
```

```
  page_id_t sibilings = index == 0 ? parent->ValueAt(1) : parent->ValueAt(index - 1);
  auto sibilings_page = reinterpret_cast<N
  ↪  *>(buffer_pool_manager_->FetchPage(sibilings)->GetData());
  if(node->GetSize() + sibilings_page->GetSize() > node->GetMaxSize()) {
    Redistribute(sibilings_page, node, index);
    buffer_pool_manager_->UnpinPage(sibilings_page->GetPageId(), true);
    buffer_pool_manager_->UnpinPage(node->GetPageId(), true);
    return false;
  }else{
    if(Coalesce(sibilings_page, node, parent, index, transaction)) {
      if(parent->IsRootPage()) {
        if(AdjustRoot(parent)) {
          buffer_pool_manager_->UnpinPage(parent->GetPageId(), false);
          buffer_pool_manager_->DeletePage(parent->GetPageId());
        } else{
          buffer_pool_manager_->UnpinPage(parent->GetPageId(), true);
        }
      }else{
        if(CoalesceOrRedistribute(parent, transaction)) {
          buffer_pool_manager_->UnpinPage(parent->GetPageId(), false);
          buffer_pool_manager_->DeletePage(parent->GetPageId());
        } else{
          buffer_pool_manager_->UnpinPage(parent->GetPageId(), true);
        }
      }
    }else{
      buffer_pool_manager_->UnpinPage(parent->GetPageId(), true);
    }
    return true;
  }
}
```

Coalesce 合并操作，将兄弟节点的 key 和 value 移动到自身，这里调用了 MoveAllTo 函数，根据 index 判断合并的方向，是合并到兄弟还是合并到自身，然后调用 MoveAllTo 函数。这里的操作也可以通过模板完成。

```
bool BPlusTree::Coalesce(LeafPage *&neighbor_node, LeafPage *&node, InternalPage
↪  *&parent, int index,
                         Txn *transaction) {
  if(index == 0){
    neighbor_node->MoveAllTo(node);
    LeafPage *temp = neighbor_node;
```

```
    neighbor_node = node;
    node = temp;
    parent->Remove(1);
  }else{
    node->MoveAllTo(neighbor_node);
    parent->Remove(index);
  }
  return parent->GetSize() < parent->GetMinSize();
}

bool BPlusTree::Coalesce(InternalPage *&neighbor_node, InternalPage *&node,
↪  InternalPage *&parent, int index,
                         Txn *transaction) {
  if(index == 0){
    neighbor_node->MoveAllTo(node, parent->KeyAt(1), buffer_pool_manager_);
    node->MoveAllTo(neighbor_node, parent->KeyAt(0), buffer_pool_manager_);
  }else{
    node->MoveAllTo(neighbor_node, parent->KeyAt(index), buffer_pool_manager_);
  }
  parent->Remove(index);
  return parent->GetSize() < parent->GetMinSize();
}
```

Redistribute 重分配操作，将兄弟节点的 key 和 value 移动到自身，这里调用了 MoveFirstToEndOf 和 MoveLastToFrontOf 函数，将兄弟节点的第一个 key 和 value 移动到自身的最后一个位置，或者将兄弟节点的最后一个 key 和 value 移动到自身的第一个位置。

```
void BPlusTree::Redistribute(LeafPage *neighbor_node, LeafPage *node, int index) {
  auto parent = reinterpret_cast<InternalPage
  ↪  *>(buffer_pool_manager_->FetchPage(node->GetParentPageId())->GetData());
  if(index == 0){
    neighbor_node->MoveFirstToEndOf(node);
    parent->SetKeyAt(1, neighbor_node->KeyAt(0));
  }else{
    neighbor_node->MoveLastToFrontOf(node);
    parent->SetKeyAt(index, node->KeyAt(0));
  }
  buffer_pool_manager_->UnpinPage(parent->GetPageId(), true);
}
void BPlusTree::Redistribute(InternalPage *neighbor_node, InternalPage *node, int
↪  index) {
```

```
  auto parent = reinterpret_cast<InternalPage
  ↪  *>(buffer_pool_manager_->FetchPage(node->GetParentPageId())->GetData());
  if(index == 0){
    neighbor_node->MoveFirstToEndOf(node, parent->KeyAt(1), buffer_pool_manager_);
    parent->SetKeyAt(1, neighbor_node->KeyAt(0));
  }else{
    neighbor_node->MoveLastToFrontOf(node, parent->KeyAt(index),
    ↪  buffer_pool_manager_);
    parent->SetKeyAt(index, node->KeyAt(0));
  }
  buffer_pool_manager_->UnpinPage(parent->GetPageId(), true);
}
```

AdjustRoot 在前面合并结点后，如果父节点是根节点的话，需要对下面两种情况进行处理：第一是删除了根节点的最后一个元素，但是根节点还有一个子节点；第二是根节点是最后一个节点了，那么就是删除整个 b+ 树。

```
bool BPlusTree::AdjustRoot(BPlusTreePage *old_root_node) {
  // case 1: when you delete the last element in root page, but root page still has
  ↪  one last child
  if(old_root_node->GetSize() == 1 && !old_root_node->IsLeafPage()){
    InternalPage *old_page = reinterpret_cast<InternalPage *>(old_root_node);
    page_id_t child_page_id = old_page->RemoveAndReturnOnlyChild();
    Page *child_page = buffer_pool_manager_->FetchPage(child_page_id);
    BPlusTreePage *child_node = reinterpret_cast<BPlusTreePage
    ↪  *>(child_page->GetData());
    child_node->SetParentPageId(INVALID_PAGE_ID);
    root_page_id_ = child_page_id;
    UpdateRootPageId(0);
    buffer_pool_manager_->UnpinPage(child_page_id, true);
    return true;
  }
  // case 2: delete the last element in whole b+ tree,
  if(old_root_node->GetSize() == 0 && old_root_node->IsLeafPage() ){
    root_page_id_ = INVALID_PAGE_ID;
    UpdateRootPageId(0);
    return true;
  }
  return false;
}
```

3.3 IndexIterator

前面我们已经实现了整个 b+ 树的创建和插入删除查找的功能。现在我们实现迭代器，用于遍历 b+ 树中的叶子节点的元素。

3.3.1 IndexIterator implementation

IndexIterator 这里是使用 page_id,BufferPoolManager,index 创建的迭代器，默认的构造函数就是一个空的迭代器。

```
IndexIterator::IndexIterator() = default;

IndexIterator::IndexIterator(page_id_t page_id, BufferPoolManager *bpm, int index)
    : current_page_id(page_id), item_index(index), buffer_pool_manager(bpm) {
      if(current_page_id != INVALID_PAGE_ID)
        page = reinterpret_cast<LeafPage
        ↪  *>(buffer_pool_manager->FetchPage(current_page_id)->GetData());
}
```

IndexIterator 析构函数，用于释放迭代器的资源，这里需要释放 page，因为每个迭代器都在构造时候 fetch 了一个 page。

```
IndexIterator::~IndexIterator() {
  if (current_page_id != INVALID_PAGE_ID)
    buffer_pool_manager->UnpinPage(current_page_id, false);
}
```

operator* * 操作符，返回当前迭代器指向的元素，这里调用了 page 的 KeyAt 和 ValueAt 函数。

```
std::pair<GenericKey *, RowId> IndexIterator::operator*() {
  return std::make_pair(page->KeyAt(item_index), page->ValueAt(item_index));
}
```

operator++ ++ 操作符，用于迭代器的自增，这里需要判断是否到达了叶子结点的末尾，如果到达了叶子结点的末尾，那么需要将当前的 page 释放，然后将 page 指向下一个叶子结点。

```
IndexIterator &IndexIterator::operator++() {
  if (item_index + 1 < page->GetSize()) {
    item_index++;
  } else {
    page_id_t next_page_id = page->GetNextPageId();
    buffer_pool_manager->UnpinPage(current_page_id, false);
    current_page_id = next_page_id;
    if (current_page_id != INVALID_PAGE_ID) {
      page = reinterpret_cast<LeafPage
      ↪  *>(buffer_pool_manager->FetchPage(current_page_id)->GetData());
      item_index = 0;
    } else{
      page = nullptr;
      current_page_id = INVALID_PAGE_ID;
      item_index = 0;
    }
  }
  return *this;
}
```

operator== == 操作符，用于判断两个迭代器是否相等，这里需要判断两个迭代器的 page_id 和 item_index 是否相等。

```
bool IndexIterator::operator==(const IndexIterator &itr) const {
  return current_page_id == itr.current_page_id && item_index == itr.item_index;
}
```

operator!= != 操作符，用于判断两个迭代器是否不相等，这里调用了 == 操作符。

```
bool IndexIterator::operator!=(const IndexIterator &itr) const {
  return !(*this == itr);
}
```

3.3.2 BPlusTree IndexIterator

Begin 返回一个迭代器，指向 b+ 树的第一个元素，这里需要找到 b+ 树的第一个叶子结点，然后返回一个迭代器，指向第一个叶子结点的第一个元素。

```
IndexIterator BPlusTree::Begin() {
  page_id_t page_id = root_page_id_;
  Page *page = buffer_pool_manager_->FetchPage(page_id);
  BPlusTreePage *node = reinterpret_cast<BPlusTreePage *>(page->GetData());
  while(!node->IsLeafPage()) {
    InternalPage *internal_node = reinterpret_cast<InternalPage *>(node);
    page_id = internal_node->ValueAt(0);
    buffer_pool_manager_->UnpinPage(node->GetPageId(), false);
    node = reinterpret_cast<BPlusTreePage
    ↪  *>(buffer_pool_manager_->FetchPage(page_id)->GetData());
  }
  return IndexIterator(page_id, buffer_pool_manager_);
}
```

另外，Begin 函数有一个 overload 版本，可以指定一个 key，返回一个迭代器，指向第一个大于等于 key 的元素。

```
IndexIterator BPlusTree::Begin(const GenericKey *key) {
  Page * page = FindLeafPage(key, root_page_id_, false);
  if(page == nullptr){
    return IndexIterator();
  }
  LeafPage *node = reinterpret_cast<LeafPage *>(page->GetData());
  int index = node->KeyIndex(key, processor_);
  buffer_pool_manager_->UnpinPage(node->GetPageId(), false);
  return IndexIterator(node->GetPageId(), buffer_pool_manager_, index);
}
```

End 返回一个迭代器，根据迭代器的定义，应当是一个空的迭代器。

```
IndexIterator BPlusTree::End() {
  return IndexIterator();
}
```

3.4 Test

使用框架提供的测试进行测试，其中 BPlusTreeTests 测试的 n 达到 1e6，测试结果如下：

本节测试全部通过。

```
└> ./test/b_plus_tree_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.SampleTest
[       OK ] BPlusTreeTests.SampleTest (105630 ms)
[----------] 1 test from BPlusTreeTests (105630 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (105630 ms total)
[  PASSED  ] 1 test.
```

图 11: test7

```
└> ./test/b_plus_tree_index_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from BPlusTreeTests
[ RUN      ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest
[       OK ] BPlusTreeTests.BPlusTreeIndexGenericKeyTest (0 ms)
[ RUN      ] BPlusTreeTests.BPlusTreeIndexSimpleTest
[       OK ] BPlusTreeTests.BPlusTreeIndexSimpleTest (135 ms)
[----------] 2 tests from BPlusTreeTests (136 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (136 ms total)
[  PASSED  ] 2 tests.
```

图 12: test8

```
└> ./test/index_iterator_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from BPlusTreeTests
[ RUN      ] BPlusTreeTests.IndexIteratorTest
[       OK ] BPlusTreeTests.IndexIteratorTest (115 ms)
[----------] 1 test from BPlusTreeTests (115 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (115 ms total)
[  PASSED  ] 1 test.
```

图 13: test9

所给的测试样例只进行了一半的删除，我添加了剩余一半的删除测试，否则 AdjustRoot 函数无法测试。

4 Summary

我还负责对系统全部实现后进行了测试，具体的测试在验收时都已经演示过了，这里就不再赘述。至此，我个人完成的部分已经全部实现并通过了测试。对于我完成的部分，b+ 树的部分是问题最大的，后续系统功能的测试中出问题的地方也几乎都是索引引发的，经过了漫长的调试，最终得到了一个可以使用的版本。对于这样的大型工程，单元测试的完善可以帮助我们更好地实现整体的功能，我认为目前框架内提供的单元测试还不足以完全测试代码的正确性。在 tableheap 这一部分可以说几乎没有测试，这些单元测试的不完备导致我们在后面的程序调试中需要往前寻找问题而这一过程是十分困难而且耗时的。我希望能够把我完善的部分测试（lab2,3）的单元测试加入到框架中，这样可以更好地帮助我们调试程序。也希望未来框架能够越来越好，现在有一些地方的说明也不是很清晰，比如 destroy 函数的默认值，这样的设计在文档中没有提及，导致了很多的困扰。报告的撰写也花费了不少的精力，感谢能看到这里的助教老师，希望我们数据库系统这门课能越来越好，也希望这样一个巨大工作量和难度的大程能在考核中获得更多的分数，而不是仅仅 15 分。